UCL
Université catholique de Louvain

Institut de Statistique, Biostatistique et Actuariat

Estimation de la fréquence des sinistres par l'application des réseaux de neurones à AdaBoost

Membres du jury :

Prof. Donatien Hainaut *(Promoteur)*
Prof. Johan Segers *(Lecteur)*

Mémoire présenté en vue de l'obtention du master en sciences des données par :

Lucien Ledune

Louvain-La-Neuve
Août 2020

Préface

Ce mémoire de fin d'étude peut être vu comme la conclusion de mon parcours dans l'enseignement supérieur. Celui-ci a représenté pour moi une grande quantité de travail, mais a également été l'occasion d'approfondir mes connaissances sur de nombreux sujets relatifs aux études entreprises. La rédaction de celui-ci a pour but de compléter et finaliser ma formation, tout en respectant les exigences de l'université en vue d'obtenir une maîtrise en sciences des données (orientation statistique) à l'UCL. Le commencement de ces études a été compliqué, notamment dû à certaines lacunes de mon bagage mathématique, mais au fil des années j'ai été en mesure de travailler sur ces problèmes. Plus particulièrement, la rédaction de ce travail m'a permis de comprendre comment apprendre de nouveaux concepts sur base de papiers scientifiques, mais aussi à travailler de manière indépendante. Le choix du sujet de mémoire a été fait en concertation avec le professeur Donatien Hainaut, et a pour but de trouver des pistes pour l'amélioration des modèles utilisés dans la prédiction des fréquences de sinistres d'une assurance non-vie. Dans cette optique, la rédaction de ce mémoire m'a aussi permi d'en apprendre plus sur ce domaine, que je ne connaissait que de nom auparavant. La recherche associée a ce travail fut laborieuse et le nombre de travaux scientifiques sur le sujet étaient assez limités, mais cette investigation a finalement été fructueuse. Afin de mener à bien ce projet, il a également fallu que j'aiguise mes compétences de programmation (particulièrement en Python) afin de produire un travail de qualité. Il s'agit probablement du plus gros projet que j'ai jamais mené, et la programmation de celui-ci m'a appris énormément de chose. Le temps de calcul de certains algorithmes utilisés étant conséquent, cela a aussi été l'occasion pour moi d'apprendre à optimiser le code et la recherche des paramètres par diverses techniques, comme l'utilisation d'un processeur graphique pour la réalisation de certains calculs.

En espérant que cette lecture vous soit agréable,

Lucien Ledune
Quaregnon, 5 août 2020

Remerciements

Un grand nombre de personnes m'ont été d'une aide précieuse au cours de la rédaction de ce mémoire de fin d'études. Je tiens à remercier mes amis et ma famille : ils ont été un grand soutien moral tout au long de l'écriture de ce travail. Plus particulièrement, je remercie ma soeur, qui a accepté avec joie d'effectuer la relecture du projet. Mes remerciements vont également à Iasonas Topsis Giotis, mon maître de stage. Si ce travail n'a pas été réalisé dans le cadre du stage, j'ai néanmoins appris une quantité innombrable de choses en travaillant avec cette personne, notamment en ce qui concerne la programmation en Python. Un grand merci également au professeur Johan Segers pour avoir accepté d'être lecteur de ce mémoire.

Enfin, je tiens surtout à remercier mon promoteur, le professeur Donatien Hainaut, pour son aide et son assistance dans la rédaction de ce mémoire de fin d'études, mais aussi pour sa patience.

Table des matières

Chapitre 1

Introduction

Partout dans le monde, la facilité montante de l'accès à une puissance de calcul importante a motivé un grand nombre de changements dans la façon d'approcher les problèmes, et ce dans la plupart des domaines de recherche. Cette avancée technologique a permi d'utiliser des techniques de modélisations et de prédictions qui étaient jusqu'à lors trop longues à mettre en place pour avoir un réel intérêt pratique.

Le secteur des assurances n'a pas été épargné par le phénomène et évolue avec son temps. Le machine learning[1], outil de modélisation très puissant, est maintenant facilement accessible et les compagnies d'assurances souhaient en tirer le meilleur parti en matière de prédiction et d'aide à la décision.
L'utilisation de prédicteurs n'est pas nouvelle dans ce secteur, mais les algorithmes utilisés changent avec l'essort actuel de la technologie. Ainsi il est maintenant possible facilement utiliser des réseaux de neurones afin de répondre aux problèmes pour lesquels des algorithmes moins complexes étaient utilisés auparavant.

Le cadre dans lequel les analyses de ce mémoire seront réalisées est la prédiction de la fréquence de sinistres des clients d'une assurance (moto). Les différentes parties du travail seront articulées autour de ce but final. Afin de prouver que les techniques utilisées dans ce travail sont efficaces, celles-ci seront comparées avec un type de modèle étant encore majoritairement utilisé pour le calcul de la fréquence des accidents : le modèle linéaire généralisé. Au coeur de ce travail se situent les réseaux de neurones ainsi que l'algorithme de boosting bien connu "AdaBoost". Plus particulièrement, les réseaux de neurones précédemment cités seront appliqués à l'algorithme AdaBoost afin de tenter de produire de meilleurs résultats que ceux du modèle linéaire généralisé.

Dans la première partie de cet ouvrage seront traités les questions relatives à l'anlyse et la préparation de la base de données utilisée. Dans la deuxième partie, les algorithmes utilisés ainsi que les concepts importants à leur compréhension seront abordés et expliqués. Enfin, la troisième partie de ce travail aura pour but d'appliquer les algorithmes au cas étudié et de comparer les résultats obtenus.

1. Apprentissage automatique : Méthodes statistique permettant à un ordinateur d'apprendre à résoudre un problème donné à l'aide d'une base de données pertinente.

Chapitre 2

Jeu de données

2.1 Présentation des données

Afin de réaliser ce travail, une base de donnée adéquate est nécessaire, celle-ci est présentée et analysée dans ce chapitre. La base de données est constituée d'informations sur les clients d'une compagnie d'assurance nommée Wasa, dans une période de temps située entre 1994 et 1998. Les véhicules assurés sont composés uniquement de motos. Il est difficile d'obtenir des données plus récentes à cause des clauses de confidentialité des assurances. La version originelle de ce jeu de données est utilisée dans une étude cas du livre "Non-life insurance pricing with GLM", écrit par Ohlsson et Johansson, et celle-ci est disponible sur le site web du livre [1]. Le jeu de données est constitué de 64505 observations des 9 variables suivantes :

- OwnersAge : L'âge du conducteur.
- Gender : Le sexe du conducteur.
- Zone : Variable catégorielle représentant la zone dans laquelle le véhicule est conduit..
- Class : Variable catégorielle représentant la classe du véhicule. Les classes sont assignées dans une des 7 catégories selon le ratio : $EV = \frac{kW \times 100}{kg+75}$.
- VehiculeAge : L'âge du véhicule en années.
- BonusClass : Le bonus du conducteur, un nouveau conducteur commence à 1 et sera incrémenté à chaque année complète passée dans la compagnie sans sinistre déclaré, jusqu'à un maximum de 7.
- Duration : Le nombre d'année passées dans la compagnie.
- NumberClaims : Le nombre de sinistres.
- ClaimCost : Le coût des sinistres.

La variable Zone est décrite dans le tableau 2.1.

1. http ://staff.math.su.se/esbj/GLMbook/case.html

TABLE 2.1.1 – Descriptions des variables catégorielles

Variable	Classe	Description
Zone géographique	1	Parties centrales et semi-centrales des trois plus grandes villes de Norvège.
	2	Banlieues et villes moyennes.
	3	Petites villes (à l'exception de celles des catégories 5 et 7).
	4	Villages (à l'exception de ceux des catégories 5 et 7).
	5	Villes du nord de la Suède.
	6	Campagnes du nord de la Suède.
	7	Gotland (Grande île).

2.2 Analyse exploratoire

Maintenant que les différentes variables ont été brièvement présentées et que leur fonction est plus claire, l'analyse exploratoire de celles-ci peut être effectuée. Le but de cette analyse est de mieux comprendre les données qui serviront à entraîner les différents algorithmes, ainsi que de repérer d'éventuelles anomalies. Durant l'analyse exploratoire d'une base de données, il est important de regarder la distribution des variables, celle-ci nous donne beaucoup d'informations quant aux données.

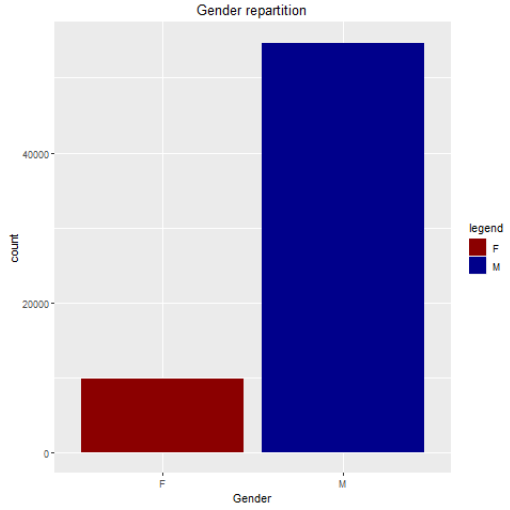


FIGURE 2.2.1 – Distribution Gender

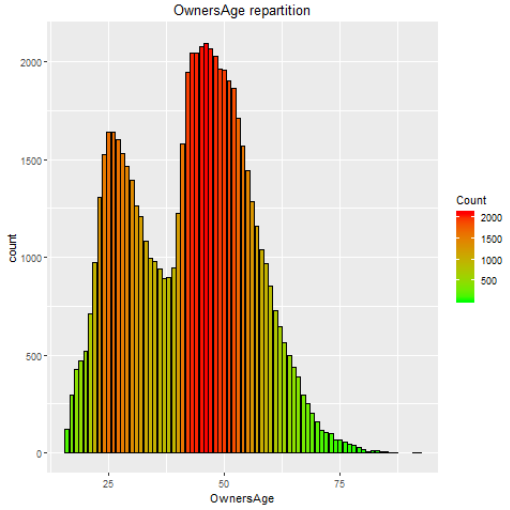


FIGURE 2.2.2 – Distribution OwnersAge

Sur ces deux premières figures, il est possible d'observer les distributions des variables "Gender" et "OwnersAge". La première chose qui saute aux yeux est la grande disparité entre le nombre de femmes et d'hommes clients auprès de l'assurance. Le jeu de données est composé d'hommes dans sa grande majorité. Pour ce qui est de la variable "OwnersAge", les valeurs sont réparties entre 16 et 92 ans, avec deux "pics" situés vers 25 et 45 ans. Les valeurs maximales et minimales de ces variables continues seront importante pour la suite, car elles sont nécessaires afin d'appliquer une normalisation des données, qui sera discutée plus tard dans ce travail. Ci-dessous sont présentées les distributions des variables "VehiculeAge" et "Zone" :

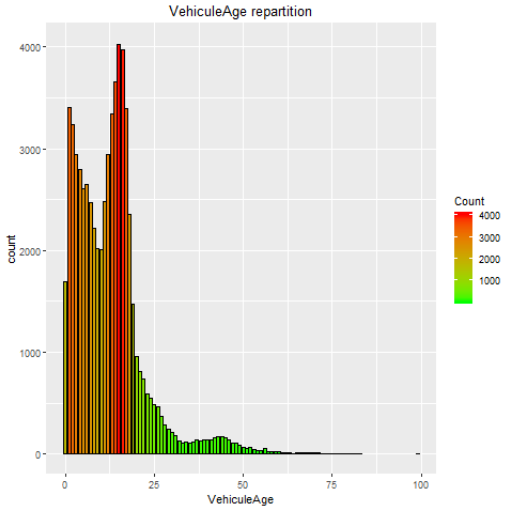


FIGURE 2.2.3 – Distribution VehiculeAge

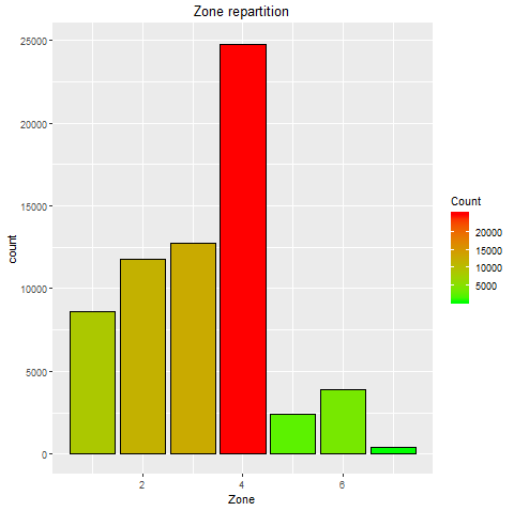


FIGURE 2.2.4 – Distribution Zone

La distribution de "VehiculeAge" est indiquée en années, et, si la plupart des véhicules assurés ont moins de 20 ans, un grand nombre de ceux-ci sont bien plus vieux, avec comme maximum 99 ans. Il est possible que ce véhicule soit considéré comme donnée aberrante [2] et il sera reconsidéré dans la sous-section 2.6.3. La distribution de la variable "Zone" est intéressante, elle révèle que la plupart des véhicules assurés sont conduits dans des villages, mais aussi que très peu d'entre eux le sont dans le Gotland. Ceci n'est pas surprenant puisque la population de la Suède est d'environ 10 millions d'habitants, pour seulement 60.000 habitants dans la région du Gotland. Les classes 5 et 6 sont elles aussi minoritaires, cela était aussi à prévoir puisque ces catégories représentent le nord de la Suède alors que la plupart de la population vit dans le sud du pays.

2. Outlier.

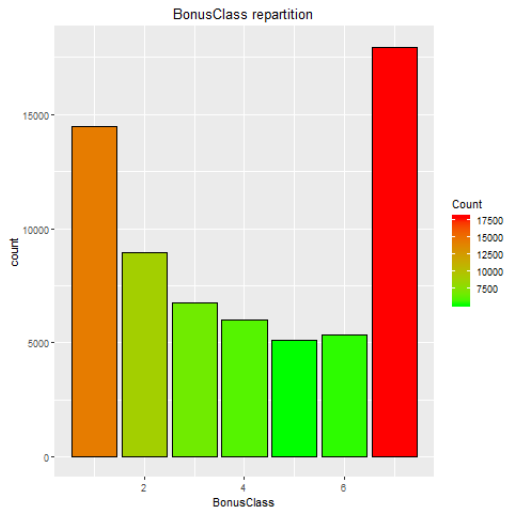


FIGURE 2.2.5 – Distribution BonusClass

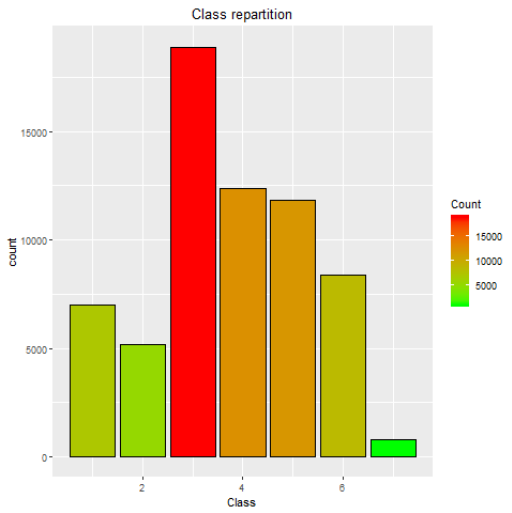


FIGURE 2.2.6 – Distribution Class

Les classes de bonus les plus fréquentes sont les classes 1 et 7, respectivement le minimum (c'est à dire l'année d'entrée dans la compagnie d'assurance) et le maximum (représentant un client fidèle depuis sept années au minimum).
Pour la variable Class nous observons qu'assez peu de véhicules appartiennent à la catégorie la plus puissante. En fait, la plupart des véhicules se situent dans les classes 3, 4 et 5, ce qui montre que les vehicules les moins puissants et les plus puissants sont minoritaires. Les graphiques suivants montrent quant à eux la répartitions des sinistres par bonus et par classe de véhicule.

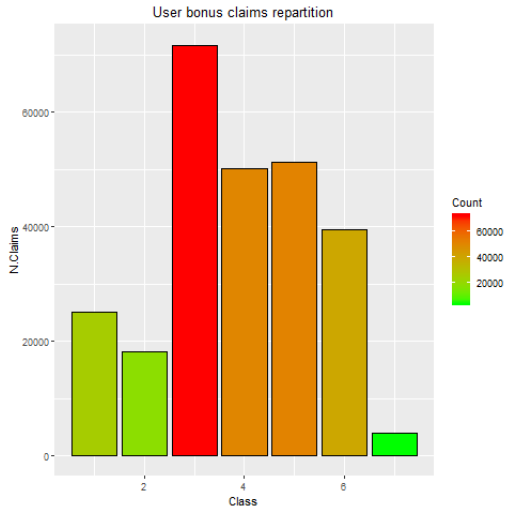


FIGURE 2.2.7 – Bonus Claims

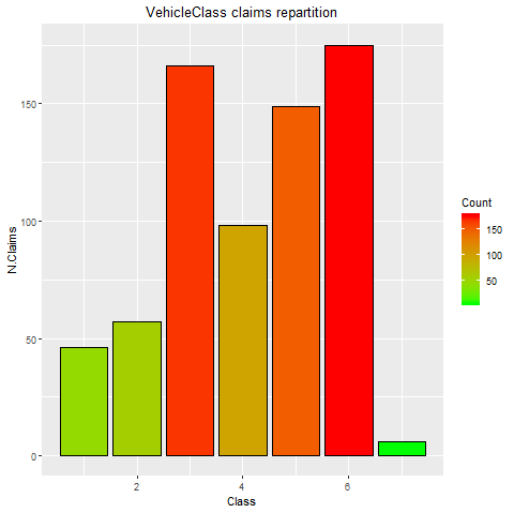


FIGURE 2.2.8 – Class Claims

Il parait logique de supposer que plus un client a un bonus élevé, moins celui-ci causera d'accidents, puisque le bonus monte uniquement si le client parvient à compléter une année

sans causer d'accident. Cependant en observant la figure 2.7, il apparait que la majorité des cas d'accidents sont déclarés par des clients appartenant aux classes 3 à 6 de bonus. Ce qui est d'autant plus étonnant lorsque l'on associe ce résultat avec la distribution de la variable "BonusClass" (figure 2.5) : les classes 3 à 6 sont celles contenant le moins d'utilisateurs. Les clients appartenant à la classe de bonus 7 semblent cependant causer très peu d'accidents malgré le fait qu'ils soient la classe de bonus majoritaire.
Les résultats de la figure 2.8 sont moins surprenant : plus un véhicule est puissant, plus le risque d'accident sera important. Les déviations de cette règle par les classes 3 et 7 sont expliquées par la distribution de la population à travers les différentes classes (figure 2.6), ainsi il y a peu d'accidents pour les véhicules de classe 7 simplement car ceux-ci sont peu nombreux, une conclusion similaire peut être énoncée concernant la classe 3.

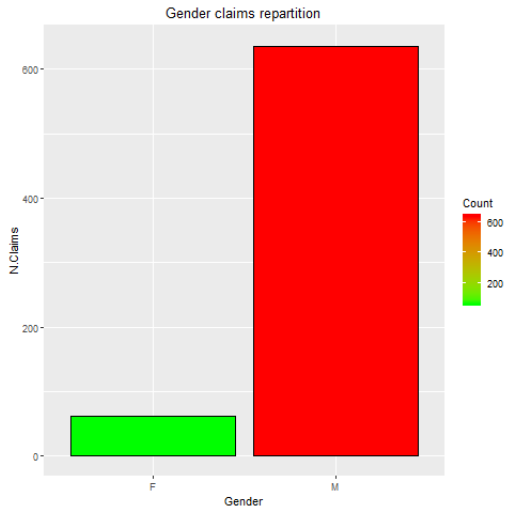


FIGURE 2.2.9 – Gender claims

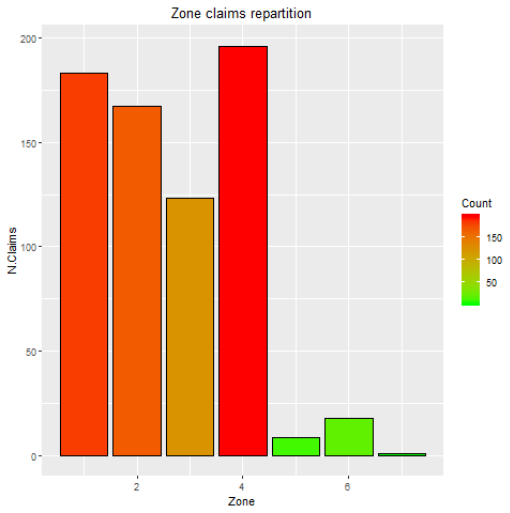


FIGURE 2.2.10 – Zone claims

La répartition des sinistres par sexe indique que les hommes sont plus susceptibles de causer des accidents que les femmes. Il faut prendre en compte que les hommes sont bien plus nombreux que les femmes dans nos données, cependant la conclusion ne change pas puisque la proportion de femmes est de 18%, alors que celles-ci ne causent que 8.7% des sinistres. Sur la figure 2.10, les classes 1 et 3 sont celles qui déclarent le plus de sinistres. La première classe étant plutôt minoritaire (figure 2.4), les personnes habitant dans les parties centrales et semi-centrales de Norvège semblent causer bien plus d'accidents que les autres catégories. La même conclusion peut être faite pour la deuxième classe (Banlieues et villes moyennes). Pour ce qui est de nord de la Suède, nous constatons le phénomène inverse puisque cette partie de la population semblent causer assez peu d'accidents.

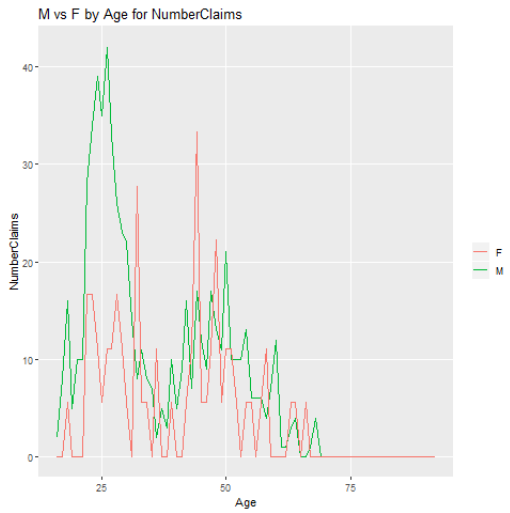


FIGURE 2.2.11 – Gender claims by Age

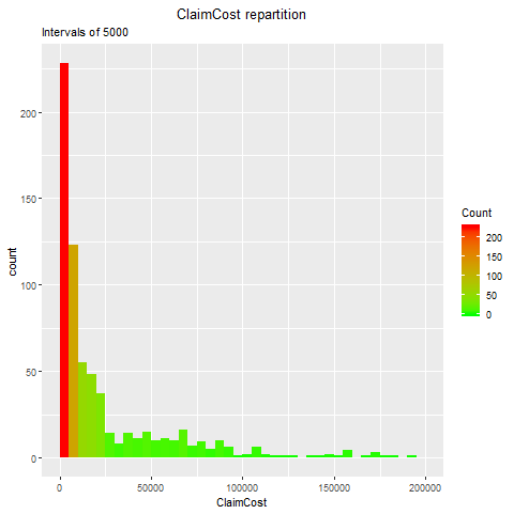


FIGURE 2.2.12 – Distribution ClaimCost

La figure 2.11 a pour but de mieux comprendre cette disparité entre les hommes et les femmes dans la déclaration des sinistres. Une comparaison par âge permet de mettre en évidence une information importante. En effet, si les hommes semblent bien causer plus d'accidents que les femmes jusqu'à la treintaine, mais passé ce cap, cette tendance s'égalise. Enfin, la figure 2.12 nous informe de la distribution de la variable ClaimCost, représentant le coût total des sinistres d'un client. Il est important de noter qu'il s'agit bien de la somme de tous les sinistres déclarés du client. Ainsi si une personne a reçu une compensation de l'assurance pour plusieurs sinistres, la variable contient la somme du coût de tous ces sinistres déclarés. Il semble que la grande majorité des clients ne dépassent pas 5000€ de compensation venant de l'assurance (Le graphique ne montre que les valeurs non-nulles, ainsi les clients n'ayant jamais déclaré de sinistre ne sont pas comptés dans l'intervalle $[0, 5000]$. La distribution de cette variable montre bien qu'assez peu de clients dépassent les 20.000€ de compensation, mais pourtant certains peuvent monter à près de 200.000€.

2.3 Matrice des corrélations

La matrice des corrélations va maintenant être analysée. Cette matrice, représentée sous forme de graphique, donne des informations quant aux relations linéaires liant les différentes variables. La corrélation entre deux variables est définis comme suit :

$$r = \frac{Cov(X, Y)}{\sigma_X \sigma_Y}$$

- $Cov(X, Y)$: Covariance des variables X et Y.

