Vision et reconnaissance d'objet :
Comparaison d'algorithmes sur MNIST

Demonet, Grasset, Puppo & Rouhard

5 Mai 2015

Table des matières

Introduction

> Those who predict, don't have knowledge.
>
> Lao Tseu

Si Lao Tseu rappelle dans *Le Livre de la Voie et de la Vertu* que le savoir est meilleur que la prédiction, en l'absence de savoir la meilleure idée du réel est celle que l'on prédit à partir de nos connaissances du passé. Ainsi, toute forme de prédiction est basée sur la connaissance du passé pour produire la meilleure estimation possible de nouveaux évènements. La classification automatique, c'est-à-dire faite par ordinateur, relève de ce principe, en effet le concept de savoir est étranger à un ordinateur, par contre il peut prédire le futur ou le nouveau, en fonction du passé ou du connu et le tout à très grande vitesse.

Sur un célèbre moteur de recherche, il existe une fonction qui permet en lui important une image de déterminer ce qu'elle représente. Ce type d'exercice relève parfaitement de la prédiction : à partir d'un passé (qui est un entraînement), on donne une prédiction sur du nouveau, de l'inconnu. C'est ce type de fonctionnalité qui va faire l'objet de ce projet, comment à partir d'une base de données sur laquelle on a de l'information, on peut prédire l'information manquante sur de nouvelles observations. Par exemple, une banque qui veut lire de manière automatique les numéros de compte sur un chèque va recourir à ce type de fonctionnalité.

Quelles méthodes appliquer pour faire de la prédiction statistique ? Comment choisir entre nos méthodes ? Comment appliquer nos méthodes de manière optimale ? Est-ce que toutes les méthodes répondent aux mêmes problématiques ? Pour répondre à ces questions, nous allons procéder en quatre parties comme suit :

- Pour commencer, nous décrirons les principes de l'apprentissage automatique et nous présenterons nos supports de travail pour répondre à ce problème, en particulier dans le cas de la lecture de chiffres manuscrits ;
- Nous ferons une présentation plus en détail de ce qu'est l'apprentissage machine, comment nous construisons et calibrons nos estimateurs et comment analyser les performances de nos algorithmes pour ensuite les comparer entre eux ;
- Ensuite, nous présenterons les différentes familles d'algorithmes que nous avons utilisées et leurs caractéristiques attendues ;
- Puis, nous analyserons les résultats obtenus en termes de performance de prédiction et de complexité algorithmique ;
- Enfin dans une dernière partie plus exploratoire, nous décrirons d'autres méthodes peu "orthodoxes" que nous avons utilisées ou que nous voulions utiliser en fonction de problématiques différentes.

Première partie
L'apprentissage automatique et la classification d'images

1 Présentation du cadre de travail

1.1 La classification automatique d'images

Le phénomène de reconnaissance d'éléments par la vision est depuis plus d'une vingtaine d'années au cœur de la recherche en intelligence artificielle : comment expliquer à une machine un des fonctionnements les plus élémentaires du cerveau humain ?

Les avancées faites dans ce domaine ont amené aujourd'hui à de nombreuses technologies dont l'utilisation est devenue quotidienne : combien de fois demande-t-on à notre moteur de recherche favori de retrouver le nom de cette fameuse actrice ? Cependant, les techniques utilisées jusque récemment n'utilisaient pas le phénomène de reconnaissance. En pratique, les images étaient référencées, associées à des mots-clés, et chaque recherche correspondant à un mot-clé donnait pour résultats les images ayant été le plus souvent choisies par d'autres utilisateurs après une requête similaire.

Mais les machines évoluent, et les algorithmes aussi : il existe à présent des techniques capables de discerner si le mot-clé correspond bien à l'image, même s'il s'agit de sa première apparition sur le Web. Et les résultats progressent d'autant plus vite que les applications se multiplient : prévision de flux financiers, sélection de contenu personnalisé sur les réseaux sociaux, détection de cellules cancéreuses...

1.2 MNIST : Caractères numériques manuscrits

Si des applications réelles aussi poussées que celles mentionnées ci-dessus existent, un problème simple, inhérent aux besoins d'automatisation des procédés administratifs, reste à résoudre : dans le traitement de formulaires, chèques, déclarations ou autres examens, comment permettre à la machine de reconnaître automatiquement les caractères manuscrits, extrêmement variables de par leur nature "personnelle" ?

Pour situer le lecteur dans un contexte d'application réel, nous allons nous poser le problème de classification décrit par les points suivants :

1. Nous sommes une banque.
2. Nous souhaitons traiter automatiquement les chèques déposés par nos clients.
3. Nous disposons d'un système d'acquisition des numéros inscrits sur ces chèques.
4. Nous souhaitons obtenir un système qui permettent de nous éviter au maximum les erreurs d'interprétation de ces numéros.

Ce système en question est un "programme informatique" ou algorithme qui, après avoir été exposé à de nombreux exemples de numéros au préalable identifiés par des experts, sera capable de reconnaître les chiffres d'un nouveau chèque.

Un professeur et chercheur renommé dans le domaine de l'apprentissage machine, Yann LeCun, a travaillé sur l'élaboration d'une base de caractères numériques manuscrits : **MNIST**[1]. Celle-ci est conçue pour les travaux de *machine learning* spécifiquement. Ainsi, elle est séparée en un ensemble d'entraînement comprenant 60 000 images, et d'un ensemble de test en comprenant 10 000 : cette séparation réside dans l'idée que l'on expose la machine au "passé" (les 60 000 images d'entraînement) que l'on lui explique (chaque image lui est décrite par sa "valeur" réelle), puis que l'on confronte cette machine à un "futur" inconnu (les 10 000 images de test) qu'elle doit reconnaître (c'est-à-dire prédire le caractère correspondant à l'image).

Ce processus est spécifique à ce que l'on appelle l'*apprentissage supervisé* (voir Chapitre 2 pour plus d'explications). Dans l'objectif de ne pas surcharger inutilement la démarche de recherche en *machine learning* sur cette base de données, MNIST est déjà *pré-traitée* (voir Chapitre 3 pour plus d'explications) : chaque image, de taille 28×28, en 8 niveaux de gris, est centrée et obtenue à partir d'une image en binaire (noir et blanc) de taille 20×20 grâce à un filtre d'*anti-aliasing*[2].

MNIST constitue donc une excellente base de données pour expérimenter des méthodes de reconnaissances d'images et des techniques d'apprentissage machine sur des données réelles, sans s'encombrer des problèmes de pré-traitement et formatage habituels. L'objectif est alors d'obtenir des estimateurs nous permettant de classifier *au mieux* (voir Chapitre 5) ces données.

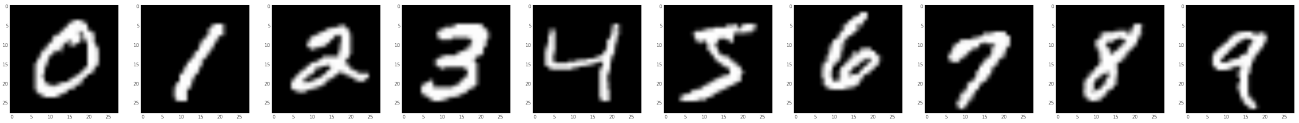

FIGURE 1 – Quelques chiffres de MNIST

Quelques statistiques descriptives sur la base MNIST

Afin de se donner une idée générale de la composition de MNIST et des sous-ensembles choisis pour l'entraînement et les tests, nous avons étudié la répartition des différentes classes. On constate (cf. l'histogramme en Annexes) que la composition générale (toutes proportions gardées) de la base de test est très fortement similaire à celle de la base d'entraînement, ce qui est statistiquement cohérent étant donné que la moyenne empirique converge rapidement avec le nombre d'observations : la fréquence d'apparition des dix classes est donc sensiblement la même sur 10 000 observations ou 60 000. Nous notons toutefois que les classes ne sont pas totalement équiréparties, nous n'obtenons pas une fréquence de 10% pour chaque chiffre.

Ensuite, nous avons voulu représenter graphiquement ce qu'on pourrait appeler les "images moyennes" de chaque chiffre, obtenues en assignant à chaque pixel la moyenne empirique des pixels correspondants de toute la classe. Nous obtenons ci-dessous des images assez représentatives, témoignant du bon pré-traitement de la base MNIST :

1. *Mixed National Institute of Standards and Technology database*
2. voir `http://fr.wikipedia.org/wiki/Anticrénelage`

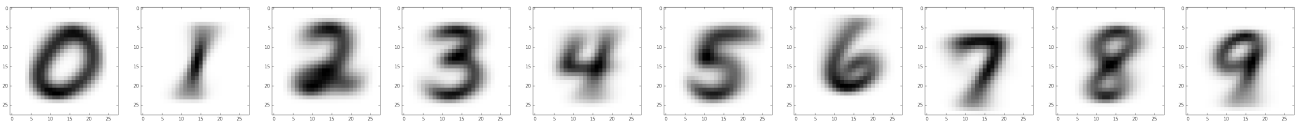

FIGURE 2 – Images moyennes

Nous avons aussi fait de même pour la variance, les points les plus clairs représentant ainsi les points les plus invariants de chaque classe. Un prétraitement optimal minimiserait la variance intraclasse et réduirait ainsi le plus possible les zones d'ombre observées sur les images ci-dessous :

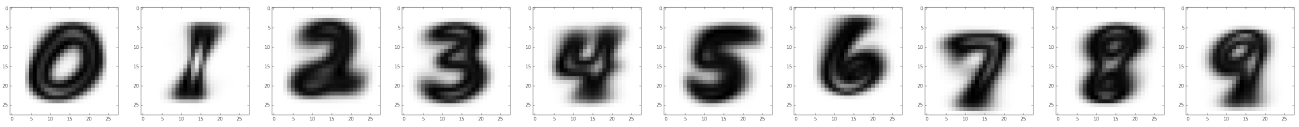

FIGURE 3 – Variances par chiffre

1.3 Python et `scikit-learn`

Pour mener ce projet, nous nous servirons du langage Python^{TM}. Ce langage est libre et est l'un des plus utilisé pour faire des statistiques et du *machine learning*. Il semblerait que ce langage soit en train de détrôner R dans ce domaine grâce à ces nombreux modules. Il présente aussi l'avantage d'être très lisible et facile à utiliser.

En particulier, nous nous servirons du module *scikit-learn*[3] qui possède des implémentations d'un grand nombre d'algorithmes dédié à l'apprentissage automatique. L'avantage principal de ce module est sa rapidité comparé à celle de Python seul[4], en effet, il est optimisé pour un certain nombre de calcul ce qui donne des temps d'entraînement acceptables pour la plupart des algorithmes. Enfin, utiliser ce module nous permet de comparer nos performances avec celle de la littérature car les algorithmes sont bien les mêmes.

2 Principes de base de l'apprentissage automatique

Introduction

L'apprentissage automatique (ou *machine learning*) est le champ d'études visant à étudier et implémenter des méthodes automatisables (arithmétiques ou heuristiques) permettant à une machine de réaliser certaines tâches complexes. Très souvent ces tâches consistent à deviner la meilleure réaction possible à une situation, la structure d'un certain jeu de données ou la valeur prise par une certaine variable d'intérêt pour un jeu de données. L'usage de l'apprentissage automatique s'est peu à peu démocratisé au cours des années 1990 et 2000 en même temps que la puissance informatique.

