UCL
Université
catholique
de Louvain

Institut de Statistique, Biostatistique et Actuariat

Application d'AdaBoost aux réseaux de neurones dans un cas de régression

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
Juillet 2020

Preface.

This dissertation has been prepared in partial fulfillment of the requirements for the master degree in actuarial sciences delivered by the Université Catholique de Louvain.

Acknowledgment.

Many people have contributed to the completion of this dissertation.

Contents

Chapter 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.
La modélisation de prédiction n'est pas nouvelle dans ce secteur, mais les algorithmes utilisés changent avec l'essort actuel de la technologie. Ainsi nous pouvons maintenant facilement utiliser des réseaux de neurones afin de répondre aux problèmes pour lesquels des algorithmes moins complexes étaient utilisés auparavant. Le but de ce travail est de montrer que ces nouveaux[2] algorithmes sont efficaces et représentent un espoir d'amélioration pour le futur. TODO : expliquer le mémoire rapidement.

[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.

[2] La plupart de ces algorithmes sont en fait assez peu récents mais étaient difficilement applicables en raison d'une puissance de calcul trop faible, citons par exemple le perceptron, élement de base des réseaux de neurones qui a été inventé dès 1957.

Chapter 2

Jeu de données

2.1 Présentation des données

Afin de réaliser ce travail, nous avons besoin d'une base de donnée adéquate, nous présentons celle-ci dans cette sous-section. La base de données est constituée d'informations sur les clients d'une compagnie d'assurance nommée Wasa, entre 1994 et 1998. Les véhicules assurées 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.

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

La variable Zone est décrite dans la table 1.

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 leur fonction plus claire, nous allons maintenant passer à l'analyse exploratoire de celles-ci. 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.

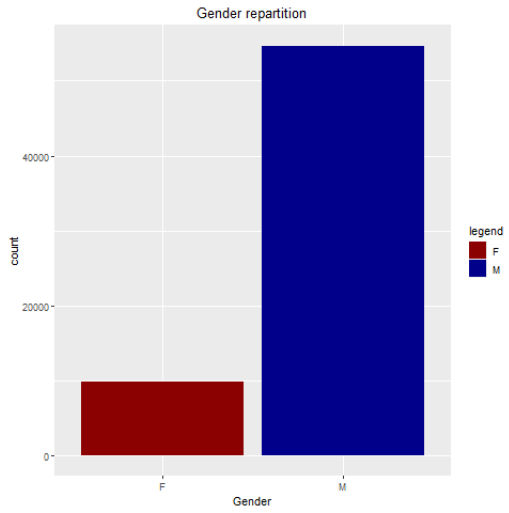


Figure 2.2.1: Distribution Gender

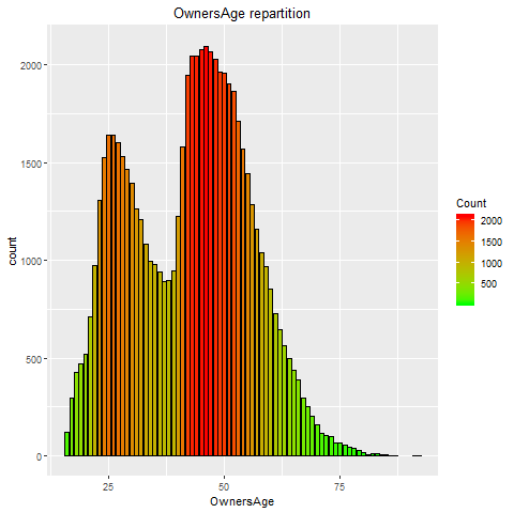


Figure 2.2.2: Distribution OwnersAge

Sur ces deux premières figures nous observons respectivement les distributions des variables Gender et OwnersAge. La première chose que nous pouvons voir est la grande disparité entre le nombre d'hommes de et femmes clients de l'assurance. Notre jeu de données est composé d'hommes pour la grande majorité. Pour ce qui est de la variable OwnersAge, nous constatons que les valeurs sont réparties entre 16 et 92 ans, avec deux "pics" vers 25 et 45 ans. Les valeurs maximales et minimales de nos 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 les distributions des variables VehiculeAge et Zone :