- σ_X : Écart-type de X.
- σ_Y : Écart-type de Y.

Il est important de noter que cette mesure ne prend en compte que les relations linéaires entre les variables, mais elle permet de se faire une première idée des relations dans la base de données lors de l'analyse exploratoire. Une corrélation positive implique que si Y augmente, alors X a tendance à augmenter aussi. Au contraire, si la corrélation est négative, X aurait tendance à diminuer. Enfin, si la corrélation est proche de 0, cela signifie que les variable n'ont pas de relation (linéaire).

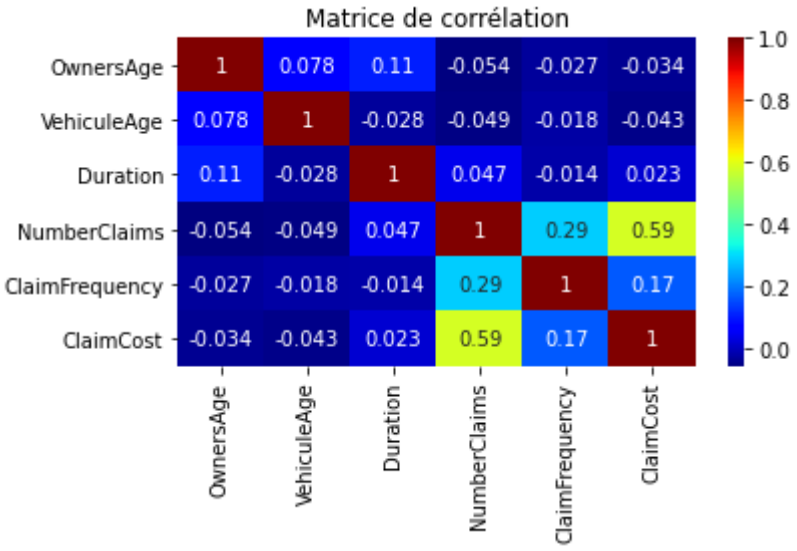


FIGURE 2.3.1 – Matrice des corrélations.

Cette métrique n'a que peu d'intérêt à être appliquée aux variables catégorielles, c'est pour cela que seules les variables continues sont représentées ici.

- Les variables "ClaimFrequency" et "ClaimCost" sont positivement corrélées, ce qui est facilement explicable par le fait que "ClaimCost" est la somme des coûts des différents sinistres survenus. Ceci implique que si aucun sinistre n'a été déclaré (ClaimFrequency =0) alors "ClaimCost" sera nul.
- Le même raisonnement peut être tenu pour la corrélation positive entre "NumberClaims" et "ClaimCost".
- La corrélation positive entre "ClaimFrequency" et "NumberClaims" est quant à elle expliquée par le fait que "ClaimFrequency" est un ratio divisant "NumberClaims" par "Duration" (voir sous-section 2.5.2).
- Les variables "Duration", "NumberClaims", "ClaimFrequency", "ClaimCost" ne sont pas utilisées dans la prédiction. Si les autres variables ("OwnersAge" et "VehiculeAge") sont mises en évidence, la corrélation presque nulle est une bonne chose puisqu'il

est bon d'éviter de mettre trop de variables corrélées dans un même modèle (ce qui ne veut cependant pas dire qu'elles n'ont pas une relation non-linéaire possible).

2.4 Observation des sinistres

En plus de la matrice des corrélations, il est possible de regarder les variables numériques deux à deux afin de mettre en évidence d'éventuelles relations.

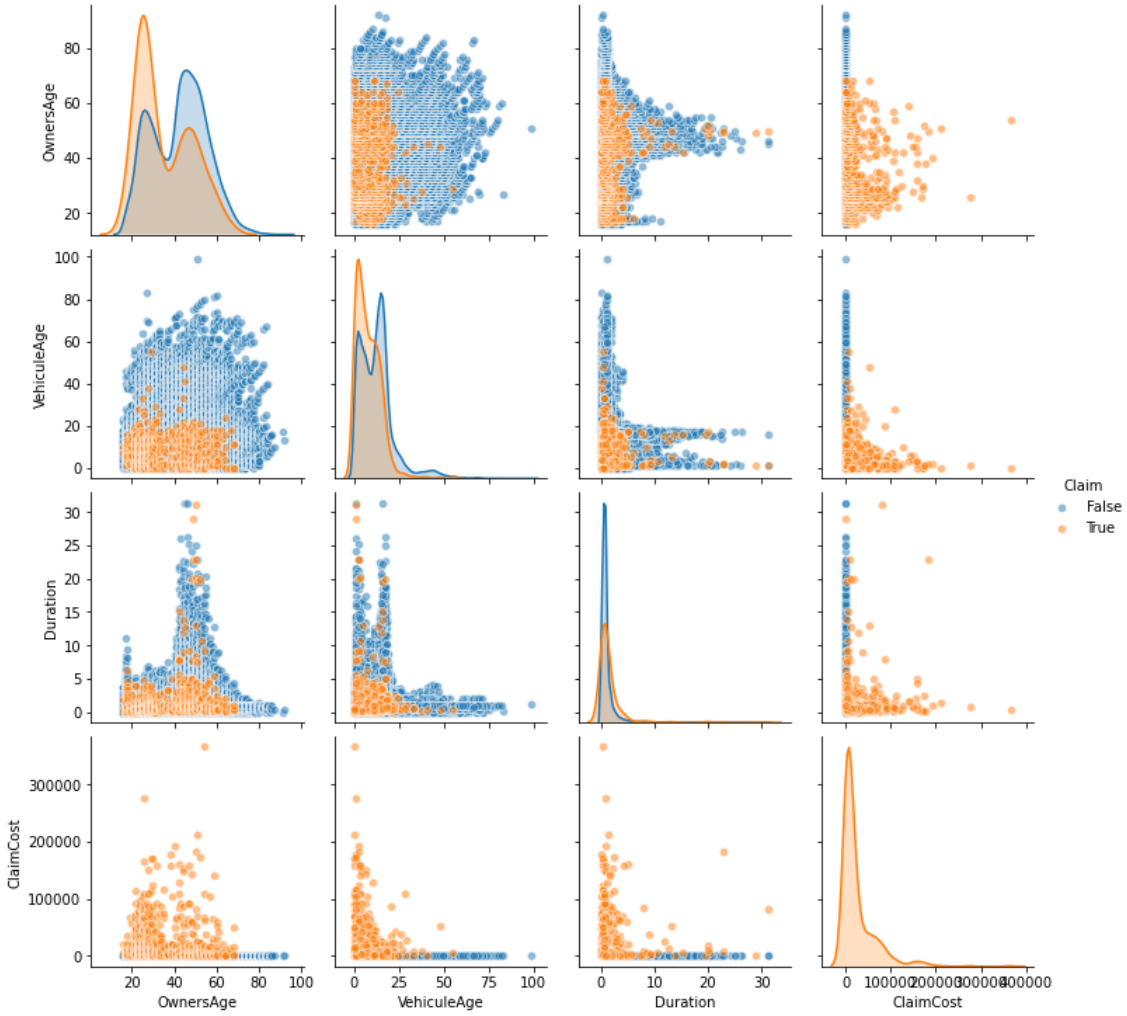


FIGURE 2.4.1 – "Pair-plot" des variables numériques en fonction de la variable "Claim".

La figure 2.14 montre plusieurs choses :

- Les individus sont colorés en fonction d'une nouvelle variable catégorielle créée pour l'occasion : "Claim". Elle indique si un individu a déclaré un sinistre ou non. Sur la diagonale se situent les fonctions de densité, scindées selon la variable "Claim". Les

autres graphiques montrent les relations entre les différentes variables numériques, deux à deux.

- En ce qui concerne la variable "OwnersAge" et sa densité, il est intéressant de noter que les deux pics observés sont inversés en fonction de la variable "Claim". Il y a plus de clients qui déclarent un sinistre vers le pic de 25 ans que de clients qui n'en déclarent pas, et il y a moins de clients qui déclarent un sinistre vers 45 ans que de clients qui en déclarent un.
- Pour la variable "VehiculeAge", la forme de la densité du groupe ayant déclaré un sinistre ou plus indique que beaucoup d'accidents surviennent lorsqu'un véhicule est encore relativement neuf. Ce qui semble aussi être indiqué par les graphiques "VehiculeAge/NumberClaims" et "VehiculeAge/ClaimFrequency".
- La durée du contrat ("Duration") semble quant à elle avoir moins d'influence sur l'appartenance à l'un des groupes de la variable "Claim".

Dans les annexes 1 à 3, des "pair-plots" similaires séparant les observations selon des critères différents sont disponibles.

2.5 Assurances

Dans cette section seront expliquées les particularités de l'analyse de données dans le cadre de l'assurance pour le cas étudié.

2.5.1 Principe

Un contrat entre l'assureur et l'assuré implique le payement d'une prime d'assurance par le second, en échange de compensations lorsqu'un sinistre est déclaré. Le fait de regrouper un grand nombre de clients va avoir un effet stabilisateur de variance pour l'assureur. En effet il est difficile de prédire précisément le nombre d'accidents qu'un assuré aura, cependant plus le nombre d'assurés sera élevé et plus le nombre (total) de sinistres sera prévisible. Ceci est dû à la loi des grands nombres qui implique :

$$\bar{X} \to \mu \, when \, n \to \infty$$

La moyenne de l'échantillon $\bar{X}$ converge vers le moyenne de la population μ lorsque la taille de l'échantillon augmente.
Dans le cadre de l'assurance, les variables explicatives sont aussi appelées facteurs d'évaluations, les deux termes seront utilisés indifféremment au cours de ce travail.

2.5.2 Ratio

Il est important de comprendre ce que l'on cherche à expliquer avec un modèle, aussi il faut savoir que cette analyse se base sur un ratio. Un ratio Y est un rapport entre une

réponse X et une exposition[3] v.

$$Y = X/v$$

Bien qu'il existe différents ratios, il ne sera présenté ici que celui qui sera utilisé dans l'analyse à venir : La fréquence des réclamations (ou fréquence des sinistres). Il s'agit du rapport entre le nombre de réclamations et la durée du contrat, une fréquence qui sera la variable prédite par les modèles est ainsi obtenue.

2.5.3 Distribution du ratio

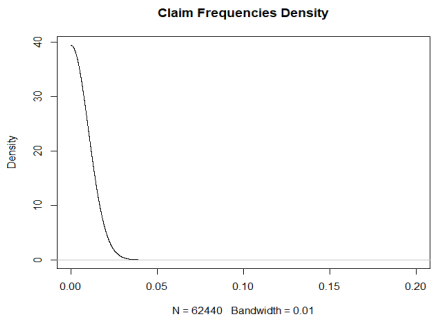
Pour la création d'un modèle cherchant à prédire la fréquence des réclamations, c'est souvent la distribution Poisson qui est utilisée. Une loi poisson de paramètre λ implique :

$$Y \sim Poisson(\lambda)$$

$$E[Y] = \lambda$$

$$Var[Y] = \lambda$$

Afin de vérifier que la distribution de la fréquence des sinistres suit bien une loi de poisson, cette dernière est affichée. En effet, la plupart des valeurs se situent près de 0, les valeurs supérieures à 1 sont présentes mais extrêmement rares et ne sont donc pas affichées afin d'observer plus précisément les valeurs types.



La loi de poisson fait partie de la famille des modèles à dispersion exponentielle (ED), une catégorie regroupant un grand nombre de lois statistiques connues comme la loi normale, gamma ou encore la loi binomiale. Une variable aléatoire Y suit une dispersion

3. L'exposition est égale à la durée du contrat dans le cadre de ce travail.

exponentielle si sa fonction de distribution suit la forme suivante (Denuit, Hainaut et Trufin 2019b) :

$$f_{Y_i}(y;\theta_i;\phi) = exp\left\{\frac{y\theta_i - a(\theta_i)}{\phi/\upsilon_i}\right\} c(y,\phi,\upsilon_i)$$

- θ_i : Paramètre dépendant de i.
- ϕ : Paramètre identique pour tous les i. Il s'agit du paramètre de dispersion.
- $c(.)$ est indépendante du paramètre θ_i.

2.6 Préparation des données (preprocessing)

Le "preprocessing" des données est une étape très importante et celui-ci n'est pas à négliger. Il consiste à préparer les données pour les algorithmes, afin que ces derniers puissent fonctionner de manière optimale. Dans ce travail, plusieurs méthodes de préparation des données ont été utilisées.

2.6.1 Variables binaires (Dummy variables)

Certaines des variables sont catégorielles, et plus particulièrement sont des variables non-ordinales. Cela signifie que les différentes catégories ne peuvent pas être ordonnées, il s'agit purement de l'assignation d'un client à un groupe donné. Ce type de variable ne peut pas être encodé tel quel et doit être transformé en une série de variables binaires. La conversion d'une variable catégorielle en variable binaire consiste à créer $n-1$ nouvelles variables dites "binaires" et qui prendront pour valeur 1 si le client appartient à cette catégorie, 0 le cas échéant. Par exemple, si x_{ij} représente la variable associée au client i et à la variable j, il est possible d'écrire :

$$x_{ij} = \Big\{0 \text{ if } x_i \notin j \quad 1 \text{ if } x_i \in j$$

Les variables "Gender", "Zone", "Class", et "BonusClass" peuvent être transformées de la sorte. Ce travail sera effectué avec le package caret [4] de R. La fonction incluse dans ce package crée n variables binaires, la dernière sera donc supprimée car celle-ci ne représente aucune information. La variable "Gender" sera prise en guise d'exemple. Si $x_{iGender} = 1$ alors le client est une femme. Cependant il n'est pas nécessaire de créer une deuxième variable binaire puisque si $x_{iGender} = 0$, il est facile d'en déduire que le client est un homme. Ceci explique pourquoi $n-1$ variables binaires sont suffisantes pour représenter toute l'information de n catégories appartenant à une variable catégorielle.

4. Classification And REgression Tools : Package de machine learning pour R.

2.6.2 Normalisation

Les données quantitatives ne doivent pas être transformées en variables binaires (bien que celles-ci peuvent être converties en intervalles puis en variables binaires). Cependant cela ne signifie pas qu'aucune méthode de préparation ne doit être appliquée à ces données. La normalisation des données quantitatives aide certains algorithmes à converger plus rapidement et peut même parfois améliorer leur précision. Cette normalisation des données est particulièrement importante pour les réseaux de neurones, car l'optimisation de ceux-ci est basée sur les résultats des fonctions d'activation [5] utilisées. Celles-ci varient rapidement entre 0 et 1 (ou -1 et 1) et sans la normalisation des données leurs réponses aux valeurs extrêmes seraient localisées dans les extrémités, ce qui peut empêcher l'algorithme de converger dans les cas les plus graves.
Plusieurs méthodes peuvent être utilisées afin de procéder à la normalisation des données. Cependant ne sera décrite dans ce travail que celle qui sera utilisée par la suite : la normalisation Min Max (Denuit, Hainaut et Trufin 2019b).
Elle consiste (pour une variable j) à convertir l'ensemble des données sur l'intervalle [0,1] en se servant de $Max(x_j)$ [6] et $Min(x_j)$ [7] par la formule :

$$z_{ij} = \frac{x_{ij} - min(x_j)}{max(x_j) - min(x_j)}$$

Cette transformation sera appliquée aux variables "OwnersAge" et "VehiculeAge".

2.6.3 Outliers

Lors de l'encodage des données, il arrive que certaines valeurs des variables quantitatives soient assez extrêmes et semblent fort distantes des autres observations. Dans ce cas il est possible de soit garder la valeur si celle-ci semble ajouter de l'information au modèle soit de la supprimer dans le but d'une meilleure généralisation par celui-ci. Dans la section 2.2 une donnée a particulièrement attiré l'attention comme outlier potentiel : une moto de 99 ans. Pour celle-ci deux possibilités sont à envisager :

- Il peut simplement s'agir d'une erreur d'encodage, auquel cas l'observation doit être supprimée pour éviter de biaiser le modèle.
- Il peut également s'agir d'une vraie moto de type "ancêtre", cependant ce genre de véhicule ne se conduit pas de la même manière qu'un véhicule utilisé au quotidien, et on peut supposer que les risques d'accidents sont bien moindres. Dans ce cas-ci la suppression de la donnée n'est pas obligatoire mais peut aider à une meilleure généralisation du modèle.

5. Voir sous-section 3.3.1.
6. Valeur maximale prise par la variable j.
7. Valeur minimale prise par la variable j.

Puisqu'aucun moyen simple de vérifier cette hypothèse n'est disponible, celle-ci sera supprimée de la base de donnée car la perte d'information est moindre.
La suppression de trop d'information peut cependant nuire au modèle, les données extrêmes mais non-aberrantes seront donc conservées.

2.6.4 Durée de contrat

Pour le bon fonctionnement du modèle, certaines autres données ont du être modifiées car celles-ci entrainaient des erreurs dans la fonction de déviance utilisée pour la calibration des poids des réseaux de neurones (limitant ou empêchant parfois la convergence). Les observations incriminées sont celles ayant une durée de contrat nulle. Ces valeurs nulles de durée de contrat sont peu cohérentes (il est en effet impossible d'avoir un sinistre déclaré au cours du contrat si celui-ci n'est pas encore en vigueur) et elles seront donc changées par la valeur correspondant à un jour ($1/365 \simeq 0.00274$).

Certaines de ces observations n'ont cependant pas une valeur nulle pour la variable NbClaims, ce qui signifie que des sinistres auraient été déclarés dans des contrats dont la durée est nulle. Ce cas de figure n'étant pas possible, elles seront supprimées car il s'agit probablement d'erreur d'encodage. De plus, changer la valeur de la durée du contrat par 0.00274 dans ce cas reviendrait à considérer une fréquence de sinistre de $\simeq 365$ soit un sinistre par jour ce qui pourrait grandement biaiser le modèle. À la fin des différentes étapes de préparation, la base de données passe de 64505 observations à 64501.

Chapitre 3

Algorithmes

Afin d'effectuer un travail de prédiction en matière de régression ou de classification, il est nécessaire d'avoir un outil puissant à disposition : l'apprentissage automatique ou "machine learning". Il s'agit d'un ensemble de méthodes basées sur des outils statistiques permettant la création de modèles pouvant être utilisés pour la prédiction.

Dans ce chapitre, les différents algorithmes utilisés lors des analyses seront présentés à des fins de compréhension du travail. Pour commencer, des explications seront données sur le fonctionnement général d'un modèle prédictif, et plus particulièrement dans le cas de la régression. Les bases nécessaires pour comprendre comment évaluer un modèle afin de déterminer la supériorité de l'un d'entre eux par rapport à d'autres seront explorées afin de disposer du bagage théorique nécessaire pour analyser les résultats obtenus dans le cadre de ce travail.

Les deux algorithmes de base que sont les modèles linéaires généralisés et les réseaux de neurones artificiels seront introduits et leur fonctionement expliqué. Enfin, le boosting sera également étudié et plus particulièrement le "méta-algorithme" qui nous intéresse ici : AdaBoost.

3.1 Explication d'un modèle

3.1.1 Définitions et apprentissage

Avant de pouvoir rentrer dans les détails, il est important de définir les notations qui seront utilisées, de définir ce qu'est un modèle et comment celui-ci se construit. La construction d'un modèle est spécifique pour la résolution d'une tâche T, que l'on cherche à améliorer par rapport à une métrique P, et ce grâce à des expériences [1] Q. La base de données est composée de X, les variables explicatives et de Y, la variable expliquée. Dans notre cas :

- T : Déterminer la fréquence des sinistres.
- P : Déviance (Explications dans la section 3.1.2).

1. Données.

- $Q = \{(x_1, y_1), ..., (x_n, y_n)\}$: Base de données $X \to Y$

Un modèle cherche à approximer une fonction cible (choisie), dans le cas étudié $f : X \to \mathbb{R}^+$ Dans cette étude seuls les modèles supervisés[2] seront abordés. Deux éléments sont nécessaires pour la création d'un modèle, une base de données et un algorithme supervisé (choix du modèle). La base de données est composée de N entrées des variables explicatives et de la variable expliquée y_i où i représente l'individu et où x_{ij} représente la valeur prise pour la variable j pour l'individu i. Le modèle va se servir des valeurs prises par les x_{ij} pour tenter d'expliquer y_i. Pour ce faire, l'algorithme doit subir une phase d'apprentissage, au cours de laquelle celui-ci va recevoir des entrées provenant de la base de données et se servir de celles-ci pour "apprendre" et se calibrer sur le problème que l'on cherche à résoudre. Cette phase d'apprentissage est différente d'un algorithme à un autre. Une fois cette phase d'apprentissage terminée le modèle ainsi créé peut être utilisé pour la prédiction. Les valeurs prédites par l'algorithme seront notées $\hat{y}_i$.

3.1.2 Évaluation et fonctions de perte

Pour l'apprentissage, les algorithmes supervisés se basent sur un critère mesurant l'erreur d'un modèle : la fonction de perte. Ce critère permet la comparaison entre différents modèles et un choix optimal parmi ceux-ci, en sélectionnant celui qui minimise l'erreur commise dans les prédictions. Il existe un grand nombre de critères pouvant servir pour l'évaluation des modèles mais le choix de celui-ci doit se faire en fonction du problème adressé et de la base de données. Le choix de ce critère est important car il peut impacter l'efficacité des modèles créés.
Le critère peut être défini comme $f(y, \hat{y})$ une fonction dépendant des valeurs prises par y et $\hat{y}$ mesurant la différence entre les prédictions et la réalité d'une certaine manière. Une fonction de perte intuitive est donnée en guise d'exemple par la racine de l'erreur quadratique moyenne (RMSE).

$$RMSE(\theta, \hat{\theta}) = \sqrt{MSE(\theta, \hat{\theta})} = \sqrt{E((\hat{\theta} - \theta)^2)}$$

Le RMSE est sûrement un des critères les plus utilisés aujourd'hui (y compris pour les réseaux de neurones), cependant utiliser celui-ci reviendrait à considérer le ratio de la fréquence des réclamations comme étant distribué selon une loi gaussienne (Denuit, Hainaut et Trufin 2019b). Les résultats s'en trouvent alors biaisés car les propriétés statistiques de la variable expliquée ne sont pas prises en compte. Afin de palier à ce problème, un autre critère statistique sera utilisé : la déviance de Poisson. La déviance est utilisée la plupart du temps pour des tests d'hypothèses et elle joue un role important dans l'analyse des distributions de la famille exponentielle. Elle peut être vue comme une forme de distance (d'où son utilisation en temps que critère). Aussi :

2. Algorithmes nécessitant une base de données $X \to Y$ pour l'apprentissage.

- D^*, la déviance mise à l'échelle, un test entre les maximums de vraisemblance du modèle entraîné et du modèle saturé [3].
- $l(\hat{y_i})$, le log-vraisemblance de $\hat{y_i}$.
- $l(y_i)$, le log-vraisemblance de y_i.

$$D^*(y_i, \hat{y_i}) = 2\left(l(y_i) - l(\hat{y_i})\right)$$

En utilisant la fonction de distribution des ED vue précédemment, le résultat suivant est obtenu :

$$D^*(y_i, \hat{y_i}) = \frac{2}{\phi}\upsilon_i(y_i h(y_i)) - a(h(y_i)) - y_i h(\hat{y_i}) + a(h(\hat{y_i})))$$

La déviance non mise à l'échelle peut ensuite être obtenue en multipliant cette formule par le paramètre de dispersion ϕ. La dérivation de cette déviance non mise à l'échelle appliquée à la loi de poisson résulte en la déviance de Poisson non-mise à l'échelle, c'est ce critère qui sera utilisé pour l'apprentissage des modèles (Denuit, Hainaut et Trufin 2019b).

- $a'(\theta) = e^\theta$
- $h(y) = ln(y)$

$$D(y_i, \hat{y_i}) = 2\upsilon_i(y_i ln(y_i) - y_i ln(\hat{y_i}) - y_i + \hat{y_i}),\ \ \forall y_i > 0$$

La base de données utilisée portant sur des événements rares, beaucoup de cas $y_i = 0$ seront étudiés, en modifiant la formule ci-dessus (en remplaçant y_i par 0) la formule adéquate pour ce cas particulier est donnée par :

$$D(y_i, \hat{y_i}) = 2\upsilon_i \hat{y_i},\ \ \forall y_i = 0$$

3.1.3 Surentraînement (Overfit)

Il est important de comprendre le but derrière la création d'un tel modèle. L'objectif principal est d' approximer une fonction de réponse Y à l'aide de celui-ci. Afin que cette fonction soit au mieux approchée, il est important que le modèle reste général. En effet, le but final du modèle est de prédire des valeurs de Y pour des nouvelles entrées X_i, ce qui amène à des méthodes permettant de vérifier que cette fonction est correctement approximée. Plus particulièrement, le but est d'éviter un maximum l'overfit. L'overfit (ou surentraînement) survient lorsqu'un modèle approxime bien les valeurs du jeu de données utilisé pour l'entraînement mais que celui-ci est incapable de déterminer des valeurs cohérentes pour des nouvelles entrées, il peut être notamment causé par un trop grand nombre

3. Modèle où les $\hat{y_i}$ sont définis comme les véritables valeurs y_i,le maximum de vraisemblance de ce modèle est le meilleur que nous pouvons obtenir du fait que les $y_i$sont parfaitement prédits.

de variables explicatives ou par une phase d'apprentissage mal configurée. Le modèle est alors inutilisable dans un scénario réel.