3. Appelé aussi `sklearn`, module faisant partie du projet **SciPy**
4. Scikit-learn est écrit en Cython, voir `http://cython.org/`

2.1 Les différentes formes d'apprentissage automatique

Il existe un grand nombre de formes d'apprentissage machine. Nous pouvons par exemple classer les différentes approches en fonction de la possibilité ou non d'entraîner notre algorithme à effectuer sa tâche sur un jeu de données et le cas échéant, des informations à notre disposition lors de la phase d'apprentissage. Ce qui nous donne la typologie suivante :

1. L'apprentissage supervisé : c'est le cas le plus commun et celui qui nous intéresse majoritairement. Ici, on dispose d'un ensemble de données d'entraînement, pour lesquelles nous disposons de la valeur de la variable d'intérêt (qu'il s'agisse de l'appartenance à une classe ou d'une variable continue). On se sert alors de l'information à disposition sur les données de l'ensemble d'entraînement pour mieux prévoir les valeurs prises par la variable d'intérêt sur un ou plusieurs autres jeux de données, appelés "ensembles de test".
2. L'apprentissage non supervisé (ou *clustering* en anglais) : dans ce cas, nous ne disposons d'aucun ensemble d'apprentissage, seulement de données à étudier pour lesquelles nous ne disposons pas d'information sur la valeur prise par la variable d'intérêt. L'apprentissage non-supervisé consiste alors à partitionner les données en plusieurs groupements (*clusters* en anglais) à choisir selon des critères de similarités, eux-mêmes à déterminer.
3. L'apprentissage partiellement supervisé : très proche de l'apprentissage supervisé à ceci près que l'information à disposition sur les données de l'ensemble d'entraînement ne sont que partielles. Plus précisément, on ne connaît pas nécessairement la valeur prise par la variable d'intérêt pour chacune des données mais on connaît au moins un sous-ensemble de valeurs possibles pour chaque donnée, potentiellement plus petit que l'ensemble de définition de la variable d'intérêt.
4. Apprentissage semi-supervisé : encore une fois très proche de l'apprentissage supervisé, mais on ne dispose ici de l'information prise par la variable d'intérêt que pour un nombre restreint (potentiellement petit) d'observations de l'ensemble d'entraînement. On peut cependant se servir de la structure de l'ensemble d'entraînement pour prédire les valeurs prises par la variable d'intérêt sur notre ensemble de test.

Il existe encore beaucoup d'autres formes d'apprentissages sur lesquelles nous ne nous sommes pas attardés telles que l'apprentissage par renforcement ou l'apprentissage par transfert. Le fait est que nous avons ici un aperçu suffisant de la diversité des formes d'apprentissage automatique.

Dans notre cas, il s'agit d'entraîner un algorithme à détecter des chiffres écrits à la main sur un ensemble d'entraînement de 60 000 images afin d'obtenir les meilleures prévisions possibles sur un autre ensemble de 10 000 images. On entre donc dans le cadre de l'apprentissage supervisé.

2.2 Les différentes formes d'apprentissage supervisé

On a vu que l'apprentissage supervisé consiste à prédire les valeurs prises par une variable d'intérêt sur un ensemble d'observations (dit ensemble test) après avoir entraîné notre algorithme sur les observations d'un ensemble d'apprentissage pour lesquelles nous connaissons les valeurs prises par la variable d'intérêt.

Il s'agit maintenant de savoir quel type de variable d'intérêt on cherche à prédire, ce qui déterminera en partie les algorithmes à notre disposition :

1. Si elle ne peut prendre qu'un nombre fini de valeurs (que l'on appelle étiquette ou *label*), on parle de **classification** (binaire ou multiple selon qu'il y ait deux labels possible ou plus).

2. Si la variable d'intérêt est réelle à support non fini, on parle de **régression**.
3. Si la variable d'intérêt n'est pas n scalaire, on parle de **prédiction structurée**.

Nous nous confrontons à de la classification multiple, puisqu'il s'agit ici de prédire des labels pouvant prendre les valeurs entières entre 0 et 9.

La différence entre classification binaire et classification multiple est importante puisqu'elle va influencer les indicateurs qui sont à notre disposition. La plupart des indicateurs de prédiction reposent sur les notions de vrai/faux positif/négatif et sont faits pour la classification binaire. Dans notre cas on doit adapter l'usage de ces indicateurs en appliquant certaines stratégies. Nous y reviendrons au cours du chapitre 5.

2.3 L'apprentissage supervisé vu par la théorie statistique de l'apprentissage

On note X l'ensemble des observations possibles et Y l'ensemble des étiquettes des classes. La théorie statistique de l'apprentissage suppose l'existence d'une loi de probabilité inconnue p sur l'espace produit $Z = (X, Y)$. On peut alors voir une observation dont on connaît la classe comme un doublet (x, y) de cet espace produit, où y est l'étiquette correspondant à l'observation $x = (x_i^1, ..., x_i^p)$ (où les x_i^j sont les caractéristiques ou *features* de l'observation x_i), et l'ensemble d'entraînement comme un échantillon $S = (x_i, y_i)_{i=1...n} \in Z^n$, de taille n tiré selon la loi de probabilité $p^{\otimes n}$.

L'objectif de l'apprentissage supervisé est alors de trouver une fonction $f : X \to Y$ parmi l'ensemble $\mathcal{F}$ des fonctions accessibles à l'algorithme choisi, telle que pour toute observation $(x, y) \in Z$, on ait $f(x) = y$.

Cet objectif étant le plus souvent impossible à atteindre, on choisit un critère de perte $L : Y \times Y \to \mathbb{R}^+$ (typiquement le critère d'exactitude $(y_1, y_2) \mapsto 1 - \mathbb{1}_{y_1=y_2}$, qui renvoie 0 si l'étiquette prédite est la bonne et 1 sinon) et on cherche une fonction f qui minimise la perte moyenne associé à ce critère $I[f] = \int_{X \otimes Y} L(f(x), y)p(x, y)dxdy$ dans l'espace des fonctions accessibles.

Étant donné que nous ne connaissons pas la probabilité p mais seulement les fréquences d'apparition des éléments dans l'échantillon d'entraînement, on approche cette erreur moyenne par l'erreur d'apprentissage sur l'échantillon de test $I_m[f] = \frac{1}{m} \sum_{i=1}^{m} L(f(x_i), y_i)$. Ce faisant, on fait l'hypothèse que les données à analyser ont été tirées selon la même loi de probabilité p que les données de l'ensemble d'entraînement.

2.4 Description succincte du fonctionnement d'un algorithme d'apprentissage automatique

Chaque classificateur correspond donc à une fonction f possible de l'espace $\mathcal{F}$, cette fonction en étant la représentation formelle. On va maintenant préciser le fonctionnement global habituel des algorithmes de *machine learning*. En parallèle, nous nous intéresserons d'un peu plus près à la manière dont ces fonctions f sont généralement construites, mais également comment cette construction peut être réalisée sur *Python* et *Scikit-Learn*.

Pour construire notre classificateur, nous allons nous servir de familles d'algorithmes assez classiques (que nous détaillerons dans la partie III) telles que les machines à vecteurs de support (SVM), les arbres

de décision, les méthodes de boosting (AdaBoost) et d'autres. Cependant nous allons parfois ne pas appliquer ces algorithmes directement sur nos données brutes $S = (x_i, y_i)_{i=1...n}$. Une étape préalable consiste à sélectionner et transformer ces données pour les présenter d'une manière plus adaptée à l'algorithme.

Il existe trois grands types de transformations de caractéristiques que sont la sélection, l'extraction et la construction. Nous reviendrons plus en détail et en exemple sur ces familles de transformations au Chapitre 3. Tout ce qu'il faut retenir pour l'instant est que ces transformations peuvent se représenter sous la forme d'une fonction h qui associe à une donnée brute x la donnée transformée $h(x)$. Ensuite seulement, on applique notre algorithme de classification (SVM ou autre) que l'on peut également représenter par une fonction g.

On a donc $f = g \circ h$, où h est une transformation de nos données brutes et g est un algorithme de classification à proprement parler.

Les fonctions g et h dépendent le plus souvent de paramètres, dont il existe deux types. Les premiers sont fixés par l'utilisateur. On les appelle les *hyperparamètres*, et on reviendra dessus dans le chapitre 4 sur les différentes manières de les calibrer. Les seconds sont spécifiques aux algorithmes d'apprentissage et sont obtenus au cours de la phase dite d'apprentissage. Le but de cette phase est d'obtenir un jeu de paramètres permettant de minimiser l'erreur effectuée sur l'ensemble d'apprentissage.

C'est en cela que la fonction g de notre algorithme et donc la fonction f de notre classificateur dépendent de nos données d'entraînement S.

Pour résumer, on a $f = g_S^\lambda \circ h^\mu$, avec h une transformation initiale de nos données (d'hyperparamètre μ) et g un algorithme de classification (d'hyperparamètre λ et entraîné grâce aux données S). On repère bien le processus de construction d'un classificateur : on choisit d'abord quelles transformations on applique à nos données, puis on se sert des données transformées comme entrées d'un algorithme de classification qu'on entraîne sur nos données d'apprentissage. Enfin on cherche la combinaison d'hyperparamètres qui nous fournit les meilleurs résultats.

2.5 Construction d'un classificateur dans `scikit-learn`

Dans `scikit-learn`, la construction de la fonction f de notre classificateur se fait à l'aide des fonctions **FeatureUnion** et **Pipeline**, et l'entraînement de notre algorithme grâce à la méthode **fit** de ce dernier.

Nous avons vu précédemment que nous pouvions transformer nos données brutes à l'aide de différentes fonctions h_1 ou h_2. La fonction *FeatureUnion* permet de concaténer différentes transformations $(h_i)_{i=1...k}$ pour obtenir une seule transformation h de forme $h(x) = (h_1(x), ...h_k(x))$. Ainsi, on peut appliquer plusieurs transformations en parallèle les unes des autres.

La fonction *Pipeline* permet quant à elle de composer un certain nombre de transformations $(h_i)_{i=1...k}$ et un estimateur g afin d'obtenir une fonction f telle que $f(x) = (g \circ h_1 \circ ... \circ h_k)(x)$. Elle permet donc d'appliquer plusieurs transformations successivement.

En combinant les fonctions *FeatureUnion* et *Pipeline*, on peut obtenir des classificateurs plus ou moins complexes. Il suffit alors d'entraîner notre classificateur sur nos données grâce à la méthode *fit*. Le classificateur n'a dès lors plus qu'à être testé sur les données de l'ensemble de test à l'aide la méthode **predict**.

Deuxième partie
Le détail des étapes du machine learning

3 La transformation de *features*

La transformation des données a permis aux chercheurs et techniciens d'observer certaines améliorations, en termes d'optimisation de taille (les données peuvent prendre beaucoup d'espace de stockage), de qualité (se concentrer sur les données utiles) et de performance (meilleures données impliquent un entraînement plus efficace) [5]. On trouve trois grands types de méthodes de transformation de données : le pré-traitement, les méthodes d'extraction/construction de caractéristiques, et les méthodes de sélection de caractéristiques [6].

3.1 Le pré-traitement des données

Dans un problème réel d'apprentissage, les données dont on dispose sont en pratique inutilisables en tant que telles : elles sont bruitées, incomplètes, inégales... Cela peut provenir de leur existence physique réelle (ici, des chiffres inscrits sur des chèques par exemple ne sauraient être parfaitement centrés dans leurs cases) ou de leur méthode d'acquisition (photos, scans peuvent êtres influencés par des variations de luminosité, inclinaison, support abîmé...).

Pour uniformiser l'allure des données et ainsi augmenter les similitudes intra-classe, on peut procéder à de nombreuses transformations manuelles. Dans le cas des images, on peut trouver par exemple des algorithmes de :

— Lissage et débruitage
— Redressement (en anglais *deskewing*)
— Centrage

La base *MNIST* étant conçue dans l'optique de simplifier cette étape de pré-traitement, nous n'avons pas eu besoin de nous pencher sur cet aspect pourtant inévitable dans les cas professionnels d'application du *machine learning*.

