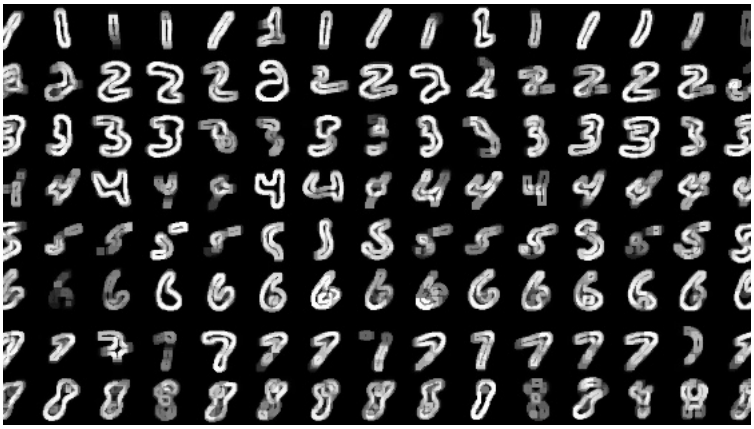
UNIVERSITÉ DE NANTES

IAE NANTES
ÉCONOMIE & MANAGEMENT

Machine Learning with Python

Kaggle project : Digit Recognizer



Riwan PERRON
Teodoro MOUNIER TEBAS
Diane THIERRY

Enseignant : M. Abrahamson
Année universitaire : 2020-2021

Master 2 Économétrie et Statistiques, parcours Économétrie Appliquée

Table des matières

1 Introduction

Le Machine Learning (ML) est une technologie d'intelligence artificielle permettant l'apprentissage automatique aux ordinateurs par le biais de différents algorithmes mis en oeuvre par l'homme. L'application de machine learning *"spam filter"* ayant vu le jour dans les années 90 en est un bon exemple. Elle permettait de filtrer les emails en les triant en non désirés lorsque ceux-ci validaient un certain nombre de caractéristiques propres aux spams, de telle manière qu'il était rare de devoir les trier soi-même. Ainsi, le machine learning consiste à donner aux ordinateurs la capacité d'apprendre et de se perfectionner sans être explicitement programmés.

Nous distinguons dans ce dossier cette science du data mining, qui peut être lui considéré comme une ressource du machine learning. En effet, le data minging consiste à extraire d'une base de données les informations les plus utiles pour détecter des tendances, des patterns - informations alors réutilisées par des individus, contrairement au ML qui vise à automatiser le processus via l'IA.

Au fil des années, de nombreuses méthodes de ML se sont développées après le 1er algorithme mis en oeuvre par Frank Rosenblatt en 1957 appelé le **Perceptron**. Cette machine a été construite pour la reconnaissance et la classification d'images, et est basée sur le même principe qu'un neurone humain, c'est à dire constitué de k entrés avec des poids différents et d'une sortie binaire.

Ce dossier a pour but de répondre à la problématique de la compétition kaggle intitulée "Digit Recognizer : Learn computer vision fundamentals with the famous MNIST data". L'objectif de cette dernière, dont les données d'images manuscrites sont issues du 'Modified National Institute of Standards and Technology', est de reconnaître les chiffres inscrits sur chacune des images à l'aide de centaines de pixels colorés en noir et blanc et qui, assemblés, forment un chiffre compris entre 0 et 9. Effectivement, il a été demandé à plusieurs milliers de personnes de dessiner des digits (chiffres) compris entre 0 et 9 ; ces représentations ont été enregistrées sous forme d'images, et le but de l'analyse est de réussir à prédire les digits à partir de la coloration des pixels qui composent l'image. Nous procéderons ainsi à différentes classifications, en cherchant chaque fois à minimiser le taux d'erreur de prédictions pour ainsi obtenir la meilleure modélisation possible.

Dans une première partie nous nous intéresserons à la théorie cachée derrière les techniques de machine learning en les explicitant une à une, après avoir déroulé le processus de modélisation en ML. Dans une seconde partie nous procéderons à l'analyse exploratoire de nos données pour bien les cerner et répondre au mieux à la problématique posée par la compétition kaggle. Puis, nous appliquerons les méthodes de ML présentées partie 1 pour finir sur l'interprétation des résultats dans une 4è partie.

2 Présentation des méthodes de Machine Learning

2.1 Processus de modélisation

2.1.1 Division train et test

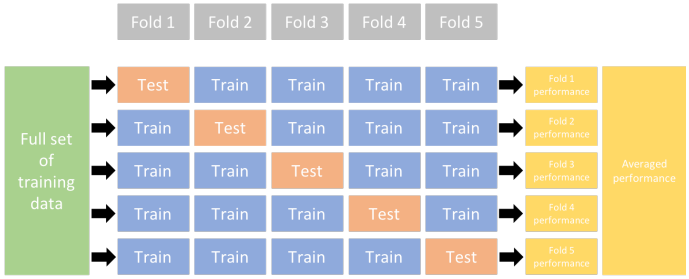
Le principal objectif du Machine Learning est que l'algorithme puisse prédire le mieux possible les valeurs souhaitées (dans notre cas les bons chiffres), à partir de données passées. L'algorithme sera intéressant seulement s'il est généralisable c'est à dire capable de prédire correctement à partir de nouvelles données. Afin de se rendre compte de cet aspect généralisable de notre modèle, il est important de diviser la base de données en deux sous-ensembles. Un ensemble d'apprentissage pour entraîner nos algorithmes, régler les hyperparamètres et comparer les modèles entres eux, et un ensemble test destiné à estimer notre modèle de manière non biaisée puisqu'il sera alors appliqué à des données n'ayant pas servi à l'apprentissage.

Dans la plupart des cas on entraîne les modèles sur 70% de la base et on teste leurs performances sur les 30% restants. Si la proportion de l'échantillon d'apprentissage est trop importante, alors il y a le risque que le modèle soit sur-ajusté aux données d'apprentissage, et donc prédise moins bien sûr l'échantillon test. À l'inverse, un échantillon trop petit pour l'apprentissage ne permettra pas d'avoir une bonne évaluation des paramètres du modèle. Dans le cas où les bases de données ont une quantité très importante d'observations, il n'est pas nécessaire de respecter ces pourcentages car cela alourdirait le temps d'apprentissage et les gains marginaux seraient insignifiants.

La similitude dans les distributions de l'échantillon test et train a une importance lors de la séparation de la population à modéliser. Dans des problèmes de classification, la variable de réponse peut être déséquilibrée avec 90% de "oui" et 10% de non. Afin de bien prédire les deux alternatives, des méthodes permettent d'obtenir des sous-échantillons équilibrés afin de ne pas biaiser les prévisions.

2.1.2 Méthodes de rééchantillonnages

Une fois notre base divisée, la phase d'apprentissage sur l'échantillon 'train' commence. Au cours de cette phase de formation, il est important d'évaluer la performance générale du modèle : cela est permis par les méthodes de ré-échantillonnage élaborées pour la "validation" du modèle. Ces méthodes consistent à diviser davantage la base train afin d'estimer les performances sur un ensemble de validation. Les deux méthodes principales sont la **Cross-validation** et le **Bootstrapping**. La méthode de validation croisée est une méthode de ré-échantillonnage qui divise aléatoirement les données d'apprentissage en k groupes de tailles égales. Le modèle se forme alors sur les k-1 groupes et s'entraîne sur le dernier - et ainsi de suite k fois. Une fois les modélisations terminées, l'estimation de la CV correspond à la moyenne des k erreurs de test. La figure ci-dessous schématise ce processus :



La méthode du Boostrapping est elle, principalement utilisée dans les **forêts aléatoires**. Un échantillon Boostrap étant construit par un tirage avec remise de *n-observations* parmi *n-observations*, il correspond à un cas particulier de l'échantillon d'origine car il garde les mêmes proportions mais est distribué de manière aléatoire. Lors de la construction de l'échantillon Bootstrap, environ 1/3 des observations ne sont pas prises en compte par le tirage ; elles constituent ainsi l'échantillon **Out of bag** (OOB). Le modèle est donc construit à partir des échantillons "bootstrappés", et validé sur les échantillons 'OOB'.

2.1.3 Tuning du modèle

Les erreurs de prédiction peuvent être décomposées en deux parties : l'erreur due au biais d'une part - c'est à dire la différence entre la prédiction attendue de notre modèle et la valeur correcte que l'on essaie de prédire, et l'erreur due à la variance d'autre part - qui peut se définir comme la variabilité d'une prédiction pour une observation donnée. Autrement dit, les modèles à variance élevée (kNN, RandomForest) prédiront très bien et auront donc des erreurs dues au biais très faibles. En revanche, ils ont un risque de sur-ajustement aux données d'apprentissage, et donc un risque que le modèle ne soit pas généralisable sur l'ensemble des données. La subtilité réside donc dans le compromis entre les erreurs dues au biais et celles dues à la variance. Pour cela, de nombreux hyper-paramètres contrôlent la complexité des modèles et, s'ils sont bien paramétrés, permettent aussi d'en optimiser les performances. De tels paramètres de réglage (des algorithmes d'apprentissage) doivent être ajustés en fonction du jeu de données et du problème traité. Cela peut se faire manuellement jusqu'à trouver la combinaison parfaite des paramètres qui minimise l'erreur de Cross-Validation, ou bien par une recherche en grille appelée **'grid search'** qui explore les valeurs des hyper-paramètres sélectionnées, et s'arrête lorsqu'il a trouvé la combinaison qui maximise la réduction de l'erreur.

2.1.4 Validation du modèle

Après estimation de différents modèles via différentes méthodes expliquées ci-dessus, il convient d'évaluer la qualité de tels modèles. Pour les modèles de classification, c'est la **matrice de confusion** qui permet d'en évaluer la qualité, sur la base de différents niveaux de performances : elle regroupe donc l'ensemble des résultats de la prédiction. Elle se décompose en *vrai-positifs* lorsque la prédiction d'un évènement (Y=1) est la même que la valeur réellement observée, et en *faux-positifs* lorsque les valeurs prédites par le modèle sont différentes des valeurs observées. De même, il s'agit de *vrai-négatifs* quand la prédiction de l'évènement Y=0 est réalisée, et de *faux-négatifs* donc le cas contraire. Le tableau ci-dessous représente cette matrice de confusion :

	Evènememement prédit	Evènememement non-prédit
Evènement	Vrai-positif	Faux-négatif
Non-évènement	Faux-positif	Vrai-négatif

A partir de cette matrice, nous pouvons calculer plusieurs indicateurs de performance du classificateur que nous cherchons à maximiser :

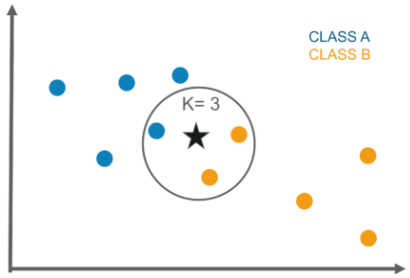
- **Accuracy** $(\frac{VP+VN}{total})$ à quelle fréquence le classificateur est-il correct ?
- **Precision** $(\frac{VP}{VP+FP})$ pour le nombre de prédictions que nous avons faites, combien étaient correctes ?
- **Sensitivity** $(\frac{VP}{VP+FN})$ parmis les évènement survenus, combien en avons nous prédits ?
- **Specificity** $(\frac{VN}{VN+FP})$ avec quelle précision le classificateur classe-t-il les non-événements réels ?

2.2 Algorithmes d'apprentissage supervisé

Nous avons ainsi développé la démarche de modélisation avec les étapes d'application que l'on suit lors de la mise en pratique des méthodes de Machine Learning. Voyons à présent les différents types d'algorithmes existants. Nous développerons ainsi, dans les sections à venir, les méthodes dites "supervisées" c'est à dire où l'on cherche à expliquer/prédire un phénomène/une variable notée Y. Cette catégorie de méthodes représente près de 95% des méthodes de ML, les autres consistant en l'apprentissage où les données ne sont pas étiquetées. Outre la méthode d'apprentissage, on distingue dans le machine learning les classifications des régressions - ces premières servant à l'explication de phénomènes qualitatifs, et les régressions à l'explication de phénomènes quantitatifs. Dans le cadre de notre étude, nous utiliserons donc des approches de classifications puisque nous travaillons à reconnaître des images où Y est composé de 10 modalités, à savoir les labels de chiffre compris entre 0 et 9.