3.1.4 Validation croisée (K-folds)

La valeur de certains paramètres d'un modèle pouvant être générée dans un premier temps de manière aléatoire lors de l'initialisation avant d'être optimisés, et la forme du jeu de données pouvant aussi influencer les résultats, deux modèles "identiques" n'auront pas toujours le même résultats pour les critères statistiques utilisés. Il est aussi parfois difficile de détecter l'overfit, c'est la raison pour laquelle la validation croisée est utilisée. Généralement la base de données utilisée est divisée en un jeu d'entraînement et un jeu de test qui ne sera pas utilisé pour l'entraînement mais pour calculer les résultats du modèle aux différents critères statistiques. Dans le cas étudié ici, les événements sont rares, et le jeu de test sera d'une taille réduite afin de garder un maximum de données pour l'entraînement. Afin de vérifier que l'entraînement se déroule correctement et que les modèles ont du sens. La validation croisée sera également utilisée sur le jeu d'entraînement.
Le principe de base est de diviser le jeu de données en K ensembles de tailles égales. Un modèle sera ensuite entrainé pour chaque ensemble en utilisant cet ensemble comme jeu de test et la totalité des autres ensembles comme jeu d'entraînement. Le résultat de la métrique est alors la moyenne des résultats des différents ensembles (Denuit, Hainaut et Trufin 2019b ; Shao 1993). Plus formellement :

- $Q = \{(x_1, y_1), ..., (x_n, y_n)\}$, la base de données.
- K, le nombre d'ensembles créés.
- M, l'algorithme d'apprentissage.
- $E(M, Q)$, la fonction de perte dépendant de y et $\hat{y}$, calculée sur Q avec le modèle M.

Algorithm 3.1 Validation croisée (K-folds)

Diviser Q en K sous-ensembles $Q_1, ..., Q_k$
for $k = 1, ..., K$
test $= Q_k$
train $= Q \setminus Q_k$
$M_k = M(train)$//Entraînement du modèle.
$E_k = E(M_k, test)$ //Erreur du modèle M_k sur Q_k.
endfor
$E(Q) = K^{-1} \sum_{i=1}^{i=k} E_k$//Moyenne des erreurs
return $E(Q)$

Cette méthode permet de détecter le surentraînement dans la plupart des cas. La valeur de la métrique donnée par la validation croisée dispose d'une variance inférieure aux résultats donnés sans celle-ci.

3.2 GLM : modèle linéaire généralisé

3.2.1 Les bases du GLM

Les GLMs représentent une généralisation de l'OLS [4] dans deux directions :

- La distribution de probabilité : Dans l'OLS, nous supposons que les données suivent une loi gaussienne. Avec les GLM, une classe de distribution (les distributions EDM) dont la fonction de distribution générale a été présentée plus tôt dans ce travail, est utilisée. Plus particulièrement, dans le cadre de ce mémoire, cela permet de construire les modèles avec la loi de Poisson.
- Modèle pour la moyenne : Dans les modèles linéaires classiques, la moyenne n'est qu'une fonction linéaire des variables exploratoires. Dans les GLMs, c'est une transformation monotone de la moyenne qui est une fonction linéaire des variables exploratoires.

Les GLMs se sont vite présentés comme un modèle très efficace pour la modélisation des ratios dans les assurances. Ceux-ci présentent en effet nombres d'avantages :

- L'existence de différentes recherches menées sur le sujet ainsi que de méthodes déjà définies pour l'estimation des différents paramètres du modèle, d'intervalles de confiance, ...
- Ils sont utilisés à la fois dans le cadre des assurances, mais pas uniquement.
- Il existe un bon nombre de programmes/outils permettant de construire de tels modèles de manière relativement simple et efficace.

Enfin, un GLM est composé de :

- Un prédicteur linéaire : $\eta_i = \beta_0 + ... + \beta_1 x_{1i} + \beta_p x_{pi}$
- La fonction de lien : $g(\mu_i) = \eta_i$
- La fonction de variance : $v(Y_i) = \phi v(\mu)$ où ϕ est le paramètre de dispersion (constant).

Ces composants seront décrit plus précisément plus loin dans ce chapitre.

3.2.2 Hypothèses

Afin que les raisonnements à suivre restent corrects, il est nécessaire de faire 3 hypothèses pour que nos modèles statistiques restent valides (Ohlsson et Johansson 2010).

Hypothèse 1 : Indépendance des polices d'assurances En considérant n polices d'assurances, avec X_i correspondant à la réponse de la police i, et en supposant également

4. Ordinary Least Squares (régression linéaire)

que $X_1, ..., X_n$ sont indépendants.

Même si cette hypothèse est primordiale, il n'est pas difficile de trouver des cas où celle-ci n'est pas respectée. Par exemple, dans le cas des grandes catastrophes naturelles qui impliquent qu'un grand nombre de personnes déclarent des sinistres pour la même raison. Pour ce genre de cas, il est nécessaire d'utiliser des modèles différents, cependant, nous ne nous y intéresserons pas dans le cadre de ce travail.

Hypothèse 2 : Indépendance temporelle En considérant n intervalles de temps différents, avec X_i correspondant à la réponse dans l'intervalle de temps i. Alors $X_1, ..., X_n$ sont indépendants.

Hypothèse 3 : Homogénéité En considérant deux polices dans la même plage de tarifs et ayant la même durée de contrat, avec X_i correspondant à la réponse pour la police i. Alors X_1 et X_2 ont la même distribution de probabilité.

Cette hypothèse n'est dans les fait pas complètement respectée. Les différents conducteurs sont répartis dans des classes relativement homogènes mais il reste de la non-homogénéité résiduelle. Celle-ci est généralement compensée par un système de bonus/malus ou d'autres moyens similaires.

Les hypothèses 1 et 2 ont pour conséquence que les coûts des sinistres sont indépendants : Ils concernent soit une police différente soit un intervalle de temps différent.

3.2.3 Modèles à dispersion exponentielle (EDM)

Les modèles à dispersion exponentielle font partie des GLMs, et généralisent la distribution normale utilisée dans les modèles linéaires. En utilisant les hypothèses ci-dessus, la distribution de probabilité d'un EDM peut être déterminée (Ohlsson et Johansson 2010 ; Denuit, Hainaut et Trufin 2019b).

$$f_{Y_i}(y; \theta_i; \phi) = exp\left\{\frac{y\theta_i - a(\theta_i)}{\phi/\upsilon_i}\right\} c(y, \phi, \upsilon_i)$$

- θ_i : paramètre pouvant dépendre de i.
- $\phi > 0$: paramètre de dispersion, ne peut pas dépendre de i.
- $b(\theta_i)$: fonction cumulante, le choix de cette fonction revient à choisir une famille de distribution de probabilité (Normale, Poisson, ...).
- $c(.)$: fonction ne dépendant pas de θ_i.

Dans le cas étudié, c'est la fréquence des sinistres qui est importante. En considérant $N(t)$ comme le nombre de sinistres d'un individu au cours de la période $[0, t]$. Beard, Pentikainen et Pesonen montrent que ce processus statistique suit une loi de Poisson (Beard, Pentikainen et Pesonen 1977). Si X_i représente le nombre de plaintes dans une classe de

durée υ_i et en prenant μ_i comme l'espérance lorsque $\upsilon_i = 1$. Alors X_i suit une distribution de Poisson et sa densité est :

$$f_{X_i}(x_i; \mu_i) = e^{-\upsilon_i \mu_i} \frac{(\upsilon_i \mu_i)^{x_i}}{x_i!},\ x_i = 0, 1, 2, ...$$

La distribution du ratio $Y_i = X_i/\upsilon_i$ est aussi étudiée et tombe également sur une distribution de Poisson (Ohlsson et Johansson 2010) [5] :

$$f_{Y_i}(y_i; \mu_i) = P(Y_i = y_i) = P(X_i = \upsilon_i y_i)$$

$$= exp\{\upsilon_i(y_i\theta_i - e^{\theta_i}) + c(y_i, \upsilon_i)\}$$

Il s'agit là une forme de la distribution de probabilité des EDM vue plus haut, avec :

- $\phi = 1$
- $b(\theta_i) = e^{\theta_i}$

3.2.4 La fonction de lien

3.2.4.1 Forme de tableau

Dans ce travail, il a jusqu'ici été considéré que les données étaient exprimées sous forme de liste, soit un vecteur $y' = (y_1, ..., y_n)$, chaque ligne i contient également la durée υ_i ainsi que les facteurs d'évaluation correspondants. Cette notation représente la forme de liste, et est celle qui sera préférée dans des logiciels tels que SAS, R, ou encore Python afin de gérer et représenter les données. Il existe cependant une autre manière de représenter les données qui s'avère très utile pour certaines démonstrations : la forme de tableau (Ohlsson et Johansson 2010).

Dans cette forme, les observations $y_{i_{1,...,i_k}}$ sont représentées de manière à avoir un indice par variable, un exemple est donné ci-dessous.

i	Married	Gender	Observation
1	Yes	M	y_1
2	Yes	F	y_2
3	No	M	y_3
4	No	F	y_4

Table 3.2.1 – Forme de liste

5. Pour être précis, il s'agit en réalité d'une transformation de la distribution de Poisson, la distribution "relative" de Poisson.

Married	Male	Female
Yes	y_{11}	y_{12}
No	y_{21}	y_{22}

TABLE 3.2.2 – Forme de tableau

3.2.4.2 Fonction de lien

Maintenant que les notations utilisées dans cette parties ont été expliquées, il faut se pencher sur la deuxième partie de la généralisation des GLMs, la fonction de lien. Il est également considéré que μ_{ij} représente l'espérence de ratio de la cellule (i, j) (forme de tableau), où i et j représentent respectivement la permière et la deuxième variable catégorielle. Pour cet exemple, il est supposé que la première variable comporte deux classes et la deuxième en comporte trois, ce qui est ci-dessous observable dans une représentation en forme de tableau.

Var1/Var2	Catégorie 1	Catégorie 2	Catégorie 3
Catégorie 1	y_{11}	y_{12}	y_{13}
Catégorie 2	y_{21}	y_{22}	y_{23}

TABLE 3.2.3 – Exemple pour la fonction de lien

La fonction de lien sert à généraliser le modèle linéaire basique (OLS), celui-ci suppose que la moyenne suit une structure additive de la forme :

$$\mu_{ij} = \gamma_0 + \gamma_{1i} + \gamma_{2j}$$

Il faut maintenant choisir une cellule comme la cellule de base, qui servira à ajouter des contraintes, en particulier que sa moyenne μ_{ij} soit égale à γ_0. Pour continuer avec cet exemple, choisissons la cellule $(1, 1)$ comme cellule de base, cela signifie que $\mu_{11} = \gamma_0 + \gamma_{11} + \gamma_{21}$ doit donner $\mu_{11} = \gamma_0$. Pour ce faire, il suffit de déclarer :

$$\gamma_{11} = \gamma_{21} = 0$$

Et donc :

$$\mu_{11} = \gamma_0$$

Le modèle est réécrit sous forme de liste et les équations suivantes sont définies :

- $\beta_1 := \gamma_0$
- $\beta_2 := \gamma_{12}$

- $\beta_3 := \gamma_{22}$
- $\beta_4 := \gamma_{23}$

En forme de liste, μ_i représente la moyenne d'une plage de tarifs. Les deux variables catégorielles sont également transformées en variables binaires, de la même manière que dans la partie 2.4.1. Est maintenant défini :

$$\mu_i = \sum_{j=1}^{4} x_{ij}\beta_j \qquad i = 1, 2, ..., 6.$$

Cette équation représente la structure linéaire et peut être écrite sous la forme matricielle :

$$\mu = X\beta$$

Il est maintenant possible de généraliser l'exemple ci-dessus, avec l'équation :

$$\mu_i = \sum_{j=1}^{r} x_{ij}\beta_j \qquad i = 1, ..., n.$$

- r : nombre de variables exploratoires.

La fonction de lien $g(.)$ est maintenant introduite :

$$g(\mu_i) = \eta_i = \sum_{j=1}^{r} x_{ij}\beta_j$$

Ici $g(.)$ représente la fonction de lien et doit être monotone et dérivable. Cette fonction est le lien entre la moyenne et la structure linéaire. Par exemple, dans le cas d'un modèle multiplicatif (et pour reprendre le cas étudié ci-dessus) le but est d'avoir $\mu_{ij} = \gamma_o\gamma_{1i}\gamma_{2j}$, ce qui peut être obtenu grâce au logarithme :

$$log(\mu_{ij}) = log(\gamma_o) + log(\gamma_{1i}) + log(\gamma_{2j})$$

$$log(\mu_i) = \sum_{j=1}^{4} x_{ij}\beta_j \qquad i = 1, 2, ..., 6.$$

$$g(\mu_i) = \eta_i = log(\mu_i)$$

La fonction de lien est donc le logarithme dans le cas des modèles multiplicatifs, bien souvent utilisés dans les problèmes d'assurances (Ohlsson et Johansson 2010).

3.2.5 Estimation des paramètres β

Dans le cadre des GLMs, les paramètres β sont estimés à l'aide de la technique du maximum de vraisemblance. Cela signifie que les estimations sont basées sur l'échantillon de données et implique la nécessité de disposer d'un bon nombre d'entrées. La fonction du maximum de vraisemblance de θ est définie comme suit :

$$l(\theta;\phi;y) = \frac{1}{\phi}\sum_i \upsilon_i(y_i\theta_i - b(\theta_i)) + \sum_i c(y_i,\phi,\upsilon_i)$$

En réécrivant l'équation en fonction de β au lieu de θ et en prenant la dérivée de $l(.)$ par rapport à β, la fonction de score est obtenue :

$$\frac{\partial l}{\partial \beta_k} = \frac{1}{\phi}\upsilon_i \frac{y_i - \mu_i}{v(\mu_i)g'(\mu_i)}x_{ij}$$

- $v(.)$: fonction de variance

Il suffit maintenant de prendre ces r dérivées partielles, de les égaliser à 0 et de les multiplier par ϕ afin d'obtenir les équations du maximum de vraisemblance :

$$\sum_i \upsilon_i \frac{y_i - \mu_i}{v(\mu_i)g'(\mu_i)}x_{ij} = 0, \qquad i = 1,...,r$$

La résolution de ce système d'équations doit se faire de manière numérique et donne l'estimation des paramètres β, qui dépendent du jeu de données utilisé (Ohlsson et Johansson 2010 ; Denuit, Hainaut et Trufin 2019a ; Denuit, Hainaut et Trufin 2019b).

3.2.6 Modélisation de Poisson

Cette sous-section traite du cas où Y suit une loi de Poisson, ce qui est le cas pour notre base de données. En supposant :

$$Y_i \sim Poisson(\lambda_i)$$

Alors les propriétés de la loi de Poisson donnent :

$$E(Y_i) = \lambda_i$$
$$var(Y_i) = \lambda_i$$

Donc la fonction de variance sera donc :

$$v(\mu_i) = \mu_i$$

Et la fonction de lien utilisée dans ce cas-ci sera le logarithme (qui permet de passer du domaine $[0, \infty]$ au domaine $[-\infty, \infty]$) :

$$g(\mu_i) = log(\mu_i)$$

3.3 Réseaux de neurones

Avant de comprendre comment cet algorithme fonctionne d'un point de vue mathématique, il est intéressant de comprendre de quoi est inspiré son fonctionnement. Comme son nom l'indique, le réseau de neurones artificiels est un algorithme d'apprentissage automatique inspiré du fonctionnement du cerveau humain. Un neurone biologique est composé du corps de la cellule qui contient les composants les plus importants, de plusieurs extrémités nommées les dendrites, ainsi que de l'axone (une extension du neurone pouvant être des milliers de fois plus longue que le corps du neurone). L'axone se subdivise en plusieurs branches appelées synapses. Les neurones fonctionnent en recevant des impulsions électriques appelées "signaux", de plus lorsqu'un neurone reçoit un certain nombre de signaux en même temps, celui-ci va déclencher son propre signal (Denuit, Hainaut et Trufin 2019b).

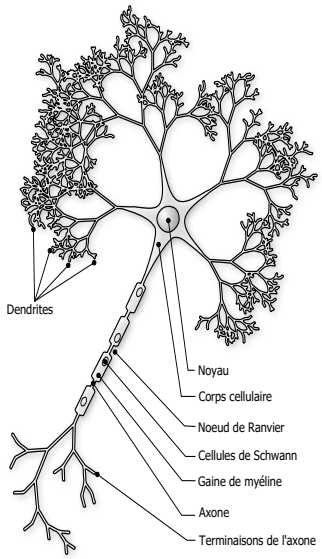


Figure 3.3.1 – Schéma d'un neurone biologique. Source : https ://fr.wikipedia.org/wiki/Fichier :Neuron-figure-fr.svg

3.3.1 Perceptron

Le perceptron est le modèle à la base des réseaux de neurones artificiels modernes, celui-ci a été inventé dès 1957 par Frank Rosenblatt. Il s'agissait alors d'un algorithme

permettant l'apprentissage automatique par l'erreur (apprentissage supervisé). Son fonctionnement est similaire à celui d'un neurone humain, puisqu'il en est inspiré. Mathématiquement, un perceptron est la somme d'un biais et de multiplications entre les valeurs des variables et les poids du perceptron passant par une fonction dite "d'activation" qui détermine la réponse du neurone. La fonction d'activation représente le potentiel d'activation du neurone. L'exemple le plus simple serait une fonction renvoyant 1 si l'entrée est positive et 0 (ou -1) si l'entrée est négative (bien que cette fonction ne sera pas utilisée dans les algorithmes de ce travail car elle n'est pas différentiable, ce qui est un problème pour la calibration), c'est d'ailleurs cette fonction qui a été utilisée en premier pour les perceptrons (Géron 2017).

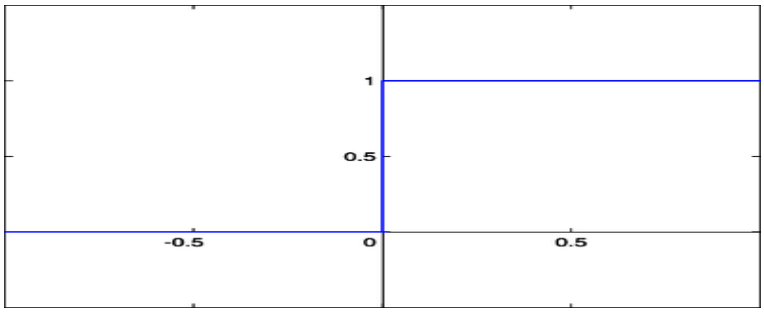


Figure 3.3.2 – Exemple de fonction d'activation : Step function

Pour en revenir au perceptron, les données d'un individu sont considérées comme un vecteur X_i et l'ensemble des données constituent la matrice de données X. L'attribut j de la personne i est noté x_{ij}. Le vecteur des poids du perceptron est noté W et w_n représente le poids n. Il existe aussi un biais noté w_0 qui n'est pas multiplié par une valeur x_i mais par 1, il n'interagit donc pas avec les valeurs prises par les variables. La fonction d'activation est quant à elle notée $\varphi(.)$. Il est important de noter que le nombre de poids correspond à J, soit le nombre de variables explicatives utilisées. Un perceptron peut être représenté comme ceci :

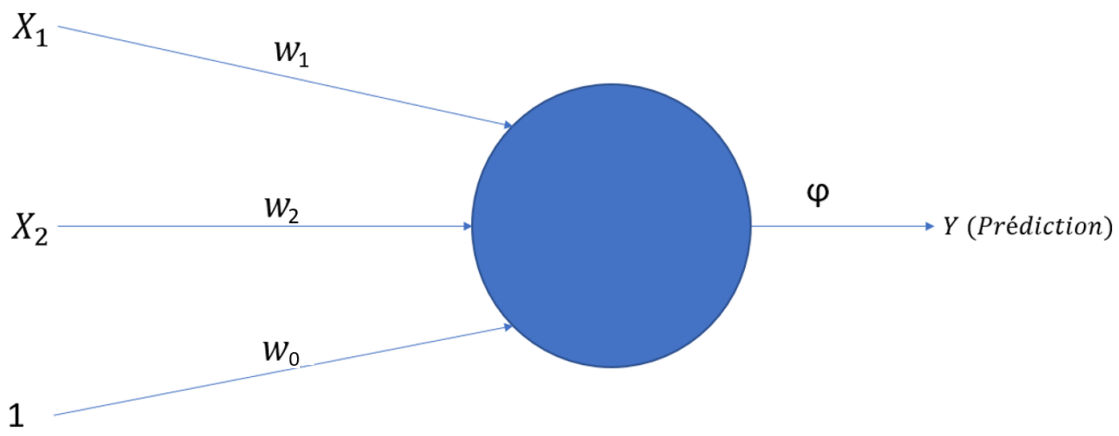


FIGURE 3.3.3 – Représentation d'un perceptron.

La manière dont un perceptron calcule la réponse Y peut être décomposée en deux étapes :

- Les poids sont multipliés par les valeurs des entrées à laquelle on ajoute le biais : $\omega = \sum_{j=1}^{J} x_j w_j + w_0$
- La valeur ainsi obtenue est prise comme argument dans la fonction d'activation : $Y = \varphi(\omega)$

C'est donc cette réponse Y qui sera considérée comme le résultat du perceptron. Le perceptron reste cependant un modèle simple et il ne peut être utilisé que pour des problèmes relevant de classification séparable linéairement, et son fonctionnement pourrait être schématisé par la recherche d'un seuil représentant la valeur des variables à partir de laquelle un exemple serait considéré comme appartenant à une classe ou à l'autre.
Lorsqu'il est question d'apprentissage automatique, cela signifie en fait que les paramètres du modèle sont optimisés en fonction de notre base de données. Pour le perceptron cela veut dire trouver les valeurs optimales w pour un problème spécifique. Pour être plus précis, l'algorithme va prendre en entrée une ligne de notre base de données et faire une prédiction. Si cette prédiction est correcte alors les poids ne sont pas modifiés, cependant, dans le cas ou la prédiction est incorrecte une règle de mise à jour des poids sera appliquée. Ce procédé sera expliqué plus en détail dans la sous-section suivante, mais nous pouvons d'ores et déjà observer l'équation de mise à jour des poids d'un perceptron simple (GÉRON 2017) :

$$w_i^{next} = w_i + \eta(\hat{y}_j - y_j)x_i$$

- w_i : le poids du neurone i
- η : le taux d'apprentissage, il s'agit d'un paramètre déterminant à quel point les poids sont modifiés lors d'une erreur, celui-ci est inclus dans l'intervalle $[0, 1]$.
- $\hat{y_j}$: la prédiction du perceptron
- y_j : la valeur cible

Comme mentionné précédemment, le perceptron reste cependant un modèle simple, et celui-ci est incapable de résoudre certains problèmes pourtant basiques. L'exemple le plus connu est le problème XOR [6], insolvable par le perceptron et mentionné par Minsky et Papert dans leur papier intitulé "Perceptrons" qui met en avant certaines faiblesses du perceptron. Suite à ces découvertes la recherche dans le domaine sera mise de côté. Cependant il est plus tard apparu que ces faiblesses pouvaient pour la plupart (y compris le problème XOR) être éliminées en empilant plusieurs perceptrons en couches de neurones artificiels (Géron 2017).

3.3.2 Perceptron multicouche

Fonctionnement Le perceptron multicouche, plus communément appelé MLP [7] est une structure composée de couches, elles-mêmes formées par plusieurs perceptrons. Pendant longtemps, les chercheurs ont tenté de trouver une manière efficace d'effectuer l'entraînement de ce type de réseau sans succès, jusqu'au jour où D. E. Rumelhart a découvert une méthode appelée la rétropropagation (1986) qui sera expliquée plus loin dans ce chapitre. Le schéma ci-dessous montre la structure d'un tél réseau :

6. "OU" exclusif, opérateur logique de l'algèbre de Boole. Pour deux opérandes, celui-ci permet à chacun d'avoir la valeur VRAI, mais pas les deux en même temps.