3.2 Extraction ou construction de *features*

Une image est une matrice $n \times n$ de pixels (chaque pixel prenant une valeur $x_i^j \in [0, 1]$ pour mesurer le niveau de gris, ou alors trois valeurs dans le cas des couleurs en canaux RGB). Dans notre cas d'étude donc, chaque image possède $n^2 (= 784)$ caractéristiques (ou *features*).

Mais ces caractéristiques peuvent être "insuffisantes", parce qu'elles ne sont pas assez significatives en elles-mêmes des spécificités des images, ou parce qu'elles sont trop peu nombreuses par exemple. Sans décrire toutes les motivations possibles d'adopter ce type de méthodes, l'extraction (ou construction) de *features* consiste en la création de nouvelles caractéristiques à partir de l'image originale, pour en réduire la dimension (dans le cas de l'extraction) ou l'augmenter (dans le cas de la construction).

5. Comme le remarquent Hiroshi Motoda et Huan Liu [ML02].

6. Ces méthodes, bien qu'intéressantes et permettant de ne se concentrer que sur les caractéristiques utiles, n'ont pas été traitées dans notre projet. Les avantages que nous aurions pu en retirer auraient été une baisse du surapprentissage, une augmentation de la vitesse d'apprentissage ou encore un gain en compréhensibilité des résultats

3.2.1 Exemples d'extraction de caractéristiques

Pour le modéliser simplement, le concept d'extraction est le suivant : on dispose de $A_1, ..., A_n$ caractéristiques initialement, et on cherche un *mapping* F tel que $F(A_1, ..., A_n) = B_1, ..., B_m$ (avec $m < n$). Les objectifs sont alors de minimiser le nombre de caractéristiques m, maximiser le critère de performance choisi (cf. Chapitre 5), ou encore simplifier/améliorer/diversifier le calcul du *mapping* F.

En guise d'illustration, nous avons implémenté quelques méthodes d'extraction[CCR10] de caractéristiques :

— **Structural Characteristics**[Kav+03] (cf. Fig. **XX**) Comme son nom l'indique, l'idée de cette méthode est d'évaluer les caractéristiques structurelles de l'image. On évaluera ainsi :
  1. le nombre de pixels noirs par ligne *(28 valeurs)*
  2. le nombre de pixels noirs par colonne *(28 valeurs)*
  3. le nombre de pixels noirs par "rayons" avec un pas de 5 deg *(72 valeurs)*
  4. le profil *In-Out*, c'est-à-dire les premiers pixels noirs venant du centre sur chaque "rayon" *(72 valeurs)*
  5. le profil *Out-In*, les derniers pixels sur chaque "rayon" *(72 valeurs)*

— **Multi-Zoning** (cf. Fig. **XX**) Cette fois, on procède à de multiples "découpages" de l'image ($2 \times 1, 7 \times 4$...), et sur chaque sous-image, on calcule le pourcentage de pixels noirs.

— **Modified Edge Maps** (cf. Fig. **XX**) Cette méthode emploie les filtres de Sobel [7] pour obtenir des cartes de contours selon plusieurs directions, qui sont ensuite subdivisées et simplifiées comme dans la méthode de *Multi-Zoning* (cf. ci-dessus).

3.2.2 Exemples de construction de caractéristiques

Historiquement, construction et extraction ont été souvent confondues. L'on s'accorde cependant pour attribuer aux méthodes de constructions l'objectif d'atteindre un nombre m de caractéristiques plus élevé qu'initialement (on crée donc artificiellement des données). Si les bases d'images ne sont pas toujours concernées par un tel besoin d'augmentation de dimension de part leur taille, il est possible que l'on cherche à drastiquement augmenter la dimension du problème pour simplifier la séparation des points.

Une de ces méthodes, réputée dans la recherche et détection de contours dans les images, est l'**Histogramme de Gradient Orienté** (HOG) [8], que nous avons utilisé [9] dans notre projet (cf. **partie Résultats**). Un HOG s'obtient en appliquant une série de trois étapes :

1. On calcule le gradient de l'image via un filtre dérivatif (dans le même esprit que les filtres de Sobel)
2. Dans des cellules (sous-images) de différentes tailles (4×4, 7×7 ou 14×14), on calcule l'histogramme des différentes orientations (de 0 à 360°) du gradient
3. En regroupant les cellules par blocs carrés de 4, on applique une normalisation L_2 au vecteur des histogrammes

7. voir `http://fr.wikipedia.org/wiki/Filtre_de_Sobel`
8. Cette méthode est proche de celle du Shape Context présenté ici : [BMP02]
9. grâce à la fonction `skimage.feature.hog`. Voir `http://scikit-image.org/`

4 Les méthodes d'estimation

4.1 Les paramètres

La plupart des algorithmes que nous avons utilisés nécessitent le calibrage de certains paramètres, souvent appelés "hyperparamètres" afin de les distinguer des paramètres calculés par les algorithmes au cours du processus d'apprentissage. Ces paramètres sont, pour la plupart des nombres réels, des réels positifs ou des entiers naturels.

Pour un algorithme et une transformation de *features* donnés, à chaque classificateur correspond un jeu de paramètres. On définit alors l'espace des paramètres comme étant l'ensemble des jeux de paramètres associés à un classificateur valide. L'espace des paramètres prend le plus souvent la forme d'un produit cartésien d'espace plus petits correspondant chacun à un paramètre.

Exemples de paramètres :
— Degré du noyau polynômial dans un SVM
— Paramètre de régularisation dans une régression logistique
— Profondeur maximale d'un arbre de décision

Le but est alors de trouver, étant donné notre ensemble d'apprentissage, le meilleur jeu de paramètres possible, c'est-à-dire celui qui permet de classifier les données de la manière la plus adéquate. Bien entendu, il n'est pas possible de tester tous les paramètres possibles puisqu'il en existe la plupart du temps une infinité. C'est pourquoi on borne l'espace des paramètres à des valeurs qui nous semblent raisonnables et on discrétise celui-ci quand certains paramètres sont réels.

La capacité d'un algorithme à classifier convenablement les données est mesurée à l'aide d'une fonction de score que l'on cherche à maximiser. Ces fonctions de score sont spécifiques au problème de classification et différentes de celles employées dans un cas de régression supervisé par exemple.

Le choix de la fonction de score retenu pour calibrer les algorithmes n'est pas anodin : il traduit les préférences de l'utilisateur au niveau du type d'erreur commise lors de la prédiction. Nous reviendrons plus en détail sur les fonctions de score usuelles ci-après.

4.2 Éviter le surapprentissage

Il pourrait être tentant de choisir comme meilleur classificateur celui qui obtient un score maximum sur notre ensemble d'apprentissage. Cependant, cela serait se tromper d'objectif : un classificateur est censé faire de la prédiction, c'est-à-dire classer convenablement des images dont il ne connaît pas la véritable étiquette. De plus, ce n'est pas parce qu'un classificateur a un très bon score sur les données de notre ensemble d'apprentissage qu'il sera apte à prédire des images extérieures à cet ensemble.

On appelle **surapprentissage** (ou *overfitting* en anglais) le phénomène par lequel un estimateur obtient de très bons résultats sur les données sur lesquelles il a été entraîné, mais n'arrive pas à prédire des cas plus ou moins éloignés de celles-ci.

Pour limiter le surapprentissage, on peut utiliser principalement deux méthodes. La première s'appelle la "régularisation" et peut être utilisée lorsque la recherche des paramètres internes d'un classificateur (à ne pas confondre avec les "hyperparamètres"). Elle consiste en un programme de minimisation d'une fonction de perte évaluée en l'ensemble d'apprentissage, c'est-à-dire en la recherche d'un compromis entre la

capacité du classificateur à prédire les données et la « taille » du vecteur des paramètres. Nous reviendrons sur la régularisation plus en détail.

L'autre solution pour éviter le surapprentissage consiste à séparer notre ensemble d'apprentissage en deux ensembles, l'ensemble d'apprentissage à proprement parler, sur lequel on construit notre classificateur, et l'ensemble de validation, grâce auquel on peut estimer la capacité de prédiction de notre classificateur en calculant nos fonctions de score. L'inconvénient de cette méthode est qu'elle nécessite de se priver d'une fraction non négligeable de nos données d'apprentissage qui vont constituer notre ensemble de vérification et qu'on aurait bien aimé pouvoir utiliser pour entraîner notre classificateur.

4.3 La validation croisée

Une parade à ce désagrément est la **validation croisée** (ou *cross validation* en anglais) et se réalise comme suit : on sépare notre ensemble d'apprentissage en un certain nombre K de sous-ensembles. Pour chaque sous-ensemble K_i, on entraîne notre classificateur sur l'ensemble total privé de notre sous-ensemble et on teste les capacités de prédiction de notre classificateur. On calcule alors le score de prédiction moyen sur les K parties et on considère comme étant optimal le jeu d'hyperparamètres qui obtient le meilleur score. On se sert alors de ce jeu d'hyperparamètres pour entraîner à nouveau notre classificateur en utilisant tout notre ensemble d'entraînement cette fois. On peut également retirer une seule observation de notre ensemble d'apprentissage une seule observation sur laquelle on effectue notre prédiction et prendre à nouveau le score moyen, ce qui revient à choisir K égal au nombre de données dans l'ensemble d'apprentissage.

Il existe plusieurs manières de découper notre ensemble d'apprentissage : on peut par exemple choisir entre tirer aléatoirement les données de chaque sous-ensemble de manière uniforme ou encore s'assurer que les proportions de chaque classe dans l'ensemble total se retrouvent dans chacun des sous-ensemble.

La validation croisée permet ainsi d'utiliser l'intégralité de notre ensemble d'apprentissage pour entraîner notre classificateur sans pour autant trop "coller" à nos données. Sur *sklearn*, la fonction qui permet de trouver le jeu de paramètres optimal dans un espace de paramètres défini est un "GridSearch" (**grid_search.GridSearchCV**).

La validation croisée nécessite cependant d'entraîner notre classificateur un très grand nombre de fois, plus précisément une fois et par jeu d'hyperparamètres et par sous-ensemble (c'est d'ailleurs pour cela que le nombre de sous-ensembles doit être judicieusement choisi), en plus de l'apprentissage du classificateur une fois les hyperparamètres fixés. Cela peut entraîner une lenteur de calibrage trop importante pour pouvoir être effectuée en pratique sur de grands espaces de paramètres.

Une solution permettant de contrôler et diminuer le temps d'entraînement est d'utiliser une fonction qui ne teste qu'un nombre fixé de configurations tirées aléatoirement dans l'espace des paramètres [10]. Ainsi, si l'évolution des fonctions de score dans l'espace des paramètres est suffisamment régulier, on peut, à défaut d'obtenir le jeu de paramètres optimal, obtenir un jeu de paramètres convenable en un temps prévisible et relativement court.

À noter qu'il peut arriver que l'espace des paramètres soit tellement grand (dans le cas d'algorithmes

10. grid_search.RandomizedSearchCV

complexes par exemple) que même en tirant aléatoirement des jeux de paramètres, on ne peut espérer recouvrir l'espace convenablement sans un grand nombre d'essais, ce qui entraîne un temps de computation trop important. On se contente alors de couper l'ensemble d'apprentissage en un ensemble d'entraînement de 50000 données et un ensemble de validation de 10000 données.

À noter également que certains algorithmes ne nécessitent pas l'utilisation de validation croisée pour la calibration de leurs hyperparamètres. On pensera ainsi aux méthodes de *bagging* sur lesquelles nous reviendrons, pour lesquelles le score *Out-Of-Bag* fait office de score de prédiction.

4.4 La courbe de validation

La courbe de validation affiche les scores obtenus sur l'ensemble d'apprentissage et ceux obtenus par validation croisée pour différentes valeurs prises par un hyperparamètre, les autres hyperparamètres étant constants. Elle permet de sélectionner la valeur de l'hyperparamètre qui limite le plus le surapprentissage.

