



Projet Mathématiques-Informatique

Groupe d'étude sur le cas du Titanic

ALBANÉSI-CLET Quentin ESPICIER Anaïs MELGET Alban
LEROY Geoffrey LIM Yves-Heng

19 Juin 2024

Table des matières

Remerciements	2
Résumé	3
Abstract	3
Introduction	4
1 Types d'apprentissages	6
1.1 Apprentissage automatique supervisé	6
1.1.1 Préliminaires	6
1.1.2 Quelques définitions	6
1.1.3 Modélisation mathématique	7
1.1.4 Recherche de modèle	8
1.1.5 Évaluation de modèle	10
1.2 Apprentissage automatique non-supervisé	10
1.2.1 Introduction	10
1.2.2 Réduction de la dimensionnalité (prétraitement)	10
1.2.3 Différents types de modèles non-supervisés	11
1.3 Apprentissage automatique semi-supervisé	12
1.3.1 Aspects généraux	12
1.3.2 Conditions d'application	12
2 Différents algorithmes d'apprentissage supervisé	14
2.1 Réseau de neurones	14
2.2 Support Vector Machine	15
2.3 Arbre de décision	18
2.4 Forêt aléatoire	20
3 Cas d'étude : Titanic	22
3.1 Dataset Titanic	22
3.2 Métriques d'évaluation	22
3.3 Prétraitement	23
3.4 Application réseau de neurones	24
3.5 Application Support Vector Machine (SVM)	26
3.6 Application arbre de décision	28
3.7 Application forêt aléatoire	30
3.8 Discussion	31
Conclusion	32

Remerciements

Nous tenons tout d'abord à exprimer notre gratitude envers M. Lahcen TAMYM pour son encadrement et sa confiance tout au long de ce travail de recherche. Ses précieux conseils et son expertise nous ont beaucoup aidé dans l'aboutissement de ce projet.

Nous souhaitons également remercier l'ensemble du corps professoral de la licence Maths-Info d'Aix-Marseille Université, pour la qualité des enseignements et les connaissances qu'ils nous ont transmises durant ces années d'études. Cet apprentissage a été précieux pour la finalisation de ce projet.

Un merci spécial à nos camarades de promotion pour leurs encouragements et leurs échanges intellectuels stimulants et à toute personne qui, de près ou de loin, a contribué à la réussite du projet et de nos études.

Résumé

Peut-on prévoir l'impact d'une catastrophe sur un groupe d'individus ? Les algorithmes d'apprentissage automatique nous permettent aujourd'hui d'envisager des réponses de plus en plus précises à cette question. Ce travail porte sur l'application de plusieurs de ces algorithmes au cas du Titanic dont le naufrage causa, en 1912, la mort de plus d'un millier de personnes. Il s'intéresse dans un premier temps aux différents types d'apprentissage automatique afin d'identifier le type le plus adapté au contexte du naufrage. La prédiction portant sur la mort ou la survie des passagers ou du personnel embarqué sur le paquebot étant essentiellement un problème de recherche de classe, nous abordons, dans un deuxième temps plusieurs modèles d'apprentissage automatique supervisé -réseaux de neurones, support vector machine, arbres de décisions et forêts aléatoires - en expliquant leur logique et arrière-plan théorique. Nous proposons, enfin, une implémentation de ces quatre modèles d'apprentissage supervisé en Python, ainsi qu'une évaluation de leurs performances respectives.

Abstract

Can we predict the impact disasters have on relevant populations ? Machine learning algorithms provide us today with increasingly accurate answers to this question. This report proposes to apply several of these algorithms to the RMS Titanic, which sank in 1912, killing more than one thousand. We first outline the different types of machine learning algorithms so as to identify the type most adequate to the sinking of the ship. Predicting the death or survival of passengers and staff onboard the ship is inherently a classification problem. In the second chapter, we therefore focus on unpacking the logic behind several supervised learning models -neural network, support vector machine, decision trees and random forests. Finally, we propose a Python implementation of each of these four models as well as an assessment of their respective performances.

Introduction

Contexte et Motivations

L'apprentissage automatique, ou apprentissage statistique, est un sous-domaine de l'intelligence artificielle ayant pour but de donner aux machines la capacité *d'apprendre* à partir de données, via des modèles mathématiques [1]. Plus précisément, il s'agit d'améliorer leurs performances à résoudre des tâches sans être explicitement programmés pour celles-ci. Cela englobe la conception, l'analyse, l'optimisation, le développement et l'implémentation de tels modèles [2]. On distingue plusieurs sous-catégories en apprentissage automatique : apprentissage supervisé, semi-supervisé et non supervisé [3].

Pour un bref point historique, l'apprentissage automatique et plus généralement l'intelligence artificielle se concrétisent avec Alan Turing et son concept de "machine universelle" en 1936 [4], puis se développent avec son ouvrage "L'ordinateur et l'intelligence" en 1950 [5]. Ensuite, c'est l'informaticien américain Arthur Samuel qui utilise pour la première fois le terme *machine learning* en 1959 pour désigner un algorithme qui jouait aux dames et s'améliorait en jouant [6]. Une avancée notable a lieu en 1997 avec le super ordinateur Deep Blue développé par IBM au début des années 1990, qui réalise l'exploit de battre le champion du monde d'échec en titre de l'époque en 1997 [7]. Ce projet en inspira de nombreux et dans les années suivantes, des applications de l'apprentissage automatique médiatisées se multiplient : en 2014, une intelligence artificielle parvient à convaincre des humains qu'elle est un garçon ukrainien de 13 ans en 5 minutes de conversation [8] ; en 2016, un réseau neuronal profond pour la reconnaissance visuelle de la parole nommé Lipnet est développé et permet de lire sur les lèvres [9], etc...

Mais les applications de l'apprentissage automatique sont également nombreuses dans d'autres domaines plus fondamentaux de notre société moderne. En marketing, les algorithmes d'apprentissage automatique par régression linéaire permettent de modéliser la relation entre les dépenses publicitaires et les chiffres de vente, prévoyant ainsi les ventes futures [10]. En médecine, la régression logistique est utilisée pour calculer le risque d'une certaine maladie sur la base des caractéristiques des patients, ce qui permet d'établir un diagnostic plus rapide et plus précis [11]. En finance, les SVM (Support Vector Machine) permettent de détecter les transactions frauduleuses : elles trouvent des lignes de démarcation claires entre les modèles de transaction normaux et suspects [12]. En industrie automobile, les réseaux de neurones sont utilisés par les véhicules autonomes pour apprendre à comprendre des modèles de trafic complexes [13]. Et les exemples sont nombreux.

Un domaine particulièrement crucial où l'apprentissage automatique joue un rôle clé est la prédiction des catastrophes [14]. En effet, il est très difficile d'anticiper des catastrophes extrêmes lorsque leur faible fréquence empêche d'alimenter les algorithmes 'classiques'. Ce problème peut être en partie contourné en utilisant des algorithmes d'apprentissage automatique qui nécessitent moins de données pour pouvoir ensuite effectuer des prédictions sur les risques de catastrophes [15]. Par exemple, un réseau de neurones à couche (voir section 2.1 pour plus de détails) nommé DeepONet, présenté en 2019 par des chercheurs de l'Université de Brown, peut être entraîné à rechercher les paramètres ou les précurseurs qui conduisent à l'apparition de vagues scélérates, même si les points de données sont peu nombreux [16]. En ce qui concerne les inondations, l'intelligence artificielle aidée de l'IoT (Internet des objets) joue un rôle crucial dans l'amélioration des capacités des systèmes de prévision et de gestion de ce type de catastrophe. En effet, des drones équipés de capteurs avancés et de caméras fournissent des données en temps réel sur les niveaux d'eau, les conditions météorologiques et les dommages aux infrastructures lors d'inondations ; toutes ces informations sont ensuite transmises en temps réel aux algorithmes d'apprentissage automatique qui les analysent et les interprètent afin de générer des prédictions d'inondations et d'aider à la décision dans la gestion de catastrophe [17]. Un autre exemple est la prédiction des tremblements de terre : un nouvel algorithme d'apprentissage automatique, nommé Earthquake Transformer et développé par des chercheurs de l'Université de Stanford, est capable de détecter plus de séismes avec un niveau de précision proche des analyses humaines, en particulier ceux de faible intensité habituellement non repérés par les méthodes de détection traditionnelles. Et le fait de pouvoir repérer et donc étudier les plus petits séismes permet d'obtenir une meilleure compréhension globale de la manière dont les tremblements de terre se déclenchent et prennent fin [18].

La catastrophe qui va nous intéresser dans ce projet est l'une des plus tristement célèbres de l'Histoire : le naufrage du Titanic. Le 15 avril 1912, le Titanic, considéré comme insubmersible, fit naufrage après avoir

heurté un iceberg dans l'Atlantique Nord. Bien que la chance ait joué un rôle dans la survie des rescapés, les faits semblent indiquer que certains groupes de personnes ont eu plus de chances de survivre que d'autres. *Notre objectif est donc de construire un modèle prédictif grâce à des algorithmes d'apprentissage automatique à partir des données disponibles sur les survivants afin de répondre à la question suivante : " Quels profils de personnes avaient le plus de chances de survivre ? "*

Classification de problèmes

Comme nous l'avons vu dans la section précédente, les problèmes résolus par l'apprentissage automatique sont nombreux et de natures très différentes. On distingue trois déclinaisons majeures des problèmes d'apprentissage automatique : la classification, la régression et l'ordonnancement. Lorsque les données sur lesquelles l'algorithme apprend sont discrètes, on parle de problème de classification : le modèle associe chaque élément à une catégorie/classe à partir de ses caractéristiques ou attributs. Lorsque les données sont continues, on parle de problème de régression. Les problèmes d'ordonnancement quant à eux se sont développés récemment et se situent à la frontière de l'apprentissage machine et de la recherche d'information.

*Le problème que nous allons chercher à résoudre est donc un **problème de classification** : à partir des caractéristiques des passagers survivants et de ceux qui ont péri, nous devons déterminer une "classe" de personnes qui aurait eu le maximum de chances de survie.*

On distingue deux types de problèmes de classification : la classification binaire et la classification multiple. En effet, la classification binaire est le type de classification le plus courant en apprentissage supervisé. Elle traite les problèmes à deux classes (par exemple, e-mail spam vs. non spam [19]). L'objectif d'un algorithme basé sur la classification binaire est donc d'associer les éléments d'un dataset à l'une des deux classes en fonction de leurs caractéristiques ou de leurs attributs. La question de la survie ou non d'un passager que nous cherchons à résoudre dans le cadre de ce projet est donc un problème de **classification binaire** puisqu'il n'y a que deux classes possibles où "ranger" les passagers : survivant ou naufragé. La classification multi-classe concerne quant à elle les problèmes de classification où plus de deux étiquettes peuvent être attribuées aux données, c'est-à-dire concerne les modèles prédisant deux événements ou plus. Par exemple, si l'on souhaite entraîner un algorithme à prédire la météo à partir d'une photo, alors il y aura plus de deux alternatives de réponse : ensoleillé, couvert, pluvieux, orageux, etc... Les données peuvent être "rangées" dans plusieurs classes, d'où l'appellation multiclass. Un large éventail de problèmes multiclass existent dans le monde réel, cependant l'application des modèles d'apprentissage automatique à ce type de problèmes est loin d'être aisée, et des recherches sont toujours en cours afin de déterminer une méthode générale pour les résoudre.

Chapitre 1

Types d'apprentissages

1.1 Apprentissage automatique supervisé

1.1.1 Préliminaires

L'apprentissage automatique supervisé est une sous-catégorie de l'apprentissage automatique s'appliquant aux problèmes de classification qui consiste à utiliser des données étiquetées, ou annotées, pour entraîner les algorithmes. Au fur et à mesure que les données étiquetées (avec la réponse attendue du modèle) alimentent ce dernier, il ajuste ses pondérations en minimisant l'écart entre sa réponse et la réponse attendue jusqu'à atteindre un niveau de précision satisfaisant. Un algorithme d'apprentissage supervisé est donc nécessairement associé à une fonction d'erreur ou de perte qui lui permettra de s'ajuster au fur et à mesure des entraînements, c'est-à-dire d'apprendre.

1.1.2 Quelques définitions

Un modèle d'apprentissage automatique construit une **fondation de prédiction** à partir d'un ensemble fini d'exemples, appelé **base d'entraînement** (ou training set). L'aspect supervisé de l'apprentissage réside dans l'étiquetage de cet ensemble de données : généralement, chacun de ses éléments est un couple constitué du vecteur représentatif d'une observation (composé des **variables explicatives**) et de sa **sortie désirée**, ou **réponse associée**. Le but de l'apprentissage est d'induire une fonction qui prédisse les réponses associées à de nouvelles observations en commettant une erreur de prédiction la plus faible possible.

L'hypothèse fondamentale de l'apprentissage automatique est que les données sont **stationnaires**, c'est-à-dire que les exemples de la base d'entraînement, sur laquelle l'algorithme apprend, sont bien représentatifs du problème général que l'on souhaite résoudre. Autrement dit, les exemples d'entraînement ainsi que les observations futures et leur sorties désirées sont supposés être issus d'une même source d'information.