7. Multilayer perceptron

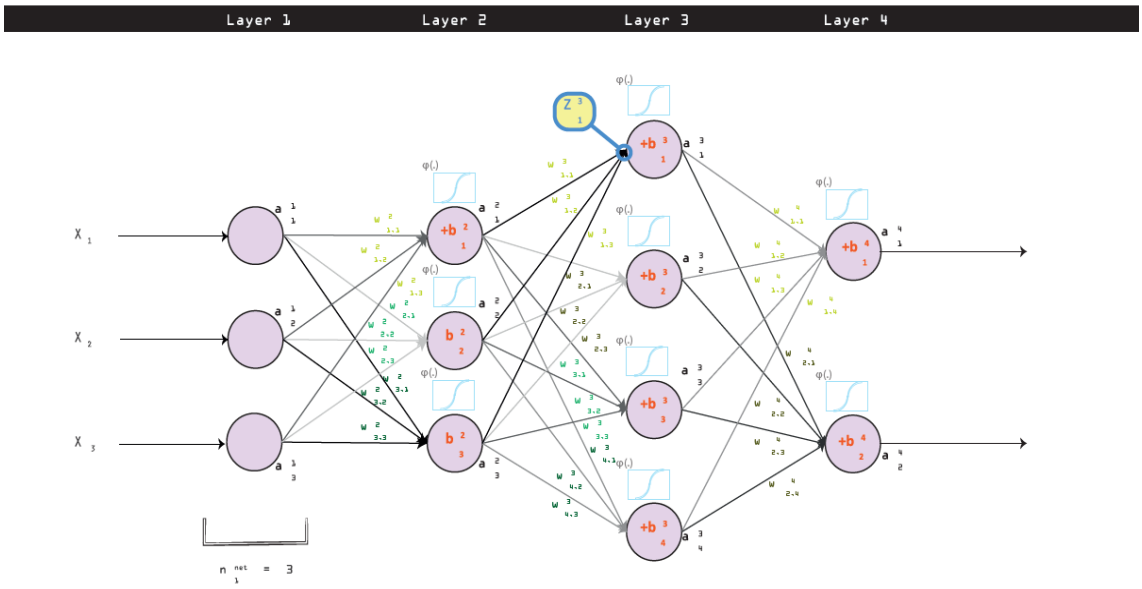


FIGURE 3.3.4 – Représentation d'un MLP "feed-forward".

Un réseau de ce type résout la plupart des problèmes du perceptron lorsqu'il est utilisé seul et compose un modèle beaucoup plus puissant, capable de détecter les relations non-linéaires et de résoudre un grand nombre de problèmes de classification et de régression. Ce type de réseau est utilisé dans un grand nombre d'applications récentes telles que les voitures autonomes, la reconnaissance vocale ou encore la vision artificielle [8]. Comme observable sur le graphique, l'information circule dans un seul sens sur le schéma. Il faut savoir qu'il existe différents types de MLP, dont certains avec des relations circulaires entre les neurones. Cependant ce type de réseau ne sera pas utilisé dans ce travail et ne sera donc pas détaillé, les réseaux de neurones artificiels utilisés seront uniquement des réseaux "feed-forward" ; l'information arrive à l'entrée du réseau, transite à travers celui-ci dans une direction unique, puis, le modèle renvoie une réponse $\hat{y}_i$ en sortie.
Un réseau tel que celui-ci est composé de plusieurs couches à travers lesquelles l'information passe successivement. La première couche est la couche d'entrée, celle qui va recevoir les données. Ensuite, il y a une ou plusieurs couches cachées [9] et enfin la couche de sortie renvoyant la réponse du réseau. Le nombre de neurones par couche et le nombre de couches cachées sont tous deux des hyperparamètres [10] à définir.
Le fonctionnement de ce réseau est similaire à celui du perceptron, cependant, son architec-

8. Computer vision.
9. Si il y a une seule couche cachée on parle de réseau peu profond (shallow network) et s'il y a plusieurs couches cachées on parle alors de réseau profond (deep network).
10. Paramètres choisis à la création du modèle par l'utilisateur, ceux-ci ne peuvent être optimisés facilement de manière automatique et nécessitent la création de plusieurs modèles avec différents hyperparamètres dont les résultats seront comparés sur la fonction de perte.

ture est plus complexe et nécessite certains changements dans les notations utilisées, aussi :

- w^i_{jk} : poids du neurone k dans le couche $(i-1)$ vers le neurone j de la couche i.
- b^i_j : le biais du neurone j dans la couche i.
- a^i_j : la valeur de la fonction d'activation du neurone j dans la couche i.
- $z^i_j = \sum_k (w^i_{jk} \cdot a^{i-1}_k) + b^i_j$: La valeur d'activation avant de passer par la fonction d'activation.
- $\varphi(.)$: la fonction d'activation.
- n^{net} : le nombre de couches du réseau (entrée et sortie comprises).
- n^{net}_j : le nombre de neurones dans la couche j.

Chaque neurone de ce réseau fonctionne en fait comme un perceptron, relié à d'autres perceptrons organisés en forme de couches, et qui transfère le résultat de sa sortie à la couche suivante. En partant de ce principe et en adaptant les notations, la conclusion suivante est trouvée :

$$a^i_j = \varphi(\sum_k (w^i_{jk} \cdot a^{i-1}_k) + b^i_j)$$

représentant la sortie du neurone j dans la couche i. Si $i = n^{net}$ alors il représente la sortie du réseau, ou sa réponse. La couche de sortie peut être composée d'un seul neurone mais cela n'est pas obligatoire. Si le but recherché est par exemple d'estimer à quelle classe un échantillon appartient il est possible de représenter chaque classe par un neurone dans cette couche.

Le fonctionnement d'un réseau de neurones étant expliqué, il est maintenant nécessaire de comprendre comment un tel réseau est calibré, car sans cette phase de calibration le réseau de neurones serait inefficace. Il existe diverses méthodes permettant d'optimiser un réseau de neurones mais nous ne détaillerons que la "backpropagation" ou rétropropagation, algorithme ayant permi la reprise de la recherche dans le milieu lors de son apparition (Denuit, Hainaut et Trufin 2019b ; Géron 2017).

Rétropropagation Avant l'apparition de la rétropropagation, les réseaux de neurones étaient calibrés grâce à la descente du gradient, algorithme consistant à calculer le gradient, la matrice Hessienne et son inverse afin de mettre à jour les paramètres du modèle (poids w^i_{jk}). Cette méthode présente cependant un inconvenient de taille : l'inversion numérique de la matrice Hessienne est très longue voir impossible lorsque celle-ci est mal conditionnée. Ces problèmes peuvent être contournés en utilisant un autre algorithme paru en 1986 : la rétropropagation. Pour une plus grande clarté dans la description de cet algorithme, le vecteur contenant les poids w^i_{jk} sera nommé Ω.

La rétropropagation consiste en une série d'instructions répétées T fois, et t sera utilisé en indexe des paramètres pour montrer la valeur d'un paramètre à l'époque [11] t. Le vecteur Ω_t est modifié d'un petit pas dans la direction opposée à celle du gradient à chaque époque

11. Époque : une période ou moment particulier dans l'exécution de l'algorithme.

pour devenir Ω_{t+1}, et la taille de ce pas est généralement déterminée par une fonction dégressive diminuant au fur et à mesure que t augmente.

Algorithm 3.2 Rétropropagation (Denuit, Hainaut et Trufin 2019b)

1: Attribution de valeurs aléatoires à Ω_0.
2: Choix du pas initial, ρ_0.
3: Choix de la fonction du pas $g(\rho_0, t)$.
4: **begin**
5: For t = 0, ..., T
6: Calcul du gradient : $\nabla R(\Omega_t)$
7: Mise à jour de la taille du pas : $\rho_{t+1} = g(\rho_0, t)$
8: Mise à jour des poids : $\Omega_{t+1} = \Omega_t - \rho_{t+1}\nabla R(\Omega_t)$
9: **end**

3.4 Boosting

Dans cette section, le boosting sera introduit et expliqué, et plus particulièrement AdaBoost et ses variations.

3.4.1 Bagging

Si le bagging n'est pas au centre de ce travail, il est tout de même intéressant de présenter cet algorithme car il est souvent comparé au boosting qui sera introduit dans la sous-section suivante. En effet, le bagging (Bootstrap Aggregating) est une méthode assez populaire dans le domaine de l'apprentissage automatique, connue pour fonctionner particulièrement bien lorsque la variance est élevée. Pour mieux comprendre pourquoi l'algorithme fonctionne de cette manière, le bagging va être introduit ci-dessous. (Breiman 1996)

- $\mathcal{L}$: Set d'apprentissage, il est composé de données sous la forme $\{(y_n, x_n), n = 1, ..., N)\}$.
- x : Les facteurs d'évaluation.
- y : La valeur à prédire.

Sur base de ceci, $\varphi(x, \mathcal{L})$ est défini, il s'agit du prédicteur qui sert à déterminer une réponse y sur base de x et $\mathcal{L}$. En supposant que plusieurs sets d'apprentissages dénotés $\{\mathcal{L}_k\}$ sont disponibles, alors il est possible de créer k prédicteurs $\varphi(x_k, \mathcal{L}_k)$ et de les combiner afin d'obtenir un résulat pour la prédiction. Dans le cas de la régression, ce résultat peut être obtenu en prenant simplement la moyenne des résultats des différents prédicteurs :

$$\hat{y} = K^{-1} \sum_{1}^{K} \varphi(x_k, \mathcal{L}_k) = E_{\mathcal{L}} \varphi(x, \mathcal{L})$$

Un raisonnement similaire peut être tenu pour le cas de la classification mais celui-ci ne sera pas expliqué ici car il ne rentre pas dans le cadre de ce travail. Le problème principal du bagging est qu'il nécessite k sets d'apprentissage, ce qui est rarement possible en pratique. Heureusement, il est possible de palier à ce problème grâce à une technique de génération de sets d'apprentissage : le bootstrap.

Celui-ci consiste en un échantillonnage aléatoire (équiprobable) avec remise. Les sets d'entraînements générés sont donc composés de n entrées sélectionnées de manière aléatoire parmi le set d'entraînement de base. Une variante pondérée de cette technique d'échantillonnage est également utilisée dans le boosting introduit dans le reste de cette section.

3.4.2 Boosting

Afin d'améliorer les performances de certains algorithmes, de nombreux chercheurs se sont penchés sur les méthodes dites de "boosting". Il s'agit ici plus de "méta-algorithme" que d'algorithme à proprement parler, c'est-à-dire qu'il s'agit d'une famille d'algorithmes utilisant d'autres algorithmes (apprenants faibles) dans certaines de ses étapes afin d'améliorer les performances que l'apprenant faible choisi aurait eu sans l'utilisation du méta-algorithme. Ces méthodes fonctionnent par l'entrainement d'apprenants faibles [12] combinés afin de créer un ensemble de prédicteurs considéré comme "fort". Ils reposent souvent sur une règle de décision elle-même basée sur les réponses des différents prédicteurs, dans le cas d'une classification binaire cela peut, par exemple, être représenté par un vote majoritaire de ces derniers. Avec le boosting, les prédicteurs sont entrainés l'un après l'autre sur un sous-ensemble des données du jeu d'entraînement. Une fois le prédicteur entraîné, toutes les données du jeu d'entraînement sont passées dans celui-ci et l'erreur du prédicteur sur les prédictions est enregistrée. Concrètement, c'est la distance entre l'observation et sa prédiction qui est observée, puisque nous tentons avec le boosting de donner l'accent aux données mal prédites pour l'entraînement des prochains prédicteurs. Les probabilités des données les plus incorrectes sont ensuite modifiées afin d'augmenter leur chances d'apparaître dans le sous-ensemble utilisé pour l'entraînement du prochain prédicteur. L'algorithme de boosting le plus connu est probablement AdaBoost, qui tient son nom de "Adaptative Boosting" [13], il en existe plusieurs versions à la fois pour la classification et la régression et c'est ce méta-algorithme qui sera l'objet principal de ce mémoire.

12. Algorithme générant des prédicteurs de relativement mauvaise qualité (mais avec de meilleures performances que le hasard pur), la plupart du temps utilisé en combinaison avec d'autres apprenants faibles afin de créer un ensemble de prédicteurs "fort".

13. Boosting adaptatif.

3.4.3 AdaBoost

À l'origine, AdaBoost est un algorithme qui a été pensé pour la classification binaire. Son fonctionnement ne prévoyait pas l'estimation d'une valeur $\hat{y}$ en régression. L'algorithme est présenté par Freund et Schapire (Freund et Schapire 1995), et la différence principale avec les autres techniques de boosting, c'est qu'il tient compte de l'erreur de chaque apprenant faible afin de tenter de les corriger lors de la création des prochains apprenants faibles. Cet algorithme a gagné énormément de popularité au cours des années en remportant un grand nombre de compétitions (surtout dans le domaine de la classification). Celui-ci a notamment montré sa supériorité par rapport au "bagging", un autre méta-algorithme populaire. Il faut cependant noter que puisque dans AdaBoost les résultats de chaque modèle influent la création du suivant, il est impossible d'entraîner les modèles de manière parallèle sur les différents coeurs d'un processeur ou d'un GPU (contrairement au bagging) [14](Drucker 1997).

AdaBoost suppose qu'un apprenant de base (apprenant faible) peut être utilisé pour créer une hypothèse de base (hypothèse faible). Cette hypothèse faible ne doit pas être particulièrement complexe ou efficace sur le jeu de données, la seule supposition faite est qu'elle doit être légèrement plus efficace qu'une sélection aléatoire de la classe finale. Cette sélection aléatoire, dans le cadre d'une classification binaire, aura une précision $\epsilon_t = 1/2$. Si γ est un nombre positif proche de 0, alors cette supposition appelée "supposition de l'apprentissage faible" peut être écrite comme :

$$\epsilon_t \leq \frac{1}{2} - \gamma$$

Afin d'améliorer la précision d'AdaBoost, des apprenants faibles sont construits, et ceux-ci nécessitent un jeu de données afin d'être entraînés. C'est ici qu'un autre problème se pose car si les apprenants faibles sont tous entraînés sur le même jeu de données alors on peut supposer que ceux-ci seront toujours très similaires et ne permettront pas d'obtenir de bons résultats. Il est donc nécessaire de modifier le jeu de données utilisé après chaque itération de l'algorithme. AdaBoost contourne cette barrière en construisant un nouveau jeu d'entraînement à chaque itération en respectant une distribution D_t modifiée a chaque itération de l'algorithme, cela permet aux apprenants de bases de produire des hypothèses de base différentes et variées, ce qui au final permet d'obtenir de bien meilleurs résultats. C'est d'ailleurs l'idée principale du boosting, choisir les jeux d'entraînement de telle manière que l'on force l'apprenant faible à déduire une nouvelle hypothèse faible.

AdaBoost fonctionne de la manière suivante :

14. Graphics Processing Unit, ou processeur graphique, souvent utilisé pour faire des calculs parallèles de manière extrêmement rapide.

Algorithm 3.3 AdaBoost (FREUND et SCHAPIRE 1995)

Avec : $(x_1, y_1), ..., (x_m, y_m)$ où $x_i \in \chi, y_i \in \{-1, +1\}$
Init : $D_1(i) = 1/m, i = 1, ..., m$
For $t = 1, ..., T$

- Entraînement d'un apprenant faible en utilisant D_t.
- Récupération de l'hypothèse faible de l'apprenant faible $h_t : \chi \to \{-1, +1\}$.
- Calcul de l'erreur pondérée : $\epsilon_t \doteq Pr_{i \sim D_t}[h_t(x_i) \neq y_i]$.
- Calcul de l'importance : $\alpha_t = \frac{1}{2} ln \left(\frac{1-\epsilon_t}{\epsilon_t} \right)$.
- Mise à jour de la distribution D_t :
  - Si $h_t(x_i) = y_i$: $D_{t+1}(i) = \frac{D_t(i)}{Z_t} e^{-\alpha_t}$
  - Si $h_t(x_i) \neq y_i$: $D_{t+1}(i) = \frac{D_t(i)}{Z_t} e^{\alpha_t}$

  Où Z_t représente le facteur de normalisation qui permet de garder l'égalité $\sum_{i=1}^{m} D_{t+1}(i) = 1$ afin que D_{t+1} reste une distribution.
- Hypothèse finale :

$$H(x) = sign \left(\sum_{t=1}^{T} \alpha_t h_t(x) \right)$$

Cette version de l'algorithme est différente de celle qui sera utilisée dans le cadre de ce travail puisqu'elle n'est pas prévue pour la régression mais il est tout de même intéressant de comprendre les différentes étapes de celui-ci car la logique reste très similaire entre les différentes versions d'AdaBoost. Premièrement, un jeu de données est sélectionné en suivant la distribution D_t.

Il est important de noter que le but de l'apprenant faible est simplement de déterminer une hypothèse faible $h_t : \chi \to \{-1, +1\}$ dépendante de la distribution D_t. À chaque itération cette hypothèse sera différente puisque $D_t \neq D_{t+1}$. La qualité de l'hypothèse est déterminée par le calcul de l'erreur pondérée (SCHAPIRE et FREUND 2013) :

$$\epsilon_t \doteq Pr_{i \sim D_t}[h_t(x_i) \neq y_i] = \sum_{i : h_t(x_i) \neq y_i} D_t(i)$$

Cette équation indique deux choses intéressantes :

- ϵ_t représente la chance de se tromper dans la classification d'un élément choisi de manière aléatoire selon la distribution D_t.
- Puisque D_t est une fonction de distribution et que sa somme est égale à 1, cette chance est représentée par la somme des poids $D_t(i)$ des éléments mal classés.

Ensuite, AdaBoost détermine l'importance α_t associée à l'hypothèse h_t, ce qui permet de donner plus ou moins d'importance à chaque hypothèse dans la décision finale. Plus une hypothèse est efficace (plus son erreur est faible) et plus α_t est élevé ce qui lui donne plus

d'importance. Le cas inverse est aussi vrai, moins une hypothèse est efficace (plus elle se rapproche des performances d'une sélection aléatoire de la réponse) et plus l'importance de l'hypothèse sera faible. La fonction d'importance dépend uniquement de l'erreur pondérée.

Pour la prochaine étape, il est nécessaire de mettre à jour la distribution D_t en fonction des performances de l'hypothèse h_t pour obtenir une distribution D_{t+1} qui permet une hypothèse h_{t+1} efficace sur les exemples mal classés par l'hypothèse précédente. L'idée principale ici est de donner plus d'importance aux éléments mal classés par h_t et moins d'importance aux éléments classés correctement par h_t. Pour ce faire, l'équation suivante est utilisée :

$$D_{t+1}(i) = \frac{D_t(i)exp(-\alpha_t y_i h_t(x_i))}{Z_t}$$

Celle-ci peut être scindée en deux cas (si l'exemple i est correctement classé ou non) comme indiqué dans l'algorithme 3.3 (Schapire et Freund 2013). Z_t est le facteur de normalisation, c'est-à-dire que c'est un facteur qui sert à maintenir l'équation $D_{t+1} = 1$. De ce fait, il est facile de déduire celui-ci : $Z_t = \sum_{i=1}^{m} \sigma_{t+1}(i)$ où σ_{t+1} représente :

$$\sigma_{t+1}(i) = D_t(i)exp(-\alpha_t y_i h_t(x_i))$$

soit D_{t+1} avant d'être divisé par Z_t.

Une fois tous les apprenants faibles entraînés, AdaBoost les combine en un modèle final H. Ceci est simplement représenté par un vote de chaque apprenant de base, pondéré par α_t. Ce vote pondéré simple est possible car cet algorithme est utilisé dans le cadre de la classification, et que les hypothèses sont représentées par $h_t : \chi \to \{-1, +1\}$.

$$H(x) = sign\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$

Pour résumer, AdaBoost se repose sur trois idées principales :

- AdaBoost combine un grand nombre d'apprenants faibles afin d'obtenir une classification la plus précise possible. Les apprenants faibles sont des modèles avec une prise de décision simple. Dans le cas de la classification, il s'agit souvent d'une série d'arbres de décision effectuant leur choix en se basant sur une seule variable, les apprenants faibles sont généralement rapides et simples à entraîner puisque c'est potentiellement un très grand nombre d'entre eux qui seront utilisés. Ceci dit, le boosting est aussi possible avec des algorithmes plus puissants, le seul réel problème dans ce cas étant le temps de calcul et que les apprenants de base soient plus efficaces qu'une sélection aléatoire (équiprobable) de la réponse.

- Tous les algorithmes entrainés n'ont pas le même poids lors de la prise de décision, en effet ceux qui sont les plus confiants quant à leur décision seront plus impactants lors de la décision finale.
- Chaque apprenant faible est entraîné en prenant en compte les erreurs commises par les apprenants faibles précédents. Ceci est représenté par le calcul de la distribution D_{t+1}.

L'algorithme a été dévié dans plusieurs formes portant toutes des noms différents, et parmi ces formes, certaines permettent de travailler avec des problèmes de régression. C'est notamment le cas de Adaboost.R2, présenté par Drucker, qui sera l'algorithme utilisé dans ce travail. Celui-ci est présenté ci-dessous dans la sous-section 3.4.4.

3.4.4 AdaBoost.R2

AdaBoost.R2 est un algorithme présenté par Drucker et permettant d'appliquer l'algorithme d'AdaBoost à un problème de régression. Il est très similaire dans son fonctionement à celui de l'algorithme originel d'AdaBoost présenté ci-dessus.

Algorithm 3.4 AdaBoost.R2 (DRUCKER 1997)

Avec : $(x_1, y_1), ..., (x_{N_1}, y_{N_1})$ où $x_i \in \chi, y_i \in \mathbb{R}$
Init : $w_i = 1 \ i = 1, ..., N_1$
While $\bar{L} < 0.5$

- Normalisation des poids $p_i = w_i / \sum w_i$.
- Création d'un jeu d'entraînement de taille N_1 selon les probabilités p_i.
- Création d'un apprenant faible t avec une hypothèse faible $h_t : x \to y$.
- Récupération des prédictions $y_i^{(p)}(x_i) \ i = 1, ..., N_1$.
- Calcul de la perte $L_i = L\left[|y_i^{(p)}(x_i) - y_i|\right]$, $L \in [0, 1]$
  - Si $D = sup|y_i^{(p)}(x_i) - y_i| \ i = 1, ..., N_1$ alors voici trois fonctions candidates :
    1. Linéaire : $L_i = \frac{|y_i^{(p)}(x_i) - y_i|}{D}$
    2. Carrée : $L_i = \frac{|y_i^{(p)}(x_i) - y_i|^2}{D^2}$
    3. Exponentielle : $L_i = exp\left[\frac{-|y_i^{(p)}(x_i) - y_i|}{D}\right]$
- Calcul de la perte moyenne : $\bar{L} = \sum_{i=1}^{N_1} L_i p_i$.
- Calcul de l'importance : $\beta = \frac{\hat{L}}{1-\hat{L}}$ (Plus beta est petit, plus l'importance est grande).
- Mise à jour des poids : $w_{i+1} = w_i \beta^{[1-L_i]}$ (Plus l'erreur est faible, plus le poids est réduit)
- Hypothèse finale h_f comme combinaisons des T hypothèses faibles h_t :

$$h_f = inf\left\{y \in Y : \sum_{t:h_t \leq y} log\left(\frac{1}{\beta_t}\right) \geq \frac{1}{2}\sum_t log\left(\frac{1}{\beta_t}\right)\right\}$$

Comme mentionné précédemment, il est clair que les deux algorithmes sont très similaires, et fonctionnent tout deux comme ceci :

- Échantillonnage d'un jeu d'entraînement selon les probabilités associées à chaque élément, D_t pour AdaBoost et p_i pour AdaBoost.R2.
- Création d'un apprenant faible et calcul de la perte.
- Calcul de l'importance de l'apprenant faible.
- Mise à jour des poids : c'est cette étape qui différencie réellement le boosting du bagging, en effet, le fait de modifier les probabilités de sélection des éléments de manière individuelle permet aux apprenants faibles suivants de se concentrer sur les exemple mal prédits.
- Agrégation des hypothèses faibles en une hypothèse forte pour la décision finale : dans AdaBoost, cela correspond à un vote majoritaire (pondéré par l'importance de l'hypothèse) et dans AdaBoost.R2 la formule suivante est utilisée :

$$h_f = inf \left\{ y \in Y : \sum_{t:h_t \leq y} log\left(\frac{1}{\beta_t}\right) \geq \frac{1}{2} \sum_t log\left(\frac{1}{\beta_t}\right) \right\}$$

Soit la médiane pondérée. Pour se faire une idée de la manière donc cette étape fonctionne dans AdaBoost.R2, une interprétation est donnée.