L'idée derrière la courbe de validation est qu'en trouvant successivement les meilleures valeurs paramètre par paramètre, on peut espérer approcher le maximum global de notre score (quitte à reboucler plusieurs fois sur nos paramètres si nécessaire). On retrouve là le principe de fonctionnement des algorithmes gloutons. Cette méthode de calibrage des paramètres a l'avantage d'être bien plus rapide que le GridSearch pour un espace de paramètre assez grand. Elle peut parfois aboutir à un maximum local non global, mais fournit des résultats plus que convenables le plus souvent.

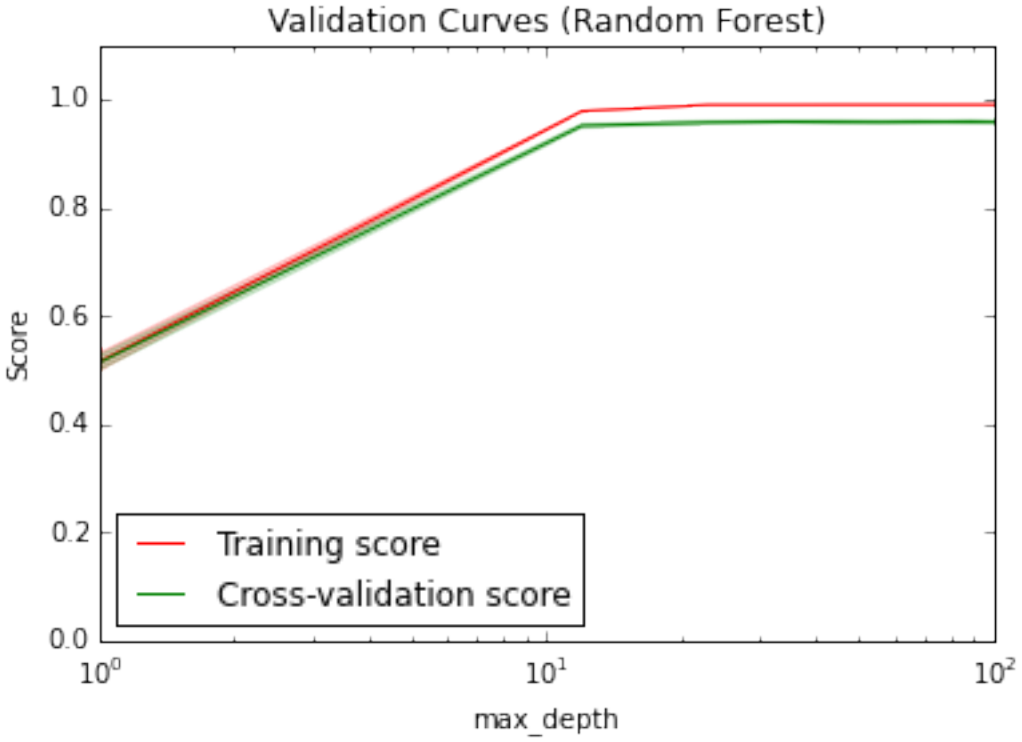

FIGURE 4 – Un exemple de courbe de validation

5 Les outils d'analyse

Pour comparer les performances de divers algorithmes, il nous faut des métriques qui permettent de les hiérarchiser en fonction de plusieurs critères. Si le temps d'entraînement est le critère le plus évident, il n'est dans le cas de MNIST que peu pertinent car tous les algorithmes que nous avons utilisés avaient des temps d'entraînement relativement courts (moins d'une journée), on va donc comparer les performances des algorithmes.

5.1 Le score d'exactitude (ou *accuracy score*)

L'efficacité d'un algorithme se mesure avec un score. Ici, comme on va chercher à prédire l'appartenance à un label, le score que l'on va principalement utiliser est un *accuracy score* qui mesure le taux de vrais positifs, c'est-à-dire :

$$Score_{accuracy} = \frac{\text{nombre d'observations correctement prédites}}{\text{nombre d'observations total}}$$

On peut donc mesurer un accuracy score à l'échelle d'un seul label (la bonne prédiction des 5 par exemple) ou sur l'ensemble des labels.

Ce score comporte un risque important, en effet s'il existe des classes dégénérées que l'on prédit très mal, on peut quand même avoir un bon accuracy score. Par exemple dans une base de donnée où l'on veut prédire l'appartenance à deux classes, si l'une des deux classes ne comporte que 5% des individus, alors un score d'accuracy de 95% peut signifier qu'on classe toute nos données dans la même classe et que l'on se trompe que sur les 5% de la petite classe ce qui est bien sûr très problématique. Néanmoins, nous nous sommes prémunis de ce risque en observant les matrices de confusion de nos algorithmes.

5.2 La matrice de confusion

La matrice de confusion est un matrice de taille $n \times n$ avec n le nombre de labels. En colonne on a les labels prédits, en ligne les vrais labels et dans chaque case on a l'effectif. Par exemple, dans la case (i, j) on trouve le nombre de i qu'on a classés comme des j. L'accuracy score peut se lire dans cette matrice. En effet le rapport entre la trace de cette matrice et la somme de tous les éléments est exactement l'accuracy score. On peut également y retrouver les accuracy score label par label en faisant, dans chaque colonne, le rapport entre le coefficient diagonal et la somme de la colonne.

Dans MNIST, comme les classes sont équilibrées, on peut mesurer la qualité de nos prédictions en prenant un code couleur sur les effectifs. Le même effectif dans deux cases diagonales signifie la même qualité de prédiction sur le label en question.

Enfin, la matrice de confusion a un atout supplémentaire qui vient du fait qu'elle est visuellement très interprétable et comme elle résume beaucoup d'information, elle permet de mesurer simplement en un coup d'oeil les performances de l'algorithme.

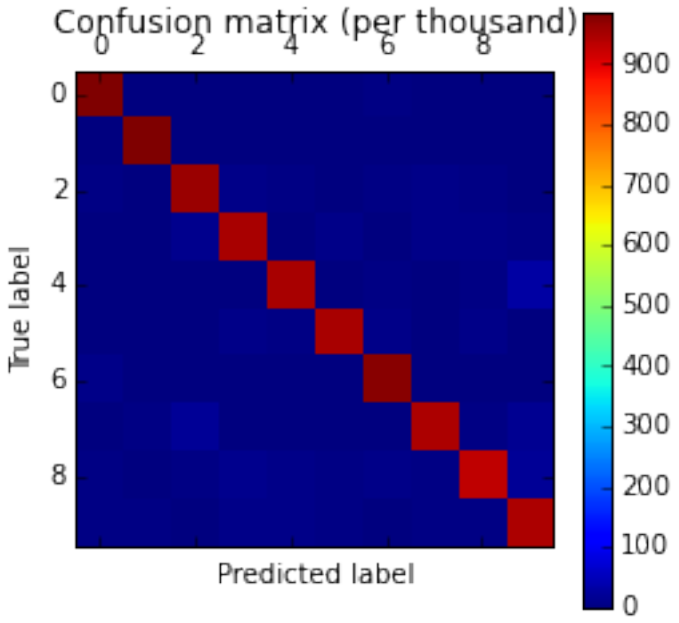

FIGURE 5 – Un exemple de matrice de confusion

5.3 Les courbes ROC

Les courbes ROC (Receiver Operating Characteristic) servent initialement à un classificateur binaire. Elles représentent le taux de "vrai positif" en fonction du taux de "faux négatif" et elles mesurent donc les performances du *classifier* en fonction de seuils de prédictions choisis.

Dans notre cas, on a dix classes ce qui signifie que l'outil n'est *a priori* pas adapté. Néanmoins, on construit une courbe de chaque label contre tous les autres ce qui nous donne donc 10 courbes ROC différentes que l'on peut comparer entre elles pour avoir une idée plus détaillée des performances de l'algorithme, mais que l'on peut aussi comparer entre différents algorithmes en comparant l'aire sous la courbe par exemple.

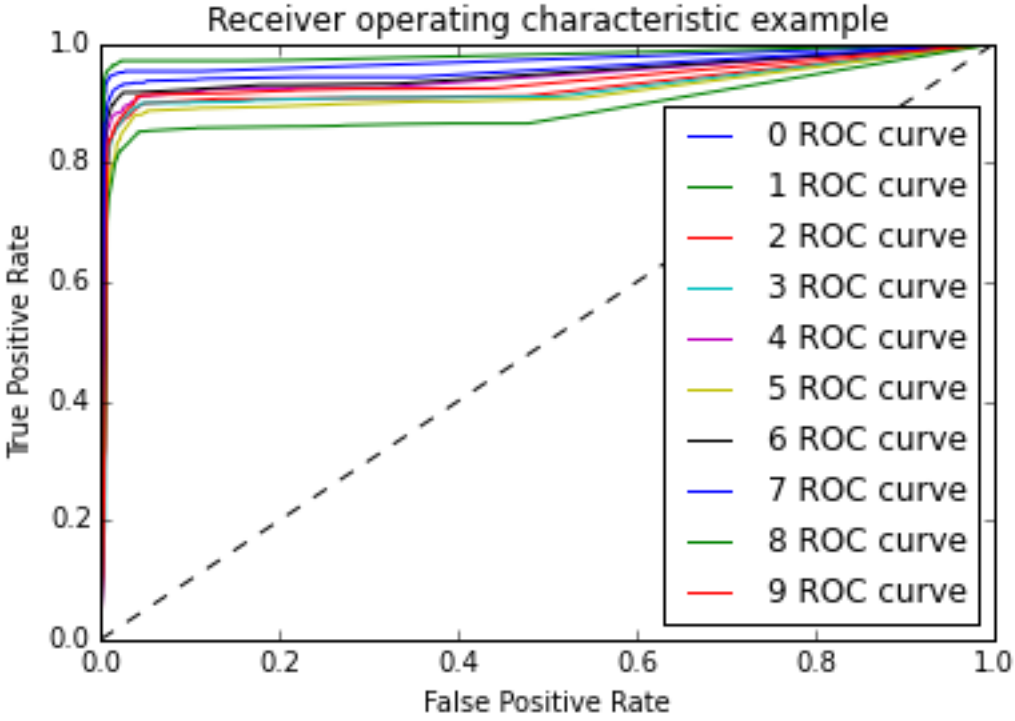

FIGURE 6 – Un exemple de courbe ROC

5.4 Les courbes d'apprentissage

Les courbes d'apprentissage sont deux courbes représentant les scores d'apprentissage et de validation croisée en fonction de la taille des sous-échantillons de l'ensemble d'entraînement. Elles servent à diagnostiquer les performances générales de l'algorithme. Si les courbes sont très éloignées l'une de l'autre, alors on peut augmenter les performances en ajoutant des observations (ce qui explique la différence décroissante entre les deux courbes). A l'inverse, si les deux courbes sont très proches et qu'elles convergent vers un score que l'on veut améliorer, alors il faut de nouvelles *features*.