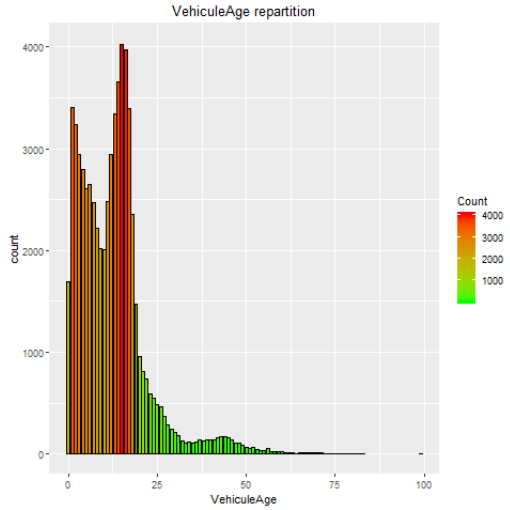


Figure 2.2.3: Distribution VehiculeAge

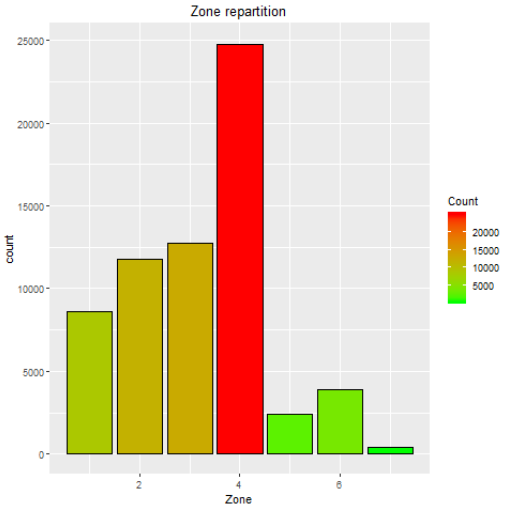


Figure 2.2.4: Distribution Zone

La distribution de vehicule est indiquée en années, et nous pouvons donc observer que 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 outlier et il sera reconsidéré dans la partie preprocessing. La distribution de la variable Zone est intéressante, elle nous 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 vis dans le sud du pays.

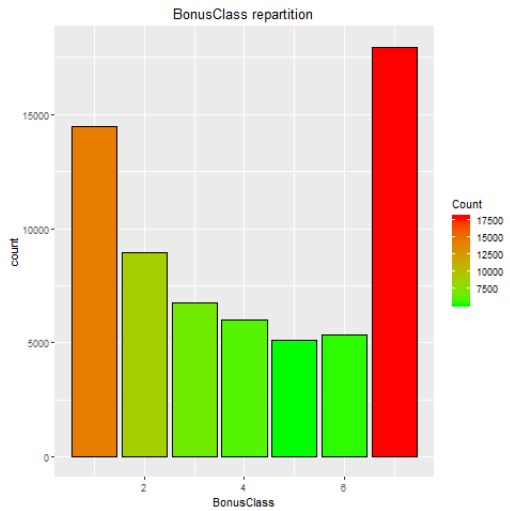


Figure 2.2.5: Distribution BonusClass

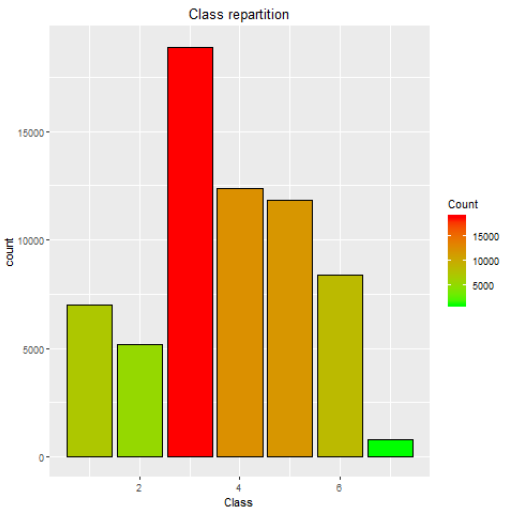


Figure 2.2.6: Distribution Class

Les classes de bonus les plus fréquentes sont les classes 1 et 7, respectivement le minimum (entrée dans la compagnie d'assurance) et le maximum (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 nous montrent la répartitions des sinistres par bonus et par classe de véhicule.