Afin de déterminer la réponse finale, toutes les hypothèses faibles sont utilisées afin d'obtenir une prédiction $y_i^{(t)}$. Chaque β_t (importance de l'hypothèse de base) est ensuite associée à la prédiction de son hypothèse faible. Les exemples sont ensuite triés de la manière suivante :

$$y_1^{(t)} \leq y_2^{(t)} \leq ... \leq y_i^{(t)}$$

Ensuite, afin d'obtenir la réponse finale de l'hypothèse forte, il suffit d'additionner (en suivant l'ordre ci-dessus) les $log(\frac{1}{\beta_t})$ tant que l'inégalité de h_f n'est pas respectée. La prédiction $y_i^{(t)}$ associée au premier t qui satisfait l'inégalité sera considéré comme la réponse finale renvoyée par l'algorithme. Cette formule, comme mentionné précédemment, représente la médiane pondérée, ce qui signifie que si tous les β_t étaient égaux (chaque hypothèse disposant de la même importance) alors la réponse finale correspondrait à la médiane simple des prédictions. Une grande différence entre AdaBoost et AdaBoost.R2 peut cependant être mentionnée : dans AdaBoost, T soit le nombre d'apprenants faibles, est choisi en avance. Dans AdaBoost.R2, des apprenants faibles seront créés tant que la perte moyenne $\bar{L}$ ne dépasse pas 0.5 (mais un nombre maximum est généralement spécifié en pratique).

3.4.5 Rasoir d'Ockham

Lorsqu'un modèle est entrainé, il est important que celui-ci soit assez général pour avoir de bonnes performances sur le jeu d'entraînement, mais aussi et surtout sur le jeu de test qui sert à évaluer les performances réelles de l'algorithme. Le principe du rasoir d'Ockham est un principe qui vient de la philosophie, mais dont le raisonnement a été largement adopté dans la communauté de recherche sur l'apprentissage automatique. Il soutient que "les hypothèses suffisantes les plus simples doivent être choisies". En d'autres termes, il s'agit de construire un modèle avec assez de paramètres pour être efficace mais assez simple pour pouvoir être généralisable. Ce principe a été largement vérifié dans la plupart des cas, mais il est intéressant de noter que le boosting peut constituer une exception à cette règle, comme cela sera expliqué dans cette sous-section (Schapire, Freund et al. 1998).

Dans une étude menée par Schapire, Freund, Bartlett et Lee, les performances d'AdaBoost ont été comparées aux performances du bagging, et plus particulièrement, c'est un fait intéressant concernant AdaBoost qui a été à l'origine de cette recherche. En observant les graphiques de performances d'AdaBoost et du bagging entrainés avec le même type d'apprenant faible et sur le même jeu de données. Il est possible de voir que dans certains

cas, l'erreur sur le jeu de test d'Adaboost continuait de descendre en tendant vers une asymptote après que l'erreur sur le jeu d'entrainement était déjà nulle, ce qui n'a pas été observé pour le bagging et constitue un comportement à la fois particulier et avantageux (Schapire, Freund et al. 1998). Il était jusque-là supposé que les méthodes de vote d'ensemble [15] fonctionnaient en réduisant la variance des algorithmes, et que ces méthodes étaient donc plus efficaces lorsque la variance de la prédiction était élevée (L. Breiman 1996). Cette explication, qui peut être utilisée comme argument en parlant du bagging (qui est connu pour fonctionner particulièrement bien lorsque la variance est grande), n'est cependant pas applicable dans le cas du boosting. En effet celui-ci ne requiert pas que la variance soit élevée afin de délivrer de bonnes performances.

Dans son rapport de recherche, et pour prouver son hypothèse, Breiman applique le boosting à un algorithme connu pour sa variance faible (Analyse discriminante linéaire) et les résultats obtenus semblent pencher en sa faveur puisque les performances de l'algorithme combiné au boosting sont plutôt mauvaises. Cependant, l'autre étude évoquée dans cette sous-section met en évidence le fait que la variance faible de l'algorithme ne serait pas en cause. Le boosting permettrait dans certains cas de faire diminuer l'erreur de test après que l'erreur d'entraînement soit déjà nulle, comme mentionné plus haut. En plus du fait qu'AdaBoost a été testé avec succès sur des problèmes à faible variance, les chercheurs évoquent deux hypothèses qui expliqueraient pourquoi le boosting peut ne pas être performant dans certains cas (Schapire, Freund et al. 1998).

1. Trop peu de données sont mises à disposition de l'algorithme pour l'apprentissage.
2. Les erreurs d'entrainement des apprenants faibles deviennent trop élevées rapidement (suite aux rééchantillonnages).

3.4.6 Erreur naturelle

En suivant la méthode d'AdaBoost, nous pouvons réduire l'erreur à chaque nouveau prédicteur ajouté. Cependant il serait naïf de penser que l'on peut utiliser cette méthode infiniment afin d'approcher une erreur nulle, d'abord à cause d'un possible overfit empêchant la généralisation sur le jeu de test, mais surtout car il est possible de démontrer qu'il existe une erreur minimum due aux bruits des données, qui est donc inévitable. Adaboost est donc très sensible au bruit des données comme il l'a été démontré. (Drucker 1997) :

- N_1 : le nombre d'observations du jeu d'entraînement.
- N_2 : le nombre d'observations du jeu de test.
- y_i : la valeur prise dans les données pour l'observation i.
- $y_i^{(t)}$: la valeur réelle exacte pour l'observation i.
- $y_i^{(p)}(x_i)$: la prédiction pour l'observation i.

Définissons l'erreur de prédiction (PE) et l'erreur d'échantillonnage du modèle (ME) :

15. Décision basée sur l'agrégation du vote de plusieurs modèles, comme dans le boosting.

$$PE = \frac{1}{N_2}\sum_{i=1}^{N_2}[y_i - y_i^{(p)}(x_i)]^2$$

$$ME = \frac{1}{N_2}\sum_{i=1}^{N_2}[y_i^{(t)} - y_i^{(p)}(x_i)]^2$$

Si le bruit peut être considéré comme additif alors :

$$y_i = y_i^{(t)} + n_i$$

où n_i est le bruit de l'observation i. De plus, si nous ajoutons que :

$$E[n] = 0$$
$$E[n_i n_j] = \delta_{ij}\sigma^2$$

Il est possible d'obtenir l'espérance par rapport à (y, x) pour terminer avec :

$$E[PE] = \sigma^2 + E[ME]$$

Ce qui laisse supposer l'existence d'une erreur de prédiction minimum due au bruit et représentée par σ^2. Il ne sert donc à rien d'augmenter indéfiniment le nombre d'estimateurs utilisés pour AdaBoost (Drucker 1997).

Chapitre 4

Étude de cas

Maintenant que les notions qui seront utilisée au cours de cette analyse ont toutes été présentées, l'étude de cas peut être réalisée. Pour rappel, les données utilisées sont issues d'une assurance moto et la variable prédite est la fréquence des sinistres, qui correspond au nombre de sinistres déclarés divisé par le durée du contrat. Dans ce chapitre la préparation des données sera revue et de nouveaux éléments relatifs à celle-ci seront introduits. Par la suite, des modèles seront construits pour chaque algorithme présenté, et les résultats seront analysés. Certaines modifications de l'algorithme original AdaBoost.R2 seront également expliquées et mises en pratique. Des notes concernant le code utilisé pour l'analyse sont disponibles dans l'annexe 7.

4.1 Préparation des données (suite)

Lorsqu'un modèle est entraîné sur un jeu de données, celui-ci est ensuite capable de prédire des valeurs pour de nouvelles observations. L'exactitude de ces observations dépends de beaucoup de facteurs, mais de manière générale, un mauvais modèle renverra de mauvaises prédictions. Comme déjà évoqué au cours de ce mémoire, un modèle peut par exemple souffrir de surentraînement ce qui se traduirait par de bonnes prédictions sur le jeu d'entraînement mais une mauvaise généralisation aux nouveaux cas. Dans le cadre de l'assurance, les bases de données sont composées d'événements rares, ce qui rends les modèles entraînés sur celles-ci très sensibles au sûrentraînement. Ce problème peut être en partie résolu grâce à la validation croisée qui a été présentée plus tôt, cependant si celle-ci est réalisée sur le jeu de données au complet, la valeur renvoyée par la fonction de perte sera toujours biaisée.

C'est pour cette raison que dans la vaste majorité des cas, les bases de données sont divisées en un jeu d'entraînement et un jeu de test.

- Jeu d'entraînement : Utilisé pour entraîner le modèle, et affiner les paramètres de celui-ci.
- Jeu de test : Utilisé uniquement pour le calcul de la perte du modèle. Le fait que

ces données n'influent pas la prise de décision sur les paramètres choisis garanti que le calcul de la perte du modèle est correct.

De manière générale, les bases de données sont donc divisées suivant des proportions proches de 80% pour le jeu d'entraînement et 20% pour le jeu de test. Dans notre cas, les événements sont plus rares et il est donc important d'en inclure un maximum dans l'entraînement pour maximiser l'efficacité du modèle.

C'est donc une proportion 87.5% (entraînement) et 12.5% (test) qui sera utilisée. Un autre problème spécifique à nos données peu équilibrée apparait cependant lors de l'échantillonnage : Les événements étant rares, la proportion de clients ayant eu un sinistre par rapport à la proportion de clients n'ayant déclaré aucun sinistre risque de ne pas être respectée.

Afin de contourner ce problème, une variable temporaire prenant pour valeur 1 si la fréquence des sinistres est supérieure à 0 ou 1 si la dite fréquence est égale à 0. Cette variable est donc très similaire à la fréquence des sinistres avec l'exception qu'elle ne tient pas compte du nombre de sinistres. L'échantillonnage est ensuite réalisé en suivant un échantillonnage stratifié sur cette variable (un échantillonnage respectant les proportions de cette variable, si 1% des clients déclarent un sinistres dans la base de donnée, alors 1% des clients de la base test exactement auront déclaré un accident). De cette manière, nous sommes assurés que les deux parties de la base de donnée contiennent la même proportion de clients ayant déclaré un accident.

4.2 Normalisation de la déviance

Pour la bonne compréhension de la suite de cette analyse, un point doit encore être expliqué. Le calcul de la déviance est réalisé comme la somme de la déviance de toutes les observations prédites. Cependant la division en jeu d'entraînent et de test qui ont un nombres d'observations différentes rend la comparaison de ces déviances inadaptées, et le jeu d'entraînement aura systématiquement une déviance totale plus élevée que le jeu de test. Pour palier à ce problème, deux solutions sont possibles.

- Travailler avec la déviance moyenne : $\bar{D} = \frac{1}{n} \sum_1^N D(y_n, \hat{y}_n)$
- Normaliser la déviance par rapport au nombre total de clients : $D = (D_{train}/N_{train}) * (N_{train} + N_{test})$ ou $D = (D_{test}/N_{test}) * (N_{train} + N_{test})$.

Normaliser la déviance d'une de ces manières rends la comparaison entre les déviances calculées sur le jeu d'entraînement et les déviances calculées sur le jeu de test possible.

4.3 GLM

4.3.1 Premier GLM

Afin de disposer d'un point de comparaison quant aux performances des algorithmes suivants, un modèle linéaire généralisé va être construit. Dans un premier temps, le modèle sera construit sur toutes les variables explicatives. La durée de contrat est également passée dans le modèle en tant que paramètre connu. Concernant la distribution ED utilisée, il s'agit de la distribution de poisson, avec le logarithme néperien comme fonction de lien. Il est important de préciser que le modèle est entraîné sur le jeu d'entraînement uniquement. Les résultats suivants sont obtenus :

Déviance moy.	Déviance tot. [1]
0.10680	6889.15

TABLE 4.3.1 – Résultat du premier modèle GLM.

4.3.2 Stepwise

Un modèle avec moins de paramètres généralise mieux les données. C'est pour cette raison qu'après la création du modèle GLM, celui-ci doit encore être modifié car il est nécessaire de supprimer les variables trop peu utiles dans les prédictions de celui-ci. Pour ce faire, la méthode "stepwise" sera utilisée. Á chaque itération, la variable dont la suppression améliore le plus le score (déviance) du modèle est retirée du prédicteur linéaire. Sans rentrer dans les détails, cela permet généralement d'améliorer le score du modèle sur le jeu de test. Les variables sélectionnées pour le modèle final sont :

Gender.F	Zone.1	Zone.2	Zone.3	Class.3
Class.4	Class.6	BonusClass.4	OwnersAge	VehiculeAge

TABLE 4.3.2 – Variables sélectionnées pour le GLM final.

Les résultats suivants sont obtenus :

Déviance moy.	Déviance tot. [2]
0.09904	6388.51

TABLE 4.3.3 – Résultat du GLM final.

Comme prévu, les performances sont meilleures que celles du premier modèle.

1. Les déviances sont calculées sur le jeu de test séparé.
2. Les déviances sont calculées sur le jeu de test séparé.

4.4 Réseaux de neurones

Les réseaux de neurones représentent un des aspects principaux de ce travail puisque ce sont eux qui serviront d'algorithme de base à AdaBoost.R2. Afin d'avoir différents points de comparaison pour la performance des méthodes de boosting sur les données utilisées, deux modèles consistant de réseaux de neurones seront entraînés. Un réseau de neurone "simple", constitué d'une seule couche cachée, et un réseau dit d'apprentissage profond, qui est constitué de plusieurs couches cachées.

Les modèles sont entraînés sur le jeu d'entraînement et les valeurs de la déviance par validation croisée seront données ainsi que les valeurs de la déviance du jeu de test séparé, normalisées sur le nombre total d'observations de la base de données.

Afin de limiter un peu plus le temps d'entraînement, la majorité des calculs seront exécutés sur un processeur graphique [3].

Un dernier point à mentionner est le fait que la sélection des variables (comme celle réalisée pour le GLM) n'est pas nécessaire en travaillant avec les réseaux de neurones, car les poids associés aux variables peuvent tendre vers 0 si la variable qui leur est associée n'est pas utile dans la prédiction.

4.4.1 Domaine

Un modèle est utilisé pour réaliser une prédiction $\hat{y}$ sur base du vecteur contenant les variables exploratoires X. Ceci peut être écrit sous la forme :

$$f(X) \to y$$

où $f(.)$ représente le modèle utilisé.

Dépendant du cas étudié, le domaine de cette fonction $f(.)$ peut différer. En effet, si dans la plupart des cas il est intéressant d'avoir une fonction avec $\mathbb{R}$ comme domaine, il existe des exceptions. La variable prédite de ce mémoire est la fréquence des sinistres. Puisqu'il s'agit d'une fréquence, il est impossible que la valeur prédite soit inférieure à 0. En addition à cela, le calcul de la déviance implique de calculer $ln(\hat{y})$, ce qui n'est pas un nombre réél si $\hat{y}$ est négatif résultant en l'impossibilité pour le modèle d'optimiser les paramètres.

Il est donc nécessaire de limiter le domaine de la fonction de réponse à $\mathbb{R}^+$. Dans certains ouvrages, ceci est réalisé en cherchant à prédire non pas $\hat{y}$ mais $exp(\hat{y})$. Le domaine de la fonction est donc modifié de $\mathbb{R}$ à $\mathbb{R}^+$ (Denuit, Hainaut et Trufin 2019b). Les prédictions de la fréquence des sinistres sont ensuite récupérées simplement grâce à la formule suivante :

$$\hat{y}_{pred} = ln(\hat{y}_{model})$$

Dans ce travail, une approche différente mais qui devrait produire des résultats similaire sera utilisée. L'approche précédente utilisait une fonction de lien similaire à celle du

3. GPU : Nvidia 1060 6Go

GLM afin de restreindre le modèle, mais celle utilisée dans ce travail se base sur la fonction d'activation de la couche de sortie. En modifiant cette fonction d'activation par une fonction ayant pour domaine $\mathbb{R}^+$, il est certain que la sortie du réseau suivra le domaine $\mathbb{R}^+$. L'avantage est que les valeurs $\hat{y}$ prédites sont directement les valeurs de la fréquence des sinistres. La fonction utilisée pour ce faire sera la fonction d'activation exponentielle, ce qui aura également comme résultat de produire des résultats très proches de la première méthode.

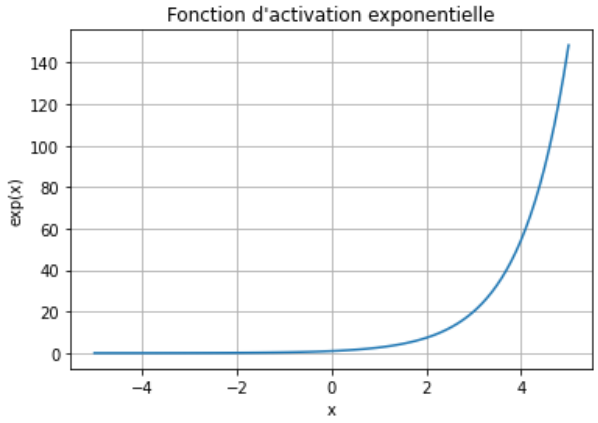


FIGURE 4.4.1 – Fonction d'activation exponentielle.

4.4.2 Hyperparamètres

Les hyperparamètres ont été cités plus tôt dans ce travail, mais pour le reste de cette section il est important de bien comprendre ce qu'ils sont et comment les choisir. Lors de l'entraînement d'un modèle, les paramètres de celui-ci sont optimisés par rapport aux données qu'il a reçu, et ce de manière "automatique". Cependant, dépendant d'un problème à un autre, cette optimisation doit changer dans le but de maximiser l'efficacité du modèle. Pour ce faire, il suffit de régler les hyperparamètres du modèle pour que ceux-ci soient adéquats pour le cas étudié.

Il y a cependant une grande différence au niveau de l'entraînement entre les paramètres optimisés par le modèle et les hyperparamètres : il n'existe pas de règle prédéfinie pour l'optimisation des seconds. En ajoutant à cela le fait qu'il est généralement difficile voir impossible de "deviner" les meilleurs hyperparamètres, le besoin d'une méthode heuristique pour le choix des hyperparamètres est bien présent.

Pour ce faire, un domaine de "recherche" sera défini pour chaque hyperparamètre. Une fois les domaines définis, N_{hyp} ensembles de paramètres seront échantillonnés depuis ces domaines (un paramètre dans chaque domaine, N_{hyp} fois). Un modèle sera ensuite entrainé avec chacun de ces ensembles de paramètres et leurs résultats seront comparés.

Dans un monde parfait, toutes les combinaisons possibles des hyperparamètres dans le domaine défini devraient être testées, ceci est cependant impossible à cause du temps de calcul nécessaire à l'entraînement du modèle, ce qui rends la tâche difficile (et encore

plus lorsque la validation croisée est utilisée, puisque celle-ci implique la création de K [4] modèles pour chaque combinaison d'hyperparamètres). C'est cette différence qui fait que cette méthode peut être qualifiée d'heuristique.

Les hyperparamètres qui seront optimisés pour les réseaux de neurones de ce travail sont décris ci-dessous :

- Nombre d'époques (Epochs) : C'est le nombre de fois que la base de donnée (d'entraînement) complète sera passée dans le réseau de neurones au cours de l'optimisation des poids de celui-ci.
- Taille des lots (Batch size) : Un lot est un ensemble de $N_{BatchSize}$ entrées, qui sera passé dans le réseau de neurone en une seule fois lors de l'optimisation. La taille du lot défini combien de lots au total seront passés dans le réseau de neurones au cours d'une époque. Par exemple, si la base de données est composée de 10.000 entrées, et que la taille des lots est de 500, alors 20 lots seront passés à travers le réseau de neurones à chaque époque. En pratique, cela influence l'optimisation des poids car un gradient sera calculé par lot, et les poids seront modifiés à chaque calcul de gradient.
- Pourcentage d'abandon (Dropout rate) : Il s'agit d'un paramètre de régularisation permettant de limiter les risques de surentraînement du modèle. Il représente un pourcentage des neurones qui sera ignoré pendant chaque époque. Cela signifie que leur contribution au réseau est temporairement ignorée.
- Nombre de neurones : Il s'agit simplement du nombre de neurones dans une couche cachée. S'il existe plusieurs couches cachées, alors un paramètre sera défini pour chacune d'entre elle.
- Taux d'apprentissage (Learning Rate) : Ce paramètre détermine à quel point les poids du réseau seront modifiés à chaque fois que les poids seront mis à jour. Il doit être compris entre 0 et 1.

4.4.3 Courbes d'apprentissage

Le surentraînement d'un modèle peut être vraiment problématique pour les prédictions de celui-ci, comme expliqué plus tôt dans ce travail. Dans le but de réduire celui-ci, la validation croisée est un outil efficace. Si toutes les itérations de la validation croisée ont une déviance similaire, ceci est une bonne indication que le modèle a été correctement entraîné. Cependant d'autres techniques permettent de s'assurer qu'un modèle n'a pas été sous-entrainé ou surentraîné. Une méthode populaire consiste à dessiner les courbes d'apprentissage d'un modèle, afin de s'assurer que les résultats sur la base de données d'entraînement et sur celle de test sont cohérents, et reflètent que le modèle est correctement paramétré.

Un graphique de courbe d'apprentissage est composé de deux axes :

4. Nombre d'ensembles créés pour la validation croisée.

- Sur l'axe des abscisses, le paramètre pour lequel il est intéressant de vérifier qu'il est correctement ajusté. Le paramètre peut être un hyperparamètre, ou la taille de la base de donnée utilisée pour l'entraînement.
- Sur l'axe des ordonnées, la déviance moyenne.

Des courbes représentant la déviance moyenne pour différentes valeurs du paramètre seront dessinées, une pour la déviance sur le jeu d'entraînement, et une autre sur le jeu de test. Sur base de ces courbes, il est possible de détecter un sous-entraînement ou un surentraînement (si l'erreur de test est inférieure à l'erreur d'entraînement, ou si l'erreur de test est trop supérieure à celle d'entraînement, respectivement). Ce type de graphique peut aussi être utilisé pour explorer le domaine de certains hyperparamètres et choisir la meilleure valeur possible pour l'un d'entre eux.

4.4.4 Réseau de neurones simple

Dans cette sous-section, la méthodologie utilisée ainsi que les résultats obtenus pour le réseau de neurones contenant une seule couche cachée seront présentés.

4.4.4.1 Domaine des hyperparamètres et résultats

Afin de choisir un des meilleurs modèles possibles, la validation croisée se fera sur le jeu d'entraînement, et les résultats finaux de la déviance seront calculés sur le jeu de test. La liste des domaines explorés pour les hyperparamètres est indiquée ci-dessous. Il est tout de même important de préciser que pour arriver à ces domaines relativement restreints, une première recherche avec des domaines plus vagues a été menée. Sur base des résultats de cette première recherche les critères ont pu être affinés.

Paramètre	Domaine
Époque	$[100, 250, 400]$
Taille du lot	$[500, 1000, 10000, 51600]$ [5]
% abandon	$[0.1, 0.2, 0.3]$
N. Neurones	$[5, 10, 15, 25]$
Taux d'apprentissage	$[0.01, 0.05, 0.1, 0.3]$

TABLE 4.4.1 – Liste des domaines des hyperparamètres recherchés pour le réseau de neurones simple.

L'entraînement d'un réseau de neurone par validation croisée prenant un temps considérable, et ce malgré l'utilisation d'un processeur graphique pour l'apprentissage. Pour cette raison, l'approche présentée plus tôt sera utilisée : 40 ensembles de paramètres seront échantillonnés depuis les domaines ci-dessus, et un modèle sera entraîné par validation croisée pour chacun d'entre eux.

5. 51600 représente la taille de la base de donnée d'entraînement, et donc un seul lot par époque.

Les résultats de la recherche sont indiqués dans le tableau ci-dessous :

Rang [6]	Époque	Lot	% abandon	N neurones	T. Apprentissage	Déviance moy.	Déviance Tot.	Std. déviance [7]
1	250	500	0.1	25	0.1	0.0908163	5857.74	0.00573
2	250	500	0.1	15	0.1	0.0908278	5858.48	0.00604
3	400	500	0.1	10	0.1	0.0912018	5882.60	0.00631

TABLE 4.4.2 – Résultats de l'analyse sur les réseau de neurones simples. Note : Les déviances de ce tableau sont calculées comme la moyenne de la deviance des jeux de test de la validation croisée. La déviance totale est égale à la déviance moyenne normalisée pour les 64501 observations.

Les performances des modèles sur le jeu de test séparé sont également présentées :

Rang	Déviance moy.	Déviance Tot.
1	0.091650	5911.54
2	0.091539	5904.41
3	0.091939	5930.13

TABLE 4.4.3 – Déviances calculées sur le jeu de test séparé.

Le choix entre les deux premiers modèles est assez difficile au vu de leurs performances très similaires. Le principe de parcimonie [8] indique de choisir le deuxième modèle, cependant le premier modèle dispose aussi d'un écart type inférieur pour la déviance sur les différents tests, ce qui indique qu'il peut potentiellement être plus général que le deuxième modèle. Les deux choix peuvent donc se défendre, mais pour le reste de cette analyse c'est le deuxième modèle qui sera utilisé puisque celui-ci dispose de moins de paramètres (330 poids contre 550) et obtient de meilleures performances sur le jeu de test séparé.

Les courbes d'apprentissage du modèle vont maintenant être présentées. Note : Pour la création des courbes d'apprentissage, les paramètres du modèle sont tous identiques aux paramètres du modèle choisi dans le tableau précédent excepté le paramètre recherché.