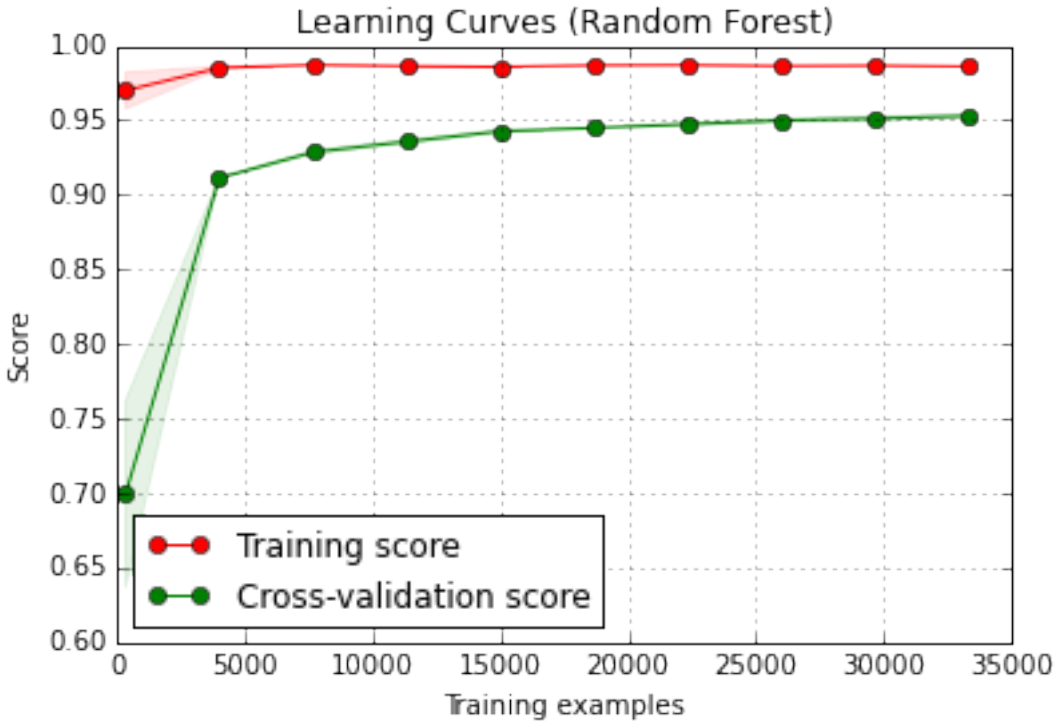

FIGURE 7 – Un exemple de courbe d'apprentissage

5.5 Comparer des algorithmes

Pour comparer des algorithmes entre eux, il y a deux familles de critères :

1. Les performances de prédiction : on compare donc à l'aide des outils vus précédemment
2. La complexité en temps et en espace : pour cela on mesure le temps d'entraînement, l'espace mémoire occupé, la durée d'entraînement de l'algorithme et le temps de classification de nouvelles *features*.

Troisième partie

Présentation des différents algorithmes utilisés et analyse des résultats

6 Les modèles linéaires

6.1 Régression logistique

La régression logistique est plus un classificateur linéaire qu'un modèle de régression à proprement parler contrairement à ce que son nom pourrait laisser entendre. Il s'agit du modèle économétrique généralisant le modèle logit binaire.

6.2 Une régularisation classique : la pénalisation L2

La pénalisation L2 est un type de régularisation qui cherche à contrôler la variance des paramètres internes de notre classificateur en pénalisant la norme euclidienne du vecteur des paramètres. Elle est une régularisation classique que l'on retrouve dans les modèles de régression Ridge. De ce fait, on limite la possibilité pour l'estimateur de coller à nos données et de subir le surapprentissage. Le choix du paramètre de régularisation C est lié à un arbitrage entre l'apprentissage de la structure des données et le risque de surapprentissage.

6.3 Perceptron

Le perceptron à simple couche n'est pas à proprement parler un modèle linéaire généralisé. C'est malgré tout un classificateur linéaire dans le sens où il sépare les données avec un hyperplan $h : wx + b = 0$, et se rapproche ainsi des machines à vecteurs de support que nous verrons dans le chapitre suivant. Le perceptron initialise le vecteur des poids w et le corrige à chaque exemple traité sans succès. Il peut parcourir plusieurs fois l'ensemble d'apprentissage en le réordonnant de manière aléatoire à chaque fois. Le perceptron est un classificateur qui s'entraîne très vite, c'est le réseau de neurones le plus simple possible.

7 Les machines à vecteurs de supports (Support Vector Machines, SVM)

Les SVM sont une classe d'algorithmes d'apprentissage, utilisée pour des problèmes de discrimination (comme c'est le cas ici avec MNIST) ou de régression. Elles ont été adoptées en raison de leur efficacité pratique et de leur capacité à traiter des données de grande dimensions avec de très bons résultats en pratique.
Après avoir détaillé le cas linéaire dans la sous-section 7.1, nous verrons dans la sous-section 7.2 comment généraliser cette approche linéaire grâce au **Kernel Trick** pour enfin présenter nos résultats obtenus sur la base MNIST dans la sous-section 7.3.

7.1 Présentation du cas linéairement séparable

Le principe des SVM est relativement simple. Le but est de parvenir à trouver un hyperplan de marge maximale qui sépare linéairement les données. On appelle marge la distance de l'hyperplan au point le plus proche. Nous nous plaçons tout d'abord dans le cas très particulier où les données sont effectivement linéairement séparables puis nous verrons dans la sous-section 8.2 comment généraliser. Formalisons un peu le problème. On veut construire $f : \chi \to \mathbb{R}$, avec χ l'ensemble des observations, telle que la classe sera donnée par le signe de f. Dans le modèle linéaire, $f(x) = w^T x + b$ et la frontière de décision correspond donc à $w^T + b = 0$. La distance d'un point à cet hyperplan est ainsi donnée par $d(x) = \frac{|w^T x+b|}{||w||_2}$. Maximiser la marge revient donc à minimiser $||w||_2$ sous contraintes. On introduit des paramètres de tolérance appelés variables ressort pour assouplir ces contraintes trop restrictives lorsqu'on traite des données réelles et le problème primal devient ainsi :

$$\begin{aligned}
&\underset{w,b,\zeta}{min} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \zeta_i \\
&\text{subject to} \\
&y_i(w^T x_i) + b) \geq 1 - \zeta_i, \\
&\zeta_i \geq 0, i = 1, ..., n
\end{aligned}$$

La recherche du meilleur paramètre C est un problème d'optimisation classique que l'on résout en annulant les dérivées partielles du Lagrangien. On obtient finalement :

$$h(x) = \sum_{i=1}^{n} \alpha_i^* y_i x^T x_i + w_0$$

Avec α_i^* les multiplicateurs de Lagrange optimaux. On trouve ainsi l'hyperplan de marge maximale qui sépare les données. La complexité de l'algorithme est quadratique en la taille des données.

Cependant, nous avons traité ici du cas simpliste où les données étaient linéairement séparables, ce qui est très rarement le cas avec des données réelles.

7.2 Le *Kernel Trick* et les différents types de noyaux

Les données sont généralement non linéairement séparables, ce qui rend la recherche d'un hyperplan séparateur impossible. On peut malgré tout se ramener au cas séparable en passant dans un espace intermédiaire (*feature space*) de plus grande dimension voire de dimension infinie en transformant les données avec une fonction ϕ bien choisie.

La solution du problème d'optimisation présenté en subsection 8.1 devient ainsi :

$$h(x) = \sum_{i=1}^{n} \alpha_i^* y_i \phi(x)^T \phi(x_i) + w_0$$

Le problème de cette transformation c'est qu'il faut maintenant calculer un produit scalaire $\phi(x)^T\phi(x_i)$ dans un espace de dimension élevée voire infinie, ce qui est extrêmement coûteux en terme de calculs, et c'est là qu'intervient le Kernel Trick.

On utilise une fonction noyau K qui vérifie $K(x_i, x_j) = \phi(x_i)^T\phi(x_j)$, ce qui présente de multiples avantages : premièrement, le calcul se fait dans l'espace d'origine des données et deuxièmement, nous n'avons plus besoin de connaître explicitement ϕ, h ne dépend plus que de la fonction noyau K.

Une des conséquences du théorème de Mercer en analyse fonctionnelle est que cette fonction noyau doit être symétrique et semi-définie positive pour correspondre à un produit scalaire dans un espace de grande dimension. Ainsi, au lieu de choisir une transformation ϕ, on choisit une fonction noyau classique vérifiant ces propriétés. Les noyaux les plus fréquemment utilisés sont les noyaux linéaire, gaussien ou polynômial [11].

Le Kernel Trick permet d'amener de la non-linéarité dans notre modèle en évitant de calculer les produits scalaires en grande dimension. Cependant les phases d'apprentissage et de prédiction des machines à vecteur de support nécessite de faire appel à un grand nombre de fois aux noyaux, ce qui augmente le temps de calcul comparativement aux modèles linéaires.

Les choix du noyau optimal et de ses paramètres (degré dans le cas polynômial, variance dans le cas gaussien) et du terme d'élasticité C sont des problématiques récurrentes dans le cadre des SVM, ils correspondent à des problèmes d'optimisation très pointus[Cha+02].

8 Les arbres de décision

Les arbres de décision sont un type d'algorithmes de base inspiré des arbres du même nom que l'on retrouve en analyse de la décision. Les arbres utilisés sont des arbres binaires. Les noeuds de l'arbre sont de trois types, à savoir la racine (qui n'a pas d'ancêtre), les noeuds internes (qui ont un ancêtre et deux descendants) et les feuilles (qui n'ont pas de descendants).

Chaque embranchement de l'arbre correspond à un critère de segmentation portant sur une des variables d'entrée, à savoir la position par rapport à un certain seuil pour les variables quantitatives ou l'appartenance ou non à un sous-ensemble de modalités pour les variables catégorielles.

À chaque noeud, on trouve un ensemble de données de l'ensemble d'entraînement qui respectent tous les critères de segmentation des embranchements en amont. Les données sont ainsi partitionnées de manière récursive selon les critères qu'elles respectent et ceux qu'elles ne respectent pas, les partitions finales correspondant aux feuilles de l'arbre.

L'intérêt de l'apprentissage est de trouver les critères et les seuils associés qui réalisent le meilleur partitionnement possible étape après étape, c'est-à-dire que les *clusters* créés après un embranchement sont ceux dont les observations sont les plus semblables entre elles.

Puisque nous travaillons avec une variable d'intérêt catégorielle et non continue, il nous faut nous tourner vers des critères d'homogénéité différents de la variance intra-classe. Les deux plus utilisés sont l'indice

11. Soit respectivement $\langle x, x' \rangle$, $exp(-\gamma|x - x'|^2)$ et $(\gamma\langle x, x' + r)^d\rangle$, où γ, r et d sont des paramètres.

de diversité de Gini (qui vaut $1 - \sum_{i=1}^{m} f_i^2$, où f_i est la proportion d'éléments appartenant à la classe i au sein de la feuille) et le gain d'information (lié à l'entropie de Shannon, et qui vaut $-\sum_{i=1}^{m} f_i log_2(f_i)$). À noter que ces deux critères sont minimaux lorsque toutes les observations associées à un noeud sont de la même classe. Dans ce cas, on arrête de segmenter et le noeud garde le statut de feuille. Il existe d'autres raisons pour lesquelles on peut arrêter la construction de l'arbre, sur lesquelles nous reviendrons.

Nous avons utilisé l'indice de diversité de Gini pour construire nos arbres. Quel que soit le critère choisi, on cherche à minimiser la valeur de sa moyenne calculée sur les feuilles. Cette méthode d'optimisation étape par étape est un algorithme glouton qui peut donc aboutir à un minimum local mais non global du critère dans certains cas.

Les arbres de décision présentent de nombreuses qualités. Ce sont des estimateurs assez simples qui se compilent très rapidement (c'est d'ailleurs une des raisons pour lesquelles ils peuvent servir de briques élémentaires pour des algorithmes plus complexes que nous verrons par la suite, tels que les méthodes de *bagging* ou de *boosting*). De plus, leur fonctionnement est suffisamment limpide pour qu'un humain puisse interpréter les choix de variables et de critères effectués par l'algorithme (d'où la qualification récurrente de "*white box*").