2.2.1 k plus proches voisins (kNN)

L'objectif de cette méthode (nommée kNN) est de classer chaque observation dans la catégorie des K observations de l'échantillon d'entraînement qui lui corresponde le plus. Chaque objet est donc classé selon la classe d'appartenance de ses plus proches voisins. On fixe ainsi k comme le nombre de plus proches voisins considérés souvent impair, de manière à ne pas avoir de classes équivalentes en nombre. La prédiction effectuée pour chaque observation à travers cette méthode est donc basée sur la similarité avec d'autres observations voisines. Ainsi, en identifiant k observations similaires ou proches d'une nouvelle entrée considérée, l'algorithme utilise la classe majoritaire de ces k observations comme sortie prévue pour cette entrée précise (dans le cadre d'une *classification*).



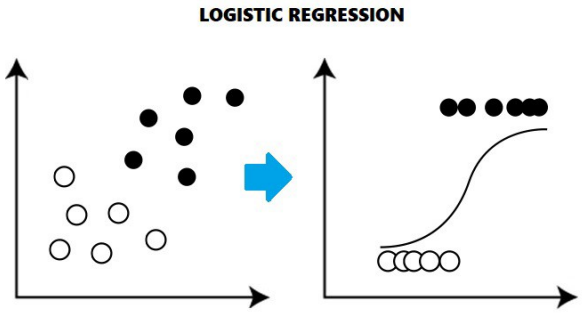
Cependant, la distance n'a pas d'influence sur la classification issue de cette méthode et il peut y avoir des erreurs liées au choix du nombre de voisins. C'est pourquoi on parle parfois de kkNN ou kNN pondéré, où les distances sont transformées de manière à connaître le poids de chacun des voisins et à déterminer plus précisément les similarités entre les observations. Il existe différentes métriques de distance dont les plus connues sont les Euclidienne, Mahnattan, de Minkowski etc.

Enfin, la réussite de la méthode des k plus proches voisins dépend fortement du nombre de voisins implémenté dans l'algorithme. Ainsi, plus les données ont des caractéristiques "bruyantes" (c'est à dire non pertinentes) plus le nombre de voisins considérés augmente - de manière à lisser ce bruit. A contrario, lorsque les données ont un signal de données faible, peut de voisins sont nécessaires à considérer dans la construction des modèles. [1]

1. Le signal des données correspond à la méthode de transfert d'informations ; binaire

2.2.2 Régressions logistiques multinomiales

La régression linéaire est parfois utilisée en machine learning pour approximer la relation entre une variable de réponse et un ensemble de variables prédictives. Cependant, lorsque la variable de réponse est binaire ou multinomiale, la régression linéaire n'est pas appropriée. C'est alors que l'on se tourne vers une méthode analogue ; la régression logistique qui a pour objectif de classer des individus dans différentes catégories (dans le cas de 2 catégories, il s'agit de régressions binaires). Cette méthode est dite "non-paramétrique" puisqu'elle n'exige pas que la distribution de la population soit caractérisée par certains paramètres.



La régression logistique multinomiale consiste à désigner une catégorie de référence et à exprimer chaque logit des autres modalités par rapport à cette référence, à l'aide d'une combinaison linéaire des variables prédictives.

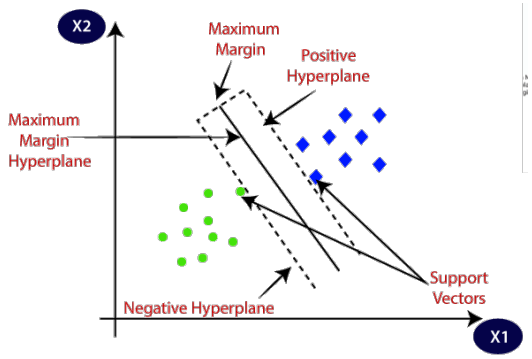
On prédit avec les régressions logistiques multinomiales la probabilité de la réponse, aussi elles permettent de comprendre l'impact de chaque variable sur les choix. Le principal avantage de cette méthode est la robustesse des résultats ; étant donné que les modèles construits sont très simples, il y a peu de risque de sur-apprentissage et les résultats ont ainsi tendance à avoir un bon pouvoir de généralisation.

Les régressions logistiques forment partie des séparateurs linéaires et peuvent ainsi avoir des résultats moins satisfaisants que dans le cadre de relations non linéaires qui ont ainsi un plus large choix de séparations dans l'espace.

2.2.3 Machine à Support Vectorielle (SVM)

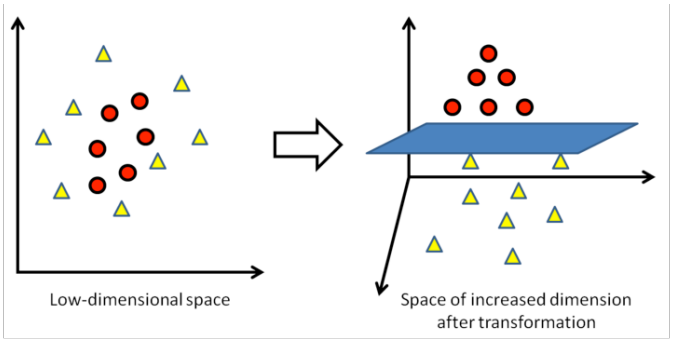
Contrairement aux méthodes de machine learning évoquées précédemment, les machines à support vectorielles ou 'séparateurs à vaste marge' (SVM) peuvent s'adapter à des relations non linéaires : elles essayent de trouver un hyperplan à p dimensions dans un espace de fonctionnalités, qui sépare au mieux les différentes classes. En ce sens les SVM viennent compléter, améliorer les régressions logistiques avec la possible non-linéarité des classifieurs. Dans le cas de séparations multiples (avec plus de 2 classes) comme dans notre analyse, on utilise deux approches générales : un contre tous (one-versus-all : OVA) et un contre un (one-versus-one : OVO). Dans OVA, nous ajustons un SVM pour chaque classe (une classe contre le reste) et classifions dans la classe pour laquelle la marge est la plus grande. Dans OVO, nous adaptons tous les SVM par paires et nous classons dans la classe qui remporte le plus de compétitions par paires.

L'hyperplan représente une frontière de décision qui partitionne l'espace de fonctionnalités en plusieurs ensembles, un pour chaque classe. Dans le cas d'une partition binaire par exemple, le SVM classera tous les points d'un côté de la frontière de décision comme appartenant à une classe, et tous ceux de l'autre côté comme appartenant à l'autre classe. Nous voyons cela par exemple dans la figure suivante.



Dans la pratique, il est cependant difficile (voire impossible) de trouver un hyperplan séparant parfaitement les classes en utilisant uniquement les fonctionnalités d'origine. La solution consiste alors utiliser l'astuce dite du noyau (kernel) pour agrandir l'espace des fonctionnalités, de telle sorte que la séparation des classes soit (plus) probable. Prenons l'exemple d'une classification linéaire où sur un plan en 2 dimensions, les observations de la catégorie A entourent les observations de la catégorie B de telle manière à former un cercle autour de celles-ci. Dans cette situation, l'utilisation de méthodes de classification linéaire seraient inefficaces puisque le problème n'est pas linéairement séparable. C'est alors qu'interviennent les kernels.

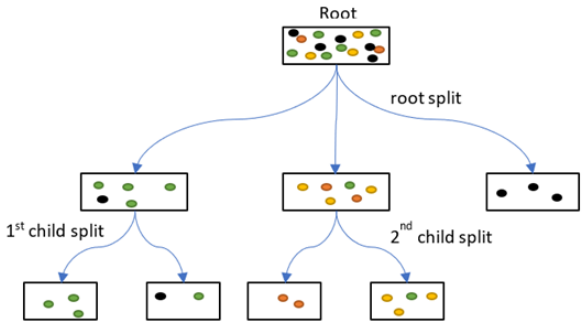
Nous pouvons agrandir l'espace des fonctionnalités en en ajoutant une troisième, de manière à obtenir un espace de caractéristique tridimensionnel. Dans ce nouvel espace, les classes sont parfaitement séparables par un hyperplan, on garde donc la frontière de décision que l'on projette alors sur l'espace de caractéristiques d'origine. On obtient ainsi une limite de décision non linéaire qui sépare parfaitement les données d'origine. Cette méthode permet de trouver un séparateur à des problèmes non séparables linéairement. Les SVM sont donc extrêmement flexibles et capables de trouver des solutions à des problèmes non linéaires complexes comme celui présenté ci-dessous.



Nous avons donc vu dans cette sous-section, qu'il existe des classifieurs SVM linéaires mais qui sont rapidement limités lorsque les relations ne le sont pas. On se tourne alors vers des classifieurs non-linéaires avec l'astuce du noyau appelée en anglais "kernel trick".

2.2.4 Arbres de décisions (Decision Trees)

Les arbres de décisions (CART : classification and regression tree) constituent eux aussi une méthode supervisée de Machine Learning utilisée dans la classification. En ML, un arbre est une classe d'algorithmes non paramétriques qui fonctionnent en partitionnant l'espace d'entités en un certain nombre de régions plus petites avec des valeurs de réponse similaires, à l'aide d'un ensemble de règles de fractionnement. Cette méthode de division a pour avantage de produire des règles simples qui sont faciles à interpréter et à visualiser avec des diagrammes en arbre. Le fonctionnement des arbres de décision est le suivant : à chaque noeud l'algorithme considère les N variables et cherche celle qui divise le jeu de données de la manière la plus optimale possible ; on cherche alors à maximiser la diminution globale de l'impureté. Plus simplement, à chaque noeud l'algorithme considère les variables les plus influentes et divise à partir de celles-ci un ensemble en 2 groupes les plus hétérogènes entre eux, et les plus homogènes en leur sein.



Il convient alors de trouver le seuil à partir duquel le jeu de données sera coupé (le nombre minimal et maximal de découpages qui correspondra à la profondeur de l'arbre), et le nombre d'itérations optimal qui permette de classer les observations sans complexifier le modèle (la situation extrême consisterait en effet en une classe pour chaque observation). Effectivement, un arbre de décision peu profond, c'est à dire avec peu de noeuds terminaux aura une faible capacité prédictive puisqu'une fonction de prédiction est associée à chaque région de l'arbre. En revanche, un arbre profond/complexe permettra de prédire parfaitement les observations du jeu d'apprentissage, mais alors le risque de sur-ajustement est existant. [2]

Pour éviter ce risque de sur-ajustement on a recours à des modèles plus parcimonieux et donc plus robustes au moment de l'estimation ; obtenus par différentes méthodes d'élagage (*pruning*) de l'arbre. Parmi ces méthodes visant la construction d'arbres moins complexes mais avec la meilleure capacité prédictive, on retrouve les méthodes backward et forward. L'une considère un arbre "complet" où elle supprime les niveaux les plus bas, tandis que l'autre part du tronc de l'arbre pour définir le mieux possible les branches. Toutes deux choisissent, par cross-validation, l'arbre qui fournira l'erreur de prédiction minimale.

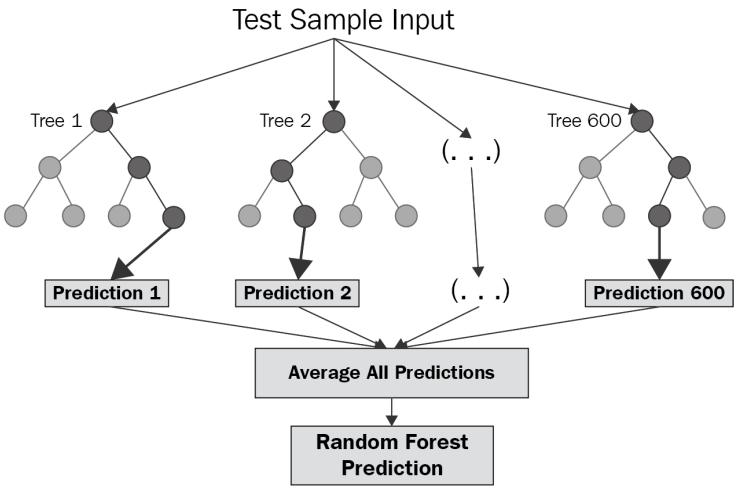
Ainsi, par rapport à un modèle linéaire, l'arbre de décision est robuste à la multicolinéarité, aux données de grandes dimensions, ainsi qu'aux données manquantes, et peut davantage traiter des interactions complexes. Malgré ses avantages certains comparé aux classifications linéaires, l'arbre de décision peut ne pas être pertinent sur les données test lorsque celles-ci sont très différentes des données d'apprentissage sur lesquelles il s'est fondé. Arrive alors le principe des forêts aléatoires (random forest).