Un autre concept de base en apprentissage est la notion de coût, aussi appelé **risque** ou **erreur**. On distingue deux notions : l'erreur réelle et l'erreur empirique.

L'espace réel n'est modélisé que par l'ensemble des données de la base d'entraînement. En apprenant à partir de celle-ci, le modèle commet deux erreurs : l'erreur d'approximation ou « biais » : utiliser un ensemble de données limité plutôt que l'espace réel, et l'erreur d'estimation ou « variance » : trouver une fonction de prédiction qui n'est malheureusement pas optimale. L'**erreur réelle** (ou erreur de généralisation) est la somme des erreurs faites : biais plus variance. Mais l'erreur réelle, et plus particulièrement le biais, sont difficilement mesurables puisque la modélisation est faite sur la base d'entraînement.

C'est pourquoi on l'estime grâce à l'**erreur empirique** (ou erreur d'apprentissage), qui correspond à l'erreur d'estimation faite par la fonction de prédiction sur la base d'entraînement. Elle mesure l'écart entre les prédictions du modèle et les sorties désirées (connues grâce à l'étiquetage des données). Elle sert ainsi à évaluer la qualité du modèle. L'erreur empirique est en quelque sorte une estimation optimiste de l'erreur réelle. Mais attention, une réduction de l'erreur d'apprentissage n'a pas nécessairement comme conséquence une diminution de l'erreur de généralisation !

De plus, il est également possible d'estimer l'erreur réelle via deux approches différentes :

1. En utilisant des **données de test** (test set) : L'erreur réelle peut être estimée par l'erreur sur les données de test, qui sont étiquetées mais qui n'ont pas été utilisées pour l'apprentissage. Cela présente cependant l'inconvénient de réduire significativement le volume de données disponibles pour l'apprentissage.

2. Sous certaines conditions, il est possible de déterminer une borne supérieure finie sur l'écart entre l'erreur réelle et l'erreur empirique (cf. mesure de la capacité). Une telle borne dépend des caractéristiques de la famille de fonctions de prédiction considérées. Malheureusement, très peu de situations pratiques permettent d'atteindre des bornes suffisamment étroites pour être utiles.

1.1.3 Modélisation mathématique

Nous supposons que les observations possèdent une représentation numérique dans un espace vectoriel de dimension finie, $\mathcal{X} \subset \mathbf{R}^d$. Les sorties désirées des observations sont supposées faire partie d'un ensemble de sortie $\mathcal{Y} \subset \mathbf{R}$.

Dans le cadre d'un problème de classification, l'ensemble de sortie \mathcal{Y} est discret et la fonction de prédiction $f : \mathcal{X} \rightarrow \mathcal{Y}$ est appelée un **classifieur**. C'est à ce type de problème que correspond notre projet : l'ensemble de sortie \mathcal{Y} n'est composé que de 2 éléments, correspondant à la survie ou non du passager considéré au naufrage du Titanic. Par exemple, $\mathcal{Y} = \{0, 1\}$ ou $\{-1, 1\}$ (0 ou -1 pour un naufrage, 1 pour la survie).

Un couple $(x, y) \in \mathcal{X} \times \mathcal{Y}$ désigne ainsi un exemple étiqueté (observation, réponse associée) et $S = (x_i, y_i)_{i=1}^m \in (\mathcal{X} \times \mathcal{Y})^m$ décrit la base d'entraînement.

De plus, afin que les données soient stationnaires, il est nécessaire que tous les exemples soient identiquement distribués (pour assurer la représentativité de la base d'entraînement). Et afin que cette dernière soit optimale, c'est à dire que chaque exemple individuel apporte un maximum d'information pour résoudre le problème de prédiction, il est fondamental que tous les exemples soient également générés indépendamment les uns des autres. Ainsi, on suppose que les exemples (x_i, y_i) de tout ensembles d'entraînement S et de test T sont identiquement et indépendamment distribués (i.i.d.) selon une distribution de probabilité fixe, mais inconnue, notée \mathcal{D} .

Pour mesurer l'erreur du modèle, on introduit une **fonction de coût instantané** définie par :

$$e : \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbf{R}^+$$

D'une manière générale, cette fonction est une distance sur l'espace des sorties \mathcal{Y} : elle mesure l'écart entre la réponse du modèle et la réponse prédictive par l'étiquetage des données. En classification bi-classe, l'erreur généralement utilisée est le coût 0/1 : 0 si la réponse du modèle et la réponse prédictive coïncident, 1 sinon. Autrement dit, pour une observation $(x, y) \in S$ et une fonction de prédiction f , on a :

$$e(f(x), y) = \mathbb{1}_{f(x) \neq y}$$

Cependant, un inconvénient de ce coût 0/1 est qu'il est symétrique : toute erreur de prédiction apporte une même pénalité (égale à 1), quelque soit l'erreur. Or, ce n'est pas toujours un choix satisfaisant en fonction du problème de classement considéré : par exemple, il est plus grave de classifier un e-mail 'normal' en tant que spam qu'un spam en tant qu'e-mail 'normal'. C'est pourquoi il existe également des coûts asymétriques, mais ils ne seront pas détaillés ici.

À partir du coût instantané et de la génération i.i.d. des exemples selon la distribution \mathcal{D} , on peut définir l'erreur réelle \mathcal{E} d'une fonction de prédiction apprise $f \in \mathcal{F}$ comme :

$$\mathcal{E}(f) = \mathbb{E}_{(x,y) \sim \mathcal{D}} e(f(x), y) = \int_{\mathcal{X} \times \mathcal{Y}} e(f(x), y) d\mathcal{D}(x, y) \quad (1)$$

où $\mathbb{E}_{(x,y) \sim \mathcal{D}} X(x, y)$ est l'espérance de la variable aléatoire X lorsque (x, y) suit la distribution de probabilité \mathcal{D} . Mais comme précisé plus haut, \mathcal{D} est inconnue donc cette erreur réelle ne peut pas être calculée explicitement. C'est ici qu'intervient l'erreur empirique $\hat{\mathcal{E}}$ de f , qui correspond tout simplement à la moyenne pour tous les exemples de la base d'entraînement de l'écart entre la réponse du modèle et la sortie désirée. Ainsi, pour une base d'entraînement S de taille m et une fonction de prédiction f , on a :

$$\hat{\mathcal{E}}(f, S) = \frac{1}{m} \sum_{i=1}^m e(f(x_i), y_i) \quad (2)$$

On remarque que l'erreur réelle ne dépend que de la fonction de prédiction considérée, tandis que l'erreur empirique dépend à la fois de la fonction f et de la base d'entraînement S .

1.1.4 Recherche de modèle

Choix d'une famille paramétrique

Une fois les données traitées et la fonction de coût instantané déterminée, il est nécessaire de choisir la famille de fonctions de prédiction parmi laquelle le modèle sera recherché, appelée famille paramétrique. Il en existe plusieurs :

- Les modèles **linéaires**, pour lesquels la fonction de prédiction est obtenue comme combinaison linéaire des différentes variables explicatives. Ils ne sont pas très précis (erreur empirique relativement élevée) mais commencer par ces modèles est une bonne pratique lors de la résolution d'un problème d'apprentissage.
- Les modèles **polynômiaux** de degré $n \in \mathbb{N}^*$ borné ($n = 1$: modèles linéaires). Dans ce cas, la dépendance entre l'entrée (les variables explicatives) et la prédiction fournie par le modèle est polynomiale. Chaque valeur de borne sur le degré n définit une famille paramétrique. La capacité d'approximation des polynômes augmente avec leur degré.
- Les modèles **non linéaires** : les perceptrons multicouches (PMC) d'architecture donnée (cf. réseaux de neurones), etc...

Une piste de choix de modèle serait de considérer uniquement les modèles possédant une capacité d'approximation élevée (sous-entendu : d'approximation des sorties désirées de la base d'entraînement) et de mettre de côté les autres. Cependant, on risque de tomber dans le piège du **sur-apprentissage** (*overfitting*) : l'erreur d'apprentissage est très faible mais l'erreur sur les données de test est comparativement élevée, autrement dit le modèle généralise mal. Cela est dû au fait qu'une capacité d'approximation trop élevée amène le modèle à apprendre par cœur les particularités des données d'entraînement, par exemple le bruit, au lieu de généraliser. Ainsi, une erreur empirique faible n'a pas nécessairement comme conséquence une erreur réelle diminuée.

Une mesure de la capacité

Nous avons parlé brièvement de la capacité d'approximation d'une famille de fonctions de prédiction, mais définissons maintenant un cadre formel pour cette notion. La mesure de capacité la plus connue est la **dimension de Vapnik-Chervonenkis**, que l'on retrouvera notamment lorsque l'on abordera les SVM (*Support Vector Machine*).

Considérons un ensemble E de N vecteurs $\{x_i\}_{1 \leq i \leq N} \in \mathbf{R}^p$. Il y a 2^N façons possibles de séparer l'ensemble E en deux (attention, l'ensemble vide et E tout entier sont des parties). On peut ainsi évaluer la capacité d'approximation d'une famille \mathcal{F} de classificateurs $f : \mathbf{R}^p \rightarrow \{-1, 1\}$ en étudiant sa capacité à séparer différents ensembles de vecteurs en deux parties P_1 et P_2 telles que $f^{-1}(\{1\}) = P_1$ et $f^{-1}(\{-1\}) = P_2$, ou inversement.

Introduisons maintenant la notion de **pulvérisation** : On dit que la famille de fonctions \mathcal{F} pulvérise E si toutes les 2^N séparations possibles peuvent être obtenues avec des fonctions de \mathcal{F} .

Ainsi, l'ensemble \mathcal{F} est de **VC-dimension** (dimension de Vapnik-Chervonenkis) n s'il pulvérise au moins un ensemble de n vecteurs et aucun ensemble de $n + 1$ vecteurs. Autrement dit, la VC-dimension de \mathcal{F} est la taille du plus grand ensemble pulvérisable par \mathcal{F} .

Par exemple, considérons l'ensemble des droites de \mathbf{R}^2 . Cet ensemble est de VC-dimension 3. En effet, considérons 3 vecteurs de \mathbf{R}^2 (le triplet de points ci-dessous). Alors les $2^3 = 8$ séparations possibles de cet ensemble peuvent être obtenues grâce à au moins une droite. Cependant, si l'on considère un quadruplet de points, cet ensemble ne peut pas être pulvérisé par une droite (on ne peut pas séparer les points rouges des points bleus).

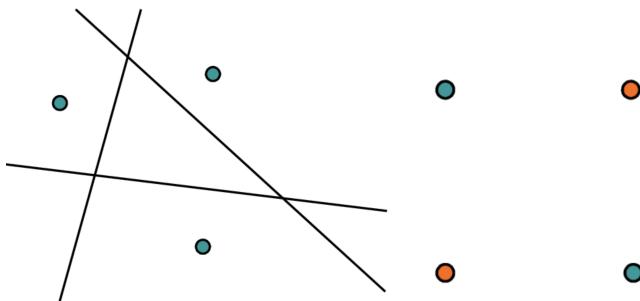


FIGURE 1.1 – Triplet de points pulvérisable et quadruplet non pulvérisable.

La VC-dimension est une mesure de capacité particulièrement intéressante car elle permet d'obtenir une borne sur l'écart entre l'erreur empirique et l'erreur réelle en fonction de la capacité de la famille de fonctions considérée \mathcal{F} . Reprenons les notations introduites précédemment : soit $e(f(x), y) = \mathbb{1}_{f(x) \neq y}$ la fonction de coût instantané, $\hat{\mathcal{E}}(f, S)$ l'erreur empirique et $\mathcal{E}(f)$ l'erreur réelle de f .

Voici le théorème central de ce principe : Si la VC-dimension de \mathcal{F} est $h < \infty$ alors pour toute fonction de prédiction $f \in \mathcal{F}$, avec une probabilité au moins égale à $1 - \delta$ avec $0 < \delta < 1$, on a :

$$\mathcal{E}(f) \leq \hat{\mathcal{E}}(f, S) + \underbrace{\sqrt{\frac{h(\log \frac{2N}{h} + 1) - \log \frac{\delta}{4}}{N}}}_{B(N, \mathcal{F})}$$

pour $N > h$ avec $N = |S|$ la taille de la base d'entraînement.

Ainsi, on obtient d'une borne $B(N, \mathcal{F})$ sur l'erreur de généralisation $\mathcal{E}(f)$ qui dépend à la fois de l'erreur empirique $\hat{\mathcal{E}}(f, S)$ ainsi que de la capacité h de la famille de fonction et de la taille N de la base d'entraînement S .

Cette borne permet d'observer que l'erreur réelle diminue lorsque le nombre de données d'entraînement augmente, et c'est logique puisqu'avec plus de données d'apprentissage, le modèle est mieux "contraint" donc plus précis. Elle peut aussi éventuellement diminuer lorsque la capacité h de \mathcal{F} est plus faible pour une taille N fixée : en effet, il est plus facile de "contraindre" un modèle moins "flexible", mais l'erreur empirique sera élevée. Enfin, elle diminue lorsque δ augmente (l'écart par rapport à une probabilité de 1), c'est-à-dire lorsque les garanties qu'apporte la borne diminuent. Cependant, il faudrait disposer d'un très grand nombre d'observations N pour obtenir une borne sur l'erreur réelle intéressante en pratique, ce qui n'est généralement pas le cas.