6. Le rang est indiqué par rapport à l'erreur de test de la validation croisée.
7. Écart-type des scores de test de la validation croisée.
8. Pour deux modèles aux résultats similaires, il est préférable de choisir celui avec le moins de paramètres, qui permet une meilleure généralisation.

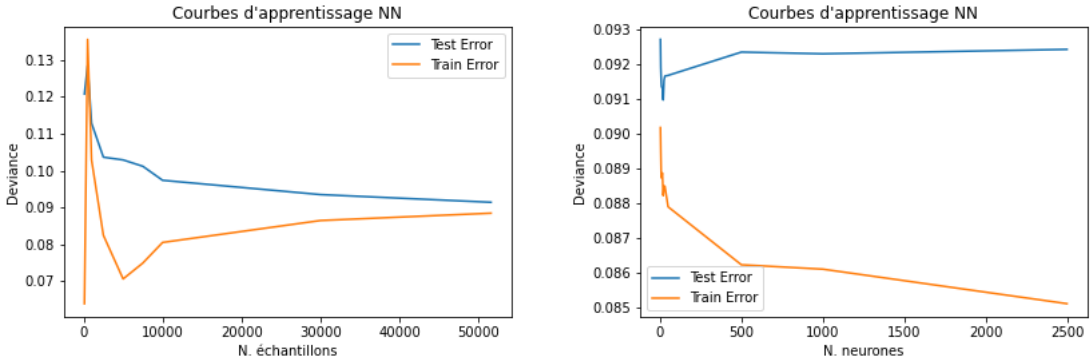


FIGURE 4.4.2 – Courbes d'apprentissage pour le nombre d'échantillons et le nombre de neurones.

- Nombre d'échantillons : Si le nombre d'échantillons est trop petit, alors l'erreur de test est trop élevée. Au fur et à mesure que le nombre d'échantillons augmentent, les courbes se rapprochent, indiquant une bonne convergence du modèle. La valeur très faible prise par l'erreur d'entraînement pour 100 échantillons peut être expliquée par le fait que l'échantillon étant trop petit, il y a probablement eu très peu d'observations ayant déclaré un sinistre. L'erreur de test élevée pour cette valeur renforce cette hypothèse. Note : L'échantillonnage est réalisé sans remise.
- Nombre de neurones : Ce graphique montre l'importance de ne pas créer une couche contenant trop de neurones. Passé un certain cap, au fur et à mesure que des neurones sont ajoutés, l'erreur de test augmente alors que l'erreur d'entraînement diminue, signe indégnable de surentraînement. Cependant il est intéressant de noter que le modèle avec 30 neurones semblent obtenir de meilleures performances que le modèle avec 25 neurones.

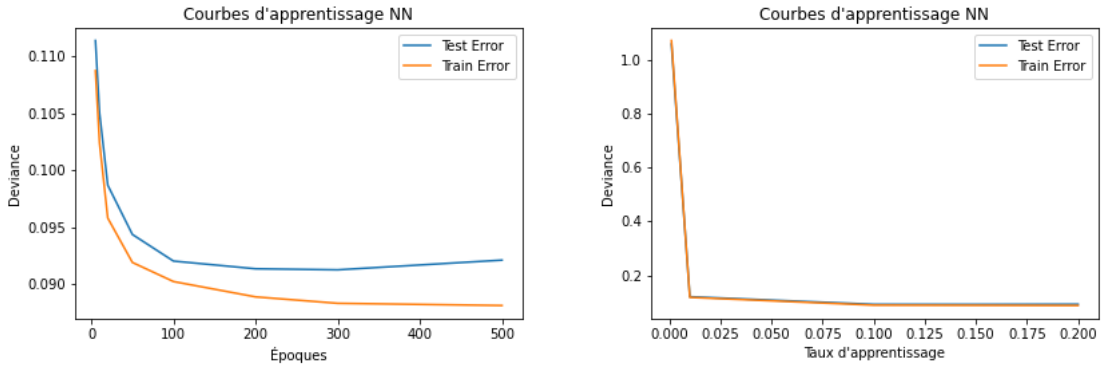


FIGURE 4.4.3 – Courbes d'apprentissage pour le nombre d'époques et le taux d'apprentissage.

- Époques : La courbe d'apprentissage indique que passé les 100 époques, l'erreur d'entraînement diminue alors que l'erreur de test semble stable. Ceci peut indi-

quer un début de surentraînement mais la différence entre les deux courbes reste raisonnable jusque la barre des 300 époques.
- Taux d'apprentissage : La courbe semble indiquer que 0.1 est une bonne valeur pour le paramètre. Au delà, l'erreur de test commence à augmenter (légèrement).

Les conclusions quant au nombre d'époques et au nombre de neurones semblent indiquer que les paramètres peuvent potentiellement bénéficier d'un dernier affinage. Un nouveau modèle est donc entraîné, avec 300 époques et 30 neurones. Celui-ci n'améliorant cependant pas les résultats (déviance et variance plus élevées), le deuxième modèle restera le modèle choisi.

4.4.5 Réseau de neurones profond

Comme évoqué précédemment, un réseau de neurones profond est simplement un réseau disposant de plusieurs couches cachées. Ce type de réseau est capable de résoudre des problèmes extrêmement complexes mais est aussi très sensible au surentraînement. En suivant la même méthodologie que pour le réseau de neurones simple, un réseau de neurones profond sera construit dans cette sous-section. Pour la création de celui-ci, le nombre de couches cachées sera limitée à deux, ce qui est suffisant pour approximer n'importe quelle fonction.

4.4.5.1 Domaine des hyperparamètres et résultats

Le domaine de recherche des hyperparamètres est indiqué ci-dessous :

Paramètre	Domaine
Époque	$[150, 250, 350, 450]$
Taille du lot	$[500, 10000, 51600]$
% abandon	$[0.1, 0.2]$
N. neurones (1) [9]	$[10, 15, 20, 40]$
N. neurones (2) [10]	$[15, 20, 25, 40]$
Taux d'apprentissage	$[0.05, 0.1, 0.2, 0.3]$

TABLE 4.4.4 – Liste des domaines des hyperparamètres recherchés pour le réseau de neurones profond.

40 modèles ont étés entraînés par validation croisée :

9. Première couche.
10. Deuxième couche.

Rang	Époques	Lot	% abandon	N. neurones (1)	N. neurones (2)	T. app.	Dév. moy.	Dév. tot.	Std. dév
1	250	10.000	0.1	40	20	0.3	0.09341	6025.04	0.00903
2	250	10.000	0.1	20	15	0.3	0.09360	6037.29	0.00594
3	150	500	0.2	10	25	0.2	0.09410	6069.54	0.01055

TABLE 4.4.5 – Résultats de l'analyse sur les réseau de neurones simples. Note : Les déviances de ce tableau sont calculées comme la moyenne de la deviance des jeux de test de la validation croisée. La déviance totale est égale à la déviance moyenne normalisée pour les 64501 observations.

Les résultats sur le jeu de test séparés sont également présentés :

Rang	Déviance moy.	Déviance tot.
1	0.09866	6004.8
2	0.09309	5977.03
3	0.09369	6042.83

TABLE 4.4.6 – Déviances calculées sur le jeu de test séparé.

Les performances du deuxième modèle sont biens meilleures sur le jeu de test séparé, et celui-ci dispose également de bien moins de paramètres que le premier modèle (735 poids contre 1655). Ceci indique que le deuxième modèle est sans aucun doute le meilleur choix ici.

Les courbes d'apprentissage vont maintenant être présentées :

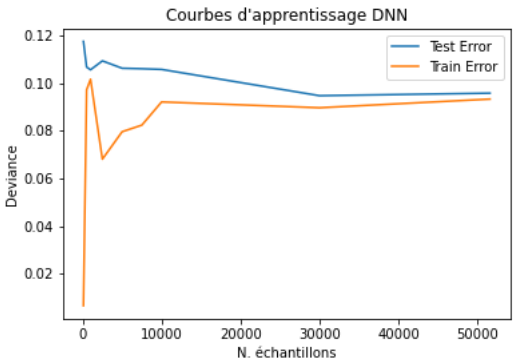


FIGURE 4.4.4 – Courbes d'apprentissage pour le nombre d'échantillons.

- Nombre d'échantillons : Les mêmes conclusions que pour le réseau de neurones simple peuvent être tirées. Les erreurs de test et d'entraînement convergent pour se rejoindre au fur et à mesure que le nombre d'échantillons utilisés sont atteints.

Note : L'échantillonnage est réalisé sans remise.

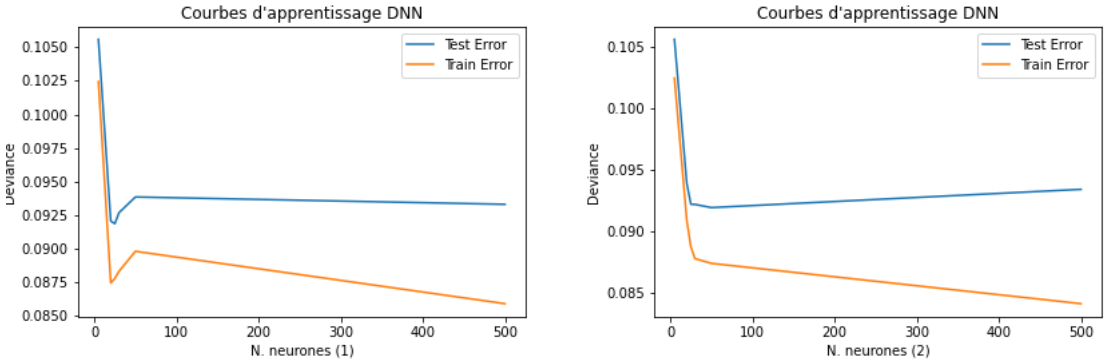


FIGURE 4.4.5 – Courbes d'apprentissage pour le nombre de neurones de chaque couche cachée.

- N. neurones (1 et 2) : Encore une fois, des conclusions très similaires au réseau de neurones simple peuvent être déduites. En effet, ajouter des neurones ne sert en rien le modèle et aura juste pour conséquence de surentraîner celui-ci à cause de tous les nouveaux paramètres (poids) créés.

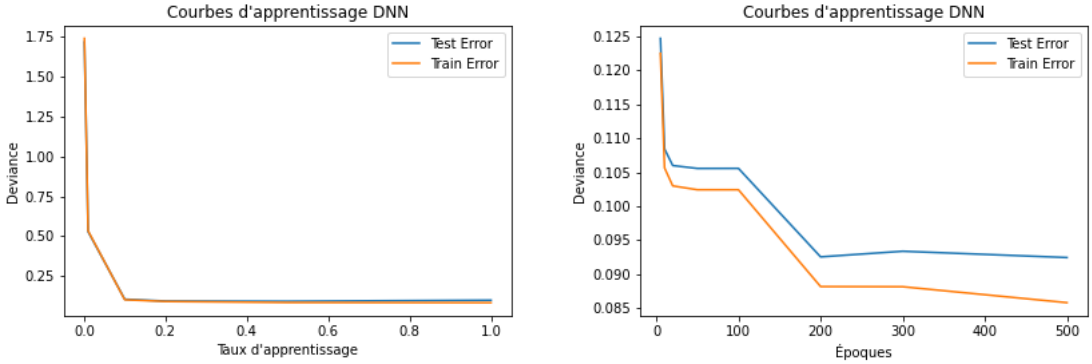


FIGURE 4.4.6 – Courbes d'apprentissage pour le taux d'apprentissage et le nombre d'époques.

- Taux d'apprentissage : La courbe montre que le modèle est dans la bonne fourchette de taux d'apprentissage. Entre 0.1 et 0.3, le modèle dispose de bonnes performances, qui commencent à se dégrader au fur et à mesure que le taux d'apprentissage augmente.
- Époques : Ici aussi, la graphique semble indiquer que le nombre d'époques paramétré est raisonnable et n'entraîne pas de surentraînement.

4.5 AdaBoost.R2

Dans cette sous-section seront présentés les résultats des modèles réalisés avec AdaBoost.R2. Comme expliqué précédemment AdaBoost est un méta-algorithme qui utilise un autre algorithme pour créer les modèles. Beaucoup de types de modèles différents peuvent être utilisés à cette fin cependant l'objet de ce travail est d'observer les résultats de cet algorithme combiné avec les réseaux de neurones, c'est donc un perceptron multicouche [11] qui sera utilisé.

Si diminuer le temps de calcul des réseaux de neurones était important dans la sous-section précédente, avec AdaBoost.R2 cela devient une nécessité. Construire un prédicteur avec AdaBoost.R2 requiert non-seulement d'échantillonner N sous-ensembles du jeu de données pour l'entraînement, mais aussi la création de N réseaux de neurones. De plus, si leur entraînement prends un certains temps, c'est loin d'être le seul problème. En effet afin de réaliser une prédiction avec AdaBoost.R2, il est nécessaire de stocker tous les apprenants faibles entrainés pour créer l'hypothèse forte du méta-algorithme. La création de 40 instances d'AdaBoost.R2 peut ainsi nécessiter l'entraînement et le stockage de milliers de perceptrons multicouches.

C'est afin de palier à ce problème que la validation croisée ne sera pas utilisée ici (l'éxecution du code de la recherche des hyperparamètres pouvant déjà prendre plus d'une journée).

4.5.1 Taux d'apprentissage d'AdaBoost.R2

Si la version d'AdaBoost.R2 originale ne dispose d'aucun hyperparamètre (hormis la fonction de perte utilisée), en pratique, il peut être intéressant d'avoir un certain contrôle sur le nombre d'estimateurs créés. Dans la version originale, l'algorithme se termine lorsque l'erreur moyenne dépasse 0.5 (souvent un nombre maximal d'apprenants faibles est tout de même spécifié).

Dans ce paragraphe sera introduit le taux d'apprentissage d'AdaBoost.R2, qui est souvent rajouté en pratique dans les implémentations du méta-algorithme.

Le but de ce paramètre est de pouvoir influer sur le nombre de modèles créés par AdaBoost.R2. Le taux d'apprentissage d'AdaBoost est noté η. Pour ce faire, deux étapes vont être modifiées :

- L'importance de l'apprenant faible, notée β, est normalement calculée comme ceci : $\beta = \frac{\hat{L}}{1-\hat{L}}$. La modification du taux d'apprentissage implique que l'importance sera maintenant calculée comme : $\alpha = \eta ln(\frac{1}{\beta})$.
- La modification des poids se fera avec la formule : $w_{i+1} = w_i \beta^{\eta[1-L_i]}$.

Ces modifications ont pour effet de diminuer l'importance des modèles et la vitesse à laquelle les poids sont modifiés (pour $\eta < 1$), et d'apporter un certain contrôle sur le

11. Une seule couche cachée.

phénomène. De ce fait, l'erreur moyenne aura tendance à augmenter moins vite, et plus de modèles seront créés avant d'arriver à une erreur moyenne de 0.5.

Pour comprendre comment ce nouveau paramètre influe sur l'apprentissage, des illustrations sont présentées ci-dessous. Les modèles illustrés ici ont été construits pour la démonstration et ne représentent pas les courbes des algorithmes entraînés plus loin dans cette section. Les paramètres utilisés sont cependant très similaires à ceux qui seront construits plus tard, mis à part le taux d'apprentissage qui est étudié ici.

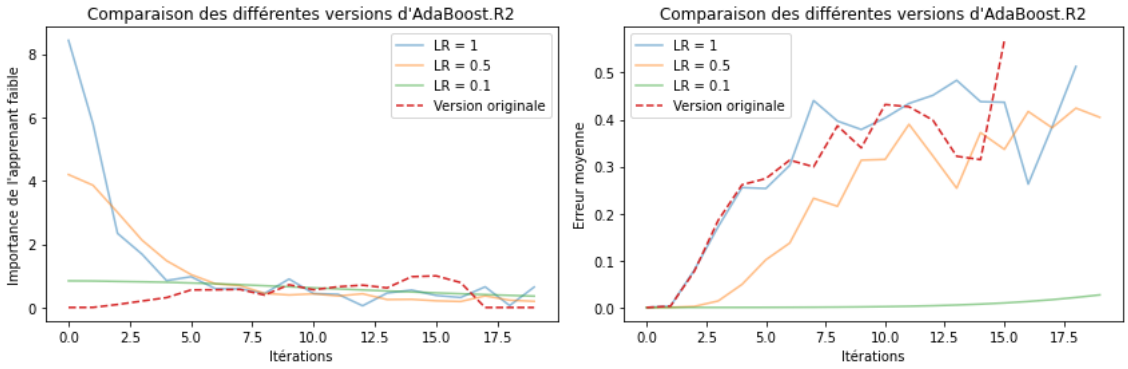


FIGURE 4.5.1 – Observations des paramètres pour différentes valeurs de taux d'apprentissage pour les 20 premières itérations.

- Le premier graphique montre l'évolution de l'importance des apprenants faibles au fil des itérations. L'influence du taux d'apprentissage est clairement visible ici, plus celui-ci se rapproche de 0 et plus l'importance sera faible dès le départ. La valeur de ce paramètre permet donc un grand contrôle sur l'entraînement des apprenants faibles. Les valeurs prises par la version originale semblent quant à elles commencer à des valeurs plus basses et monter progressivement. La courbe pour le taux d'apprentissage le plus faible semble bien plus lisse que les autres et moins sensibles aux changements brusques.
- Le deuxième graphique montre l'évolution de l'erreur moyenne sur les 20 première itérations. Puisque l'algorithme s'arrête lorsque cette erreur dépasse 0.5, il est normal que certaines des courbes s'arrêtent avant les 20 itérations (une fois qu'elle ont dépassé la valeur maximale permise). C'est notamment le cas pour la version originale d'AdaBoost.R2 mais également pour la version modifiée avec un taux d'apprentissage de 1. On s'apperçoit d'abord que ces deux courbes sont assez similaires, surtout pour les 5 premières itérations. Un taux d'apprentissage de 1 permet donc de se rapprocher des résultats qui seraient obtenus avec la version originale d'AdaBoost. Un autre fait intéressant peut être observé en regardant la courbe pour le taux d'apprentissage de 0.1. En effet, celle-ci évolue bien plus lentement que les autres. En fait, plus le taux d'apprentissage est bas et plus cette augmentation va être lente. Le taux d'apprentissage permet donc en effet un grand contrôle sur le nombre d'apprenants faibles qui seront créés.

4.5.2 Domaine des hyperparamètres et résultats

La méthodologie restera très similaire à celle des réseaux de neurones (hors validation croisée qui n'aura pas lieu ici). La plupart des hyperparamètres proviennent du réseau de neurones, à l'exception du taux d'apprentissage d'AdaBoost, présenté ci-dessus. La fonction de perte utilisée pour AdaBoost est la fonction exponentielle présentée dans la sous-section 3.4.4. Voici le premier domaine des hyperparamètres :

Paramètre	Domaine
N. estimateurs max.	$[10, 50, 100]$
Taux d'apprentissage (Adaboost)	$[0.1, 0.5, 1]$
Taux d'apprentissage (NN)	$[0.01, 0.1, 0.5]$
Époques	$[50, 100, 250]$
Taux d'abandon	$[0.1, 0.2]$
N. neurones	$[5, 10, 15]$
Taille du lot	$[500, 10000, 51600]$

TABLE 4.5.1 – Domaines des hyperparamètres pour AdaBoost.R2 (première itération).

10 instances d'AdaBoost.R2 ont été entrainées pour cette première itération, pour un total d'environ 500 réseaux de neurones créés. Les résultats des meilleurs modèles sont présentés dans le tableau suivant :

Rang	N. max. est.	N. est.	T. app. ADB	T. app. NN	T. aband.
1	10	10	0.1	0.5	0.1
2	10	10	0.1	0.1	0.2
3	10	10	1	0.1	0.1

Rang	Taille du lot	Époques	N. neurones	Déviance moy.	Déviance Tot. [12]
1	10000	100	10	0.0913674	5893.29
2	500	100	10	0.0926793	5977.91
3	500	250	10	0.0944176	6090.03

TABLE 4.5.2 – Résultat de la première recherche des hyperparamètres.

Ces résultats sont plutôt encourageants mais un autre constat intéressant peut être observé. En effet, le modèle le plus efficace est un modèle ayant un taux d'apprentissage (NN) plus élevé, et une taille de lot relativement grande également. Si ce résultat est si intéressant, c'est parce que le transit des données vers le GPU se fait à chaque époque, et ceci a pour effet de ralentir l'éxécution. Non pas parce que les calculs sont plus compliqués mais bien parce que les données mettent plus de temps à être acheminées vers le processeur graphique qu'à être traitées par celui-ci. Dernier constat : il semble que l'association d'un

12. Les déviances sont calculées sur le jeu de test séparé.

taux d'apprentissage (NN) élevé associé à un taux d'apprentissage (AdaBoost) bas permet à l'algorithme d'obtenir d'excellentes performances sur le cas étudié. Le fait que le modèle le plus performant de cette recherche dispose de ces particularité motive l'éxecution d'une deuxième recherche, avec des domaines différents (basés sur les paramètres du meilleur modèle), cette recherche devrait être bien plus efficiente que la première.

Les nouveaux domaines sont présentés ci-dessous :

Paramètre	Domaine
N. estimateurs max.	$[10, 15, 20]$
Taux d'apprentissage (Adaboost)	$[0.1, 0.25, 0.5]$
Taux d'apprentissage (NN)	$[0.3, 0.5, 0.7]$
Époques	$[100, 200, 300]$
Taux d'abandon	$[0.1]$
N. neurones	$[8, 10, 12]$
Taille du lot	$[10000, 25000]$

TABLE 4.5.3 – Domaines des hyperparamètres pour AdaBoost.R2 (deuxième itération).

Relancer une recherche avec ce nouveau domaine semble très concluant, au vu des résultats présentés dans le tableau suivant (8 itérations d'AdaBoost sur les 20 entraînées produisent de meilleurs résultats que le meilleur modèle de la recherche précédente) :

Rang	N. max. est.	N. est.	T. app. ADB	T. app. NN	T. aband.
1	10	10	0.5	0.5	0.1
2	15	15	0.1	0.5	0.1
3	15	15	0.1	0.7	0.1

Rang	Taille du lot	Époques	N. neurones	Déviance moy.	Déviance Tot. [13]
1	10.000	300	12	0.0904123	5831.7
2	10.000	100	10	0.0908329	5858.81
3	25.000	200	8	0.0908461	5859.66

TABLE 4.5.4 – Résultat de la deuxième recherche des hyperparamètres (20 itérations ont été entraînées).

Le premier modèle de cette itération est sans aucun doute le meilleur choix, il dispose de performances grandement supérieures aux deux autres, et il s'agit d'ailleurs du meilleur modèle construit pour l'instant dans le cadre de ce travail. Au vu de ces résultats, AdaBoost.R2 semble être très performant associé aux réseaux de neurones dans le cadre de l'estimation de la fréquence des sinistres.

De la même manière que pour les modèles précédents, les courbes d'apprentissages ont été construites :

13. Les déviances sont calculées sur le jeu de test séparé.

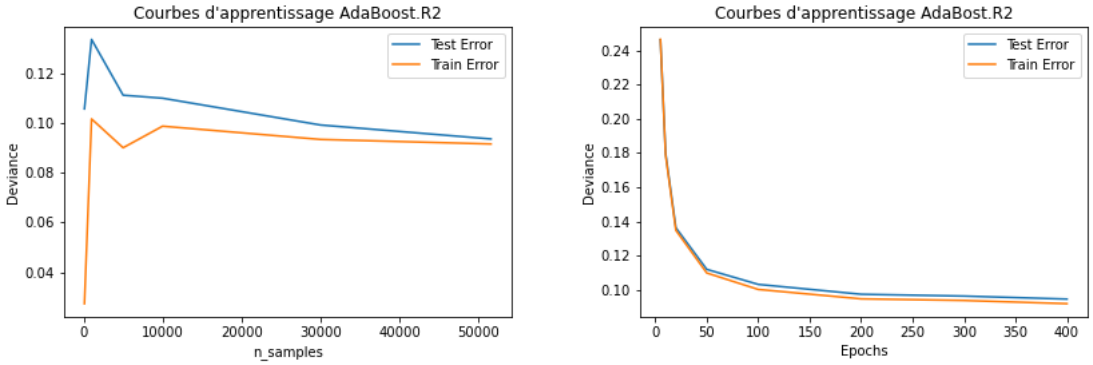


FIGURE 4.5.2 – Courbes d'apprentissage pour le taux d'apprentissage et le nombre d'époques.

- Nombre d'échantillons : Une bonne convergence est observée, si l'échantillon est trop petit, le modèle aura de mauvaises performances. Vers la fin du graphique, la courbe d'erreur su jeu de test est très proche de l'erreur d'entraînement, ce qui indique qu'assez d'échantillons sont disponibles.
- Époques : Encore une fois, le nombre d'époque choisi parait raisonnable, il est cependant intéressant de noter qu'une augmentation du nombre d'époques pourrait induire une faible amélioration du modèle.