2. Le sur-ajustement correspond à un modèle de prévision très instable car fortement dépendant des échantillons qui ont permis son estimation.

2.2.5 Forêts aléatoires (Random Forest)

Proposées par Ho en 1995 puis développées par Breiman en 2001, les forêts aléatoires sont un ensemble de CART décorrélés construits après randomisation des observations, dans le but de savoir répondre à tous les échantillons et variables possibles - et ainsi améliorer les précisions des prédictions. Une forêt aléatoire se construit de la manière suivante :

- Sélection d'un échantillon bootstrap (méthode de ré-échantillonnage présentée précédemment)
- Construction d'un arbre avec à chaque noeud le choix de la meilleure variable candidate pour la coupure, sélectionnée **au hasard** parmi un sous-ensemble de prédicteurs afin de rendre les arbres de la forêt le plus indépendants possible [3].
- Collecte de tous les arbres de la forêt et prédiction finale ; valeur moyenne ou classe majoritaire observée sur l'ensemble des prédictions individuelles



Dans le but d'améliorer les prévisions, il est possible de contrôler la profondeur et la complexité des arbres de décisions composant la forêt grâce aux hyperparamètres. Cela comprendra la taille du nœud, la profondeur maximale, le nombre maximal de nœuds terminaux ou la taille de nœud requise pour permettre des fractionnements supplémentaires.

La construction d'une forêt aléatoire peut aussi être utile pour la sélection de variables. En effet, il est possible de construire une forêt composée d'un certain nombre d'arbres, de répéter la procédure x fois, de classer les variables en fonction de leur importance et d'évaluer le RMSE (root mean square error) pour pouvoir sélectionner le meilleur sous-ensemble de variables.

En comparaison aux CART, l'algorithme d'apprentissage 'random forest' bénéficie de bonnes performances prédictives avec relativement peu de réglages d'hyperparamètres et permet d'améliorer significativement les performances prédictives, grâce à l'injection d'un caractère aléatoire dans le processus de croissance des arbres. Voyons à présent, dans une dernière sous-section, une autre méthode de Machine Learning : les réseaux de neurones, issus eux aussi de l'intelligence artificielle.

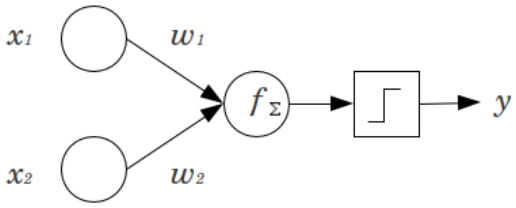
3. A chaque noeud de chaque arbre on tire un sous-ensemble de prédicteur aléatoire pour que les arbres ne soient pas corrélés entre eux lors de la construction de la forêt aléatoire.

2.2.6 Réseaux de neurones (Neural Networks)

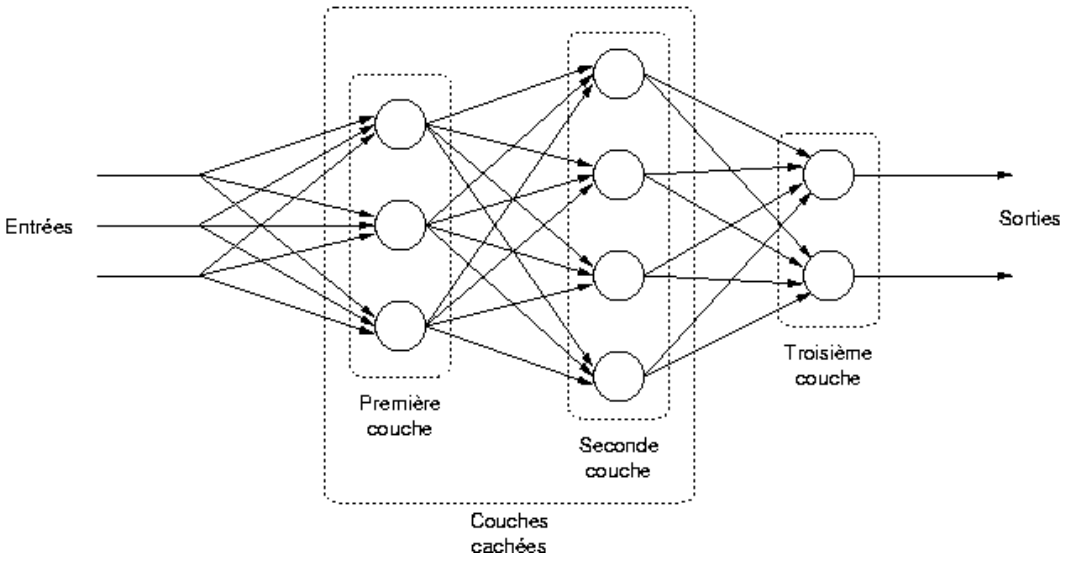
Comme leur nom l'indique, les réseaux de neurones artificiels (ANN) se basent sur un fonctionnement neuronal du cerveau humain pour apprendre, avec la motivation sous-jacente de créer des modèles imitant le raisonnement humain. Ce concept des réseaux de neurones ou 'connexionisme' fut inventé en 1943 par 2 chercheurs de l'Université de Chicago.

Pour former le système nerveux, les neurones sont connectés entre eux suivant des répartitions spatiales complexes. Les connexions entre deux neurones se font dans les synapses où ils sont séparés par un espace synaptique. Un neurone artificiel se compose d'entrées pondérées ou non, d'une fonction d'activation des entrées vers la sortie, et d'une sortie notée Y. Le réseau de neurones correspond donc à une interconnexion d'éléments simples et d'échanges d'informations via ces connexions.

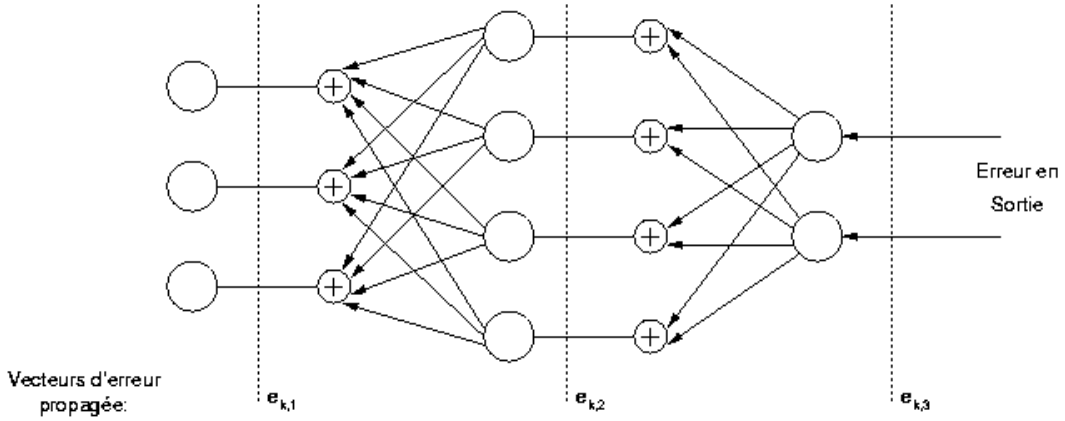
Comme expliqué en introduction, le premier algorithme de ML fut développé en 1957 par Rosenbalt ; il s'agissait du Perceptron pour l'apprentissage supervisé. Le but de l'apprentissage est de déterminer l'équation d'une surface qui sépare les points de classes différentes, appelée "surface discriminante" la plus adéquate possible.



L'objectif du Perceptron 'simple' est de classer des données en 2 classes, il est pour cela composé d'un seul neurone binaire. Il permet ainsi de construire des séparations linéaires à condition que les données soient linéairement séparables - sinon, l'algorithme itère à l'infini sans trouver de solution : il ne converge pas. De telles limites ont été mises en avant dès 1969 et avec elles, le besoin d'architectures plus complexes pour l'apprentissage : c'est alors le passage du modèle linéaire au modèle non-linéaire : le **perceptron multicouches**.



Le perceptron multi-couches (PMC) est caractérisé par une architecture *feedforward*, c'est à dire avec différents types d'interconnexions, une couche d'entrée, et plusieurs couches actives. Contrairement au Perceptron initial, ce deuxième algorithme est composé de fonctions d'activation non linéaires (excepté pour la couche de sortie), et de poids mis à jour par rétro-propagation du gradient. Les valeurs de x sont effectivement propagées de couche en couche de manière à optimiser le calcul de sortie du réseau, et à le rapprocher le plus de la sortie attendue. De telle sorte, si une erreur est mesurée en sortie, l'algorithme procède à la correction des poids synaptiques menant à la couche de sortie, et continue cette propagation (en arrière) des corrections dans les couches intermédiaires : ceci est appelé "descente de gradient". La rétro-propagation fonctionne comme la descente de gradient avec produit scalaire, et est donc plus adaptée à des calculs complexes. Son fonctionnement est le même, à savoir réduire le **coût** du réseau en passant de couche en couche depuis la sortie jusqu'à l'entrée, pour moduler les poids afin d'obtenir la sortie souhaitée. Ainsi, le PMC est capable d'approximer toute fonction continue, pourvu que l'on fixe convenablement le nombre de neurones dans la couche cachée.



Ainsi, l'apprentissage supervisé par les réseaux de neurones peut présenter un beau complément aux autres méthodes de machine learning, avec la possibilité de passer du modèle linéaire au modèle non-linéaire. Cependant, pour leur utilisation il est nécessaire de calibrer un certain nombre de paramètres comme le nombre de neurones de la couche cachée, le pas d'apprentissage et le nombre d'Epochs. [4]

4. Epoch correspond au nombre de passages sur la base.

3 Analyse exploratoire et traitement des données

Nous disposons pour cette étude d'une base composée de 785 variables où la première nommée **"label"** indique le digit représenté sur l'image, et les 784 autres contiennent les valeurs des pixels associés aux images. En effet, chaque image est constituée comme une matrice de 28 colonnes et 28 lignes où chaque pixel est une valeur de cette matrice à la i-ème colonne et la j-ème ligne. Les valeurs de ce dernier sont comprises entre 0 et 255 : plus le nombre est élevé et plus le pixel est sombre. Nous disposons de 42.000 images différentes pour cette analyse que nous réaliserons entièrement sous Python.

Pour la compétition kaggle à partir de laquelle nous travaillons pour ce dossier, 3 bases de données sont mises à disposition :

- **train** : la base d'apprentissage composée de 42.000 images et leurs pixels.
- **test** : la base de test composée de 28.000 observations où cette fois le chiffre représenté sur l'image n'est pas précisé. L'objectif de la compétition consiste à entraîner les meilleurs modèles possibles sur la base train, les appliquer au jeu de données 'test' puis envoyer les résultats obtenus (c'est à dire selon la coloration des 784 pixels, prédire à quel digit cela correspond).
- **sample_submission** : c'est le format auquel on doit rendre les résultats pour la compétition kaggle, avec d'une part les images et d'autre le digit prédit.

Étant donné que le but de ce dossier n'est non pas de participer à la compétition kaggle mais plutôt de nous entraîner au machine learning en appliquant les méthodes étudiées en cours, nous ne nous servirons que de la base d'entraînement contenant la colonne "label". Aussi, de manière à tester la performance de nos modèles nous procédons à un partitionnement de notre base en un jeu train et un jeu test avec de la fonction '*train_test_split*' de la librairie **sklearn**. Pour la construction de nos modèles nous travaillerons principalement avec cette libraire dont nous importerons les différentes méthodes de ML.

Nous appliquons donc la fonction de partitionnement sur notre base de 42.000 observations, en spécifiant les tailles que nous voulons pour les sous-bases train et test à partir desquelles nous réaliserons l'analyse. Pour réduire les temps de calcul et de manière à garder un large échantillon, nous décidons de faire apprendre les modèles sur 5000 observations et les tester sur 1000 observations des 42.000 initiales. Nous ajoutons alors les options $test\_size = 1/42$ et $train\_size = 5/42$ à notre fonction.