Estimation de modèle

Une fois la famille paramétrique déterminée, nous recherchons le modèle présentant l'erreur empirique la plus faible, tout en gardant à l'esprit le risque du sur-apprentissage. Ainsi, il est nécessaire de contrôler à la fois le risque empirique mais aussi la capacité d'approximation de la famille de fonctions considérée. Évoquons pour cela trois principes.

- **Principe de minimisation du risque empirique** (MRE) : Ce principe consiste à rechercher dans la famille de fonctions considérée \mathcal{F} la fonction f qui minimise le risque empirique sur les données de la base d'entraînement.
 $f_S^* = \arg \min_{f \in \mathcal{F}} \hat{\mathcal{E}}(f, S)$
- **Principe de minimisation du risque empirique régularisé** (MRER) : On recherche la fonction f dans \mathcal{F} qui minimise la somme entre l'erreur empirique et un terme de régularisation $G(f)$ qui pénalise la capacité de la famille de fonction, pondéré par un hyperparamètre α :
 $f_S^* = \arg \min_{f \in \mathcal{F}} [\hat{\mathcal{E}}(f, S) + \alpha G(f)]$
- **Principe de minimisation du risque structurel** (MRS) : On considère une suite de familles de fonctions croissante (i.e incluses les unes dans les autres) et on effectue une estimation MRE pour chaque famille. Le choix final d'un modèle tiendra compte à la fois du risque empirique $\hat{\mathcal{E}}(f, S)$ du modèle et de la capacité d'approximation de la famille de fonctions dont il est issu.

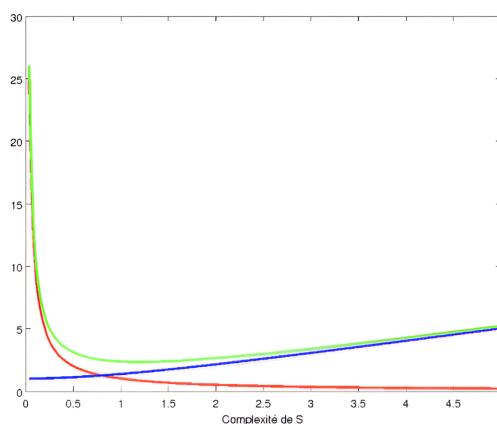


FIGURE 1.2 – En fonction de la capacité d'approximation, en rouge le risque empirique, en bleu le risque réel, en vert somme des deux.

1.1.5 Évaluation de modèle

On l'a dit, l'estimation directe du risque réel par le risque empirique (estimation *in-sample*) est très optimiste et n'est donc pas optimale. Lorsque que la capacité d'approximation de \mathcal{F} est minime et que le nombre de données d'apprentissage $N = |S|$ est très élevé, on a vu qu'il existe une borne de généralisation $B(N, \mathcal{F})$ permettant de majorer le risque réel du modèle.

De façon générale, une méthode pour estimer le risque réel est d'utiliser des **données de test** (estimation *out-of-sample*), distinctes de celles la base d'entraînement. En présentant ces données inconnues au modèle puis en comparant sa réponse aux sorties attendues associées aux données (cadre de l'apprentissage supervisé), on étudie ainsi sa capacité à généraliser.

Mais l'estimation du risque réel ainsi obtenue est très variable et dépend du découpage des données de départ en base d'entraînement et échantillon de test. Si l'on note T_n un échantillon de test et T_m un autre échantillon sur les mêmes données initiales, alors l'estimation du risque réel obtenue en testant le modèle sur T_n ou sur T_m peut être très différente. Afin de réduire la variance de cette estimation, une méthode est de moyenner les résultats issus de plusieurs découpages d'échantillons de test distincts : c'est ce que l'on appelle la **validation croisée**. Il existe plusieurs méthodes de validation croisée en fonction du type de découpage réalisé :

- méthodes exhaustives (tous les découpages possibles respectant certains effectifs sont utilisés) : *Leave P Out* (LPO), *Leave One Out* (LOO), etc...
- méthodes non-exhaustives : *k-fold*, échantillonnage répété (*shuffle and split*), etc...

1.2 Apprentissage automatique non-supervisé

1.2.1 Introduction

L'apprentissage non-supervisé est une branche du machine learning encore nouvelle et avec peu de cas d'utilisation concrets. Le développement de ce dernier est le résultat d'un questionnement scientifique et philosophique sur la capacité d'une machine à avoir un raisonnement humain.

L'objectif de l'apprentissage non-supervisé découle de ce questionnement. Il repose alors sur l'utilisation de jeux de données non étiquetées. Le modèle doit donc lui même engendrer une distribution à partir des données.

On observe donc une suite de vecteurs aléatoires X_1, \dots, X_n , $X_i \in \mathbb{R}^n$ à partir de laquelle nous utilisons les algorithmes non-supervisés pour essayer de trouver une densité pour ces données.

Nous pouvons classer les algorithmes d'apprentissage non supervisé en 3 grandes catégories :

- Algorithmes de clustering
- Algorithmes d'apprentissage par règles d'association
- Autoencodeurs (Réseaux de neurones)

Avant de rentrer plus en détail dans le fonctionnement de ces algorithmes il est primordial d'aborder le sujet du prétraitement des données, dont les méthodes utilisées diffèrent du cas supervisé.

1.2.2 Réduction de la dimensionnalité (prétraitement)

La réduction de la dimensionnalité consiste à simplifier un problème complexe en se concentrant sur ses aspects les plus importants. Il s'agit de techniques qui réduisent le nombre de caractéristiques d'un ensemble de données tout en conservant ses informations essentielles. Mathématiquement, cela revient à passer de vecteurs de \mathbb{R}^n à des vecteurs de \mathbb{R}^p avec $p < n$, ce qui allège la complexité des calculs et rend plus facile la visualisation des données.

Les méthodes utilisées pour accomplir cette tâche sont :

- L'analyse en composantes principales (ACP)
- La t-Distributed Stochastic Neighbor Embedding (t-SNE)

L'application de méthodes de réduction de dimensionnalité permet de réduire les coûts en espace mémoire nécessaire au stockage des données ainsi que de diminuer les temps de calcul requis aux modèles.

1.2.3 Différents types de modèles non-supervisés

Les algorithmes de clustering

Le clustering, appelé aussi segmentation, est une technique qui consiste à regrouper dans un ensemble des objets sur la base de leurs similitudes. Pour rendre cela possible, ils mobilisent la notion de distance entre des objets dans un espace métrique.

Les modèles que l'on peut classer dans la catégorie des clustering sont les suivants :

- K-Means
- DBSCAN
- Clustering hiérarchique - Déplacement de la moyenne

Un distance, $d : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}_+$, $n \in \mathbb{N}$, permet d'établir la ressemblance ou la dissemblance entre 2 vecteurs ou même 2 clusters.

On peut citer, de manière non-exhaustive, la distance Euclidienne :

$$d(x, y) = \left(\sum_{i=1}^n (x_i - y_i)^2 \right)^{1/2} \quad x, y \in \mathbb{R}^n$$

La distance de Manhattan :

$$d(x, y) = \sum_{i=1}^n |x_i - y_i| \quad x, y \in \mathbb{R}^n$$

Il existe beaucoup d'autres distances aussi très utilisées dans la création de clusters.

Les algorithmes d'apprentissage par règles d'association

L'apprentissage par règles d'association permet, par exemple, de trouver des liens intéressants entre les articles achetés dans un magasin. Il s'agit d'une technique d'apprentissage automatique qui permet de découvrir des relations dans les données et d'identifier des modèles, des tendances et des associations entre des éléments ou des événements. L'apprentissage par règles d'association est un outil précieux pour comprendre le comportement des consommateurs, optimiser les stocks et améliorer les recommandations personnalisées.

Les algorithmes dans la catégorie Apprentissage par règles d'association :

- Apriori
- FP-Growth (croissance des motifs fréquents)
- Algorithme Eclat

Les concepts de base du modèle des "règles d'association" reposent sur les trois concepts clés que sont le support, la confiance et le lift.

Le support est la fréquence relative d'une combinaison d'objets dans l'ensemble de données. Il est calculé comme le nombre de transactions contenant un objet (ou un ensemble d'objets) divisé par le nombre total de transactions. La formule est :

$$\text{Support}(A) = \frac{n(A)}{T}$$

où $n(A)$ est le nombre de transactions contenant l'item A, et T est le nombre total de transactions.

La confiance est une mesure de la fiabilité de la règle. Elle est définie comme la fréquence à laquelle les items du côté droit de la règle apparaissent dans les transactions qui contiennent les items du côté gauche. La formule est :

$$\text{Confiance}(A \rightarrow B) = \frac{n(A \cup B)}{n(A)}$$

où $A \rightarrow B$ indique la règle "si A, alors B", $n(A \cup B)$ est le nombre de transactions contenant à la fois A et B. Le lift est une mesure de la force de la règle. Il compare la probabilité d'observer A et B ensemble avec la probabilité d'observer A et B indépendamment. La formule est :

$$\text{Lift}(A \rightarrow B) = \frac{P(B|A)}{P(B)} = \frac{\text{Confiance}(A \rightarrow B)}{\text{support}(B)}$$

ou, en utilisant les nombres de transactions :

$$Lift(A \rightarrow B) = \frac{n(A \cup B) \times T}{n(A) \times n(B)}$$

Un lift supérieur à 1 suggère que A et B sont achetés ensemble plus souvent que ce qui serait attendu si ils étaient indépendants ; un lift inférieur à 1 suggère le contraire.

Les autoencoders

Il s'agit d'un type de réseau de neurones qui prend des données complexes, les compresse en un code, puis tente de recréer les données d'entrée à partir d'un code résumé. Ce processus de compression-décompression peut être utilisé pour supprimer le bruit des données visuelles telles que les images, les vidéos et les scanners médicaux afin d'en améliorer la qualité.

1.3 Apprentissage automatique semi-supervisé

1.3.1 Aspects généraux

L'apprentissage semi-supervisé intègre à la fois des éléments des méthodes d'apprentissage supervisé et non-supervisé. L'apprentissage semi-supervisé présente donc un avantage vis-à-vis de l'apprentissage supervisé lorsque les données labélisées sont rares ou couteuses à produire –ce qui est généralement le cas.

L'ensemble des données traitées par les méthodes d'apprentissage semi-supervisé peuvent être divisées en un premier ensemble de données déjà labélisées et d'un second ensemble de données sans label. Parmi les multiples modèles d'apprentissage semi-supervisé, on peut notamment distinguer le modèle de classification supervisée qui entraîne le modèle sur un ensemble de données comprenant des données labélisées et d'autres non-labélisées (généralement beaucoup plus nombreuses) dans le but de créer un classificateur fonctionnant pour les deux types de données [20]. Les frontières entre les classes peuvent être généralement déduites de l'ensemble des données et le label adéquat déduit des données labélisées.

1.3.2 Conditions d'application

Les modèles d'apprentissage semi-supervisé associent à toute donnée un vecteur décrivant les attributs d'une instance observée. En d'autre terme, pour tout donnée x , on a :

$$x = (x_1, x_2, \dots x_n)$$

Le fonctionnement des modèles d'apprentissage semi-supervisé suppose une hypothèse sur l'ensemble des données non-labélisées et le label sur lequel le modèle travaille, c'est-à-dire une hypothèse sur la distribution de probabilité du label pour les données non-labélisées [21] [20]. Les hypothèses clefs peuvent être identifiées comme suit [21] :

* Hypothèse de continuité (smoothness) qui implique que deux points proches dans une région de haute densité devraient appartenir à la même classe.

* Hypothèse portant sur les cluster : si des points font parties d'un même ensemble (cluster), ils doivent généralement appartenir à la même classe.

* Hypothèse portant sur la dimensionnalité (manifold hypothesis) : en traitant chaque donnée comme un vecteur, les données peuvent être de dimension élevée, mais le degré de liberté des composantes est limité ce qui permet de simplifier le problème par réduction de dimensionnalité.

* Hypothèse de transduction : bien qu'il soit parfois possible de trouver des lois générales pour la distribution des classes, le problème est généralement complexe. Le principe de Vanik propose que lorsqu'un problème est posé, les étapes intermédiaires de résolution ne doivent pas être plus difficile à résoudre que le problème lui-même. L'approche transductive propose en ce sens de trouver un modèle de classes pour les données sans tenter de résoudre la question des lois.

L'apprentissage semi-supervisé peut prendre deux formes générales. Une première est l'apprentissage induc-tif (inductive learning) qui, à partir d'un ensemble de données d'entraînement, vise à créer un classifieur qui soit capable de prédictions fiables sur des données futures. Par contraste, l'apprentissage transductif (transductive learning) entraîne un classificateur qui vise à une prédiction correcte sur l'ensemble des données non-labélisées contenue dans l'ensemble d'entraînement [20].