Du point de vue statistique, ils se caractérisent par un biais faible mais une grande variance. Ils sont de plus sujets à l'overfitting. Il existe cependant plusieurs parades à ce problème. La première (que nous n'avons pas expérimentée) s'appelle l'élagage (ou *pruning*) et consiste en la suppression *a posteriori* des branches les moins discriminantes. Une autre solution consiste à limiter l'expansion de l'arbre en autorisant une profondeur d'arbre maximale et/ou une taille des feuilles minimale. Dans le cas où une feuille n'est pas "pure" et que deux classes ou plus apparaissent dans la même feuille, on applique un vote majoritaire. Ainsi, on évite d'avoir des feuilles qui sont liées à un trop petit nombre d'observations.

Dans notre cas, nous avons utilisé un arbre de profondeur maximale 14 [12] et qui comporte au moins 5 observations par feuille.

9 Les méthodes d'ensemble : *bagging* et *boosting*

Introduction

Les méthodes d'ensemble forment une catégorie de classificateurs qui basent leurs prédictions sur celles d'un grand nombre d'autres classificateurs. Ces méthodes sont répartis en deux catégories : les méthodes d'agrégation et les méthodes de boosting.

Dans les cas des méthodes d'agrégation, on moyenne (par vote majoritaire par exemple) les prédictions faites par des classificateurs indépendants, tandis que dans le cas des méthodes de boosting, on combine des estimateurs construits successivement de sorte que les estimateurs construits en aval corrige les défauts de prédiction des estimateurs construits en amont.

Les estimateurs combinés par ses deux types de méthodes sont des estimateurs dits **faibles**, dont on exige qu'ils prévoient mieux que le hasard sur au moins deux classes. Nous avons utilisé une méthode

12. Ce qui le rend difficilement représentable puisqu'il peut avoir jusqu'à plusieurs milliers de feuilles

d'agrégation (les forêts d'arbres aléatoires) et une méthode de boosting (le boosting adaptatif, ou Adaboost).

9.1 Présentation du bagging : Random Forest

La Random forest ou forêt aléatoire est, comme son nom l'indique, un ensemble d'arbres de décision. Elle est basée sur un principe appelé *bagging* (pour *boostrap aggregating*). [13]

Le bagging consiste à tirer avec remise des observations parmi la base d'entraînement et d'entraîner des arbres de décision avec les sous échantillons ainsi tirés. Notre random forest est composée de 80 arbres et la prédiction est un vote majoritaire au sein de ces 80 arbres. En effet, chacun des arbres prédit un label et le label que prédit la random forest est celui qui est le plus souvent prédit par les 80 arbres.

L'avantage de cette méthode est que l'on pondère peu les observations rares et qui peuvent poser des problèmes de prédiction. Comme on tire avec remise, on va plus plus souvent tirer les groupes d'observations qui se ressemblent que les observations aberrantes. Quand bien même nous tirerions parfois ces observations problématiques, les arbres entraînés dessus sont noyés dans le vote majoritaire. D'autre part, nous avions relevé dans la partie sur les arbres de décision que ceux-ci avaient beaucoup de variance, en passant à la random forest qui combine 80 estimateurs, on opère ainsi une réduction de variance ce qui améliore les performances.

Notons que dans Scikit-Learn, ce n'est pas un vrai vote majoritaire entre les arbres mais une moyenne des probabilités que prédisent les arbres de décision ce qui permet de se prémunir de sous-pondérer des labels s'ils n'arrivent qu'en seconde position dans certains des sous-arbres par exemple.

9.2 Présentation du boosting : AdaBoost

AdaBoost (pour "boosting adaptatif") est un algorithme qui pondère de manière successive les prédictions de classificateurs faibles sélectionnés un par un pour leur capacité à corriger les erreurs de prédictions faits par les classificateurs sélectionnés en amont.

9.2.1 Principe

Lors de l'initialisation de l'algorithme, toutes les images de l'ensemble d'apprentissage sont pondérées de la même manière dans la fonction de score (exactitude). On dispose d'une liste de classificateurs faibles. AdaBoost sélectionne un premier classificateur en fonction de sa capacité à prédire les images de l'ensemble d'entraînement, c'est-à dire à obtenir le meilleur score. AdaBoost s'assure que le classificateur est faible, puis attribue un poids à cet algorithme en fonction du score obtenu, puis il renforce le poids dans la fonction de score des images mal prédites par l'algorithme sélectionné et diminue le poids des images convenablement prédites. On réévalue alors les algorithmes sur la fonction de score avec les nouveaux poids, ce qui permet de sélectionner un nouveau "meilleur classificateur faible" que l'on pondère et en fonction duquel on réévalue le poids de chaque image. On modifie également les poids du classificateur déjà sélectionné afin de normaliser la somme des poids. On itère un grand nombre de fois pour obtenir une liste de classificateurs sélectionnés et les poids qui leur sont attribués. La prédiction effectuée par AdaBoost est alors la moyenne des prédictions des classificateurs sélectionnés, pondérée par les poids qui leur ont été attribués.

13. Le principe et l'avenir de la random forest est présenté par Vrushali Y Kulkarni et Pradeep K Sinha dans l'article "*Random Forest Classifiers :A Survey and Future Research Directions*"[KS13]

9.2.2 Caractéristiques d'AdaBoost

AdaBoost est un algorithme puissant qui est connu pour ses bonnes prédictions. Il est très long à entraîner mais très rapide pour ses prédictions (puisqu'il ne calcule qu'une moyenne pondérée). Dans les faits nous avons utilisé une variante d'AdaBoost appelée Real AdaBoost qui s'entraîne légèrement plus rapidement.

Il est courant de se servir de perceptrons simple couche ou d'arbre de décisions de faible taille comme classificateurs faibles. Dans notre cas nous nous sommes concentrés sur deux types d'arbres de décision de profondeurs maximales respectives 2 et 14.

10 La méthodes des plus proches voisins

La méthode des plus proches voisins consiste à attribuer à une observation à classer l'étiquette ressortissant majoritairement parmi les éléments de l'ensemble d'apprentissage qui lui ressemblent le plus. Plus précisément, on récupère les étiquettes des K éléments de la base d'apprentissage les plus proches de l'observation au sens de la distance euclidienne, K étant un paramètre à fixer.

La méthode des plus proches voisins est un algorithme basé sur la mémoire (ou *instance-based learning* en anglais). En effet, la phase d'apprentissage consiste simplement à garder en mémoire les éléments de l'ensemble d'entraînement, le choix de la fonction de prédiction ne s'effectuant que lorsqu'on demande à prédire une observation (ce qu'on appelle de l'apprentissage fainéant, ou *lazy learning*, à l'opposé des algorithmes classiques dits *eager* qui calculent la méthode de prédiction en amont de l'accès aux données de test).

Il existe différents algorithmes permettant d'effectuer la recherche des plus proches voisins, dont l'efficacité dépend différemment du nombre et de la dimension des observations. Comme leur nom l'indique, les algorithmes *ball tree* et *KD tree* sont tous deux basés sur des arbres de décision. Puisque la dimension de nos données est assez grande [14], nous avons utilisé l'algorithme *ball tree*.

L'algorithme des plus proches voisins est très efficace et le temps d'apprentissage est réduit au minimum, cependant la nature même de l'algorithme implique un temps de prédiction très long.

Dans notre problème, les meilleurs résultats ont été obtenu en s'intéressant aux 3 plus proches voisins.

14. Le seuil critique étant souvent localisé aux alentours de D égale à 30.

Quatrième partie
Résultats

Dans toute cette partie, il peut être utile de se référer au tableau 14.6 situé en annexe.

11 Performance prédictive des algorithmes

11.1 Performance globale

Comme prévu les méthodes linéaires sont celles qui fournissent les moins bons résultats. En effet, l'accuracy score du perceptron est de 86,86% et celui de la régression logistique multinomiale est de 92,07%. Ces méthodes font donc figure de mauvais élèves en terme de performance même si elles font partie des plus rapides.

Même si les arbres de décision ont un score de seulement 87,69%, les méthodes d'ensembles les employant en tant que classificateur faibles obtiennent de très bons scores puisque les random forests ont un accuracy score de 95,88% et avec un AdaBoost sur des arbres de bonne taille, nous avons obtenu un score de 99,01%. Les arbres de décision jouent donc bien leur rôle de brique élémentaire.

Quant aux SVM, elles obtiennent de très bons résultats avec un accuracy score de 98,23% en sélectionnant les meilleurs paramètres avec le noyau gaussien. Enfin, les méthodes des plus proches voisins sont presque tout autant efficaces avec un accuracy score de 97,05%.

11.2 Quels labels posent problème ?

À la lumière des matrices de confusion et des courbes ROC des différents algorithmes, on remarque que les performances des algorithmes sont similaires qualitativement et très dépendantes des classes étudiées. Certaines classes sont ainsi bien moins bien prédites que la moyenne, ce qui est le cas de la classe 9 qui est la classe la moins bien prédite et que nos algorithmes confondent tous avec l'ensemble des autres labels. Les classes 3, 4, 5, 7 et 8 sont également régulièrement mal prédites. A contrario, les classes 0, 1 et 6 sont en moyenne mieux prévues que les autres par l'ensemble de nos algorithmes et même très bien prévues par nos algorithmes compétitifs.

Surtout, on peut voir certains liens se dessiner entre les classes. Les classes 3 et 5 sont ainsi régulièrement confondus entre elles, tout comme les classes 6 et 8. Ce fait nous a amené à proposer des méthodes de prévision un peu plus hétérodoxes que nous vous présentons en partie 5.

12 Complexité des algorithmes

12.1 Complexité en temps de calcul

On remarque une grande diversité entre nos différents algorithmes au niveau des temps de calcul nécessaires qui dépendent grandement de la nature de ces derniers. Les phases d'apprentissage et de prédiction des modèles linéaires sont courtes, de même pour les arbres de décision. Cela est lié à la simplicité des modèles étudiés. Les forêts aléatoires sont légèrement plus lentes à entraîner en raison du grand nombre

d'arbres requis mais la phase de prédiction s'effectue rapidement. Les machines à vecteurs de support avec noyaux non-linéaires sont plus lentes tant à l'entraînement qu'en prédiction en raison du temps de computation nécessaire au grand nombre d'appels des fonctions noyaux. Comme prévu, l'algorithme AdaBoost est extrêmement lent à entraîner en raison de sélections successives des classificateurs faibles, mais il est l'un des plus rapides en prédiction. À l'opposé, l'entraînement des algorithmes de plus proches voisins est quasiment instantanée puisqu'elle se limite à stocker les données d'apprentissage en mémoire, mais le temps de prédiction est le plus important, les plus proches voisins étant les seuls algorithmes *lazy* que nous avons étudiés.

Il est cependant difficile dans notre cas de fournir des données chiffrées solides concernant les temps d'entraînement et de prédiction de nos algorithmes puisque nous avons dû les entraîner sur des machines très différentes en termes de puissance en raison des soucis de serveur que nous avons subis.

En résumé, on a :
— Les algorithmes simples qui s'entraînent et prédisent vite : le perceptron, l'arbre de décision ou la régression logistique multinomiale.
— Un algorithme qui s'entraîne moyennement vite et qui prédit vite : les forêts d'arbres aléatoires.
— Un algorithme qui s'entraîne lentement et qui prédit moyennement vite : les machines à vecteurs de support).
— Un algorithme qui s'entraîne très lentement et prédit très vite : AdaBoost
— Un algorithme qui s'entraîne vite et qui prédit lentement : les plus proches voisins.

12.2 Complexité en espace mémoire

La diversité en termes de complexité en espace s'est beaucoup moins faite remarquer que la complexité en temps à quelques exceptions près. Il est vrai que les méthodes de *bagging* et de *boosting* nécessite le stockage de nombreux classificateurs faibles, ce qui peut se faire légèrement ressentir. Cependant leur complexité n'est pas comparable à celle des plus proches voisins, qui nécessite le stockage de toutes les données d'entraînement. Malgré cela, la complexité en espace a rarement été un frein au cours de notre projet, la mémoire vive des machines sur lesquelles nous compilions nos algorithmes étant tout à fait respectable.