3.1 Statistiques descriptives

D'après la figure n°1, on voit que le digit 1 est celui le plus représenté dans la sous base que nous avons créée aléatoirement, avec 554 observations soit 11.1% de notre base - tandis que le digit le moins représenté est le numéro 5 contenant 460 observations, soit 9.2%. La répartition parfaite entre les digits voudrait que chacun d'eux contienne 10% des observations de la base, cependant notre répartition n'en étant que très peu éloignée nous considérons qu'il n'y a pas de biais et que notre échantillon est représentatif du jeu de données initial, et continuons donc notre analyse.

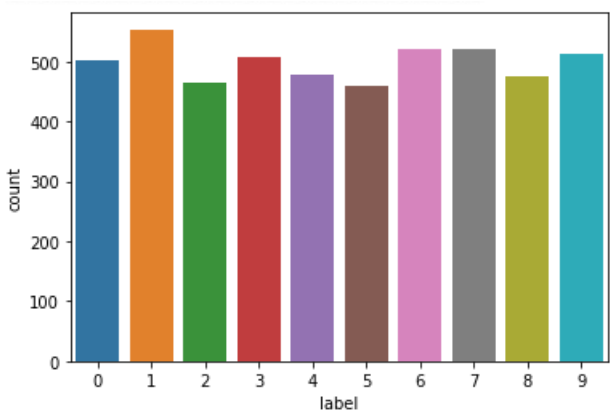


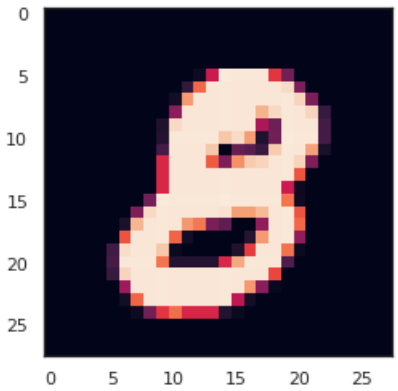
FIGURE 1 – Répartition des digits sur notre base de travail

3.2 Manipulations des données

Nous procédons dans un deuxième temps, à la normalisation de nos données. Cette étape est en effet essentielle au bon fonctionnement de nos futurs modèles, puisqu'elle permet d'une part de réduire leur complexité d'interprétations, et d'autre part de les appliquer (c'est un préalable à l'application de certains algorithmes tel que les SVM ou réseaux de neurones). La normalisation consiste à transformer les données pour que tous les prédicteurs soient à la même échelle.

La solution est de centrer, c'est à dire retirer la moyenne, et réduire, c'est à dire diviser par l'écart-type chaque observation. Ainsi, tous les prédicteurs seront normalement distribués avec une moyenne de 0 et un écart-type compris entre 0 et 1 en valeur absolue.

De plus, nous transformons la variable à expliquer en un vecteur composé de différentes catégories grâce à la fonction **to_categorical** pour que le logiciel ne considère pas Y comme une variable quantitative mais bien comme une variable catégorielle avec des images représentant des chiffres compris entre 0 et 9.



Tout comme le processus de normalisation, nous devons re-dimensionner les données de la base d'entraînement en trois dimensions afin de pouvoir les observer visuellement dans un graphique. L'image ci-dessus représente cette manipulation des données qui définit une hauteur ainsi qu'une largeur égale à 28 pixels et une intensité du trait comprise entre 0 et 1.

4 Application des méthodes à nos données

Une fois les données d'entraînement ciblées, nettoyées, normalisées et donc prêtent à utilisation, l'objectif est de sélectionner l'algorithme qui va le mieux prédire notre réponse. Dans notre cas l'algorithme devra être capable de prédire le bon chiffre entre 0 et 9. Dans la librairie **scikit-learn** il y un objet *estimator* qui contient le nom de l'algorithme, ses paramètres et hyperparamètres ainsi que les méthodes *fit()* et *predict()* qui permettent d'apprendre des données et prédire de nouvelles observations.

4.1 k plus proches voisins (kNN)

Dans le cadre de notre analyse, cette méthode considère une image dont on ne connaît pas le *label*, c'est à dire le chiffre représenté - l'algorithme cherchera à identifier les images qui ressemblent le plus à celle considérée que ce soit en termes de distance ou de caractéristiques de pixels communes. Ainsi, la méthode attribuera un chiffre entre 0 et 9 à cette image, selon le label majoritaire des plus proches voisins. Nous l'appliquerons de 3 manières différentes :

- **test in-sample** ; nous créerons un modèle sur l'ensemble de la base et nous l'appliquerons sur cette même base
- **test par cross-validation hors échantillon d'entraînement** ; nous testerons plusieurs découpages par validation croisée de manière à avoir une meilleure précision du taux d'erreur out-of-sample
- **test out-of-sample** ; nous ajusterons les paramètres du modèle sans passer par la cross-validation

a) Test in sample

Comme précisé précédemment, toutes les méthodes de machine learning que nous utiliserons dans notre analyse sous Python seront importées depuis la bibliothèque 'sklearn'. Pour les k plus proches voisins nous importons la fonction "KNeighborsClassifier" à laquelle nous imposons le nombre de plus proches voisins à considérer à 3, pour ne pas avoir de calculs trop complexes et pouvoir interpréter facilement les résultats. Dans cette partie "test in sample" nous entraînons le modèle sur la base 'train' et nous les testerons sur cette même base - ce qui peut être comparé à un élève qui calcule son niveau en faisant les exercices qu'il a déjà fait et regarde ensuite sa proportion de bonnes réponses. Le taux de bonne prédiction donné par la fonction "score" pour la méthode des plus proches voisins avec k=3 et évaluée sur l'échantillon de l'apprentissage, est de 96.64%. Ainsi sur 100 classifications d'observations sur lesquelles le modèle s'est déjà entraîné, près de 97 sont justes. On remarque donc que les prévisions ne sont pas parfaites par la méthode des kNN alors que le modèle s'est entraîné sur ces données.

b) Test par Cross-Validation

Bien que le taux de bonne prédiction soit bon dans le modèle précédent, il ne révèle pas forcément la qualité de la méthode puisque nous le rappelons, celui-ci était évalué sur la même base que celle sur laquelle il s'est construit. Il existe donc un biais et nous pouvons supposer que s'il était appliqué à un jeu de données différent du jeu train, alors le taux d'erreur augmenterait considérablement. Ainsi, pour rendre notre modèle plus robuste nous allons cette fois le construire par cross-validation, dans le but d'augmenter sa qualité et de

réduire son taux d'erreur hors échantillon d'entraînement. Nous testons donc plusieurs découpages différents par CV, en suivant le processus suivant (énoncé en partie théorique mais réexpliqué sur notre cas d'étude) :

- on divise notre base **train** de départ contenant 5000 observations en 10 sous-ensembles ("K-folds") composés ainsi de 500 observations par fold
- on en laisse un de côté et on crée une base d'entraînement à partir des 9 autres
- on ajuste le modèle sur cette base contenant 90% des observations soit 4500
- on le teste sur les 10% restants de la base initiale et on regarde le taux de prédiction
- on réitère ces étapes pour tous les sous-ensembles, de manière à ce que le modèle apprenne sur toutes les combinaisons possibles de sous-ensembles et améliore ainsi ses prédictions en étant plus entraîné

Le taux de prédiction qu'on obtient via cette méthode est simplement la moyenne des taux d'erreur sur chacune de combinaisons testées. Dans notre cas il s'élève à 8.76%, pour un taux de prédictions justes de 93.24%. Sans étonnements il est plus élevé que lors des prévisions in-sample puisqu'ici, le modèle se teste sur une base différente de son entraînement et a donc une plus grande probabilité de se tromper. Par ailleurs, quoique plus faible que dans la section précédente, ce taux est satisfaisant car il montre que sur 100 digits, moins de 9 sont mal reconnus par le modèle - en suivant le principe des plus proches voisins. Aussi le taux de bonnes prédictions est moins bon mais il est plus révélateur de ce que pourra nous fournir le modèle sur l'échantillon test.

c) Test out-of-sample

Dans le cas des prévisions out-of-sample, le modèle est ajusté sans passer par la validation croisée et donne ainsi un bon aperçu de ce que peut être le véritable taux d'erreur sur une base sur laquelle le modèle ne s'est pas entraîné. Il s'agit ici de la méthode la plus couramment utilisée en machine learning à savoir entraîner sur une partie de la base et tester sur une autre pour avoir un taux d'erreur 'transparent'. Dans ce cas, le taux de bien classé est de 92.4% ; comme dit en partie n°2, il s'agit du nombre d'observations sur la diagonale de la matrice de confusion qui représentent celles bien classées par le modèle - comme visible en figure x. Concrètement sur notre étude, cela signifie que lorsque la modélisation des 3 plus proches voisins s'entraîne sur 5000 digits avec leurs caractéristiques propres en termes de pixélisation, et qu'elle est testée sur 1000 autres digits, alors elle reconnaît correctement 924 d'entre eux. Autrement dit, à partir des contrastes des 784 pixels d'une image, seuls 76 chiffres sont mal classés par le modèle.

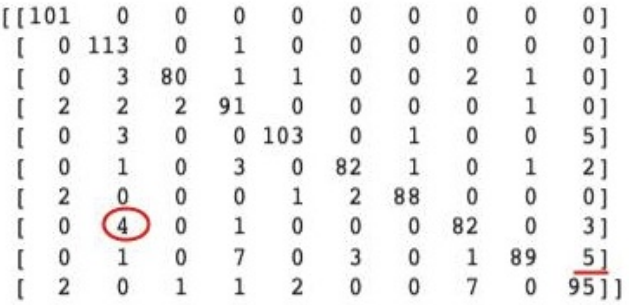


FIGURE 2 – Matrice de confusion des prévisions out-of-sample, méthode des kNN

Interprétons par exemple le chiffre 4 de la matrice, il signifie que 4 digits '7' ont été classés en digit '1' : cela est compréhensible puisque les chiffres lors de leur écriture se ressemblent beaucoup. Il en va de même pour 5 digits '8' qui ont été classés en digit '9' par la méthode des plus proches voisins. On remarque par ailleurs, que tous les digits du chiffre '0' ont été correctement classés par le modèle, si bien que l'on ne retrouve aucune valeur hors diagonale sur la première ligne de la matrice.

	Bien classés	**Mal classés**	**Total**
digit n°0	101 (100%)	0 (0%)	101 (10.1%)
digit n°1	113 (99.12%)	1 (0.82%)	114 (11.4%)
digit n°2	80 (90.91%)	8 (9.09%)	88 (8.8%)
digit n°3	91 (92.86%)	7 (7.14%)	98 (9.8%)
digit n°4	103 (91.96%)	9 (8.04%)	112 (11.2%)
digit n°5	82 (91.11%)	8 (8.89%)	90 (9%)
digit n°6	88 (94.62%)	5 (5.38%)	93 (9.3%)
digit n°7	82 (91.11%)	8 (8.89%)	90 (9%)
digit n°8	89 (83.96%)	17 (16.04%)	106 (10.6%)
digit n°9	95 (87.96%)	13 (12.04%)	108 (10.8%)
Total	924 (92.4%)	76 (7.6%)	1000 (100%)

TABLE 1 – Prédictions out-of-sample du modèle kNN

Comme visible en table n°1, en sommant les valeurs mal classées par digit (c'est à dire les valeurs par ligne en dehors de la valeur inscrite sur la diagonale), on se rend compte que c'est le digit n°8 qui est le plus souvent mal classé par le modèle, puisque pour lui, 17 chiffres sont mal classés soit 16% des digits n°8. Il apparaît aussi que le digit le mieux classé est celui '0' qui l'est à 100%, cela est compréhensible puisque sa forme étant simplement ronde il y a moins de chance de le confondre avec d'autres digits.