Il est possible de considérer les modèles d'apprentissage semi-supervisé soit comme extension des modèles supervisés, soit comme extension des modèles non-supervisés, bien que la 'famille' des modèles semi-supervisés inclut de multiples modèles [20]. En d'autre termes, les modèles d'apprentissage semi-supervisé peuvent être conçus comme une extension des problèmes de classification et de régression propres à l'apprentissage supervisé et adaptés de manière à ce qu'ils puissent opérer sur des données non-labelisées. L'ensemble d'entraînement inclut alors à la fois des données labelisées et non-labelisées, les premières fournissant généralement les labels. Ils peuvent également être considérés comme des extensions des problèmes de modèles non-supervisés lorsque le modèle entreprend un clustering qui est contraint par certaines données (notamment des relations de must-link ou cannot-link entre certains datapoints).

Chapitre 2

Différents algorithmes d'apprentissage supervisé

2.1 Réseau de neurones

Un réseau de neurones est un exemple particulier d'un algorithme d'Apprentissage Automatique Supervisé (AAS). Nous allons détailler ce modèle défini par des règles particulières.

Historique des réseaux de neurones

Le cerveau humain passionne depuis longtemps les scientifiques par son aspect mystérieux et sa grande capacité d'apprentissage. Dès l'avènement des sciences informatiques, le copier, ou au moins s'en approcher, est devenu un *leitmotiv* de nombreux chercheurs.

De manière plutôt inattendue, le premier véritable réseau de neurones date de 1957, alors que l'informatique est encore balbutiante. C'est un psychologue et informaticien américain, Frank Rosenbalt, qui, inspiré des travaux réalisés par différents neuroscientifiques, crée le "perceptron", un modèle pouvant reconnaître des formes simples.

Par la suite, les réseaux de neurones souffrent d'une mauvaise image, due à un ouvrage des scientifiques américains Marvin Lee Minsky et Seymour Papert, qui mettent en avant les faiblesses des réseaux neuronaux primitifs, incapables par exemple de réaliser un XOR (ou exclusif).

Il faudra attendre 1982 pour voir les premiers réseaux neuronaux multicouches apparaître : ces réseaux neuronaux pallient aux manques des premiers modèles, et relancent la recherche à leur sujet. Parallèlement, l'informatique se développe, et permet ainsi d'explorer plus de possibilités [22].

Présentation d'un réseau de neurones

Tout d'abord, il convient de présenter un neurone formel, qui construit les réseaux de neurones.

Inspirés des travaux sur les neurones biologiques, les premiers neurones artificiels sont une représentation mathématique simplifiée d'un modèle développé en 1881 par l'Allemand Waldeyer. Ce modèle prend un nombre fini d'entrées, et retourne une valeur binaire (0 ou 1) [23]. Pour cela, un neurone formel est composé de deux fonctions distinctes :

- Une fonction de combinaison qui, à partir de toutes les entrées, retourne une valeur. Dans le cadre de modèles simplifiés, il s'agit souvent d'une somme des valeurs d'entrée pondérées par des valeurs fixes. Dans le cadre de modèles plus élaborés, il peut s'agir de fonctions plus poussées.
- Une fonction d'activation qui va retourner 0 ou 1, en fonction de la valeur retournée par la première fonction. Cette fonction s'apparente à une fonction en "marche d'escalier", dont le seuil est fixe.

Cependant, comme vu plus haut, un neurone seul est incapable de réaliser des opérations mathématiques simples, ce qui limite son utilisation, et l'engouement des chercheurs à développer ces modèles. L'ingéniosité du système repose sur le principe régissant les neurones biologiques, c'est-à-dire l'intercommunication : les neurones peuvent communiquer entre eux, ce qui permet de se servir de la valeur de sortie de plusieurs neurones comme valeur d'entrée pour un autre neurone. Une telle organisation permet de varier les fonctions employées, et donc de réaliser une multitude d'opérations [22].

Revenons à présent à notre réseau de neurones. Comme son nom l'indique, un réseau de neurones est constitué de différents neurones interconnectés. Cela peut aller de deux à un nombre bien plus grand, cependant, afin

d'optimiser le plus possible le réseau, des modèles se sont dessinés au fil des recherches. L'un des plus utilisés actuellement est le modèle de réseau de neurones à couche.

Les réseaux de neurones à couche

Un réseau de neurones à couche peut être vu comme un assemblage d'au moins trois couches :

- La couche d'entrée, qui prend en charge les données, et les exploite une première fois
 - La ou les couche(s) cachée(s) qui utilisent les résultats des premiers neurones pour trouver la sortie attendue
 - La couche de sortie, composée d'un ou plusieurs neurones, qui renvoie le résultat attendu par l'utilisateur
- Un schéma simplifié de ce type de réseau est le suivant :

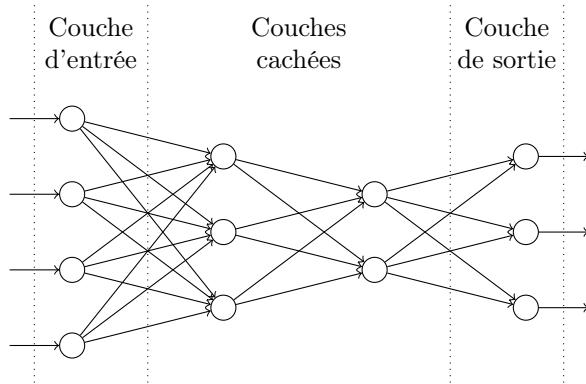


FIGURE 2.1 – Illustration d'un réseau de neurones

2.2 Support Vector Machine

Aspects historiques

Le modèle Support Vector Machine (SVM) est un modèle d'apprentissage supervisé. L'invention des SVM est relativement récente -tournant des années 1980-90- et due à Vladimir Vapnik qui propose comme un modèle permettant une classification binaire. Le modèle est rapidement devenu extrêmement populaire. Une simple recherche sur GoogleScholars pour le terme 'Support Vector Machine' renvoie à près de deux millions de documents référencant le terme.

Caractéristiques générales

Le modèle SVM est un modèle de classification. Il se focalise sur la recherche d'un plan ou hyperplan permettant une séparation maximale entre deux classes de données [24]. L'hyperplan frontière est défini par un vecteur qui lui est orthogonal et qui sert de 'support' (d'où le nom de support vector machine). Le modèle requiert que l'ensemble des données puisse être défini de manière à ce que 'tous les attributs [d'une donnée] puissent être convertis en nombres réels, indépendamment du fait qu'ils soient catégoriels, discrets, continus,... Ceci nous permet de manipuler des vecteurs attributs comme s'ils étaient représentés par des points dans un espace euclidien' [25]. En d'autres termes, une donnée X peut être représentée par :

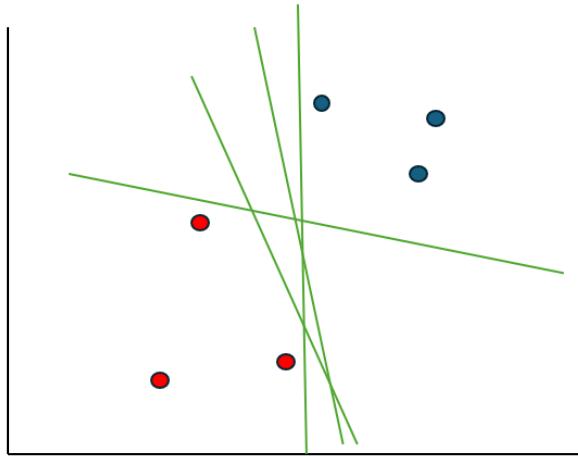
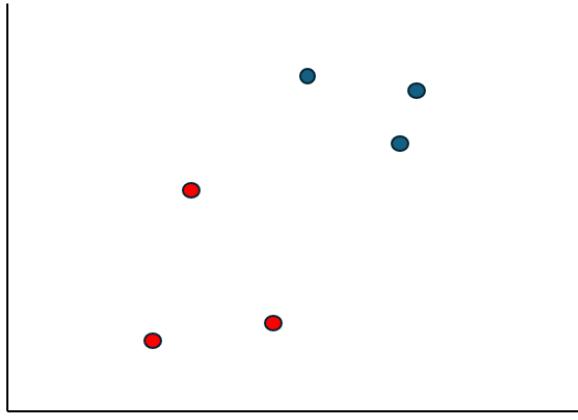
$$X = (x_1, x_2, \dots, x_m).$$

Le nombre de dimension des données (n) définit la dimension du plan qui les sépare (n-1).

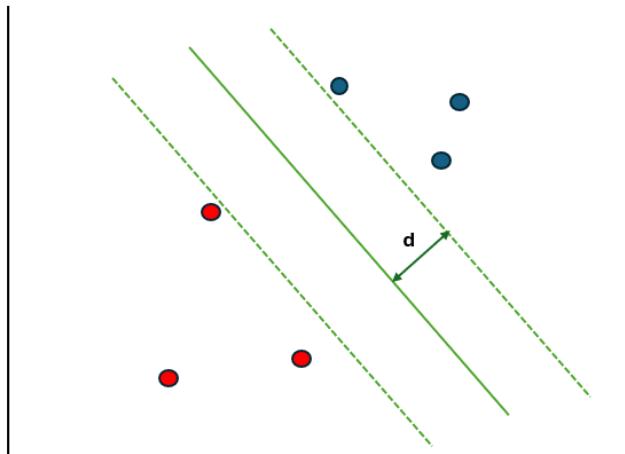
Aspects mathématiques

Soit un ensemble de données {X1, X2, ..., Xn}. L'idée sous-tendant le modèle SVM est de trouver une frontière rectiligne séparant les classes (le schéma ci-dessous pose pour deux classes, on prend rouge une classe - et bleue une classe +).

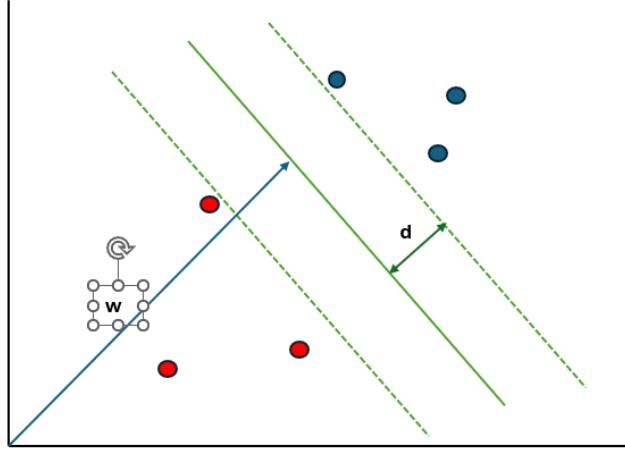
Le problème qui se pose est qu'une infinité de frontières rectilignes sont possibles. L'exemple donne une illustration de ce problème.



Le modèle SVM se propose de trouver la frontière permettant de former une marge maximale -ou ce que [26] dénomme ‘la rue la plus large’ (‘the widest street approach’)- entre les deux classes. En d’autres termes, en posant que la marge est de largeur d de chaque côté de la frontière, on cherche à maximiser d . Dans une modèle de marge linéaire, l’existence d’une marge telle que présentée ci-dessus peut dépendre du critère plus ou moins strict que l’on impose à la définition de la marge [27]. Dans un ensemble où les données positives et négatives sont au moins partiellement imbriquées les unes dans les autres, un critère de séparation stricte entre les classes empêche potentiellement de trouver une frontière. Un modèle plus souple permettant l’existence valeur aberrantes (*outliers*) est la plupart du temps nécessaire à la définition d’une marge et d’une frontière. En fin de compte, dans la construction du modèle, il s’agit de trouver un bon équilibre entre la possibilité de construire une marge maximale et le besoin de limiter le nombre de cas de violation de la marge (attribution à la mauvaise classe).



On peut alors définir la frontière à partir d’un vecteur w qui lui est orthogonal et dont on va chercher la norme.



Pour tout point, il est possible de définir un vecteur u . On s'intéresse alors à la projection de ce vecteur sur le vecteur w . Cette projection nous permet d'établir dans quelle partie de l'espace se trouve le point lié au vecteur u et donc de le lier à une prédition quant à sa classe.

En définissant un vecteur u associé à un des datapoints, on définit les datapoints positifs comme ceux vérifiant [26] :

$$\vec{u} \cdot \vec{w} \geq c$$

En travaillant sur l'ensemble d'apprentissage (labelisé), on définit les propriétés suivantes. Pour les datapoints connus positifs,

$$\vec{u}_+ \cdot \vec{w} - b \geq 1$$

Pour les datapoints connus négatifs,

$$\vec{u}_- \cdot \vec{w} - b \leq -1$$

Et en simplifiant, on choisit

$$y_i = 1$$

pour les datapoints connus positifs

$$y_i = -1$$

pour les datapoints connus négatifs.