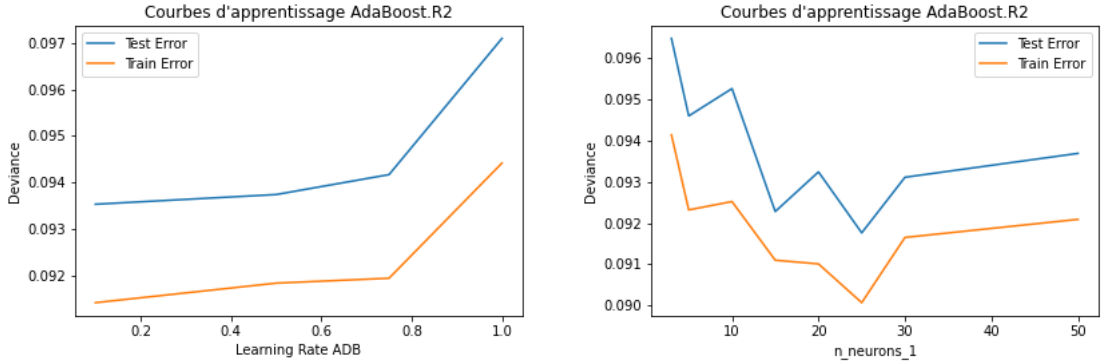


FIGURE 4.5.3 – Courbes d'apprentissage pour le taux d'apprentissage et le nombre d'époques.

- Taux d'apprentissage (AdaBoost.R2) : le graphique semble confirmer le fait que le taux d'apprentissage rajouté dans l'algorithme AdaBoost.R2 améliore considérablement les performances. Le fait de pouvoir contrôler l'importance et le nombre de modèles créés est un avantage réel.
- N. neurones : Le graphique semble indiquer qu'un nombre de neurones entre 15 et 30 pourrait peut-être donner de meilleurs résultats. Passé ce cap l'addition de neurones va progressivement dégrader les performances du modèle.

Au vu des courbes d'apprentissage du taux d'apprentissage, du nombre de neurones et du nombre d'époques, il parait intéressant de relancer une recherche avec un domaine légèrement différent. Les résultats sont présentés ci-dessous :

Rang	N. max. est.	N. est.	T. app. ADB	T. app. NN	T. aband.
1	12	12	0.2	0.3	0.1
2	8	8	0.3	0.5	0.1
3	12	12	0.2	0.7	0.1

Rang	Taille du lot	Époques	N. neurones	Déviance moy.	Déviance Tot. [14]
1	10.000	300	25	0.09031	5825.43
2	10.000	350	12	0.09067	5845.72
3	10.000	350	15	0.09068	5849.13

TABLE 4.5.5 – Résultat de la troisième recherche des hyperparamètres (20 itérations ont été entraînées).

Si le premier modèle obtient des résultats légèrement meilleurs que le meilleur modèle de l'itération précédente, la différence est trop petite pour justifier une telle augmentation dans le nombre de paramètres. En effet le nouveau modèle de base est constitué du double de neurones.

Le prédicteur choisi reste donc pour l'instant reste donc le meilleur modèle de l'itération précédente d'AdaBoost.R2.

4.5.3 Déviance et AdaBoost.R2

Dans la sous-section 3.4.4, AdaBoost.R2 est présenté. Celui-ci fait usage d'une fonction de perte qui doit avoir comme domaine $[0, 1]$. Hormis cette particularité, n'importe quelle métrique peut être utilisée pour l'entraînement du méta-algorithme.

Dans cette sous-section il sera tenté d'utiliser la déviance comme fonction de perte d'AdaBoost.R2. Afin de pouvoir utiliser celle-ci, il est nécessaire de modifier sa formule, afin que son domaine respecte bien les contraintes imposées. Si $max(D^*)$ représente la valeur maximum prise par la déviance de toutes les observations, alors il suffit de diviser chaque valeur par ce maximum afin de normaliser les résultats et qu'ils se situent tous dans le domaine $[0, 1]$ tout en conservant les proportions des résultats. La formule de la déviance devient donc :

$$D(y_i, \hat{y}_i) = \frac{2v_i(y_i ln(y_i) - y_i ln(\hat{y}_i) - y_i + \hat{y}_i)}{max(D^*)}, \quad \forall y_i > 0$$

$$D(y_i, \hat{y}_i) = \frac{2v_i\hat{y}_i}{max(D^*)}, \quad \forall y_i = 0$$

14. Les déviances sont calculées sur le jeu de test séparé.

Les résultats sont présentés dans le tableau suivant :

Rang	N. max. est.	N. est.	T. app. ADB	T. app. NN	T. aband.
1	10	10	0.2	0.3	0.1
2	12	12	0.2	0.7	0.1
3	12	12	0.2	0.7	0.1

Rang	Taille du lot	Époques	N. neurones	Déviance moy.	Déviance Tot. [15]
1	10.000	300	12	0.11361	7328.02
2	10.000	400	25	0.12139	7829.86
3	10.000	300	25	0.12233	7890.39

TABLE 4.5.6 – Résultats avec la déviance comme fonction de perte d'AdaBoost (20 itérations ont été entraînées).

Les résultats sont plutôt décevant et il semblerait que la déviance ne soit pas une bonne métrique pour calculer la perte d'AdaBoost. Dans la sous-section 3.4.5, une hypothèse avait été énoncée par des chercheurs sur la raison pour laquelle AdaBoost produisait parfois de mauvais résultats (SCHAPIRE, FREUND et al. 1998). Celle-ci évoquait qu'un nombre trop peu important de données mises à la disposition d'AdaBoost pourrait être en cause, de même qu'une erreur d'entraînement des apprenants faibles qui se rapproche trop vite de 0.5. Il n'est pas impossible que la deuxième raison soit en cause de ces résultats.

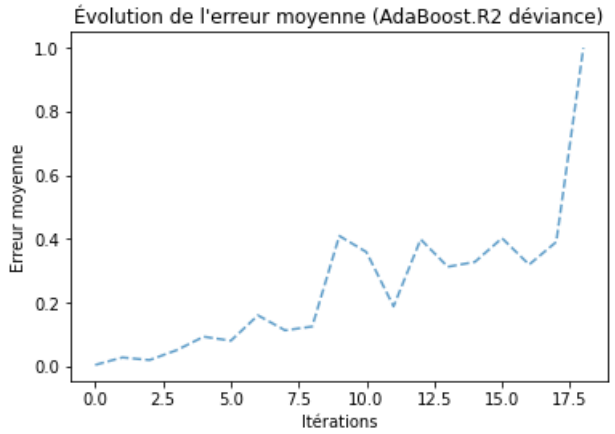


FIGURE 4.5.4 – Erreur moyenne d'un modèle entraîné avec la déviance comme fonction de perte d'AdaBoost.

15. Les déviances sont calculées sur le jeu de test.

4.6 Analyses complémentaires

Maintenant que tous les modèles ont été entraînés et choisis, leurs résultats vont être analysés de plus près. Dans cette partie seront présentés les graphiques résultant de l'analyse des prédicteurs.

4.6.1 Prédiction des fréquences de sinistres par classe

Afin de pouvoir tarifer correctement les clients, il est intéressant de regarder la fréquence moyenne des sinistres prédite par groupe de clients. En effet, s'il est difficile de prédire la fréquence exacte pour un client donné, la prédiction pour un groupe peut parfois être meilleure. Note : Les algorithmes sont entraînés sur le jeu d'entraînement et les estimations sont faites sur le jeu de test.

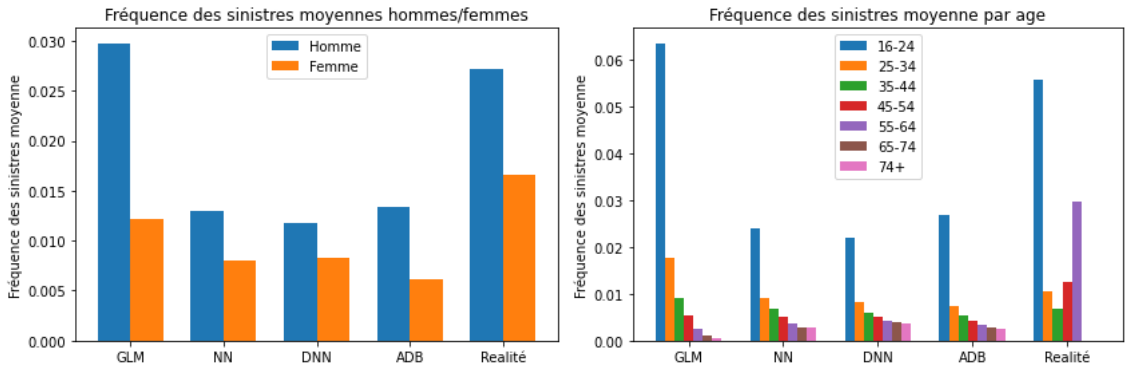


FIGURE 4.6.1 – Fréquence des sinistres par âge et par sexe.

Les graphiques ci-dessus nous apprennent des choses intéressantes :

- Sexe :
  - Le constat est clair, tout les modèles basés sur les réseaux de neurones, y compris AdaBoost.R2, sous-estime la fréquence moyenne. Le modèle GLM, bien que moins performant en terme de déviance, s'en rapproche beaucoup plus.
  - La différence entre homme et femme (en termes de proportion) également être mieux représentée par le GLM.

- Age :
  - Tous les modèles semblent opter pour une fonction dégressive en fonction de l'âge, qui est légèrement différente de la réalité (du moins pour le jeu de données utilisé ici).
  - Les modèles basés sur les RNA [16] semblent grandement sous-estimer la fréquence des sinistres moyenne pour les clients âgés de 16 à 24 ans.

16. Réseaux de neurones artificiels.

– Le GLM surrestime légèrement la moyenne du groupe 16-24 et sous-estime ceux des groupes 45-54 et 55-64.

Les résultats sont assez inattendus, le modèle GLM semble être le seul à être à même d'estimer assez bien les moyennes par groupes.

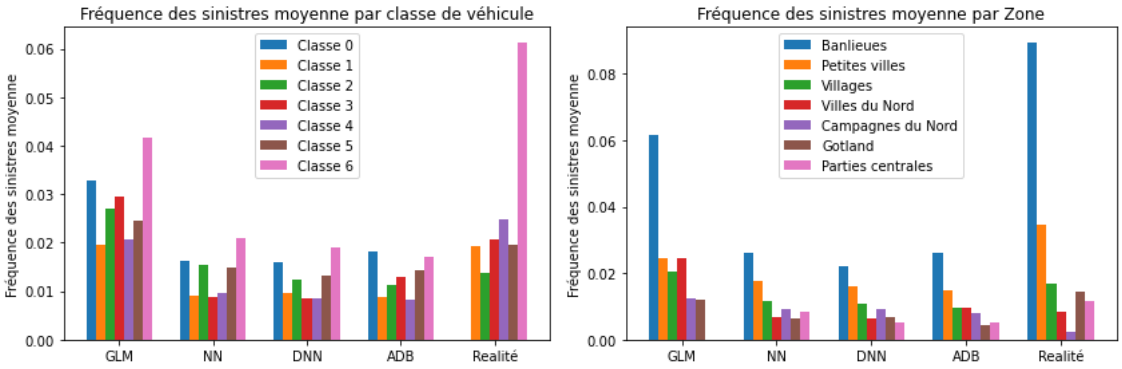


FIGURE 4.6.2 – Fréquence des sinistres moyenne par classe de véhicule et par région.

Les mêmes conclusions que pour les graphiques précédents peuvent être tirées. Dans l'annexe 4 se trouve un graphique similaire pour la fréquence moyenne prédite de tout le jeu d'entraînement.

4.6.2 Effet du taux d'apprentissage sur les clients types

Afin d'analyser la manière dont les modèles se comportent, il est possible de créer des profils fictifs de clients "typiques" et d'observer la manière dont les prédictions varient selon certains critères. Ceci est particulièrement intéressant pour les valeurs continues.

La fonction de décision utilisée dans AdaBoost.R2, pour rappel, s'écrit comme ceci :

$$h_f = inf\left\{y \in Y : \sum_{t:h_t \leq y} log\left(\frac{1}{\beta_t}\right) \geq \frac{1}{2}\sum_t log\left(\frac{1}{\beta_t}\right)\right\}$$

Comme décrit dans l'équation, si des dizaines voir des centaines de prédicteurs peuvent être entraînés, c'est au final la prédiction d'un seul des apprenants faibles qui sera choisie et renvoyée comme réponse finale. Cette propriété implique que la fonction finale peut être vue comme une combinaisons des différentes fonctions des apprenants faibles et qu'un contrôle sur le nombre d'estimateurs utilisés dans cette combinaison est possible par la ocnfiguration du taux d'apprentissage d'AdaBoost.

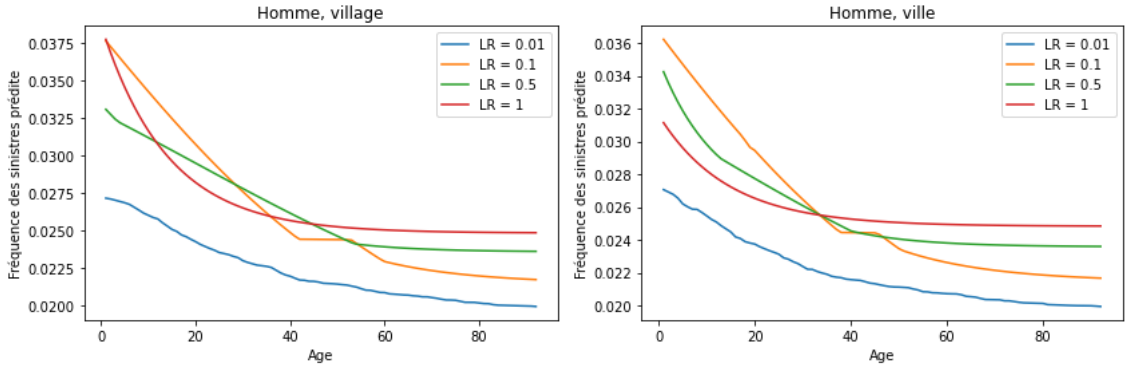


FIGURE 4.6.3 – Fréquence des sinistres prédites par âge pour un homme habitant dans un village et en banlieue.

Sur ces graphiques, il est possible d'observer l'influence du taux d'apprentissage introduit plus tôt sur la déviance prédite des clients types en fonction de l'âge. Plusieurs fait peuvent être notés :

- Plus le taux d'apprentissage est élevé, plus la valeur vers laquelle la valeur prédite converge est élevée.
- Plus le taux d'apprentissage est élevé, et plus la fonction de réponse va ressembler à une aggrégation des fonctions de réponses des apprenants faibles.
  - La courbe représentant le taux d'apprentissage de 1 semble donc naturellement assez lisse, car moins de modèles sont utilisés pour la prédiction.
  - La courbe représentant le taux d'apprentissage de 0.01 semble par contre être beaucoup moins nette, ce qui est clairement dû au nombre bien plus grand d'apprenants faibles entraînés, et peut permettre d'atteindre des fonctions de réponses plus complexes.

Des graphiques similaires représentant les prédictions pour les femmes sont disponibles dans l'annexe 5.

4.6.3 Prédictions des modèles entraînés pour les clients types

Dans cette sous-section seront présentés les résultats des prédictions des modèles entraînés pour les différents clients types. Ces représentations peuvent être intéressantes pour comprendre comment les modèles réagissent par rapport à une variable, tout choses égales par ailleurs. Premièrement, les graphiques en fonction de l'âge du conducteur sont présentés :

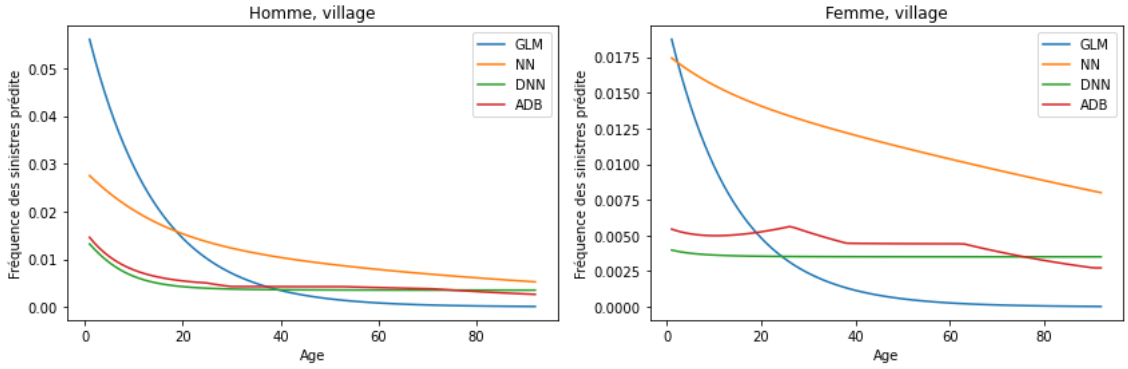


FIGURE 4.6.4 – Fréquence des sinistres prédite pour les clients types.

- En regardant ces graphiques, il est facile de remarquer que les modèles basés sur les RNA semblent estimer une fréquence des sinistres bien plus basses que les prédictions du GLM dans la quasi majorité des cas. Ce résultat peut en partie expliquer pourquoi les fréquences de sinistres par âge ont été sous-estimées dans la partie 4.4.7.1.
- La fréquence de sinistre prédite semble également être plus faible pour les femmes, et ce particulièrement lorsque les clients sont jeunes.

Les graphiques suivants représente l'évolution des prédictions en fonction de l'âge du véhicule :

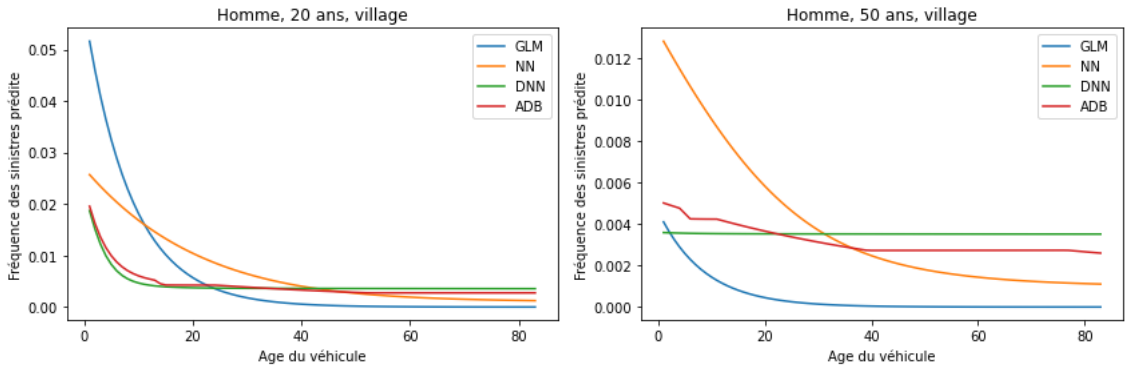


FIGURE 4.6.5 – Fréquence des sinistres prédite pour les clients types.

- La fréquence des sinistres est bien moins grande pour le client type de 50 ans, et ce pour tout les modèles.
- Des conclusions similaires aux graphiques de la figure 4.14 peuvent être prises par rapport à la forme des courbes.

Plus de graphiques similaires sont disponibles pour d'autres clients types dans l'annexe 6.

4.6.4 Comparaison des prédictions et des valeurs réelles

Dans le cadre du calcul de la prime d'un assuré, deux modèles sont généralement construits. Tout d'abord, un modèle estimant la fréquence des sinistres (objet de ce travail), et deuxièmement un modèle estimant la gravité des sinistres (montant que le sinistre va coûter à l'assureur). Une analyse de ces primes pourrait être intéressante, mais comme aucun modèle ayant pour but de prédire la gravité des sinistres n'a été construit, ce sont directement les fréquences des sinistres qui seront comparées.

Dans cette sous-section seront présentées les prédictions des différents modèles choisis, en comparaison avec les valeurs réelles. Pour ce faire, 4 clients sont choisis au hasard dans la base de donnée de test (2 clients avec une fréquence des sinistres nulle, et deux avec une fréquence positive). Les 4 clients qui seront utilisés dans cette analyse sont présentés ici :

- Homme, 49 ans, habitant dans les villes du nord de la Suède. (fréquence positive)
- Homme, 29 ans, habitant une ville de taille moyenne. (fréquence positive)
- Femme, 43 ans, habitant les campagnes du nord. (fréquence nulle)
- Homme, 27 ans, habitant une ville de taille moyenne. (fréquence nulle)

Les résultats sont présentés dans ce graphique :

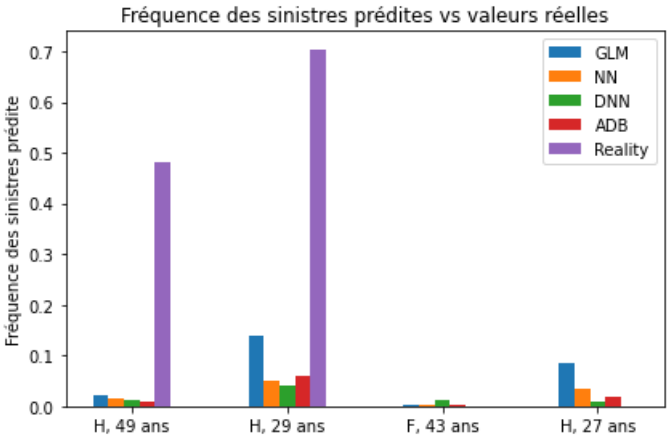


FIGURE 4.6.6 – Fréquences des sinistres prédite par les différents modèles comparées au valeurs réelles.

Plusieurs constats intéressants peuvent être évoqués :

- La tendance du GLM à prédire des valeurs plus élevées que les autres modèles se retrouve bien ici, à part contre le réseau de neurones profond pour le troisième client.
- Tous les modèles sous-estiment la valeur de la fréquence prédite lorsque celle-ci n'est pas nulle.
- Tous les modèles surestiment la valeur de la fréquence prédite lorsque celle-ci est nulle.

Chapitre 5

Conclusion

Pour rappel, le but principal de ce mémoire était à la fois de prouver que l'essort actuel de l'apprentissage automatique pouvait être bénéfique dans le cadre de l'estimation de la fréquence des sinistres pour les clients d'une assurance, mais également de tester l'efficacité du boosting sur ce problème, et en particulier tester l'application des réseaux de neurones comme apprenant faible du méta-algorithme AdaBoost. Afin de disposer d'un point de comparaison pertinent, un modèle linéaire généralisé a été construit sur la base de donnée, ce qui permet de déterminer si les modèles construits dans ce travail sont réellement efficaces.

Un élément central de cette analyse est la recherche des hyperparamètres adéquats. Trouver de bonnes valeurs pour les domaines de ceux-ci a pris du temps, mais a également amélioré les performances des algorithmes de manière considérable.

Premièrement, un réseau de neurones simple a été entrainé, produisant de biens meilleurs résultats que la référence. Par la suite, un réseau de neurones profond a également été construit. Celui-ci dispose également de meilleures performances que le modèle linéaire généralisé, cependant le nombre important de paramètres induit par les nombreux neurones des différentes couches le rend sensible au surentraînement, et celui-ci ne parvient pas à battre le premier réseau de neurones construit.

Enfin, AdaBoost.R2, variation d'AdaBoost modifiée pour fonctionner dans le cas de la régression, a été utilisé en combinaison avec des réseaux de neurones afin de tenter de produire de meilleurs résultats que tout les modèles précédents. Les résultats de cette combinaison sont réellement encourageants, les modèles entraînés avec le boosting associé aux réseaux de neurones produisent de loin les meilleurs résultats de ce travail en terme de déviance.

Une normalisation de la déviance, fonction de perte utilisée pour l'optimisation des paramètres des perceptrons multicouches, a ensuite été incorporée dans l'algorithme d'AdaBoost.R2 en tant que fonction de perte. Les résultats obtenus sont cependant décevants.

Par la suite, les prédictions des différents algorithmes ont été analysées plus en détail et plusieurs constats sont à mentionner.

Tout d'abord, en terme de fréquence des sinistres moyenne (par groupe de clients), le GLM apparait clairement comme un meilleur estimateur, même si celui-ci tends à parfois

surestimer les valeurs prédites. Les modèles basés sur les réseaux de neurones ont quant à eux une grande tendance à sous-estimer ces fréquences moyennes par groupe.

Dans le cadre de l'assurance, il est difficile de prédire la fréquence de sinistre de manière individuelle, y arriver voudrait dire réussire à obtenir une discrimination parfaite des clients et une connaissance exacte de tout les profils. Ceci étant en pratique impossible, c'est souvent par groupes d'individus que les primes seront calculées, et le GLM semble bien plus adapté pour cette tâche.