Nous avons pu observer la répartition des digits dans la base train tout à l'heure et avions constaté qu'elle était principalement bonne allant de 9.2% à 11.1%. Dans notre base test, le digit le moins présent est le numéro 2 présent à 8.8% tandis que le mieux représenté est le numéro 4 regroupant 11.2% des observations. Outre le digit 0 qui est parfaitement classé du fait de sa forme qui n'induit pas d'ambiguïté, c'est le digit n°1 qui a le taux de bien classé le plus élevé à savoir 99.12%. Nous pouvons rapprocher cette valeur au fait que c'est sur ce digit que le modèle s'est le mieux entraîné puisqu'il est le mieux représenté dans la base **train** (à hauteur de 11.1%).

4.2 Régressions logistiques multinomiales

Après avoir appliqué une première méthode de machine learning sur notre jeu de données qui s'est avérée pertinente au vu des taux d'erreurs constatés, nous allons appliquer une deuxième méthode étudiée théoriquement en partie n°2 ; il s'agit de la régression logistique multinomiale. Sur notre jeu de données, cette dernière a pour objectif de trouver des lignes (boundary decision) séparant les 10 groupes des images chiffrées de 0 à 9. Avant de commencer l'analyse, nous procédons à la standardisation de nos données de telle manière que l'algorithme converge vers des coefficients finis - à l'aide de la fonction '**preprocessing.scale()**'.

De même, c'est la fonction '**LogisticRegression**' de la librairie que nous utilisons depuis le début (*sklearn*), que nous utilisons pour les modélisations.

Tout comme pour les plus proches voisins, nous calculons 3 taux d'erreur différents pour 3 approches différentes, à savoir test in-sample, test par validation croisée et test out-of-sample. Ceux-ci sont disponibles dans la table suivante :

	taux de bien classé
Test in-sample	100%
Test par CV	88.88%
Test out-of-sample	87.8%

TABLE 2 – Taux d'erreur associés aux modélisations logistiques

C'est évidemment les prévisions réalisées par le modèle sur le même jeu qui a servi à l'apprentissage, c'est à dire '*in-sample*', qui ont le taux de bien classé le plus élevé à savoir 100%. L'apprentissage et les prévisions sont donc parfaits et elles font mieux que la méthode des plus proches voisins de 3.36%. Ainsi, sur 10 reconnaissances d'image, notre modèle de régression logistique les classe toutes correctement - les performances du modèle sont donc évidentes par rapport aux kNN, mais rappelons que les prévisions in-sample ne sont pas vraiment révélatrices du pouvoir de prédiction du modèle puisqu'il s'agit des données d'apprentissage. Voyons donc les taux de bien classé par méthode de CV puis des prédictions out-of-sample.

L'entraînement par cross-validation qui a pour objectif de mieux entraîner le modèle en le confrontant à des situations de classement différentes, fait en revanche une moins bonne prédiction qu'une modélisation simple avec un test sur les observations in sample. Les 10 taux d'erreur constatés pour les 10 découpages effectués par validation croisée sont les suivants : 0.884%, 0.878%, 0.912%, 0.874%, 0.894%, 0.874%, 0.878%, 0.9%, 0.906% et 0.888%. Ainsi le taux moyen est de 11.12% soit 2.36% de plus que pour la même situation en kNN. Sachant que le test s'effectue sur les données train, cela signifie qu'en moyenne sur les 10 estimations effectuées par validation croisée, 56 digits sur 500 (1/10 de 5000 observations dans la base train) sont mal reconnus et donc mal classés par la régression logistique.

Enfin, la dernière situation que nous testons est de faire les prévisions sur les données 'test' qui sont différentes des données d'apprentissage. On s'attend ainsi à avoir un taux d'erreur plus élevé étant donné qu'il s'agit de données inconnues au modèle. Le taux de bien classé est effectivement de 87.8% : ainsi sur 1000 observations '*test*' 878 digits sont reconnus correctement par le modèle logistique. Parmi les 122 restants mal prédits, 24 concernent le digit '8', le constat est donc le même que pour la méthode des k plus proches voisins. De même, le digit le mieux prédit est le '0' ici encore. Pour les digits '4' et '5' par exemple, les taux d'erreurs sont de respectivement 10.71% et 16.67% (12/112 pour le digit n°4 et 15/90 pour le digit n°5).

Ainsi la régression logistique multinomiale est moins efficace que la méthode des 3 plus proches voisins sur nos données dans le but de reconnaître les digits grâce aux couleurs des pixels, pour les tests par CV et out-of-sample. En revanche, nous avons pu constater que la prédiction sur les données connues (in-sample forecasting) était parfaite et donc meilleure que celle des kNN. Ces méthodes sont caractérisées précisément par leur linéarité dans leurs décisions. Voyons à présent les machines à vecteur de support qui elles, peuvent être à la fois linéaires et non linéaires et offrent ainsi une nouvelle clef de réussite du modèle pour classer les digits.

4.3 Machine à Support Vectorielle (SVM)

a) SVM linéaire

Pour cette partie qui vise l'application des SVM linéaires sur notre jeu de données, nous utilisons la fonction **SVC** de la bibliothèque 'sklearn'. La commande "*model_linear = SVC(kernel='linear')*" nous permet d'obtenir un SVM linéaire. Dans notre cas, le SVM vise à séparer les observations selon leur correspondance à un digit. Nous pouvons aisément deviner que le problème est difficilement séparable et qu'un SVM non-linéaire sera nécessaire pour améliorer les prévisions.

Nous suivons toujours le même fonctionnement, nous testons les performances du modèle sur les données de l'échantillon avec et sans cross-validation, et sur les données hors de l'échantillon d'entraînement. Nous obtenons les taux de bien classé suivants, desquels nous déduisons les taux d'erreur (taux d'erreur=100-taux de bien classé) :

	taux de bien classé
Test in-sample	100%
Test par CV	91.08%
Test out-of-sample	90.5%

TABLE 3 – Taux d'erreur associés aux SVM linéaires

Au regard de la table 3, on constate d'emblée que la prédiction est parfaite pour la méthode de machine à support vectorielle qui est testée sur sa base d'apprentissage, tout comme les prédictions issues des régressions logistiques. La méthode des SVM devance donc les k plus voisins en termes de taux prédictif in-sample. Concernant les modélisations par CV, on voit que le taux d'erreur est cette fois de 8.92% ce qui reste convenable sachant que la classification se fait pour l'instant uniquement sur une base linéaire. Il semble donc que les observations soient réparties de manière relativement linéaire puisque depuis le début de l'analyse, le taux maximum d'erreur de prédictions avec des méthodes linéaires que nous ayons trouvé est de 12.2% qui correspond au test out-of-sample de la régression logistique multinomiale. Enfin, on voit que la "vraie" efficacité du modèle est de 90.5% - 'vraie' puisque nous testons alors le modèle sur des données lui étant inconnues, et le taux d'erreur révèle alors vraiment la qualité de prévision du modèle SVM linéaire. Sur des données nouvelles, le modèle reconnaît correctement 9 digits sur 10.

Ainsi, malgré sa limite aux séparations linéaires entre les digits, ce premier modèle SVM répond correctement à notre problématique de reconnaissance d'images puisqu'il classe au mieux 100% des digits sur les données connues, et au pire 90.5% sur les données inconnues. Cependant, des situations comme celle expliquée en partie n°2 où on voyait qu'il était impossible de séparer linéairement des points du groupe 1 entourés par les points du groupe B, peuvent arriver et nous décidons pour pallier à cela de construire des modèles SVM non linéaires. En agrandissant l'espace des fonctionnalités, nous serons à même de séparer des groupes d'observations n'appartenant pas au même digit mais qui se trouvent proches dans la représentation graphique - et ainsi améliorer, nous l'espérons, la qualité prévisionnelle du modèle.

b) SVM non-linéaire (c, gamma)

Dans cette section nous utilisons donc l'astuce du noyau, évoquée en partie théorique du modèle partie n°2. Cette astuce permet de trouver une frontière de décision non linéaire et est ainsi plus représentative des situations que l'on trouve généralement avec les données. De même, nous utiliserons la validation croisée dans le but d'optimiser les paramètres du modèle. Nous fixons ainsi C qui représente le compromis entre la classification correcte des exemples de formation et la maximisation de la marge de la fonction de décision, à 3 et 5. Le paramètre γ quant à lui, définit l'influence d'un seul exemple d'entraînement ; plus il est grand plus l'influence est proche. Nous le fixons à 0.000001 et 0.0000001 pour que les influences de chaque vecteur de support soient grandes. 4 combinaisons de ces paramètres C et gamma seront testés par cross validation puisque nous définissons pour chacun d'entre eux 2 possibilités dans le but de voir quelle valeur optimise le modèle SVM non linéaire.

Nous utilisons la fonction *model=SVC(kernel="rbf")* où RBF signifie "Radial Basis Function" qui correspond à la fonction de noyau appliquée. On spécifie ensuite un tableau de paramètres 'GridSearchCV' qui a pour fonction de trouver pour chaque combinaison de C et Gamma, le meilleur modèle en termes de qualité moyenne de prédiction. Après une recherche en grille comme évoqué précédemment, il apparaît que le meilleur modèle sélectionné par cette méthode soit celui aux paramètres C=3 et Gamma=1e-07 pour lequel le taux de bien classé est de 94.46% par CV ce qui est considérable par rapport à toutes les autres méthodes testées jusqu'à présent. On comprend facilement cela puisqu'il est en réalité très difficile de séparer linéairement les chiffres les uns des autres. De plus nous remarquons par la sortie python que le temps de modélisation lorsque γ=1e-07 est de 28 minutes environ tandis qu'il est de 5 minutes lorsque celui-ci est égal à 1e-06. Ainsi, plus l'influence d'un exemple d'entraînement est lointaine et plus la modélisation prend du temps à se faire. Ces sorties nous ont été données grâce aux fonctions *model_cv.best_score_* et *model_cv.best_params_* sous Python, ainsi que *pd.DataFrame(model_cv.cv_results_)* pour les temps d'estimation.

Maintenant que nous avons trouvé grâce à la grille de validation croisée 'GridSearch', les hyperparamètres optimisant la qualité du modèle en minimisant son taux d'erreur, il peut être intéressant de l'appliquer sur notre jeu 'test' et donc voir la performance d'un tel modèle sur des données qui lui sont inconnues.

```
[[101   0   0   0   0   0   0   0   0   0]
 [  0 112   0   1   0   1   0   0   0   0]
 [  0   1  84   0   1   0   0   1   1   0]
 [  0   0   3  92   0   1   1   0   1   0]
 [  0   1   0   0 106   0   2   0   0   3]
 [  2   0   0   3   0  80   0   0   5   0]
 [  2   0   0   0   1   2  88   0   0   0]
 [  0   2   1   1   1   0   0  84   0   1]
 [  0   1   0   0   1   3   0   0  99   2]
 [  2   1   0   1   2   0   0   3   0  99]]
```

FIGURE 3 – Matrice de confusion du modèle avec les paramètres trouvés par Gridsearch CV

La performance du modèle SVM non-linéaire est donc bonne puisque son taux de prédictions bonnes dépasse celui des autres modèles ; il est de 94.5%. On voit ainsi qu'il est quasiment égal au taux de bien classé sur des prédictions in-sample réalisées par validation croisée précédemment. La robustesse de ce modèle est telle qu'il est capable de prédire correctement près de 95 digits sur 100. Nous pouvons, de la même manière, regarder le tableau n°4 qui résume l'information liée aux taux d'erreur constatés sur le modèle non-linéaire dont nous avons au préalable tuné les paramètres par CV. Le chiffre entouré sur la matrice de confusion signifie que le modèle a classé 5 digits en n°8 alors qu'il s'agissait en réalité du digit n°5, la ressemblance entre ces 2 chiffres n'est pas nulle donc nous comprenons que ce soit la fréquence de mal-classés la plus présente sur ces prévisions.