On a donc

$$y_i \cdot (\vec{u}_+ \cdot \vec{w} - b) \geq 1$$

$$y_i \cdot (\vec{u}_+ \cdot \vec{w} - b) - 1 \geq 0$$

Les datapoints qui sont à la frontière de chaque côté de la ligne de partage vérifient [26] :

$$y_i \cdot (\vec{u}_+ \cdot \vec{w} - b) - 1 = 0$$

Il est intéressant de noter que l'addition de datapoints positifs ou négatifs du 'bon' côté de la marge n'a pas d'impact sur cette dernière ni sur la frontière qui est entièrement définie par les points de chaque classe qui sont les plus proches. On s'intéresse donc aux points sur les frontières négative et positive pour tenter d'estimer la largeur de la marge en projetant les vecteurs liés à ces deux points sur le vecteur w [26].

$$\begin{aligned} &(\vec{u}_+ - \vec{u}_-) \cdot \frac{\vec{w}}{\|\vec{w}\|} \\ &(\vec{u}_+ - \vec{u}_-) \cdot \frac{\vec{w}}{\|\vec{w}\|} = \frac{\vec{u}_+ \cdot \vec{w} - \vec{u}_- \cdot \vec{w}}{\|\vec{w}\|} \end{aligned}$$

Et donc

$$(\vec{u}_+ - \vec{u}_-) \cdot \frac{\vec{w}}{\|\vec{w}\|} = \frac{1 + b - (-1 + b)}{\|\vec{w}\|} = \frac{2}{\|\vec{w}\|}$$

On a donc trouvé une relation entre la largeur de la marge maximale et le vecteur supportant cette marge. Il s'agit ensuite de maximiser cette marge -ou de minimiser w . L'utilisation du multiplicateur de Lagrange permet de trouver une solution.

$$L = \frac{1}{2} \|\vec{w}\|^2 - \sum (a_i \cdot (y_i \cdot \vec{w} \cdot \vec{x}_i - 1))$$

En fin de compte, on obtient la règle de décision :

$$\sum (a_i \cdot (y_i \cdot \vec{w} \cdot \vec{x}_i + b)) \geq 0 \text{ pour les points positifs}$$

Il est intéressant de noter que certains ensembles de données ne présentent pas de frontière immédiatement identifiable par une SVM. Il est alors possible de projeter les données sur un espace de dimensions supérieur par une procédure nommée kernel function qui permet de calculer et maximiser la distance entre les datapoints.

2.3 Arbre de décision

Présentation générale

L'apprentissage par arbre de décision est un modèle d'apprentissage automatique qui utilise un arbre de décision comme modèle prédictif. L'algorithme permettant de construire ce genre d'arbre a été introduit par Leo Breiman en 1984 sous l'acronyme CART (*Classification And Regression Trees*). Son but est d'obtenir un modèle prédisant la valeur d'une variable de sortie à partir des valeurs des différentes variables d'entrée. D'autres algorithmes de construction d'arbres de décision sont apparus plus tard, comme ID3 (pour *Iterative Dichotomiser 3*) développé par Ross Quinlan en 1986, puis l'algorithme C4.5 (une version améliorée de ID3) élaboré en 1993 également par Quinlan.

Dans cette structure arborescente, une des variables d'entrée est sélectionnée à chaque noeud interne (i.e de degré > 1), et chaque arête vers un noeud-fils correspond à un ensemble de valeurs que peut prendre cette variable, de manière à ce que toutes ses valeurs possibles soient couvertes. Chaque feuille représente quant à elle soit une valeur de la variable de sortie, soit une distribution de probabilité de ses différentes valeurs possibles. Ainsi, la combinaison des valeurs des variables d'entrée permettant d'obtenir une certaine valeur/probabilité de la variable de sortie est représentée par le chemin de la racine jusqu'à la feuille en question.

L'arbre en lui-même est construit par partitionnement récursif, c'est-à-dire que l'on divise récursivement les données en sous-ensembles distincts en fonction de la valeur d'une certaine variable d'entrée. La récursion se termine dans l'un ou l'autre de ces cas :

- tous les sous-ensembles ont la même valeur pour cette variable d'entrée
- le partitionnement n'améliore plus la prédiction du modèle

Il s'agit donc d'un algorithme glouton (on réalise à chaque étape un choix optimum local afin d'obtenir un résultat optimum global) appelé **induction descendante d'arbres de décision** (ou *top-down induction of decision trees*). C'est la méthode la plus utilisée pour apprendre un arbre de décision à partir des données.

Mise en oeuvre

Comme évoqué précédemment, les algorithmes de construction d'arbres de décision consistent généralement en divisions successives de l'arbre de la racine vers les feuilles en choisissant à chaque étape la variable d'entrée qui réalise le partage optimal de l'ensemble des données, i.e celle qui maximise un critère donné. On l'appelle **variable de segmentation**.

Il existe différents critères d'évaluation qui mesurent l'homogénéité des sous-ensembles obtenus par une partition via une variable de segmentation. Les plus courants sont l'indice de diversité de Gini, l'indice d'entropie de Shannon et leurs variantes.

L'indice de diversité de Gini, utilisé par l'algorithme CART, mesure avec quelle fréquence un élément aléatoire de l'ensemble serait mal classé si son étiquette de classe était choisie aléatoirement selon la distribution des étiquettes dans le sous-ensemble. Il peut donc être calculé en sommant pour chaque élément de l'ensemble la probabilité qu'il soit choisi multipliée par la probabilité qu'il soit dans la mauvaise classe. Dans la pratique, on utilise la formule suivante pour un cas de classification binaire :

$$I_G(f) = \sum_{i=1,2} f_i(1 - f_i) = \sum_{i=1,2} (f_i - f_i^2) = \sum_{i=1,2} f_i - \sum_{i=1,2} f_i^2 = 1 - \sum_{i=1,2} f_i^2$$

où f_i désigne la proportion des éléments de l'ensemble avec l'étiquette i (puisque la probabilité sur les éléments de l'ensemble est la probabilité uniforme).

Comme mentionné dans la section précédente, à chaque ensemble de valeurs de la variable de segmentation correspond un noeud-fils. Cependant, les algorithmes diffèrent sur le nombre de noeud-fils produits, et donc sur la façon de fractionner les ensembles de valeurs de cette variable. L'algorithme CART produit uniquement

des arbres binaires, ce qui signifie qu'il détermine la partition binaire optimale des valeurs de la variable de segmentation. D'autres algorithmes déterminent une partition optimale selon des critères statistiques, et elle n'est donc pas nécessairement binaire (exemple : algorithme CHAID). Ainsi, la largeur de l'arbre dépend de l'algorithme de construction choisi. Mais pour que l'arbre soit efficace, il ne faut pas partitionner à l'excès les données, sans quoi des groupes d'effectifs trop faibles seront produits qui ne correspondent à aucune réalité statistique. On risquerait alors de tomber dans le piège de l'*overfitting* (voir section 2.4.1).

En effet, plus les feuilles sont homogènes par rapport à la variable de sortie, i.e tous les éléments d'une feuille ont exactement la même étiquette de classe (cas limite : il existe une feuille pour chaque élément), plus l'arbre est volumineux et complexe. Il commettra alors peu ou pas d'erreur sur les données d'apprentissage, mais ne se généralisera pas bien à de nouveaux échantillons. L'enjeu est donc de construire l'arbre le plus petit possible, car plus un arbre sera simple, mieux il se généralisera. Il faut donc maintenir un équilibre entre performance et complexité dans la construction de nos arbres de décision.

Une technique efficace pour contrôler la taille de l'arbre et d'éviter ainsi l'*overfitting* est **l'élagage des arbres de décision**. En effet, il est possible de stopper la croissance de l'arbre en spécifiant des critères d'arrêt brutaux lors de la phase de construction, mais l'élagage est un moyen beaucoup plus fiable. Pour cela, il faut séparer notre ensemble de données initial en un ensemble de données servant à construire l'arbre (base d'entraînement), et un autre ensemble servant à l'élaguer (sous-ensemble des données de test) appelé **échantillon d'élagage**. Le processus d'élagage prend ainsi en entrée notre arbre et l'échantillon d'élagage, et consiste à supprimer tous les sous-arbres qui n'améliorent pas la précision des prédictions du modèle sur cet ensemble de données. On appelle également ce processus post-élagage car il est effectué après la construction de l'arbre, par opposition au pré-élagage qui correspond aux critères d'arrêt lors de la construction de l'arbre mentionnés précédemment.

Enfin, la classe attribuée à chaque feuille est déterminée en fonction de la classe majoritaire chez les éléments qui la composent, i.e de la classe la plus représentée (méthode optimale dans le cas d'un échantillon de données non-biaisé, or c'est une hypothèse fondamentale de l'apprentissage supervisé).

Points forts du modèle

Tout d'abord, les arbres de décision sont très faciles à comprendre et à interpréter une fois construits. De plus, le pré-traitement des données à effectuer pour ce type de modèle est très léger (pas de normalisation, de valeurs vides à supprimer, ou de variables muettes). Et au contraire des réseaux neuronaux par exemple, le modèle peut traiter des données numériques mais aussi des catégories. Des tests statistiques peuvent également être construits afin de rendre compte de la fiabilité du modèle. Enfin, les arbres de décision sont performants sur de grands jeux de données car ils consomment relativement peu de ressources de calcul.

Inconvénients et limites

Les arbres de décision sont sujets à l'*overfitting* et se généralisent mal, bien que l'élagage pallie grandement à ce problème. De plus, ce modèle est très sensible aux petites variations sur les données, c'est-à-dire que de petites différences sur celles-ci peuvent produire des arbres très différents. Certains concepts sont également difficiles à traduire en termes d'arbres de décision, comme par exemple le XOR ou la parité, ce qui produit des arbres extrêmement larges. Enfin, la recherche de l'arbre de décision optimal est NP-complet concernant plusieurs aspects de l'optimalité, ce qui entraîne le fait que les algorithmes d'apprentissage par arbre de décision sont basés sur des algorithmes gloutons comme vu précédemment, et ne garantissent donc pas de trouver l'optimum global.

2.4 Forêt aléatoire

Historique

Introduite en 2001 par Leo Breiman, la Random Forest ou Forêt aléatoire est un algorithme de machine learning faisant partie des "ensemble learners", c'est-à-dire dont le fonctionnement repose sur plusieurs modèles de machine learning sous-jacents, à savoir ici dans le cas de la Forêt aléatoire le modèle de l'Arbre de décision (Decision Tree).

L'algorithme de Forêt aléatoire a été créé dans le but d'améliorer l'Arbre de décision en corrigeant ses principaux défauts que sont :

- sa tendance à ne pas utiliser toutes les dimensions d'un jeu de données.
- son potentiel d'optimisation faible malgré l'ajustement des hyperparamètres.
- sa haute variance ou overfitting, autrement dit l'influence très importante qu'ont les données d'entraînement sur les prédictions du modèle au détriment de sa capacité à généraliser à partir de nouvelles données.

Fonctionnement

La Random Forest ressemble beaucoup à l'algorithme Bagging Tree qui lui aussi implémente un ensemble d'arbres de décision. Ces deux méthodes succèdent d'ailleurs à l'arbre de décision.

Cependant la Random Forest possède une particularité : à chaque split dans l'un des arbres de décision, l'algorithme sélectionne aléatoirement une partie des dimensions du jeu de données, laissant à l'arbre la possibilité de splitter uniquement sur ces dimensions précises.

Ainsi, supposons que nous ayons un jeu de données de dimension k , alors à chaque split un arbre de décision aura le choix entre n dimensions du jeu de données choisies aléatoirement parmi les k dimensions, avec $n < k$. Nous reviendrons sur le nombre de dimensions choisies à propos des hyperparamètres.

Par ailleurs, chaque arbre de la forêt est entraîné sur une partie seulement des lignes de données du jeu de données qui sont elles aussi choisies aléatoirement.

L'intérêt de cette technique est qu'elle permet de décorreler les arbres de décision, réduisant ainsi la variance du modèle. En effet, choisir de façon aléatoire un nombre limité de dimensions sur lesquelles l'arbre va pouvoir splitter et n'entraîner le modèle que sur une partie des données permet d'obtenir des arbres très différents les uns des autres.

Comment la Forêt aléatoire choisit-elle la prédiction finale ? Comme notre problème ne concerne que la classification, nous n'évoquerons ici que ce cas précis, bien que la Random Forest soit également utilisable pour la régression. Pour choisir le résultat final, l'algorithme effectue un vote à la majorité : la classe la plus récurrente prédite par les arbres de décision désignera la prédiction \hat{y} de la ligne de données.

Bootstrapping : créer des échantillons aléatoires du jeu de données.

Bagging : acronyme pour (Bootstrap aggregation), entraîne chaque arbre sur un des échantillons créés par le bootstrapping.