Cinquième partie
Un peu de cuisine

13 46 nuances de *Random Forests* : Réduction du problème en classifications binaires

13.1 L'idée

La random forest, comme la plupart des algorithmes dans Scikit-Learn, permet d'accéder à des probabilités d'appartenance aux différentes classes. En effet, l'output d'une random forest est la moyenne des probabilités d'appartenance aux classes des n sous-arbres. Ainsi, avec un *.predict_probas()* on obtient pour chaque image un vecteur de 10 probabilités d'appartenance aux 10 classes.

En mesurant le taux d'erreur brute sur les deux premiers labels, on constate que plus de 99% de nos bons labels sont prédits par les deux plus grandes probabilités. D'autre part, nous avons constaté que des random forests binaires (qui ne distinguent que deux labels) ont des *accuracy scores* qui varient entre 98% et 99%. Une fois ces deux informations obtenues, nous avons essayé d'augmenter nos performances en combinant ces deux constats.

Pour ce faire, pour chacun de nos éléments de notre ensemble de test, nous avons récupéré les deux premiers labels prédits par la random forest entraînée sur les dix labels. Nous avons ensuite repassé chacune de ces observations dans la random forest entraînée sur les deux classes en question. Ce qui fait 45 nouvelles random forests.

13.2 Les résultats

Les résultats sont meilleurs que ceux de la random forest multiclasses. En effet, cette dernière avait un score de 95,88% et avec notre nouvelle méthode, le nouveau score est de 96,41% soit une amélioration de 0,53% et donc une meilleur prédiction moyenne sur 53 images. Néanmoins, les labels qui bénéficient de cette amélioration sont les 3, 4, 5, 7 et 8. C'est-à-dire que l'on ne s'améliore pas sur le label le plus difficile à prévoir (le 9) mais on gagne sur toute la galaxie de labels sur laquelle nos scores étaient intermédiaires. En bref, on se renforce sur les labels qu'on prédisait déjà plutôt bien mais on ne s'améliore pas sur le label qui nous posait le plus de problèmes.

Si la différence n'est pas immédiatement visible sur les matrices de confusion ci-après, en s'y attardant, on remarque que le nombre de cases bleues visibles (c'est-à-dire que l'on confondait beaucoup) semble réduire.

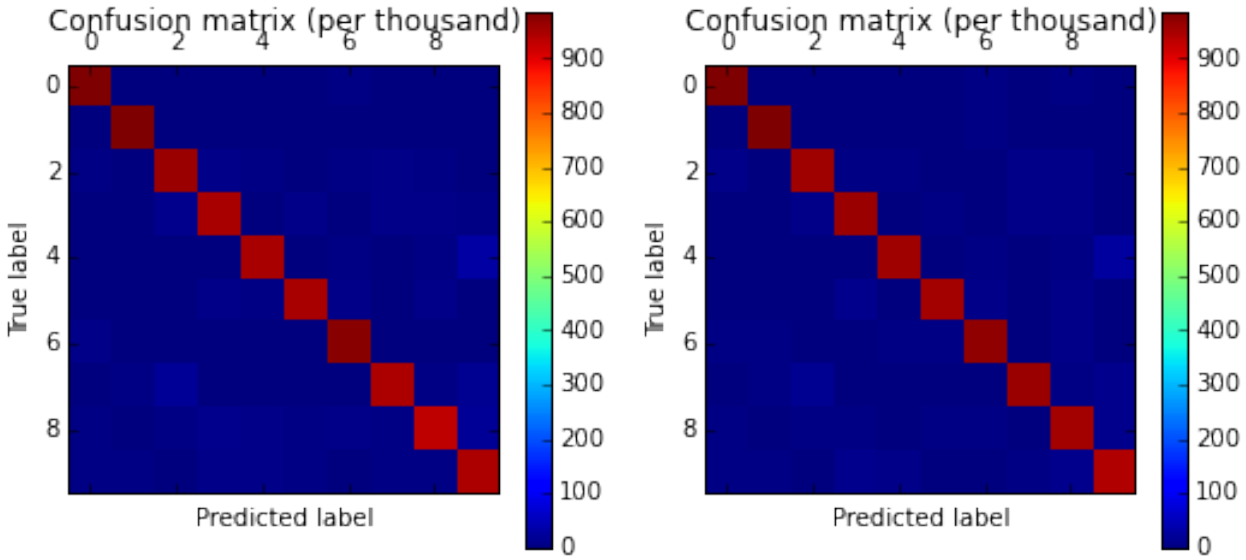

FIGURE 8 – Les deux matrices de confusion - Random forest & 46 shades

Conclusion

Bien que nous ne soyons pas un établissement bancaire, les avancées lors de ce projet nous confortent dans l'idée que nous aurions les capacités d'automatiser le traitement de nos chèques et satisfaire de fait nos clients. Nos algorithmes se rapprochant fortement des précisions humaines [15], il nous semble envisageable de construire, avec les moyens adaptés, un excellent système pour répondre à nos contraintes de **rapidité** et **précision** de prédiction.

Nous retiendrons de ce projet :

— La pratique approfondie de Scikit-Learn : de l'utilisation à la construction d'estimateurs
— Les fondements théoriques de l'apprentissage supervisé
— La gestion d'une machine virtuelle... et ses aléas
— La connaissance pratique d'un large panel d'algorithmes de *machine learning*
— De bonnes idées qui ont amélioré nos algorithmes

Si nous devions aller plus loin, nous envisagerions de plus amples transformations préalables des données (avec des extractions de caractéristiques plus poussées, ou simplement le redressement des chiffres) ou encore des combinaisons d'algorithmes plus ambitieuses. En effet, le fait de n'avoir pu que partiellement implémenter des agrégats d'algorithmes reste pour nous une frustration. Néanmoins, nous terminons ce projet avec le sentiment d'avoir beaucoup appris dans un domaine passionnant.

Quelques chiffres clés :

— Plus de 5000 lignes de codes
— 4 machines distantes
— Une dizaine d'algorithmes
— Un accuracy score de plus de 0,99
— 5 changements au *leader board* avec dans l'ordre chronologique : régression logistique multinomiale, random forest, 46 random forest, machine à vecteurs de support, machine à vecteurs de support sur données transformées (HOG), adaboost sur des arbres de profondeur 17

15. Une erreur moyenne d'environ 0.2%

Sixième partie
Bibliographie

Références

[BMP02] E Belongie, J Malik et J Puzicha. « Shape Matching and Object Recognition Using Shape Contexts ». In : *Transactions on pattern analysis and machine intelligence* (2002).

[Cha+02] O Chapell et al. « Choosing Multiple Parameters for Support Vector Machines ». In : *Machine Learning* (2002).

[ML02] H Motoda et H Liu. « Feature Selection, Extraction and Construction ». In : *Pacific-Asia Conference on Knowledge Discovery and Data Mining* (2002).

[Kav+03] E. Kavallieratou et al. « Handwritten Word Recognition based on Structural Characteristics and Lexical Support ». In : *International Conference on Document Analysis and Recognition* (2003).

[CCR10] R Cruz, G Cavalcanti et T.I. Ren. « Handwritten Digit Recognition Using Multiple Feature Extraction Techniques and Classifier Ensemble ». In : *International Conference on Systems, Signals and Image Processing* (2010).

[KS13] V. Kulkarni et P Sinha. « Random Forest Classifiers :A Survey and Future Research Directions ». In : *International Journal of Advanced Computing* (2013).