	Bien classés	**Mal classés**	**Total**
digit n°0	101 (100%)	0 (0%)	101 (10.1%)
digit n°1	112 (98.25%)	2 (1.75%)	114 (11.4%)
digit n°2	84 (95.45%)	4 (4.55%)	88 (8.8%)
digit n°3	92 (93.88%)	6 (6.12%)	98 (9.8%)
digit n°4	106 (94.64%)	6 (5.36%)	112 (11.2%)
digit n°5	80 (88.88%)	10 (11.12%)	90 (9%)
digit n°6	88 (94.62%)	5 (5.38%)	93 (9.3%)
digit n°7	84 (93.33%)	6 (6.67%)	90 (9%)
digit n°8	99 (93.40%)	7 (6.60%)	106 (10.6%)
digit n°9	99 (91.67%)	9 (8.33%)	108 (10.8%)
Total	945 (94.5%)	55 (5.5%)	1000 (100%)

TABLE 4 – Prédictions out-of-sample du modèle SVM non linéaire

Sur la table n°4, on observe qu'en moyenne le modèle reconnaît mal 55 digits sur 1000 soit 21 de moins que pour la méthode des K plus proches voisins présentée antérieurement. De même, comparé aux kNN le modèle SVM non linéaire prédit mieux les digits 3, 4, 7, 8 et 9 (cf. taux d'erreur plus faibles pour ces chiffres). On note aussi que cette fois le digit le moins bien reconnut grâce aux caractéristiques des pixels qui le composent, est le numéro 5 pour lequel le taux d'erreur est de 11.12%. En revanche, le digit '0' comporte toujours une classification parfaite et donc un taux de mal classés nul. Le digit n°1 est lui très bien reconnu par le modèle puisque son taux de bonnes prédictions est de 98.25% : on constate par ailleurs que le modèle le confond seulement 2 fois avec le digit '7' ce qui est plus satisfaisant que la méthode kNN.

Nous avons ainsi constaté que le modèle SVM non-linéaire permettait de mieux répondre à notre problème puisque ses prédictions étaient meilleures que tous les autres modèles, y compris le SVM linéaire : ainsi les taux d'erreurs étaient respectivement de 5.55% et de 5.5%. L'approche non linéaire est donc assez intéressante et elle devient plus pertinente encore lorsqu'elle est couplée au tuning qui vise à identifier les hyperparamètres optimaux qui minimisent le taux de digits mal classés.

4.4 Arbre de décision (Decision Trees)

Dans cette section nous utilisons la fonction DecisionTreeClassifier de la bibliothèque 'sklearn.Tree' dans le but de réaliser un arbre de décision capable de classer les chiffres de 0 à 9. Comme pour les algorithmes précédents il est nécessaire qu'il s'entraîne sur un échantillon 'train' avant d'être validé par cross validation, puis nous l'appliquerons sur un échantillon 'test' afin de juger de son aspect généralisable.

Nous avons fait le choix de construire un arbre de décision en conservant les paramètres par défaut pour sa construction. Le taux de prédiction sur l'échantillon d'entraînement est de 100% ce qui signifie qu'il a appris à prédire parfaitement les chiffres présents dans la base d'entraînement. Cela n'est pas surprenant car les arbres de décisions sont justement réputés pour s'ajuster parfaitement aux données d'entraînements. La conséquence de cela est le sur-ajustement du modèle, *"overfitting"* en anglais, qui réduit la capacité prédictive lorsque le modèle est appliqué à de nouvelles données. En effet, lorsque le modèle est construit par cross-validation ou lorsqu'il est testé sur l'échantillon 'test', le taux de bonne prédiction chute de 22% en moyenne - ce qui correspond au pire résultat depuis le début de l'analyse. Le Tableau ci-dessous résume ces pourcentages de bonnes prédictions à partir des différents échantillons.

	taux de bien classé
Test in-sample	100%
Test par CV	77.8%
Test out-of-sample	77.1%

TABLE 5 – Taux d'erreur associés à l'arbre de décision

Il est intéressant de comprendre quels ont été les chiffres pour lesquels l'algorithme a eu des difficultés à prédire : nous regardons pour cela la matrice de confusion. Comme nous pouvons le constater ci-dessous, l'arbre n'a eu aucun mal à prédire les chiffres '0' qui sont très distincts des autres, ou les '2', mais en ce qui concerne les digits '4' et '8', les erreurs sont bien plus élevées. L'arbre de décision n'a réussi à prédire que les 3/4 des chiffres '8'.

```
[ 91   1   1   2   1   1   1   0   2   1]
[  0 102   2   4   1   3   0   1   1   0]
[  1   3  59   6   4   2   1   9   3   0]
[  0   2   7  76   1   5   2   0   2   3]
[  1   1   2   2  88   4   0   3   1  10]
[  5   3   3   5   1  60   2   2   6   3]
[  0   2   1   0   5  11  71   1   1   1]
[  1   3   4   1   2   0   0  75   2   2]
[  1   2   3  11   6   3   2   2  67   9]
[  1   0   1   5   7   0   0  11   1  82]]
```

FIGURE 4 – Matrice de confusion des prévisions out-of-sample, méthode arbre de décision

Ces résultats peu convaincants des arbres de décisions peuvent être dus à différents facteurs tels que la profondeur de l'arbre ou la taille des noeuds avant séparation, qui correspondent à des hyperparamètres ajustables à chaque échantillon et à chaque problème donné. Nous ne sommes pas allés plus loin dans l'ajustement des hyperparamètres, car les arbres de décisions, tout comme les forêts aléatoires, ne font pas partie des algorithmes de ML mis en avant dans cette étude.

4.5 Forêt Aléatoire (Random Forest)

Comme expliqué précédemment, une forêt aléatoire est formée par un ensemble d'arbres de décision individuels où chaque arbre s'est entraîné avec un échantillon légèrement différent, grâce à la méthode de boostrapping. La prédiction d'une nouvelle observation s'obtient en faisant la somme des prédictions des arbres individuels qui forment le modèle. On peut donc s'attendre à de meilleures prédictions que l'arbre de décision de la section précédente.

De la même façon que les autres algorithmes, il est possible d'améliorer les prédictions en ajustant les hyperparamètres tels que le nombre d'arbres composant la forêt, le nombre de variables tirées aléatoirement pour la construction de l'arbre, ou encore la profondeur et la complexité des arbres. Dans notre cas, de même manière que pour le modèle SVM non linéaire, nous utilisons la fonction *Grid Search* de **sklearn** afin de tester différentes combinaisons de paramètres. Les paramètres retenus sont : une profondeur d'arbre max_depth = 30 et un nombre d'arbres n_estimators = 800.

	taux de bien classé
Test in-sample	100%
Test par CV	93.6%
Test out-of-sample	92.5%

TABLE 6 – Taux d'erreur associés aux forêts aléatoires

On peut constater à partir du tableau ci-dessus que les taux d'erreurs associés aux forêts aléatoires sont nettement meilleurs que ceux de l'arbre de décision. Les résultats sont cependant légèrement inférieurs aux prédictions réalisées à partir des SVM non linéaires avec un taux de 92.5% de bien classés pour la forêt aléatoire contre 94.5% pour les SVM non linéaires. À partir de la matrice de confusion présente ci-dessous, nous pouvons observer où la forêt aléatoire a mal prédit et quelles ont été ses confusions.

```
[[ 99   0   0   0   0   0   0   0   2   0]
 [  0 112   0   2   0   0   0   0   0   0]
 [  0   3  79   0   1   0   1   3   1   0]
 [  0   0   3  88   0   3   1   0   3   0]
 [  0   0   1   0 106   0   2   0   0   3]
 [  1   1   0   1   1  81   0   0   4   1]
 [  1   0   0   0   1   2  88   0   1   0]
 [  1   1   2   0   1   0   0  83   0   2]
 [  0   1   0   1   2   3   0   0  94   5]
 [  2   0   1   1   3   0   0   6   0  95]]
```

FIGURE 5 – Matrice de confusion des prévisions out-of-sample, méthode des forêts aléatoires

Nous voyons ainsi que la forêt aléatoire a eu du mal à différencié le chiffre '8' du '9'. En effet, 6 chiffres ont été prédits comme '9' alors que c'était des '8', et inversement : 5 chiffres ont été prédits comme '8' alors qu'il s'agissait en réalité de digits n°9. Cette confusion peut s'expliquer par la ressemblance dans l'écriture de ces deux chiffres composés de beaucoup de courbes, le '9' se différenciant seulement par sa boucle inférieure ouverte.

	Bien classés	**Mal classés**	**Total**
digit n°0	99 (98.02%)	2 (1.98%)	101 (10.1%)
digit n°1	112 (98.25%)	2 (1.75%)	114 (11.4%)
digit n°2	79 (89.77%)	9 (10.23%)	88 (8.8%)
digit n°3	88 (89.80%)	10 (10.20%)	98 (9.8%)
digit n°4	106 (94.64%)	6 (5.36%)	112 (11.2%)
digit n°5	81 (90.00%)	9 (10.00%)	90 (9%)
digit n°6	88 (94.62%)	5 (5.38%)	93 (9.3%)
digit n°7	83 (92.22%)	7 (7.78%)	90 (9%)
digit n°8	94 (88.68%)	12 (11.32%)	106 (10.6%)
digit n°9	95 (87.96%)	13 (12.04%)	108 (10.8%)
Total	925 (92.5%)	75 (7.5%)	1000 (100%)

TABLE 7 – Prédictions out-of-sample du modèle Random Forest

La table n°7 résume les deux tables précédentes en une seule avec les effectifs ainsi que les pourcentages. On peut constater que sur 1000 images, l'algorithme s'est trompé 75 fois. Ce résultat correspond parfaitement à ce à quoi nous nous attendions dans le sens où il s'ajuste mieux que l'arbre de décision unique, mais est tout de même moins bon que les classifieurs de type SVM non linéaires. Les chiffres ayant le pourcentage de mal classés le plus important sont les '8' et '9' comme constaté avec la matrice de confusion. On observe ici que les digits n°2, 3 et 5 ont aussi un pourcentage élevé de mal classés avec des taux d'erreur supérieurs à 10%.

De manière générale, les forêts aléatoires fournissent un algorithme prêt à l'emploi et qui a souvent une grande précision prédictive, c'est pour toutes ces raisons que nous souhaitions utiliser cet algorithme dans notre étude. Cependant, au vu des résultats, ce ne sera pas le modèle que l'on conservera comme modèle final, car il prédit moins bien que d'autres algorithmes pour la classification de digits.

4.6 Réseaux de Neurones (Neural Networks)

Dans cette partie, nous allons nous intéresser à deux librairies qui nous permettent de réaliser des réseaux de neurones. Dans un premier temps nous réaliserons un réseau avec la librairie **Scikit-Learn** en optimisant nos hyper-paramètres avec la fonction *Grid Search*. Dans une seconde partie, nous découvrirons la librairie **Keras** qui permet, en plus de contrôler les hyper-paramètres, de choisir précisément les caractéristiques des différentes couches.

4.6.1 Avec la library Sklearn

La librairie Scikit-learn nous permet de gérer une liste de différents paramètres, voici un descriptif de ceux que nous avons cherché à optimiser pour notre étude :

- "**hidden layer sizes**" : Un tuple représentant le nombre de neurones dans la n-ième couche. Par exemple, pour créer 2 couches cachées de 16 et 32 neurones, on utilise comme argument : $hidden\_layer\_sizes = (16, 32)$.
- "**activation**" : La fonction d'activation, qui sera la même pour tous les neurones au sein des couches.
- "**solver**" : L'algorithme utilisé afin de minimiser la fonction de perte en sortie. Dans notre cas, nous utiliserons la descente de gradient stochastique de type "adam". La descente de gradient de type stochastique mélange de manière aléatoire les variables avant les itérations afin d'avoir un bruit stochastique lors de l'estimation des poids, il ne sera donc pas induit en erreur lors de l'estimation des coefficients et il trouvera plus rapidement les poids de chaque branche entre les neurones avec un petit nombre de passages.L'apprentissage peut être beaucoup plus rapide avec une descente stochastique de gradient pour de très grands ensembles de données d'entraînement, et souvent il suffit d'un petit nombre de passages dans l'ensemble de données pour atteindre un bon ou un assez bon ensemble de coefficients, par exemple 1 à 10 passages dans l'ensemble de données.
- "**batch size**" : La taille des échantillons utilisés lors de la descente de gradient afin de minimiser la fonction de perte.
- "**tol**" et "**n iter no change**" : Ceci représente la tolérance de notre modèle, si la variation de perte est inférieure à "tol" lors de "n iter no change" itérations successives, alors le solver s'arrête.
- "**max iter**" indique le nombre maximum d'itérations du "solver" si jamais la perte est supérieure à la perte limite.