Hyperparamètres

Une partie des hyperparamètres de la Random Forest sont les mêmes que pour le Decision Tree. Cependant, 4 paramètres sont spécifiques à l'algorithme de Random Forest et importants à connaître :

- le nombre d'estimateurs
- le nombre de dimensions sélectionnées
- la métrique *Out-of-Bag*
- l'application ou non de bootstrapping

Nombre d'estimateurs

Le nombre d'estimateurs est le paramètre qui va décider du nombre d'arbres qui vont être créés à l'exécution de l'algorithme. Il est alors nécessaire de choisir un nombre d'arbres assez élevé pour maximiser la possibilité d'obtenir une prédiction juste. Il faut également noter qu'à partir d'un certain nombre d'arbres l'algorithme atteindra un seuil de saturation à partir duquel l'augmentation de ce paramètre ne sera plus en mesure d'améliorer

les performances du modèle. Cela s'explique simplement par le fait qu'au fil de l'ajout de nouveaux arbres, ces derniers seront de plus en plus similaires, jusqu'à ce qu'ils n'entraînent plus l'apport de nouvelles informations.

Nombre de dimensions selectionnées

Ce paramètre représente le nombre de dimensions que va choisir aléatoirement l'algorithme à chaque split. C'est un paramètre qu'il est très important de choisir soigneusement car il est responsable de la diversité des arbres de la forêt. Il est conseillé, pour la classification, de définir ce paramètre par \sqrt{n} avec n le nombre de dimensions du jeu de données [28]. À noter qu'il s'agit seulement de la valeur recommandée par défaut et qu'il faudra l'adapter selon les spécificités du jeu de données.

Bootstrapping

Ce paramètre concourt aussi à diversifier la forêt car, une fois activé, il va permettre d'entraîner les arbres uniquement sur un sous-ensemble du jeu de données d'entraînement où les lignes seront sélectionnées de manière aléatoire et avec remise.

Métrique Out-Of-Bag

Il s'agit d'une métrique qui permet de calculer l'erreur du modèle en utilisant les lignes de données qui n'ont jamais été sélectionnées pour l'entraînement des arbres par le bootstrapping.

Attention : l'Out-Of-Bag suppose que le Bootstrapping soit activé.

Chapitre 3

Cas d'étude : Titanic

3.1 Dataset Titanic

Les données que nous utilisons dans ce cas d'étude proviennent du site de compétition en machine learning Kaggle [29]. Il met à disposition un dataset d'entraînement avec les informations d'environ 900 passagers. Les colonnes de ce dataset sont :

- PassengerId : l'identifiant du passager
- Pclass : la répartition des passagers en 3 classes (1ère, 2nde, 3e), celles-ci nous permettant d'en déduire leur situation socio-économique
- Name : le nom complet du passager
- Sex : le genre du passager
- Age : l'âge du passager
- Sibsp : le nombre de frères et soeurs et époux/épouses à bord
- Parch : le nombre d'enfants et de parents à bord
- Ticket : l'identifiant du ticket
- Fare : le tarif du trajet
- Cabin : l'identifiant de la cabine du passager
- Embarked : le port d'embarquement du passager
- Survived : si le passager a survécu au drame

Nous étudierons plus en détails ces colonnes dans la partie concernant le prétraitement.

3.2 Métriques d'évaluation

En machine learning, à chaque type de problème est associé un certain nombre de métriques afin d'en savoir davantage sur un modèle. Dans notre étude, nous serons amenés à ranger des données dans 2 classes distinctes. Il s'agit là d'un problème de classification binaire. Nous allons donc tout d'abord présenter les différentes métriques que nous utiliserons tout au long de ce chapitre afin d'évaluer les performances de nos modèles :

- L'accuracy score
- La confusion matrix
- Le classification report

L'accuracy score calcule de façon générale la précision du modèle sur nos données de test. Elle est la métrique la plus fréquemment utilisée pour la classification binaire. Plus cette valeur est élevée et plus les prédictions du modèle sont correctes. La formule utilisée est la suivante :

$$\frac{\text{nombre de prédictions réussies}}{\text{nombre de prédictions totales}}$$

La confusion matrix permet de visualiser sous forme de matrice les prédictions et de déceler les erreurs de classification. Dans cette structure, chaque ligne et chaque colonne représentent une des classes à prédire du jeu de données. Elle nous permet de voir où s'est trompé notre modèle dans ses prédictions et quelles furent ses erreurs : avec quelle(s) classe(s) a-t-il confondu une autre classe (d'où le nom matrice de "confusion"). Dans notre cas, comme nous n'avons que 2 classes 0 et 1, la matrice est de taille 2×2 . Cependant, il faut garder à l'esprit que la matrice de confusion est aussi utilisable dans des problèmes de type multi-classes. Pour n classes elle est de taille $n \times n$. Les données correctement classifiées sont représentées sur la diagonale de la matrice. Voici en image un exemple de matrice de confusion binaire :

		Actual Values	
		Positive (1)	Negative (0)
Predicted Values	Positive (1)	TP	FP
	Negative (0)	FN	TN

Dans un problème de classification binaire, il y a 4 états possibles pour chaque prédiction réalisée par un modèle :

- True Positive (TP) - La prédiction du modèle est la classe 1 et la donnée appartient en effet à la classe 1
- False Positive (FP) - La prédiction du modèle est la classe 1 mais la donnée appartient à la classe 0
- True Negative (TN) - La prédiction du modèle est la classe 0 et la donnée appartient en effet à la classe 0
- False Negative (FN) - La prédiction du modèle est la classe 0 mais la donnée appartient à la classe 1.

Il s'agit de l'ensemble des informations qui composent la matrice de confusion [30]. Elles nous seront par ailleurs très utiles à la compréhension de la métrique suivante.

Le classification report décrit de manière plus détaillée les performances d'un modèle. Il nous donne accès à 3 informations différentes : la précision, le recall et le f1-score pour chaque classe. Tout comme pour la matrice de confusion, il est possible d'utiliser le classification report dans un problème multi-classes.

La précision permet de mesurer le taux de true positive d'une classe à partir du nombre de données ayant été prédites comme appartenant à cette dernière. Elle se calcule comme ceci :

$$\frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}}$$

Le recall, aussi appelé sensibilité d'un modèle, permet de mesurer la proportion d'éléments correctement prédits pour une classe, c'est-à-dire son taux de true positive :

$$\frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}}$$

Pour finir, nous avons le F1-score qui combine les 2 informations décrites précédemment. Il mesure la moyenne harmonique de la précision et du recall. Il se calcule de la manière suivante :

$$\frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

À noter que plus le F1-score est élevé, plus le modèle a de hautes performances que ce soit en précision ou en recall.

3.3 Prétraitement

Les algorithmes sélectionnés sont particulièrement pratiques en ce sens qu'ils ne nécessitent pas, au contraire d'autres algorithmes, un prétraitement trop lourd des données. Il n'y a, par exemple, pas besoin de faire de feature scaling, c'est-à-dire de normaliser ou standardiser les données.

Voici l'état initial du jeu de données :

```

df.isnull().sum()

```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	dtype: int64
	0	0	0	0	0	177	0	0	0	0	687	2	

Premièrement, nous remarquons que la colonne Cabin est majoritairement composée de valeurs manquantes. Une solution consistait à remplir cette colonne avec des valeurs quelconques. Cependant, estimant que le numéro de cabine des passagers n'apportait pas un supplément d'information exploitable en l'état et que les compléter avec des valeurs quelconques n'apporterait rien à la prédiction (ce qui aurait pu avoir un intérêt si nous avions disposé des données réelles qui nous auraient permis de connaître la localisation, à tribord ou bâbord par exemple, des passagers), nous avons choisi de supprimer cette colonne.

Nous avons également choisi de supprimer les colonnes Name, PassengerId et Ticket, celles-ci n'ayant pas d'intérêt pour l'entraînement de nos algorithmes.

Deuxièmement, nous remarquons qu'il manque un certain nombre de valeurs dans la colonne Age. L'âge nous semblant en revanche une donnée et un facteur importants dans la prédiction des chances de survie des passagers, nous avons jugé nécessaire de conserver cette colonne. Deux options s'offraient alors :

- supprimer les lignes où l'âge n'était pas spécifié
- imputer les valeurs manquantes

En considération du nombre déjà limité de lignes du jeu de données, nous avons opté pour la seconde solution et pris comme valeur la moyenne de l'âge des passagers à bord du Titanic.

Enfin, 2 valeurs manquaient dans la colonne du port d'embarquement que nous avons pu retrouver sur Internet.

Les bibliothèques d'implémentation des algorithmes ne supportant pas les chaînes de caractères, il faut par ailleurs transformer celles qui nous intéressent sous forme de nombres que l'algorithme peut interpréter. Nous avons donc à modifier les colonnes Age et Embarked.

Nettoyage des données

```

# Importation du jeu de données
df = pd.read_csv("../titanic/train.csv")

# Suppression de la colonne Cabin
df = df.drop("Cabin", axis=1)

# Remplissage de la colonne Age avec l'âge médian des passagers
df["Age"] = df["Age"].fillna(round(df["Age"].mean()))

# Ajout du port d'embarquement manquant pour les 2 passagers
df.at[122, "Embarked"] = "S"
df.at[141, "Embarked"] = "S"

# Suppression des colonnes Name, PassengerId, Ticker qui ne nous intéresse pas
df = df.drop(["Name", "PassengerId", "Ticket"], axis=1)

# Conversion des chaînes de caractère en valeurs booléennes
df = pd.get_dummies(df, drop_first=True)

```

3.4 Application réseau de neurones

Application concrète sur le jeu de données

Premièrement, détaillons les bibliothèques employées :

- *pandas*, qui va nous permettre de manipuler plus aisément le jeu de données.
- *sklearn*, dont la fonction *StandardScaler* nous permettra d'harmoniser la répartition des données pour leur future manipulation.
- *keras*, dont les classes *models* et *layers* nous permettront d'instancier le réseau de neurones.

```
#Importation de la bibliothèque pandas pour gérer le jeu de données
import pandas as pd

#Importation de StandardScaler de la bibliothèque sklearn afin d'harmoniser le jeu de données
from sklearn.preprocessing import StandardScaler

#Importation des classes models et layers de la bibliothèque keras afin d'instancier le réseau de neurones
from keras import models
from keras import layers
```

Réseau de neurones – bibliothèques employées

Une fois le prétraitement des données effectuées, nous séparons les données relatives aux passagers, qui nous serviront à déterminer si ceux-ci ont survécu ou non (toutes les colonnes sauf *Survived*), des données nous indiquant s'ils ont effectivement survécu au naufrage (colonne *Survived*).

En suivant, nous utilisons *StandardScaler* pour harmoniser la répartition des données ; nous avons en effet plusieurs échelles différentes pour les données numériques (l'âge n'est par exemple pas réparti sur la même échelle que la classe de voyage), ce qui peut nuit à l'efficacité du réseau de neurones. Nous devons ensuite transformer la liste retournée en *dataframe* pour assurer le fonctionnement de notre code.

```
#Crée une variable contenant toutes les colonnes servant au réseau de neurones
X = training_data[training_data.columns.drop(['Survived'])]

#Crée une variable contenant uniquement les valeurs de survie des passagers
Y = training_data['Survived']

#Mise à l'échelle du jeu de données afin d'améliorer les performances du réseau de neurones
X = pd.DataFrame(StandardScaler().fit_transform(X), columns=X.columns)
```

Séparation des données dans deux variables, et mise à l"échelle

Nous pouvons maintenant créer notre réseau de neurones, selon le schéma suivant :

- 7 neurones d'entrée pour chacune des variables du jeu de données.
- 21 neurones de calcul en première couche intermédiaire afin d'assurer un brassage conséquent.
- 10 neurones de calcul en deuxième couche intermédiaire afin de mutualiser les résultats.
- 1 neurone de sortie s'activant selon un mode différent pour renvoyer 0 ou 1

```
#Création d'un modèle de réseau de neurones
model = models.Sequential()

#Ajout des neurones par couches
#Première couche : 7 neurones d'entrée, 21 neurones de calcul connectés aux 7 d'entrée
model.add(layers.Dense(21, input_dim=7, activation='relu'))
#Deuxième couche : 10 neurones de calcul interconnectés aux 21 précédentes
model.add(layers.Dense(10, activation='relu'))
#Troisième et dernière couche : 1 neurone de sortie (fonction d'activation renvoyant 0 ou 1)
model.add(layers.Dense(1, activation='sigmoid'))
```

Construction du réseau de neurones

Il nous faut à présent entraîner notre réseau de neurones. Pour cela, nous le compilons, puis lui faisons effectuer un certain nombre de répétitions sur des groupes d'individus afin qu'il puisse réduire la marge d'erreur. Plusieurs tests ont démontré que la fonction de perte *binary_crossentropy* est la plus adaptée à notre jeu de données. Des *optimizer* auraient également pu être précisés, mais leur impact s'est révélé négligeable.

Cependant, le choix a été fait d'afficher l'*accuracy* et le *mse* des répétitions, afin de disposer de moyens de comparaison des différentes phases de test.

Les paramètres de répétition ont fait l'objet de nombreux tests, et les valeurs présentées (5000 répétitions, 32 individus par phase de test) sont un bon compromis entre un programme trop long à l'exécution, et un programme plus court mais plus imparfait.

```
#Compilation du modèle selon les données renseignées
model.compile(loss='binary_crossentropy', metrics=['accuracy', 'mse'])

#Entraînement du modèle, sur 5000 répétitions, par lots de 32 individus
history=model.fit(X, Y, epochs=5000, batch_size=32)
```

Entraînement du réseau de neurones sur le jeu de données

Nous pouvons enfin tester le réseau de neurones en le compilant. Pour simplifier l'algorithme, nous effectuons la compilation sur le jeu de données précédemment prétraité. Nous pourrions tout à fait, et cela a été réalisé en parallèle, l'effectuer sur le jeu de données de test fourni, ou sur d'autres jeux de données, pourvu que le prétraitement soit similaire.