Il faut cependant noter que si le GLM semble produire de très bonnes estimations pour les classes disposant d'un nombre d'individus élevés [1], celui-ci est plutôt mauvais si le nombre d'individus est faible [2]. Ce comportement vis-à-vis des groupes peu peuplés se retrouve également dans les autres modèles entraînés au cours de ce travail mais en moindre mesure.

Au vu de ces résultats, il est clair que l'application de nouveaux algorithmes à ce type de problème peut être bénéfique, mais nécessiterait encore des améliorations.

Cependant, AdaBoost n'est pas la seule méthode de boosting existante (les GBM [3] peuvent être cités comme exemple), et l'application de ces autres techniques à la problématique de ce mémoire pourrait également produire de bons résultats, voir de meilleurs résultats. Si ce travail était surtout articulé autout de l'algorithme AdaBoost, il serait donc tout de même pertinent de continuer la recherche sur les autres algorithmes de boosting existants [4].

1. Comparaison entre hommes et femmes de la figure 4.11 par exemple.
2. Groupe 54-65 ans de la figure 4.11 par exemple.
3. Gradient Boosting Machine.
4. Autres versions d'AdaBoost, ou encore GBM.

Bibliographie

Beard, Robert Eric, T Pentikainen et Erkki Pesonen (1977). "Risk theory ; the stochastic basis of insurance-2". In :

Breiman (1996). "Bagging predictors". In : *Machine Learning* 24, p. 123-140.

Breiman, Leo (1996). *Bias, variance, and arcing classifiers*. Rapp. tech. Tech. Rep. 460, Statistics Department, University of California, Berkeley.

Burnham, Kenneth P et David R Anderson (2004). "Multimodel inference : understanding AIC and BIC in model selection". In : *Sociological methods & research* 33.2, p. 261-304.

Denuit, Michel, Donatien Hainaut et Julien Trufin (2019a). *Effective Statistical Learning Methods for Actuaries I, GLMs and extensions*. Sous la dir. de Donatien Hainaut. Springer.

— (2019b). *Effective Statistical Learning Methods for Actuaries III, Neural Networks and extensions*. Sous la dir. de Donatien Hainaut. Springer.

— (2019c). "Ensemble of Neural Networks". In : *Effective Statistical Learning Methods for Actuaries III*. Springer, p. 147-166.

Drucker, Harris (1997). "Improving regressors using boosting techniques". In : *ICML*. T. 97, p. 107-115.

Freund, Yoav et Robert E Schapire (1995). "A decision-theoretic generalization of on-line learning and an application to boosting". In : *European conference on computational learning theory*. Springer, p. 23-37.

Freund, Yoav, Robert E Schapire et al. (1996). "Experiments with a new boosting algorithm". In : *icml*. T. 96. Citeseer, p. 148-156.

Géron, Aurélien (2017). *Hands On Machine Learning with Scikit-Learn and Tensorflow*. O'Reilly Media.

Minsky, Marvin L et Seymour A Papert (1988). "Perceptrons : expanded edition". In :

Ohlsson, Esbjörn et Björn Johansson (2010). *Non-Life Insurance Pricing with Generalized Linear Models*. Springer.

Schapire, Robert E (2013). "Explaining adaboost". In : *Empirical inference*. Springer, p. 37-52.

Schapire, Robert E et Yoav Freund (2013). "Boosting : Foundations and algorithms". In : *Kybernetes*.

Schapire, Robert E, Yoav Freund et al. (1998). "Boosting the margin : A new explanation for the effectiveness of voting methods". In : *The annals of statistics* 26.5, p. 1651-1686.

Schwenk, Holger et Yoshua Bengio (2000). "Boosting neural networks". In : *Neural computation* 12.8, p. 1869-1887.

Shao, Jun (1993). "Linear model selection by cross-validation". In : *Journal of the American statistical Association* 88.422, p. 486-494.

Shrestha, Durga L et Dimitri P Solomatine (2006). "Experiments with AdaBoost. RT, an improved boosting scheme for regression". In : *Neural computation* 18.7, p. 1678-1710.

StackExchange contributors (2020). *A list of cost functions used in neural networks, alongside applications*. `https://stats.stackexchange.com/questions/154879/a-list-of-cost-functions-used-in-neural-networks-alongside-applications`. [Online ; accessed 14-July-2020].

Wikipedia contributors (2020a). *Deviance (statistics)*. `https://en.wikipedia.org/w/index.php?title=Deviance_(statistics)&oldid=967483369`. [Online ; accessed 14-July-2020].

— (2020b). *Loss function*. `https://en.wikipedia.org/w/index.php?title=Loss_function&oldid=959425040`. [Online ; accessed 14-July-2020].

Chapitre 6

Annexes

Annexe 1

Pair-plot (sexe)

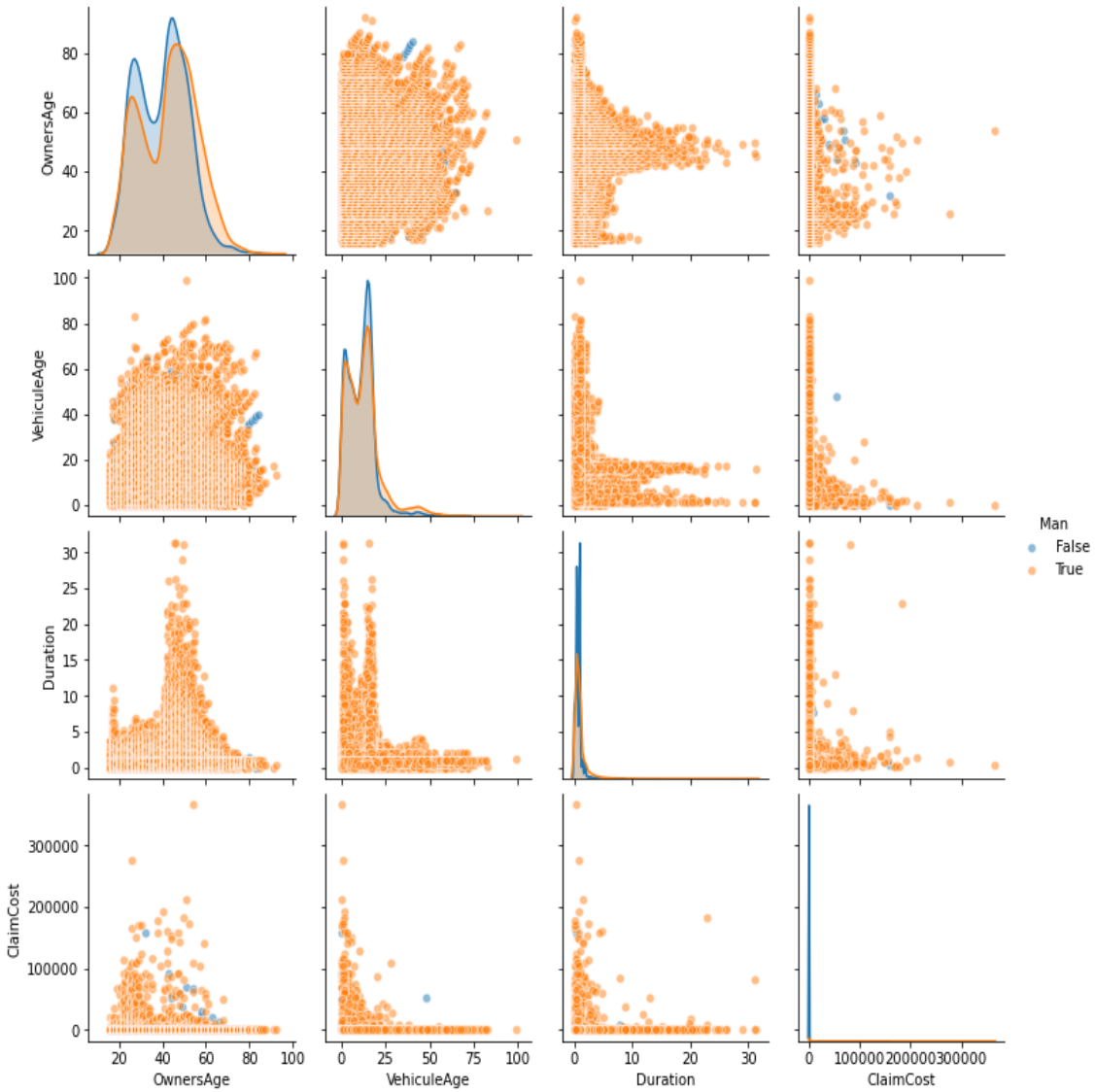


FIGURE 6.0.1 – "Pair-plot" des variables numériques en fonction du sexe.

Annexe 2

Pair-plot (Zone)

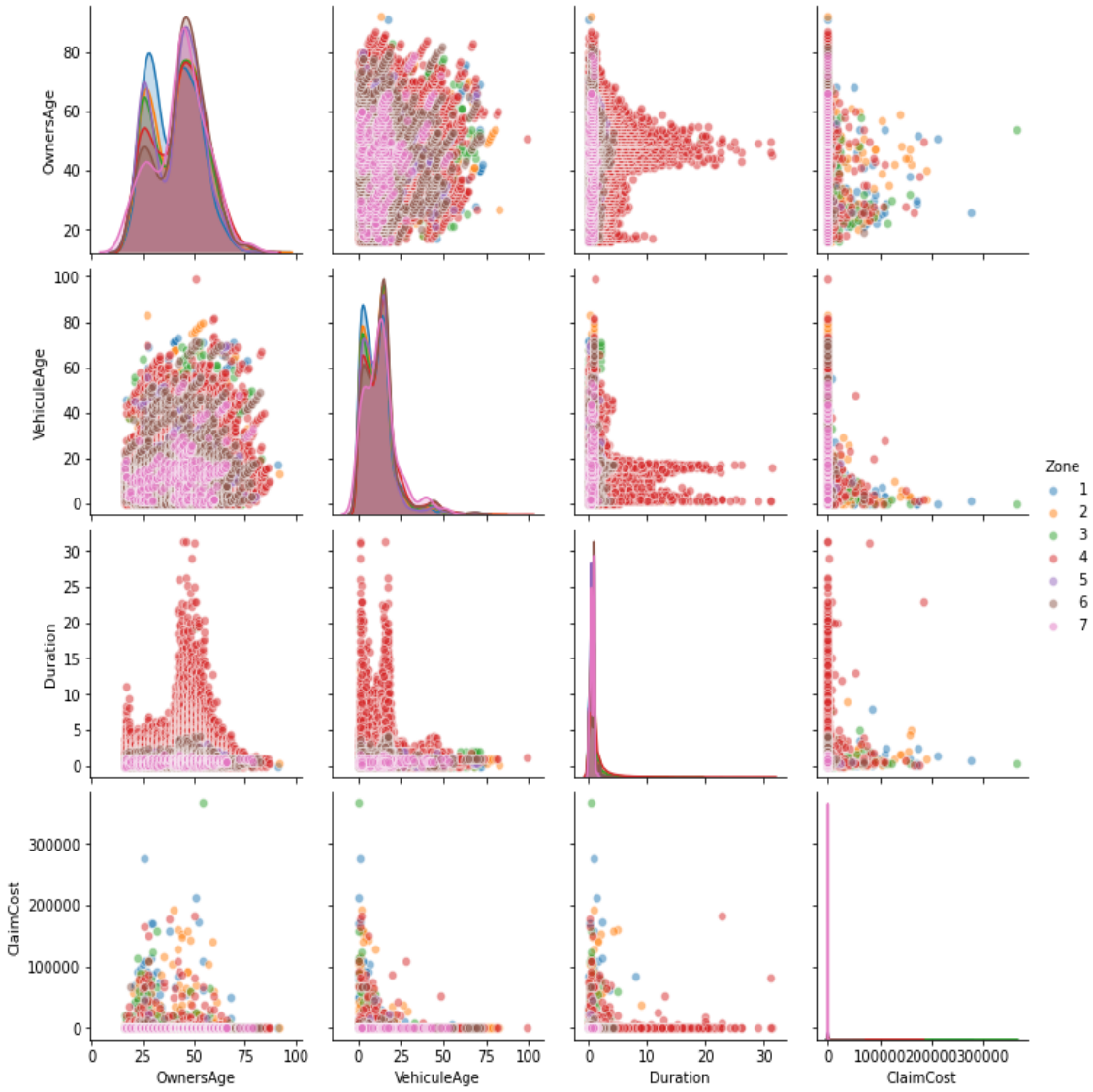


FIGURE 6.0.2 – "Pair-plot" des variables numériques en fonction de la variable "Zone".

Annexe 3

Pair-plot (Class)

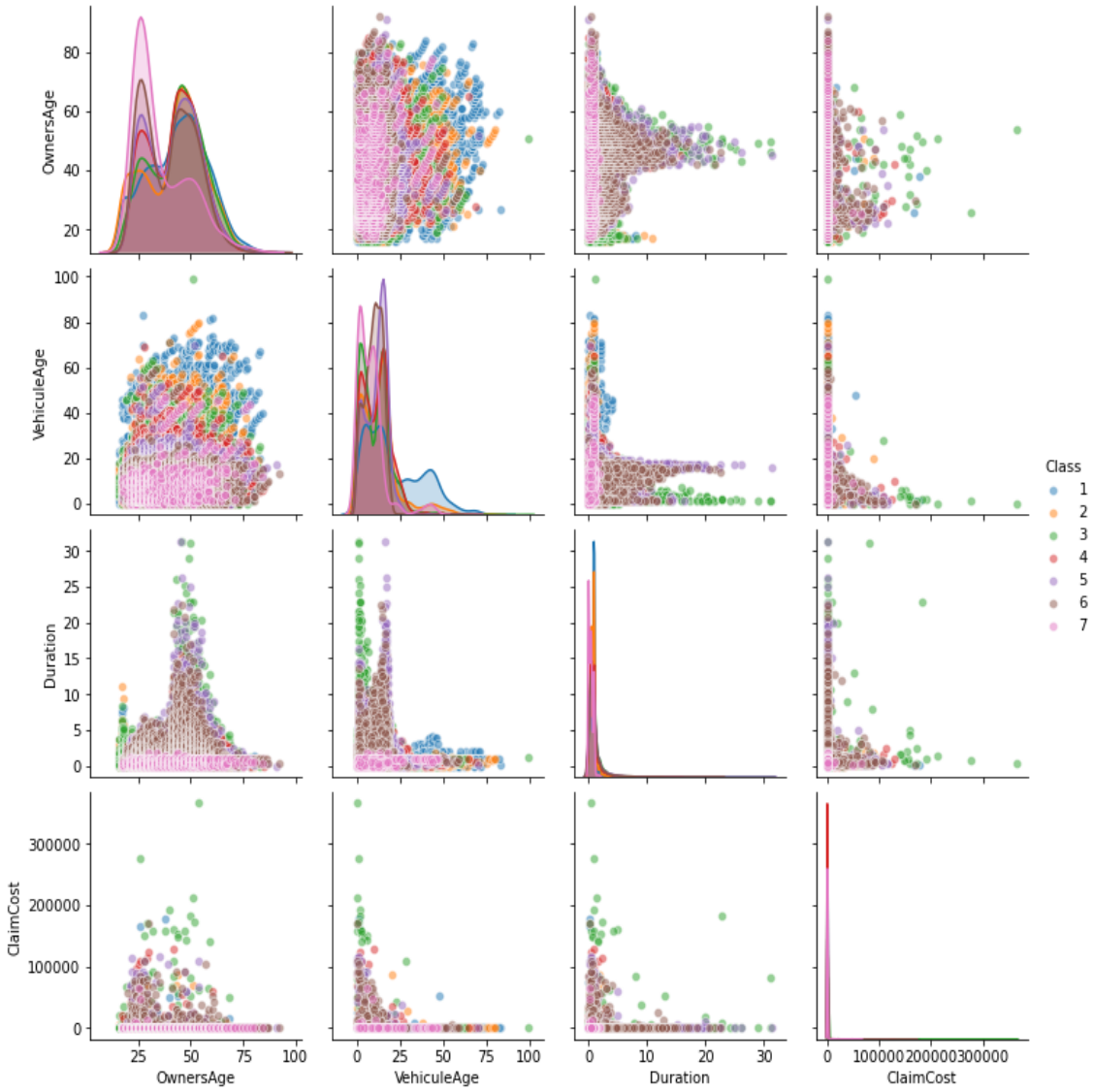


FIGURE 6.0.3 – "Pair-plot" des variables numériques en fonction de la classe du véhicule.

Annexe 4

Fréquence moyenne des sinistres

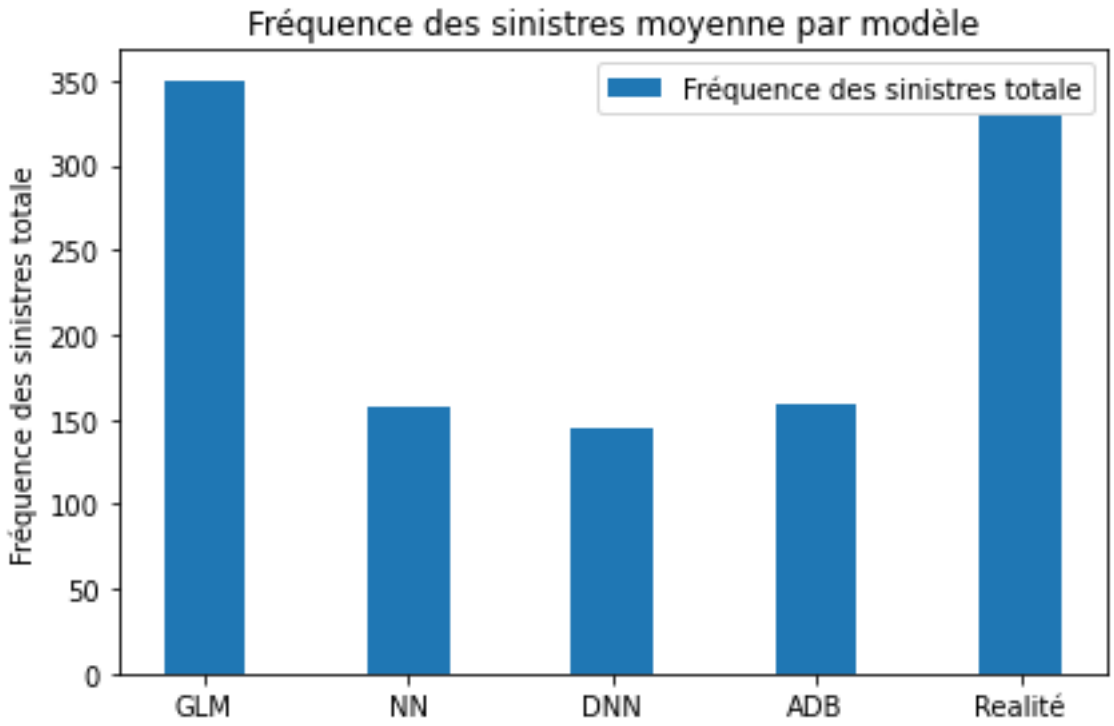


FIGURE 6.0.4 – Fréquence moyenne des sinistres par algorithme utilisé.

Annexe 5

Effet du taux d'apprentissage sur les clients types (femmes)

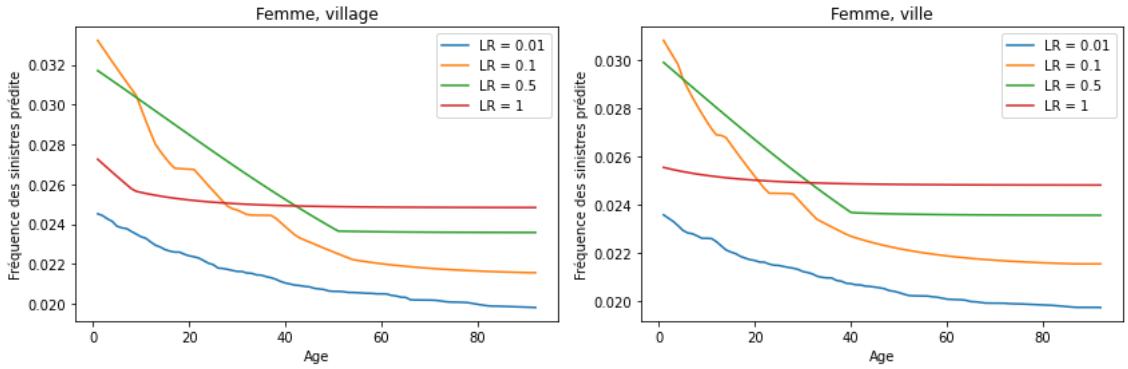


FIGURE 6.0.5 – Fréquence des sinistres prédites par âge pour une femme habitant dans un village et en banlieue.

Annexe 6

Prédictions des modèles entraînés sur les clients types

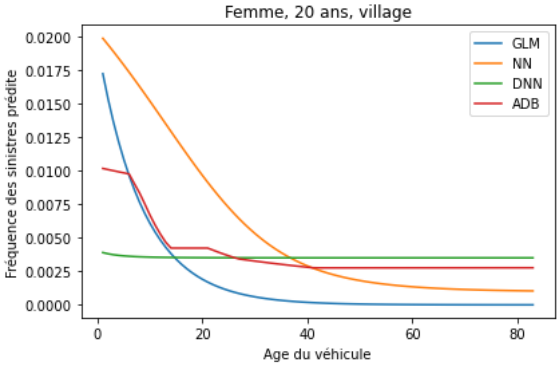


FIGURE 6.0.6 – Fréquence des sinistres prédites en fonction de l'âge du véhicule pour une femme de 20 ans habitant dans un village.

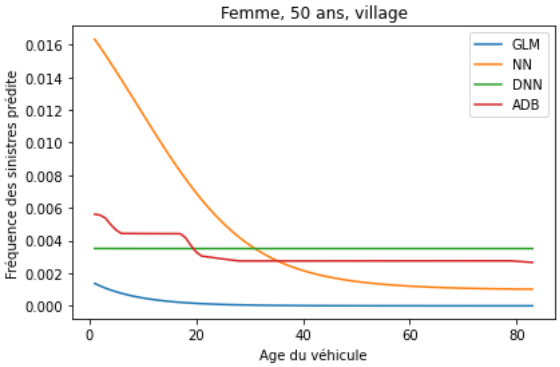


FIGURE 6.0.7 – Fréquence des sinistres prédites en fonction de l'âge du véhicule pour une femme de 50 ans habitant dans un village.

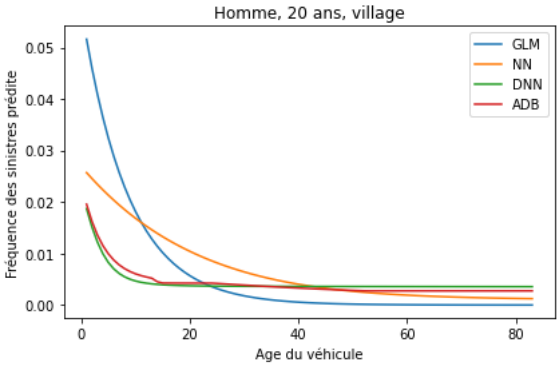


FIGURE 6.0.8 – Fréquence des sinistres prédites en fonction de l'âge du véhicule pour un homme de 20 ans habitant dans un village.

Annexe 7

Notes concernant le code

La grande majorité du code est écrit en Python, sauf une partie de la préparation des données qui a été écrite est R. Il est important de préciser que le code des différents réseaux de neurones utilise Keras [1] ainsi que Cuda [2]. Il est donc possible que pour lancer les codes relatifs aux RNA ou à AdaBoost, les deux soient installés (et configurés pour fonctionner ensemble) sur la machine qui lancera le script. Les différents fichiers sont :

- AdaBoostR2Class.py : Implémentation des deux versions d'AdaBoost (Originale, et avec taux d'apprentissage), recherche des hyperparamètres et courbes d'apprentissage.
- NN.py : Recherche des hyperparamètres pour le réseau de neurones simple et courbes d'apprentissage.
- DNN.py : Recherche des hyperparamètres pour le réseau de neurones profond et courbes d'apprentissage.
- GLM .py : Construction du GLM en python.
- Corrmat.py : Matrice des corrélations.
- Pairdata.py : Pair-plots.
- PreprocTrainTest.py : Division du jeu de données en jeu de test et jeu d'entraînement.
- LRComparisons.py : Comparaison des taux d'apprentissage AdaBoost.R2.
- ModelsComparisons.py : Comparaison des résultats des différents modèles (graphiques).
- Les résultats des différentes recherches des hyperparamètres sont disponibles dans des fichiers .csv (/Python/Param_search_results).

Le code est disponible sur github :
https ://github.com/Lledune/Poisson-neural-network-insurance-pricing/tree/master/Python

1. Outil Python permettant de travailler avec les réseaux de neurones artificiels. Interface pour TensorFlow, la librairie d'apprentissage profond de Google.
2. Programme permettant l'utilisation du GPU pour le calcul des gradients des RNA.