Nous allons donc rentrer des valeurs pour ces différents paramètres et à l'aide de la fonction *Grid search*, trouver la combinaison optimale de ces paramètres (c'est à dire la combinaison qui maximise le taux de prédiction de notre modèle). La meilleure qualité de prévision correspond aux hyper-paramètres : activation = "logistic", hidden layer sizes=(200,100) , random state=0, batch size = 100 , max iter = 200, solver = 'adam' , tol = 0.001. Cette combinaison d'hyper-paramètres donne un taux de bonnes prédictions sur le jeu de données test de 0.93 quand il est de 91.8% sur le jeu de données train.

Nous allons maintenant nous intéresser à la matrice de confusion :

```
[[113   0   0   1   0   0   5   0   0   0]
 [  0 151   2   0   0   1   0   0   1   0]
 [  0   0 151   0   1   0   1   1   2   0]
 [  1   1   2 125   0   6   0   0   4   1]
 [  0   0   0   0 112   0   0   0   0   3]
 [  1   2   0   5   0 101   3   0   2   3]
 [  0   0   1   0   0   2 131   0   1   0]
 [  0   2   3   0   2   0   0 130   0   3]
 [  1   3   5   1   0   1   2   1 110   3]
 [  0   0   1   1   4   0   0   5   0 118]]
```

FIGURE 6 – Matrice de confusion des prévisions OOS, méthode de RNN avec Scikit learn

On voit que le Perceptron multi-couches a du mal à classer certains numéros, cependant, ces numéros mal classés semblent également répartis parmi toutes les catégories (chiffres de 1 à 10).

4.6.2 Avec la library Keras

Les hyper-paramètres que nous pouvons contrôler sur notre modèle sont très nombreux, nous modifierons les suivants et utiliserons les paramètres par défaut pour les autres hyper-paramètres :

- **Optimizer** : Il s'agit là aussi de l'algorithme utilisé afin de minimiser la fonction de perte en sortie. Nous utiliserons la descente de gradient stochastique de type "adam".
- **Loss** : Ici, on décide de la fonction de perte à utiliser lors des différentes itérations de notre modèle, nous utiliserons pour notre variable à expliquer *CatégoricalCrossentropy* : une fonction qui calcule la perte d'entropie croisée entre les prédictions et les valeurs réelles.
- **Metrics** : Une liste de mesure qui ressort pour évaluer la qualité du modèle lors des différentes itérations de modification des poids. Nous utiliserons ici "*accuracy*" qui est le taux de prédiction du modèle sur l'échantillon d'entraînement.

a) Une couche simple

Avec la librairie **Keras** nous construirons 3 modèles avec différents type et nombre de couches, nous commençons par créer un modèle simple avec une seule couche cachée. Cette couche sera de type "*Dense*", c'est à dire que tous les neurones en amont et en aval de la couche sont reliés entre eux. Nous allons utiliser une couche très simple composée de 10 neurones dont la fonction d'activation de la couche sera de type "**softmax**".

Voici les résultats de la fonction de perte et du taux de prédiction calculé sur notre échantillon d'entraînement et sur notre jeu de donnée test (validation).

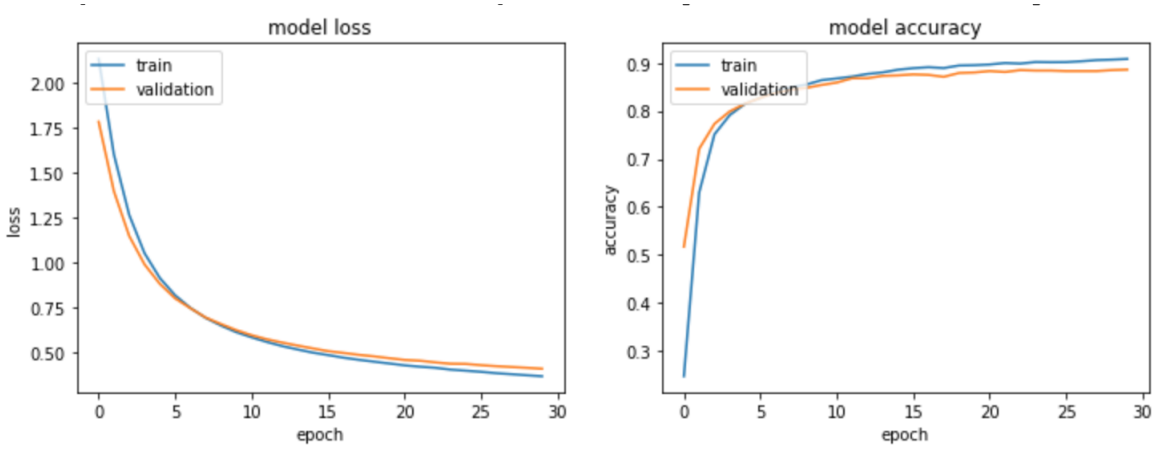


FIGURE 7 – Perte et taux de prédiction lors de chaque "epoch" pour le modèle "1 couche"

On peut observer que les fonctions de perte et de taux de prédictions sont très proches sur l'échantillon d'entraînement et sur l'échantillon de test. Ceci nous montre que ce modèle simple ne souffre pas de problème de sur-apprentissage, il ne prédit pas mieux les données d'entraînement que les données test. Le taux de bien classé sur l'échantillon d'entraînement est 90% tandis qu'il est de 86% sur l'échantillon test. La matrice de confusion de l'échantillon test est la suivante :

```
Accuracy : 0.886
[[ 96   0   0   0   0   0   1   0   0   1]
 [  0  93   1   0   0   0   0   0   2   0]
 [  1   1  96   3   1   1   0   4   2   2]
 [  1   1   4  83   0   8   2   1   2   1]
 [  1   1   0   0  89   1   1   0   2   4]
 [  1   4   0   4   0  55   3   0   4   4]
 [  0   0   1   0   3   2  90   0   2   0]
 [  1   0   2   0   1   0   0 107   0   6]
 [  0   3   2   7   0   0   0   0  81   3]
 [  0   0   1   1   4   1   0   2   2  96]]
```

FIGURE 8 – Matrice de confusion des prévisions OOS, méthode RNN avec Keras 1 couche simple

On remarque que le modèle de réseaux de neurones avec la librairie Keras et composé d'une seule couche cachée simple prédit le moins bien le digit '5' pour lequel on retrouve 20 digits mal classés. Le modèle a aussi du mal à classer les digits 8 et 9, ceux-ci sont répartis dans les autres classes de manière aléatoire. Les performances de ce modèle sont donc moins bonnes que celles du SVM non linéaires par exemple (avec un taux de bien prédits de 86% sur l'échantillon test pour celui-ci, et de 94.5% pour le SVM non linéaires).

b) Trois couches cachées simples

Nous allons maintenant réaliser un modèle avec 3 couches cachées simples (de type Dense, avec fonction d'activation de type "relu", respectivement de 100, 200 et 100 neurones) ainsi que notre couche finale déjà présente dans le modèle précédent, de manière à prédire plus efficacement nos digits. Nous retrouvons ci-dessous les résultats de la fonction de perte et du taux de prédiction calculés sur notre échantillon d'entraînement et sur notre jeu de données test (validation) pour cette méthode :

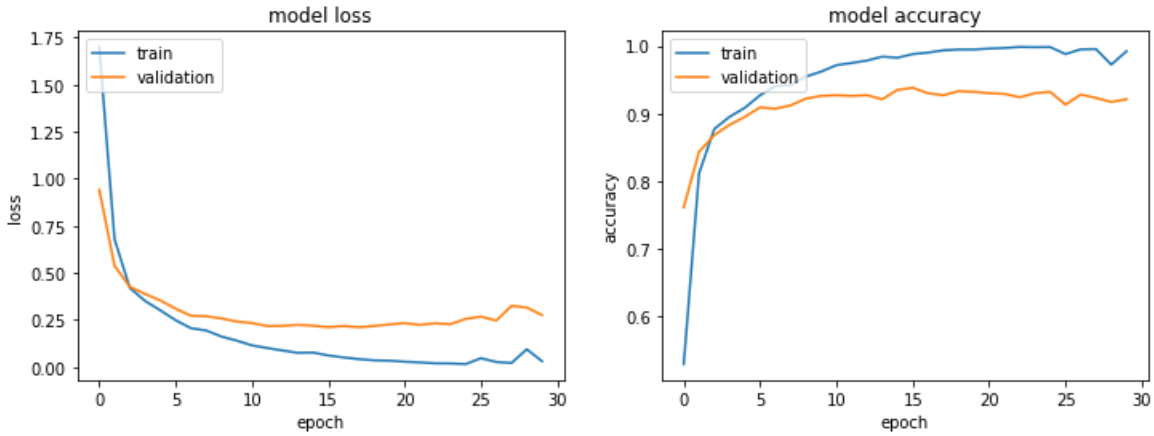


FIGURE 9 – Perte et taux de prédiction lors de chaque "epoch" pour le modèle "4 couches"

On remarque que la fonction de perte et de taux de prédiction diffèrent entre notre échantillon test et notre jeu d'apprentissage, au fur et à mesure que le nombre d'itérations d'ajustement augmente. Ceci est dû au sur-apprentissage de notre modèle ; les paramètres sont trop importants et compliqués et s'ajustent trop au jeu de données d'entraînement. Au final, le taux de prédiction sur notre échantillon d'entraînement est de 99.2% alors que ce taux n'est que de 93.5% sur notre jeu de donnée test. Voici la matrice de confusion de notre jeu de données test :

```
0.921
[[ 97   0   0   0   0   0   0   0   1   0]
 [  0  95   1   0   0   0   0   0   0   0]
 [  1   1 100   2   1   2   0   3   1   0]
 [  1   0   1  87   0  11   0   0   2   1]
 [  0   0   0   0  96   2   1   0   0   0]
 [  1   2   1   0   0  69   1   0   1   0]
 [  1   0   0   0   2   2  92   0   1   0]
 [  1   0   3   0   1   0   0 112   0   0]
 [  0   3   4   2   0   1   0   1  83   2]
 [  0   0   0   1   4   2   0   5   5  90]]
```

FIGURE 10 – Matrice de confusion des prévisions OOS, méthode RNN avec Keras 4 couches

On remarque ici que le modèle a mal classé onze chiffres 6 dans la catégorie 4, ceci est un des symptômes du sur-apprentissage : il a identifié un mauvais pattern sur l'échantillon d'entraînement qui avait peut-être quelques 6 qui ressemblaient à des 4 et donc sur l'échantillon de test il confond des 6 avec des 4.

c) Une couche convulotionnelle

Nous procédons maintenant à la réalisation d'un modèle avec des couches de convolutions. Les couches de convolutions permettent de prendre en compte la dimension spatiale de l'image.

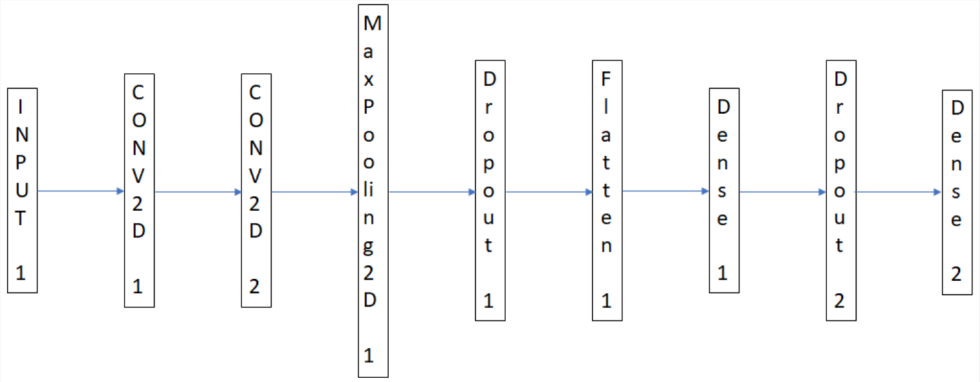


FIGURE 11 – Schéma des différentes couches présentes dans notre modèle

Voici la description des différentes couches :

- La **première couche** avec 32 filtres pour un maillage de taille 3*3 pixels, une fonction d'activation de type "relu".
- Une **seconde couche** avec 64 filtres pour un maillage toujours de taille 3*3 pixels, une fonction d'activation de type "relu".
- Une **couche Max pooling** afin de limiter le sur-apprentissage, avec un maillage 2*2.
- Une **couche dropout** qui mettre à zéro 25% des neurones de la couche précédente lors des itérations afin de limiter le sur apprentissage.
- Une **couche de type flattent** pour permettre de passer à une couche basique à une dimension.
- Une **couche dense classique** avec 128 neurones ayant une fonction d'activation de type "relu".
- Une **nouvelle couche dropout** où cette fois 50% des poids sont mis à zéro.
- Une **couche dense** avec 10 neurones de sortie qui correspondront à nos catégories (chiffres) finales.