Une fois les valeurs de sortie du réseau de neurones arrondies et stockées, nous pouvons les comparer avec les valeurs attendues, et afficher le nombre de différences.

Les réglages fournis plus haut font état de 90% de valeurs justes, et nous pouvons monter à 95% avec 25000 répétitions, mais le temps d'exécution de l'algorithme est alors de l'ordre de 15 min, contre environ 3 min pour 5000 répétitions.

```
#Test du modèle, et stockage des valeurs de retour dans une liste
y_prime = [round(x[0]) for x in model.predict(X)]

#Vérification du nombre de différences entre les valeurs attendues et les valeurs effectives
differences=0
for i in range(Y.shape[0]):
    if y_prime[i] != Y[i]:
        differences+=1

#Affichage des différences observées
print("Différences : ",differences)
```

Test du réseau de neurones et affichage des résultats

3.5 Application Support Vector Machine (SVM)

Application avec Python sklearn

Une tentative d'application est présentée dans ce qui suit.

Une première tentative a été conduite sans ajustement à partir du code suivant (code complet incluant la matrice de confusion correspondant au fichier '*Clean Titanic Code SVM.py*').

```

42 # Ouvre le fichier et le nettoie
43 train_df = pandas.read_csv('C:/Users/yvesh/OneDrive/Documents/Python Scripts/Titanic/train.csv')
44 train_df = nettoyage(train_df)
45
46 # Prepare les donnees pour le module sklean.svm
47 X = train_df[['PassengerId', 'Sex', 'Pclass', 'Age', 'Fare', 'Embarked',
48 'Parch', 'SibSp']]
49 y = train_df['Survived']
50
51 # Fit le classificateur avec les donnees nettoyees precedentes.
52 classificateur_SVM = svm.SVC()
53 classificateur_SVM.fit(X, y)
54
55 # Joint les tableaux de test (donnees et resultat) pour le nettoyage.
56 test_df = pandas.read_csv('C:/Users/yvesh/OneDrive/Documents/Python Scripts/Titanic/test.csv')
57 alive_df = pandas.read_csv('C:/Users/yvesh/OneDrive/Documents/Python Scripts/Titanic/gender_submission.csv')
58 test_with_result = test_df.merge(alive_df, how = 'inner', on = 'PassengerId')
59
60 # Applique la fonction de nettoyage
61 test_with_result = nettoyage (test_with_result)
62
63 # Prepare les donnees pour le module sklean.svm
64 test_final = test_with_result[['PassengerId', 'Sex', 'Pclass', 'Age', 'Fare', 'Embarked',
65 'Parch', 'SibSp']]
66 survived_final = test_with_result['Survived']
67
68 # Applique le predict du module sklearn.svm aux donnees test.
69 prediction = classificateur_SVM.predict(test_final)
70
71 # Cree un tableau comparant les predictions et les valeurs reelles.
72 compare = pandas.DataFrame ({'Prediction': prediction, 'Actual': survived_final})
73 compare['Score'] = compare['Prediction'] + compare ['Actual']
74 num_fail = compare['Score'].value_counts()[1]
75

```

FIGURE 3.1 – Code SVM sans ajustement

Ce premier essai est peu concluant du point de vue de sa fiabilité avec seulement 64.75 pour cent de prédictions correctes. La matrice de confusion montre notamment une surreprésentation des cas de personnes ayant survécues mais prédites mortes. Le problème est probablement en partie du à l’overfitting du modèle sur les données d’entraînement.

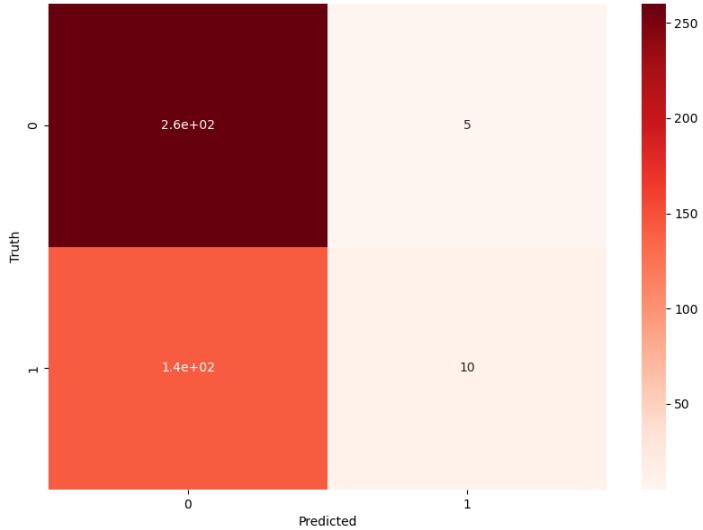


FIGURE 3.2 – SVM configuration basique

Le second essai joue sur la normalisation de l’ensemble des attributs, les SVMs étant sensibles à l’existence d’échelles différentes. D’autre part, nous jouons sur le paramètre C qui permet d’accepter dans l’ensemble d’entraînement certaines violations de frontières (code complet incluant la matrice de confusion correspondant au fichier ‘Clean Titanic Code SVM Alternate.py’).

```

43 # Ouvre le fichier et le nettoie
44 train_df = pandas.read_csv('C:/Users/yvesh/OneDrive/Documents/Python Scripts/Titanic/train.csv')
45 train_df = nettoyage(train_df)
46
47 # Prepare les donnees pour le module sklean.svm
48 X = train_df[['PassengerId', 'Sex', 'Pclass', 'Age', 'Fare', 'Embarked',
49               'Parch', 'SibSp']]
50 y = train_df['Survived']
51
52 #le scaler permet de normer l'ensemble des donnees sur [0, 1]
53 std_scaler = StandardScaler()
54 X = std_scaler.fit_transform(X)
55 y = train_df['Survived']
56
57 # Le classificateur inclut cette fois un parametre C permettant d'ajuster les
58 # violations de marge dans le set d'entraînement et d'éviter l'hyperfitting.
59
60 classificateur_SVM = svm.SVC(kernel = 'poly', C = 0.7)
61 classificateur_SVM.fit(X, y)
62
63 # Joint les tableaux de test (donnees et resultat) pour le nettoyage.
64 test_df = pandas.read_csv('C:/Users/yvesh/OneDrive/Documents/Python Scripts/Titanic/test.csv')
65 alive_df = pandas.read_csv('C:/Users/yvesh/OneDrive/Documents/Python Scripts/Titanic/gender_submission.csv')
66 test_with_result = test_df.merge(alive_df, how = 'inner', on = 'PassengerId')
67
68 # Applique la fonction de nettoyage
69 test_with_result = nettoyage (test_with_result)
70
71
72 # Prepare les donnees pour le module sklean.svm
73 test_final = test_with_result[['PassengerId', 'Sex', 'Pclass', 'Age', 'Fare', 'Embarked',
74                               'Parch', 'SibSp']]
75 test_final = std_scaler.transform(test_final)
76 survived_final = test_with_result['Survived']
77
78 # Applique le predict du module sklearn.svm aux donnees test.
79 prediction = classificateur_SVM.predict(test_final)
80
81 # Cree un tableau comparant les predictions et les valeurs reelles.
82 compare = pandas.DataFrame ({'Prediction': prediction, 'Actual': survived_final})
83 compare['Score'] = compare['Prediction'] + compare['Actual']
84 num_fail = compare['Score'].value_counts()[1]

```

FIGURE 3.3 – Code de la SVM avec scaler et ajustement de C

Le résultat obtenu est plus convaincant : le taux de prédictions juste monte à 94.24 pourcent, avec une matrice de confusion qui montre un modèle relativement équilibré.

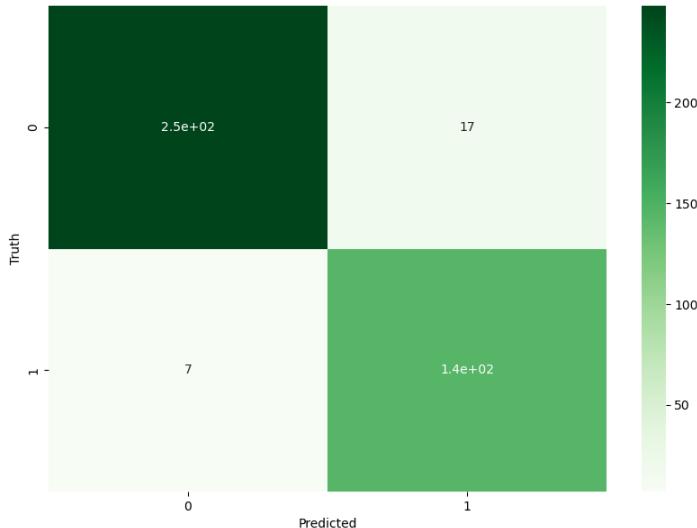


FIGURE 3.4 – SVM : 2e tentative avec scaler et ajustement de C

3.6 Application arbre de décision

Application à un cas concret : le Titanic

Le prétraitement des données est identique à celui de l'algorithme Random Forest (voir section 5.5.1). Le langage utilisé ici est Python, et les modules importés sont Scikit-learning pour l'algorithme *Decision Tree Classifier* et Graphviz pour l'affichage de l'arbre de décision.

```

Entrée [3]: from sklearn.model_selection import train_test_split
from sklearn import tree
from sklearn.metrics import confusion_matrix
import graphviz

# colonne utilisée comme support pour la prédiction du modèle
X = data.drop("Survived", axis=1)

# information que l'on veut prédire
y = data["Survived"]

# création des bases d'entraînement et de test avec #25% du volume total de données utilisé en test et les 75%
# restants pour entraîner l'algorithme, et random_state sert à mélanger les données avant d'appliquer l'algorithme
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=0)

# création du modèle en utilisant le critère de Gini, avec la variable de segmentation étant choisie comme la
# meilleure partition selon ce critère (splitter='best')
dtm = tree.DecisionTreeClassifier(criterion='gini', splitter='best', random_state=0)

# apprentissage de l'algorithme
dtm.fit(X_train, y_train)

# création fichier au format .dot pour affichage
with open("dtm.dot", 'w') as f:
    f = tree.export_graphviz(dtm, out_file=f, filled=True)

# évaluation performance modèle
dtm.score(X_test, y_test)

```

Out[3]: 0.7668161434977578

FIGURE 3.5 – implémentation d'un arbre de décision

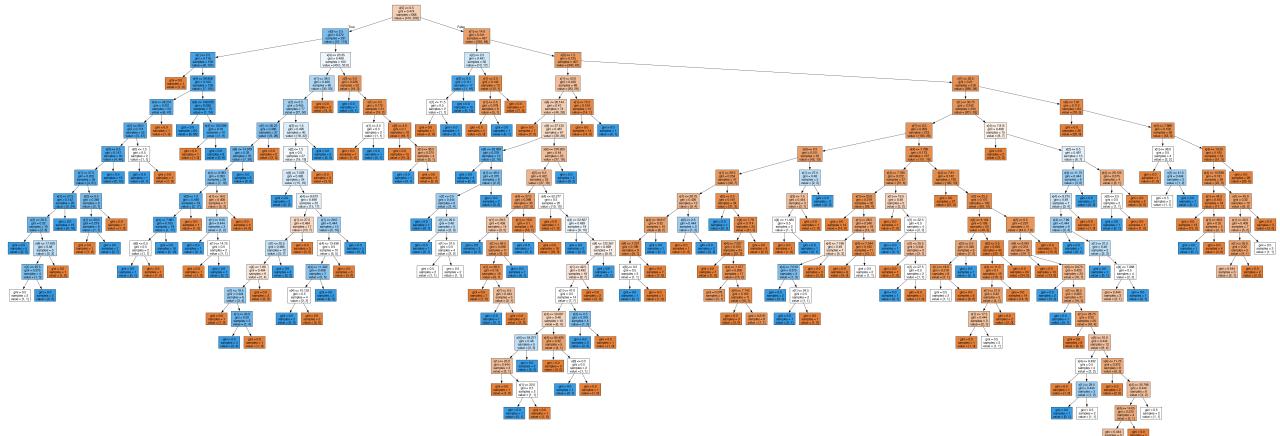


FIGURE 3.6 – arbre de décision produit

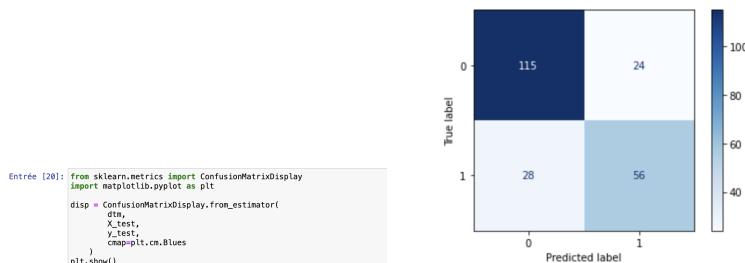


FIGURE 3.7 – matrice de confusion du modèle

3.7 Application forêt aléatoire

Pour cette partie, nous utiliserons le langage de programmation Python avec les bibliothèques Numpy et Pandas pour la préparation des données, Matplotlib et Seaborn pour la visualisation des données et Scikit-learning pour l'application de l'algorithme Random Forest.