Septième partie
Annexes

14 Annexes

14.1 Histogramme de répartition des données utilisées

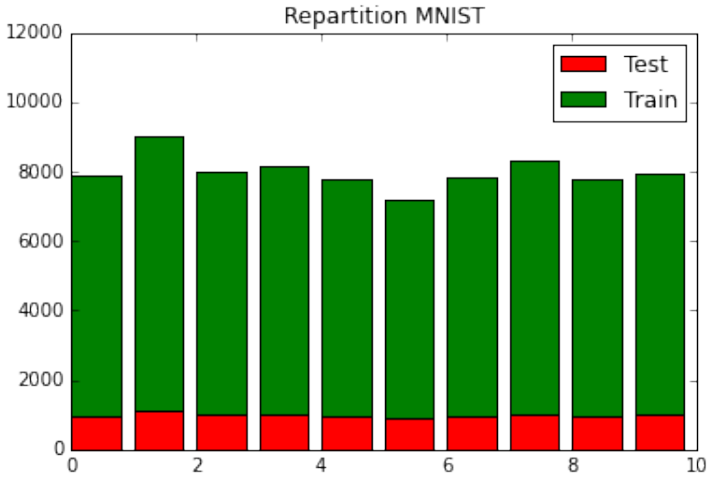

FIGURE 9 – Histogramme de répartition des bases utilisées

14.2 Résultats du perceptron

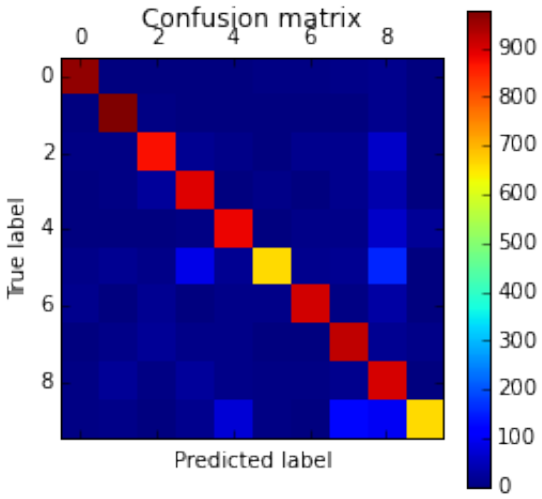

FIGURE 10 – La matrice de confusion de perceptron

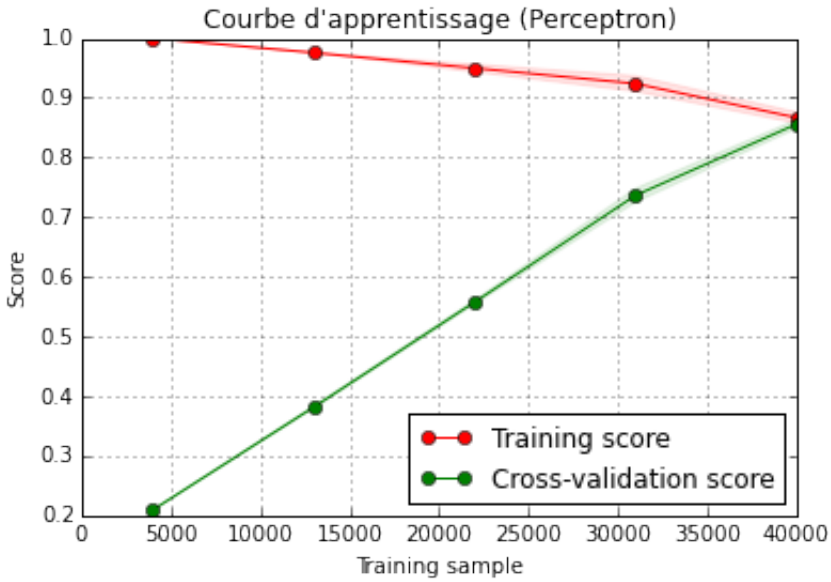

FIGURE 11 – Les courbes d'apprentissage du perceptron

14.3 Résultats de la régression logistique multinomiale

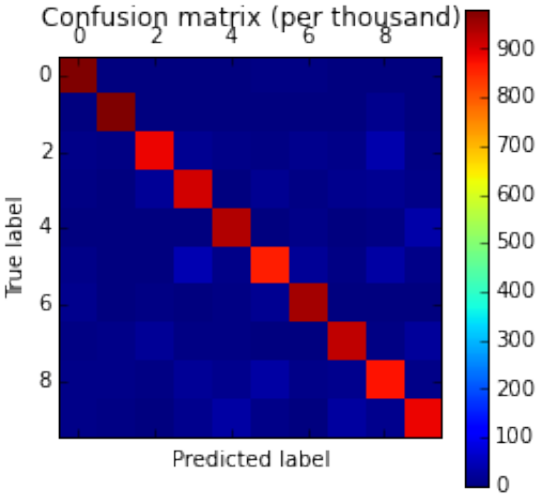

FIGURE 12 – La matrice de confusion de la régression logistique multinomiale

14.4 Résultats de l'arbre de décision

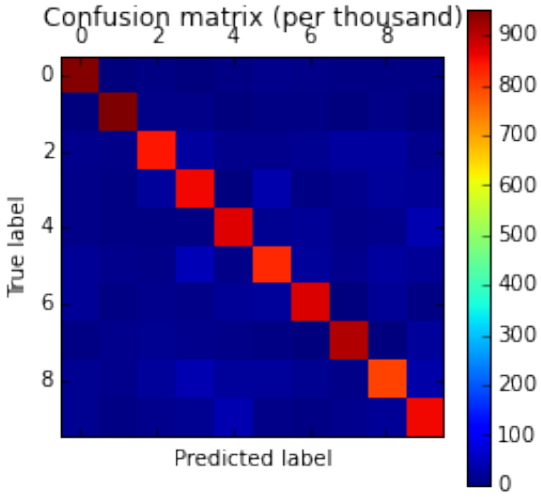

FIGURE 13 – La matrice de confusion de l'Arbre de Décision

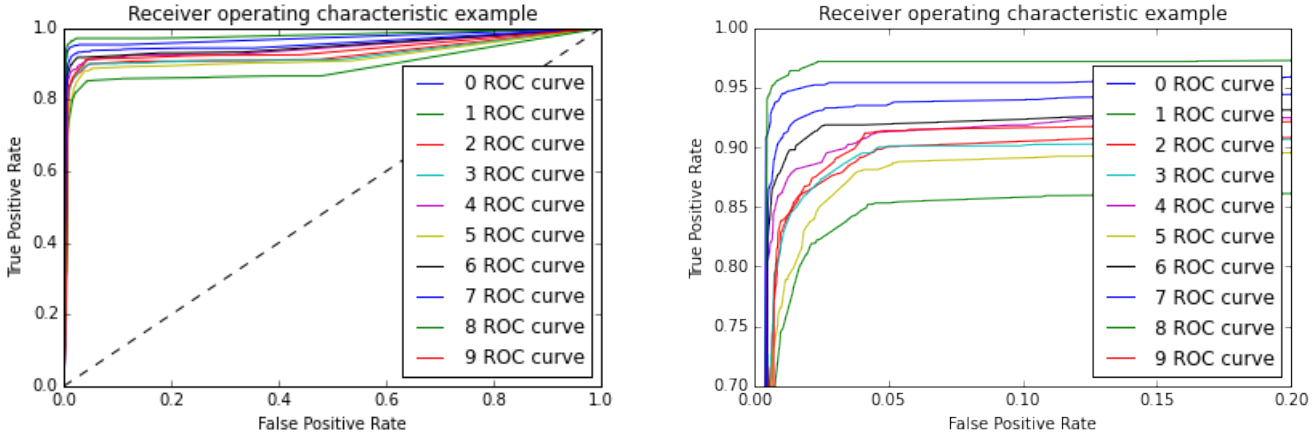

FIGURE 14 – Les courbes ROC de l'Arbre de Décision

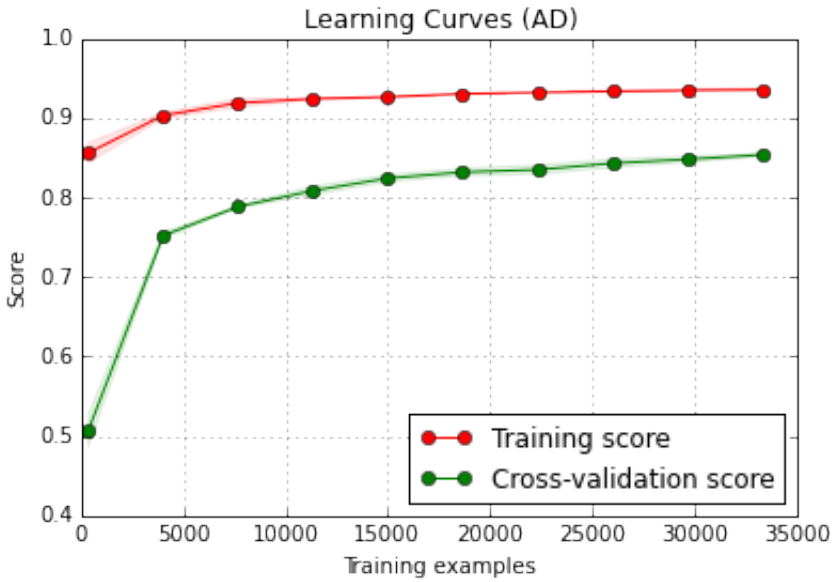

FIGURE 15 – Les courbes d'apprentissage de l'Arbre de Décision

14.5 Résultats des Random Forests

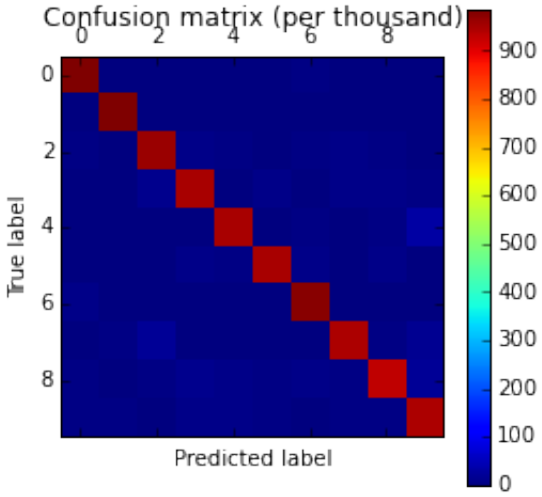

FIGURE 16 – La matrice de confusion de la random forest

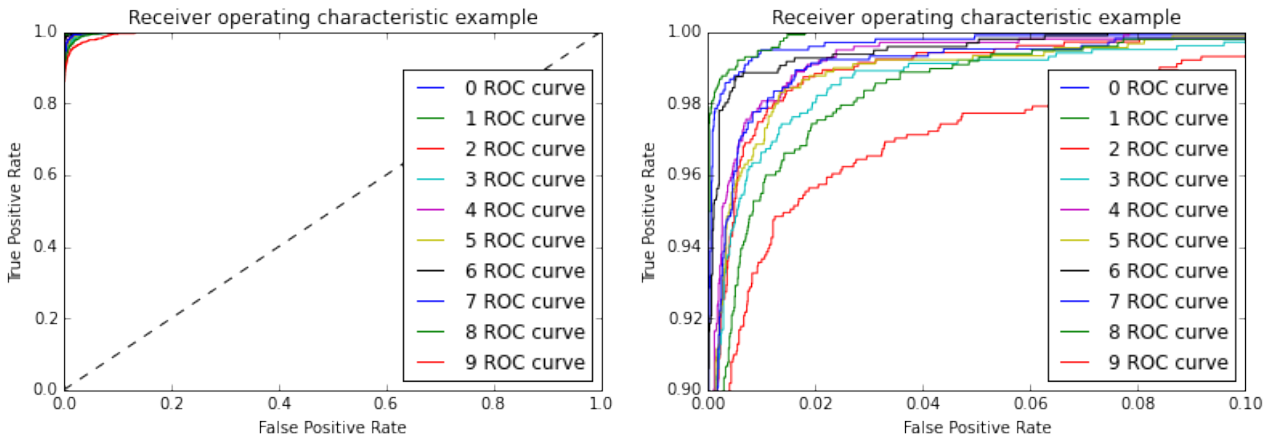

FIGURE 17 – Les courbes ROC de la random forest

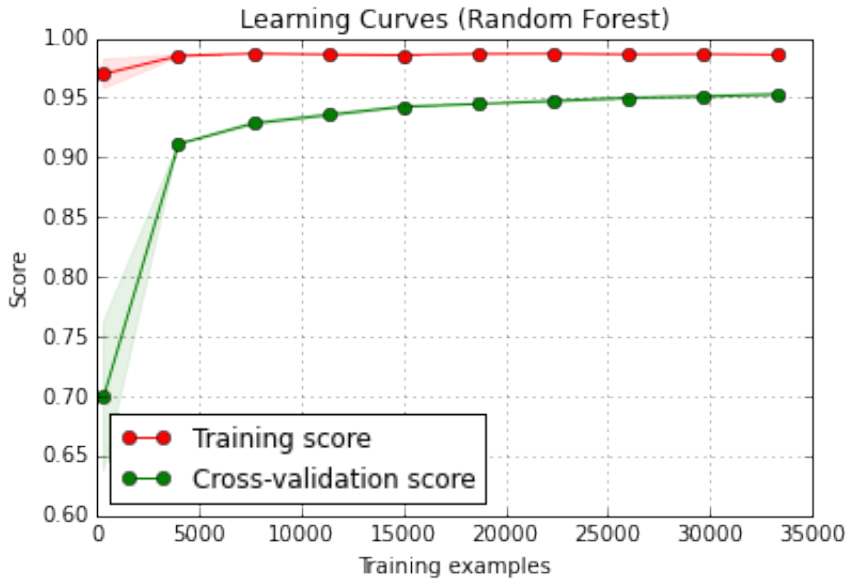

FIGURE 18 – Les courbes d'apprentissage de la random forest

14.6 Tableau des performances

Algorithme (paramètre)	Temps fit (s)	Temps prédict (s)	Accuracy (%)	AUC0	AUC1	AUC2
Perceptron	3.7	0.04	86.86	×	×	×
SVM polynomial (degré = 3, C =...)	×	×	×	×	×	×
SVM linéaire	×	×	×	×	×	×
SVM gaussien	×	×	×	×	×	×
Logistique	×	×	92.07	0.9986	0.9985	0.9831
Arbre de décision	×	×	87.69	0.9718	0.9807	0.9245
Adaboost(Arbre de taille 2)	×	×	×	×	×	×
Adaboost(Arbre de taille 17)	×	×	99.01	×	×	×
Random forest	23.94	0.30	95.88	0.9996	0.9999	0.9986
"46 shades of" Random forest	22450	×	96.41	×	×	×
K_nn (n=3)	9.89975214005	645.689776897	97.05	×	×	×

Algorithme (paramètre)	AUC3	AUC4	AUC5	AUC6	AUC7	AUC8	AUC9
Perceptron	×	×	×	×	×	×	×
SVM polynomial (degré = 3, C =...)	×	×	×	×	×	×	×
SVM linéaire	×	×	×	×	×	×	×
SVM gaussien	×	×	×	×	×	×	×
Logistique	0.9887	0.9945	0.9802	0.9971	0.9921	0.9920	0.9838
Arbre de décision	0.9172	0.9425	0.9132	0.9458	0.9548	0.8937	0.9397
Adaboost(Arbre de taille 2)	×	×	×	×	×	×	×
Adaboost(Arbre de taille 17)	×	×	×	×	×	×	×
Random forest	0.997617	0.999096	0.998692	0.999470	0.998681	0.998063	0.996168
"46 shades of" Random forest	×	×	×	×	×	×	×
K_nn (n=3)	×	×	×	×	×	×	×