Nous retrouvons sur la figure 12 ci-après les résultats de la fonction de perte et du taux de prédiction calculé sur notre échantillon d'entraînement et sur notre jeu de donnée test (validation) :

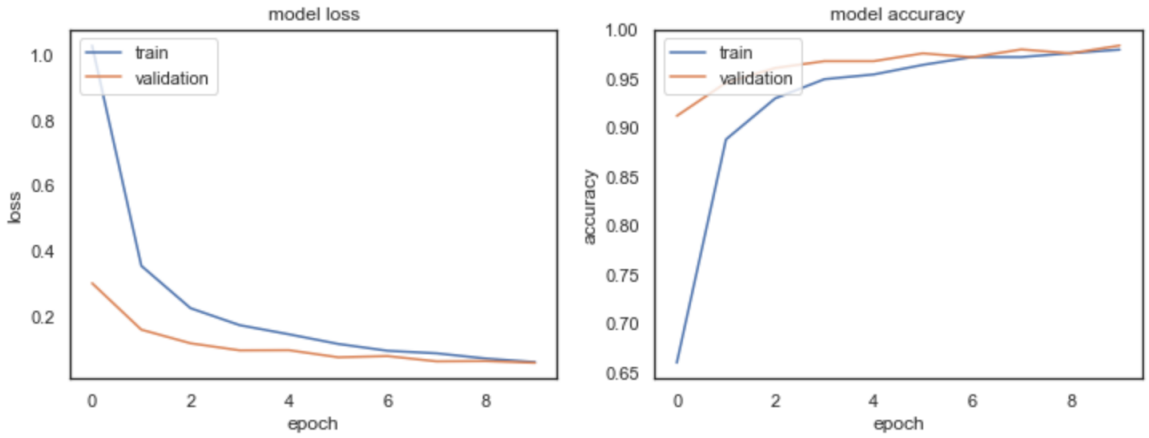


FIGURE 12 – Perte et taux de prédiction lors de chaque "epoch" pour le modèle à couche convolutionnelle

Cette fois ci le sur-apprentissage est très peu présent, le taux de prédiction sur l'échantillon d'apprentissage (98%) est le même que sur l'échantillon d'entraînement (98%). Cela est du aux deux couches *drop out* et à la couche de *max pooling*. Voici la matrice de confusion de notre jeu de données test.

```
[[110   0   0   0   0   0   0   0   1   1]
 [  0 112   2   0   0   0   0   0   0   0]
 [  0   0 102   0   0   0   0   0   0   0]
 [  1   0   0  87   0   0   0   0   1   0]
 [  0   0   0   0  81   0   0   0   0   1]
 [  0   0   0   1   0  90   0   0   0   0]
 [  0   0   0   0   1   0 101   0   0   0]
 [  0   0   0   0   0   0   0 110   0   0]
 [  0   0   0   0   0   1   1   0  86   1]
 [  0   1   0   0   2   0   0   1   0 105]]
```

FIGURE 13 – Matrice de confusion des prévisions OOS, méthode de RNN avec couches à convolution

On remarque dans la matrice de confusion que le modèle prédit avec difficultés les digits 8 et 9, peut-être y en avait-il de mal écrits dans l'échantillon d'entraînement.

Nous avons dans cette partie réalisé différents modèles de réseaux de neurones avec 2 librairies différentes : Sklearn et Keras. Pour cette deuxième nous avons construit 3 modèles ; le premier avec 1 couche simple, le deuxième avec 4 couches et enfin le dernier avec une couche convolutionnelle qui s'est avéré être le meilleur des 4 modèles en termes de qualité prédictive.

4.6.3 Temps de calcul, qualité de prévision, visualisation des erreurs

Ci-dessous sont répertoriés les temps d'ajustement en secondes sur l'échantillon d'entraînement de 5000 observations ("train tps"), le temps pour prédire les classes des 1000 observations de l'échantillon test (test tps) ainsi que la qualité de prédiction sur l'échantillon test (score test).

	Arbre décisionel	Logreg	SVM linéaire	MPL	KNN	Random Forest	SVM non linéaire
train_tps	0.873724	1.250517	4.637741	24.190109	0.388883	4.876571	6.241124
test_tps	0.008960	0.002226	1.479518	0.023884	5.646025	0.077394	1.794981
score_test	0.776000	0.878000	0.909000	0.912000	0.938000	0.944000	0.951000

FIGURE 14 – Temps de calcul et qualité d'ajustement hors échantillon pour les modèles réalisés avec Sklearn

	ANN_0	ANN_3	CNN_1
train_tps	3.507606	7.682284	119.132746
test_tps	0.038226	0.067973	0.824011
score_test	0.916000	0.959000	0.983000

FIGURE 15 – Temps de calcul et qualité d'ajustement hors échantillon pour les modèles réalisés avec Keras

On remarque que le temps d'ajustement n'est pas toujours proportionnel à la qualité de prévision hors échantillon, par exemple, le modèle SVM non linéaire prévoit avec une qualité de 95% en seulement 8 secondes sur sklearn. De plus on peut remarquer la différence entre la librairie **keras** et la librairie **sklearn** pour les réseaux de neurones, le modèle MLP a une qualité de prévision de 91% pour 24 secondes de calcul lors de l'ajustement alors que le modèle sans couche cachée de keras (CNN 1) prédit avec la même qualité pour 3,5 secondes.

Le modèle de type CNN reste lui bien meilleur que les autres modèles : les techniques d'extraction et de recherche de forme par maillage et par filtrage sont effectivement très utiles pour la reconnaissance d'image.

5 Conclusion

Nous avons dans ce dossier, développé théoriquement différentes méthodes de Machine Learning en expliquant en quoi cela consiste, en énonçant aussi la démarche de mise en oeuvre de tels modèles. Dans une 3è partie, nous nous sommes intéressés à nos données issues de la compétition kaggle '*Digit Recognizer*', données que nous avons manipulées au vu des modélisations ML. Enfin, dans une 4è partie nous avons appliqué les méthodes étudiées théoriquement en deuxième partie - en suivant pour chacune d'entre elles un même processus, visant à la comparaison *in fine* des modèles de prédictions. Nous cherchions en effet le modèle qui reconnaît le mieux le chiffre ('digit') dessiné par différents utilisateurs sur une image. Au vu des modélisations réalisées, nous avons constaté que la méthode répondant le mieux à cette problématique est celle des réseaux de neurones avec une couche convolutionelle pour laquelle les taux d'erreur de prévisions out-of-sample sont les plus faibles. Nous retrouvons les différents taux de bien classé des méthodes appliquées dans l'analyse, en table n°8 pour les modèles réalisés avec sklearn.

	kNN	**RLM**	**SVM linéaire**	**SVM non lin.**	**CART**	**Forêt**	**ANN**
Test in-sample	96.64%	100%	100%	100%	100%	100%	98%
Test par CV	93.24%	88.88%	91.08%	94.46%	77.8%	93.6%	-
Test out-of-sample	92.4%	87.8%	90.5%	94.5%	77.1%	92.5%	98.3%
Digit mieux classé	0	0	0	0	0	0 et 1	0 et 6
2è digit mieux classé	1	1 et 6	1	1	1	6	1
Digit moins bien classé	8	8	9	5	8	9	5

TABLE 8 – Résumé des informations liées aux modélisations de l'analyse avec sklearn

La méthode des SVM détient un très bon compromis entre temps de calcul et qualité de prévision hors échantillon pour reconnaître et classer les digits de 0 à 9 (pour lequel nous le rappelons les meilleurs hyper-paramètres choisis par CV étaient C=3 et γ=1e-07). On remarque aussi que l'arbre de décision CART est celui qui prédit le moins bien : son taux de bien classé est de seulement 77% sur l'échantillon test et constitue donc la pire méthode que nous ayons appliquée. Concernant les réseaux de neurones, nous avons résumé les taux de bien classé en prenant seulement le meilleur modèle parmi tous ceux estimés en RNN.

Nous voyons alors que la meilleure méthode en termes de prévisions out-of-sample est celle du RNN avec 1 couche convolutionnelle pour laquelle le taux de bien classé est de 98% tandis qu'en moyenne sur les autres modèles il est de 89.13 soit neuf points de pourcentage en moins - représentant 90 digits mal classés sur les 1000 de l'échantillon test. Par ailleurs, si on regarde les prévisions par cross-validation pour les autres méthodes on voit qu'elles sont relativement bonnes allant d'un taux de bonnes prédictions de 77.1% (CART) à 94.46% (SVM non linéaires). De même, nous voyons que de manière générale, les digits les mieux classés sont ceux comportant des formes rondes tels que le 0 ou le 6. Pareillement, le digit n°1 arrive souvent en deuxième position du chiffre le mieux reconnu. Il apparaît aussi d'après la table 8 que les digits les moins bien reconnus quant à eux sont les '5', '8' et '9'.

Aussi, il peut être intéressant de regarder quelques digits bien ou mal reconnus par un modèle pour ainsi mieux comprendre sur les taux d'erreur de chacun d'entre eux liés soit au sur-apprentissage comme nous

avons pu l'étudier dans ce dossier, soit à la mauvaise écriture du chiffre en question. Nous regardons ces images pour le modèle SVM non linéaire.

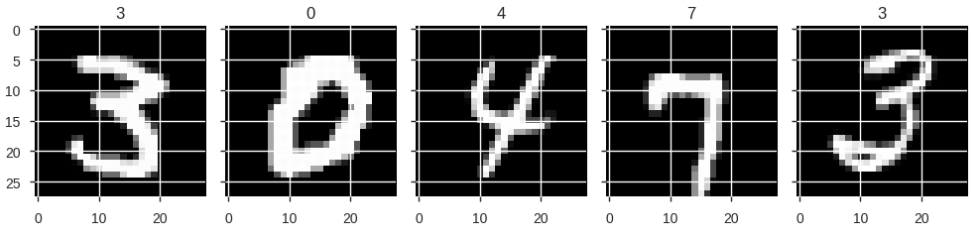


FIGURE 16 – Digits bien prédits par le modèle SVM non linéaire

Grâce à la coloration des 784 pixels qui composent les images (28 lignes * 28 colonnes), le modèle a pu reconnaître correctement les digits puisque nous voyons au dessus de l'image la classe de chiffre dans laquelle a été rangée chaque image analysée par la méthode SVM non linéaire. On voit ainsi les chiffres '3', '0', '4' et '7'.

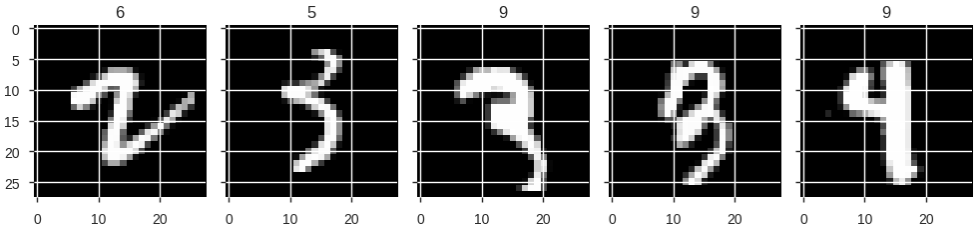


FIGURE 17 – Digits mal prédits par le modèle SVM non linéaire

Ces digits en revanche sont globalement "mal écrits" et le modèle SVM n'arrive pas à les reconnaître, ils ont trop de pixels en communs avec des chiffres différents bien écrits. Le premier chiffre 2 a ainsi été classé en n°6 tandis que les numéros 3 ont été rangés dans les classes '5' et '9'. On voit alors que la reconnaissance d'images est rendue possible grâce aux différents algorithmes de Machine Learning constituant une avancée technologique importante, mais des erreurs persistent du fait notamment de la mauvaise écriture des chiffres par les utilisateurs.