Utilisation de l'algorithme

Afin d'appliquer l'algorithme et l'évaluer par la suite, nous devons diviser notre jeu de données en 2 parties : une partie d'entraînement qu'utilisera l'algorithme et une partie de test pour juger de son efficacité.

Pour obtenir les meilleures prédictions possibles, il faut adapter les hyperparamètres à notre jeu de données. J'ai donc effectué une grid search sur les hyperparamètres concernant le nombre d'arbres de la forêt et le nombre de dimensions sélectionnées aléatoirement.

La grid search va entraîner un modèle avec toutes les combinaisons de paramètres qu'on lui donne pour trouver lesquelles produisent le meilleur résultat. Après avoir appliqué cette technique, j'ai constaté que le meilleur modèle utilise 400 arbres avec 4 dimensions choisies aléatoirement.

Préparation du modèle

```
# Colonnes utilisées pour la prédiction
X = df.drop("Survived",axis=1)

# Colonne que l'on veut prédire
y = df["Survived"]

# Séparation des données en échantillons d'entraînement et échantillon de test
train_X, verif_X, train_y, verif_y = train_test_split(X,y,random_state=9)

# Creation d'un modèle avec une seed
rfc = RandomForestClassifier(random_state=99)

# Recherche de la meilleure combinaison de paramètres:

# array allant de 100 jusqu'à 1000 avec un pas de 100 (test de 100 à 1000 arbres)
n_estimators = np.arange(100,1001,100)

# test du nombre de dimension choisies aléatoirement allant de 2 à 6 (7 n'est pas inclus)
max_features = np.arange(2,7)

# Dictionnaire qui stock les paramètres à tester
params = {"n_estimators":n_estimators,"max_features":max_features}

# creation de la grid search avec nos paramètres
most_accurate_model = GridSearchCV(rfc,param_grid=params)

# exécution de la grid search
most_accurate_model.fit(train_X,train_y)

# prédiction avec le meilleure modèle
predictions = most_accurate_model.predict(verif_X)
```

Après avoir appliqué le modèle pour prédire les données de test, nous pouvons passer à l'étape suivante.

Mesure de précision

Pour mesurer la précision de notre modèle, j'ai utilisé 3 outils :

- L'accuracy score
- La confusion matrix
- La classification report

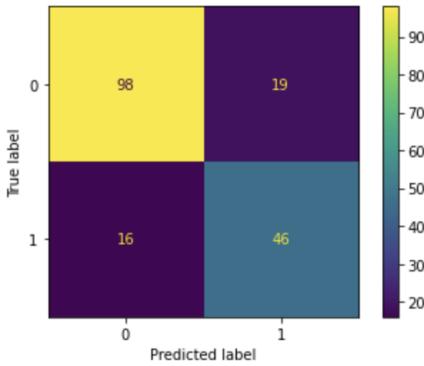
L'accuracy score calcule de façon générale la précision du modèle sur nos données de test. La formule utilisée est la suivante :

$$\frac{\text{nombre de predictions réussies}}{\text{nombre de predictions totales}}$$

Notre modèle atteint un score 81%.

La classification report décrit de manière plus précise les performances de notre modèle. Il nous donne accès à la précision, le recall et le f1-score pour chaque classe.

La confusion matrix permet de visualiser sous forme de matrice les prédictions et de déceler les erreurs de classification. Voici celle de notre modèle :



La première ligne nous indique qu'il y a dans nos données de test un total de 115 voyageurs de classe 0 (morts). Nous voyons que 96 voyageurs qui n'ont pas survécu au naufrage ont été correctement classifiés. Cependant 19 d'entre eux ont été classifiés dans les survivants (classe 1). Pour la deuxième ligne, on recense 62 voyageurs de classe 1 (survivants). Parmi ceux-ci, 46 ont été correctement classifiés. Par contre, 16 d'entre eux ont été classifiés dans les morts (classe 0).

On peut en conclure que le modèle que nous avons produit a plus de mal à identifier les survivants parmi les passagers. Cette matrice est donc un outil pratique car elle nous permet d'identifier les faiblesses d'un modèle (pour les problèmes de la classification).

3.8 Discussion

Enfin, comparons les performances des algorithmes. Le plus simple à implémenter et le plus léger en terme de ressources de calcul est évidemment l'arbre de décision, mais il s'agit également du modèle le moins fiable avec un score de précision de 76,7%. Cela est notamment dû au problème de l'overfitting quasiment inévitable avec ce type de modèle.

L'algorithme le plus performant suivant est la forêt aléatoire avec un score de précision de 81%, et en effet ce modèle permet de décorrélérer les arbres entre eux, réduisant ainsi la variance du modèle et le risque de surapprentissage. Cependant, l'algorithme a plus de mal à identifier les survivants parmi les passagers, et il est assez chronophage puisque le calcul des données est effectué pour chaque arbre de décision.

Ensuite, l'algorithme le plus performant est la SVM (*Support Vector Machine*) avec un score de précision de 94% après normalisation des données et ajustement du paramètre contrôlant l'acceptation des violations de frontières. Le modèle est également relativement équilibré, avec une matrice de confusion plutôt symétrique. Mais il est chronophage lui aussi et très sensible au bruit des données et aux *outliers* (données aberrantes), ce qui pourrait expliquer en partie les erreurs qu'il commet.

Finalement, sans surprise, le modèle le plus performant est le réseau de neurones avec un score de précision de 95% qu'il est possible d'améliorer en augmentant le nombre de répétitions. Il est cependant difficile d'expliquer d'où viennent ces 5% d'erreur en raison du problème de la 'boîte noire' inhérent aux réseaux neuronaux : en bref, il est très difficile de déterminer ce qui a amené l'algorithme à telle conclusion.

Conclusion

Ce dossier s'est intéressé à l'utilisation des algorithmes d'apprentissage automatique dans le contexte d'une catastrophe particulière, celle du naufrage du Titanic en 1912. Une première partie de ce dossier nous a permis de faire le point sur les différents types d'algorithmes d'apprentissage automatique et d'identifier le problème du Titanic comme appartenant clairement aux problématiques de classification. Nous avons choisi, dans un deuxième temps, de focaliser notre attention sur les algorithmes d'apprentissage automatique supervisé avec l'objectif de dégager les principes fondamentaux de quatre algorithmes : les réseaux de neurones, les support vector machines, les arbres de décisions et les forêts aléatoires. En nous appuyant sur ces éléments, nous avons ensuite entrepris de mettre en oeuvre ces quatre algorithmes en Python. Les résultats de ces implémentations ont été comparés ci-dessus et il n'est pas besoin d'y revenir ici en détail si ce n'est pour souligner que ces algorithmes présentent des niveaux de performance différenciés.

La bonne performance des algorithmes d'apprentissage automatique supervisé dans le cas assez étroit du Titanic ouvre naturellement une question sur l'utilisation de ces algorithmes dans le champ, plus large, de l'étude des catastrophes. A un niveau élémentaire, il serait utile de considérer l'application de nos algorithmes à des cas proches, à défaut d'être similaires. En 1915, le paquebot *Lusitania* est coulé par un sous-marin allemand, avec près de 2.000 personnes civiles à bord dont 1.198 périrent. Les catastrophes maritimes sont hélas loin d'avoir disparues et plus proches de nous, les accidents comme celui de *Le Joola* au Sénégal qui fit en 2002 plus de 1.800 victimes ou celui du *Princess of the Stars*, un ferry philippin qui coula en 2008, faisant officiellement 690 victimes sont des cas similaires à celui du Titanic sur lesquels nos algorithmes peuvent être testés -bien que des ajustements soient sans doute nécessaires. Au-delà de ces considérations, les performances des algorithmes ouvrent également la voie à des mesures plus efficaces de préventions et de sauvetage dans la mesure où l'identification de catégories plus vulnérables et des facteurs de vulnérabilité devient possible.

Bibliographie

1. DREYFUS, G. *et al.* Apprentissage statistique. *Groupe Eyrolles*, 77 (2002).
2. RADJA, M. & SALIHA, M. *Les modèles hybrides basés optimisation et apprentissage automatique pour la résolution de problèmes dans le domaine de la bioinformatique* thèse de doct. (Abdelhafid Boussouf University centre-Mila, 2016).
3. KESSLER, R., TORRES-MORENO, J. M. & EL-BÈZE, M. Classification thématique de courriels avec apprentissage supervisé, semi-supervisé et non supervisé. *les actes de VSST*, 493-504 (2004).
4. TURING, A. On computable real numbers, with an application to the Entscheidungsproblem, Proceedings of the London Mathematical Society. *J. of Math* **58**, 230-265 (1936).
5. TURING, A. M. *Mind* **59**, 433-460 (1950).
6. SAMUEL, A. L. Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development* **3**, 210-229 (1959).
7. HSU, F.-H. *Behind Deep Blue : Building the computer that defeated the world chess champion* (Princeton University Press, 2002).
8. POLOVA, M. & MAHAS, L. Modern trends in the development of artificial intelligence (2015).
9. ASSAEL, Y. M., SHILLINGFORD, B., WHITESON, S. & DE FREITAS, N. Lipnet : End-to-end sentence-level lipreading. *arXiv preprint arXiv :1611.01599* (2016).
10. KOTLER, P., KARTAJAYA, H., SETIAWAN, I. & VANDERCAMMEN, M. Chapitre 9. Le marketing prédictif. *Marketing*, 169-183 (2022).
11. AMINOT, I. & DAMON, M. Régression logistique : intérêt dans l'analyse de données relatives aux pratiques médicales. *Revue médicale de l'assurance maladie* **33**, 137-143 (2002).
12. CHERGUI, H., ABROUK, L., CULLOT, N. & CABIOCH, N. *Détection de fraude financière dans un système de transactions interbancaires* in *INFORSID* **22** (2022), 141-156.
13. GAUTHIER, E. *Utilisation des réseaux de neurones artificiels pour la commande d'un véhicule autonome* thèse de doct. (Institut National Polytechnique de Grenoble-INPG, 1999).
14. KRICHEN, M. *Méthodes formelles pour une gestion améliorée des catastrophes naturelles* working paper or preprint. Jan. 2024. <https://hal.science/hal-04368657>.
15. PICKERING E., K. G. Discovering and forecasting extreme events via active learning in neural operators. *Nat Comput Sci* **2**, 823-833 (2022).
16. Et AL., L. L. J. P. P. G. Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators. *Nat Mach Intell* **3**, 218-229 (2021).
17. KRICHEN, M. *Utilisation de l'intelligence artificielle pour prédire et atténuer efficacement les inondations* working paper or preprint. Jan. 2024. <https://hal.science/hal-04396345>.
18. MOUSAVI S.M. ELLSWORTH W.L. ZHU, W. e. a. Earthquake transformer—an attentive deep-learning model for simultaneous earthquake detection and phase picking. *Nat Commun* **11**, 3952 (2020).
19. KESSLER, R., TORRES-MORENO, J. M. & EL-BÈZE, M. Classification thématique de courriels avec apprentissage supervisé, semi-supervisé et non supervisé. *les actes de VSST*, 493-504 (2004).
20. ZHU, X. & GOLDBERG, A. B. *Introduction to Semi-Supervised Learning* (Springer Cham, 2009).
21. *Semi-Supervised Learning, Chapitre 1 et 2* (éd. CHAPELLE, O., SCHÖLKOPF, B. & ZIEN, A.) (The MIT Press, 2010).
22. TOUZET, C. *les réseaux de neurones artificiels, introduction au connexionnisme* (Ec2, 1992).
23. MASSINE, G. *Implémentation d'un réseau de neurones dans un microcontrôleur* thèse de doct. (Université Mouloud Mammeri, 2016).

24. IBM. *Support Vector Machine Topic* <https://www.ibm.com/topics/support-vector-machine>. 2023.
25. SCHAPIRE, R. *COS 511 : Theoretical Machine Learning Lecture #13* https://www.cs.princeton.edu/courses/archive/spring14/cos511/scribe_notes/0325.pdf. 2014.
26. WINSTON, P. *MIT 6.034 Artificial Intelligence* <http://ocw.mit.edu/6-034F10>. Accessed : 18 Juin 2024. 2014.
27. GÉRON, A. *Understanding Support Vector Machines, Chapitre 2* (O'Reilly Media, Inc., 2017).
28. JAMES, G., WITTEN, D., HASTIE, T. & TIBSHIRANI, R. *An Introduction to Statistical learning* (Springer Science Business Media, 2013).
29. CUKIERSKI., W. 2012. <https://kaggle.com/competitions/titanic>.
30. NARKHEDE, S. *Understanding Confusion Matrix* <https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62>. 2018.