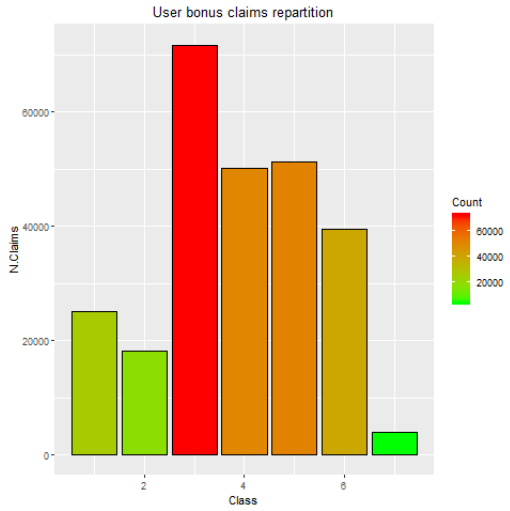


Figure 2.2.7: Bonus Claims

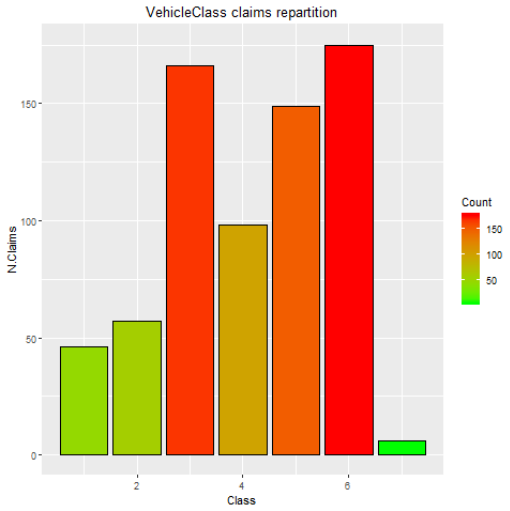


Figure 2.2.8: Class Claims

Il parait logique de supposer que plus un client a un bonus élevé, moins celui-ci causera d'accident, puisque le bonus monte uniquement si le client parvient à compléter une année sans causer d'accident. Cependant en observant la figure 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 Class (figure 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 8 sont moins surprenant : plus un véhicule est puissant, plus le risque d'accident sera important. Les déviations de cette régle par les classes 3 et 7 sont expliquées par la distribution de la population à travers les différentes classes (figure 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 pour la classe 3.

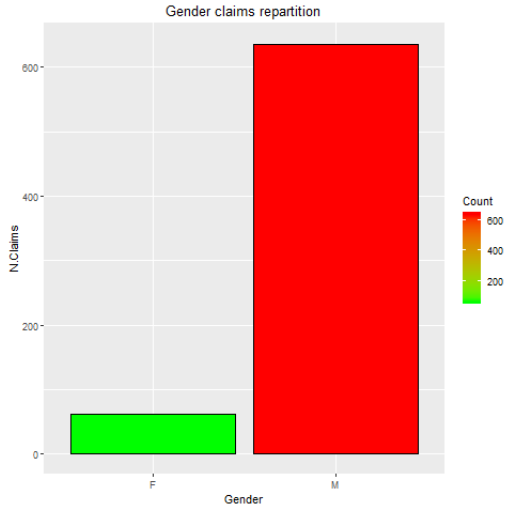


Figure 2.2.9: Gender claims

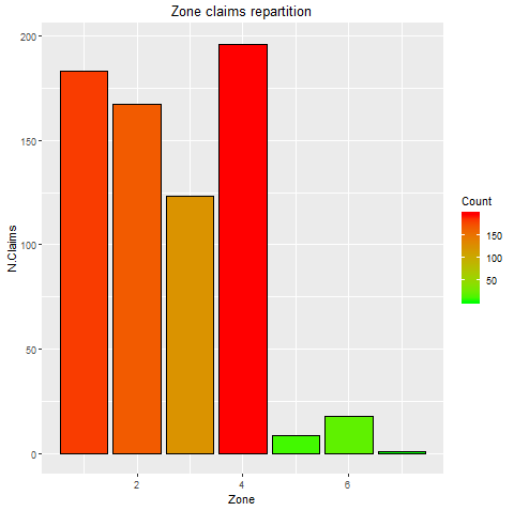


Figure 2.2.10: Zone claims

La répartition des sinistres par sexe indique que les hommes sont plus suceptibles 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, mais 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 10, nous observons que les classes 1 et 3 sont celles qui déclarent le plus de sinistres. La première classe étant plutôt minoritaire (figure 4) nous observons que 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 l'inverse puisque ceux-ci semblent causer assez peu d'accidents.

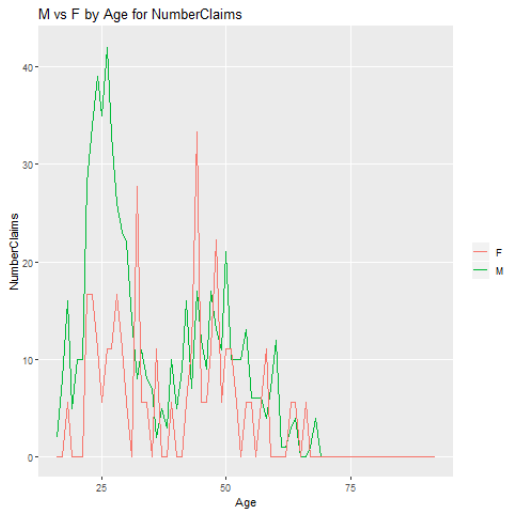


Figure 2.2.11: Gender claims by Age

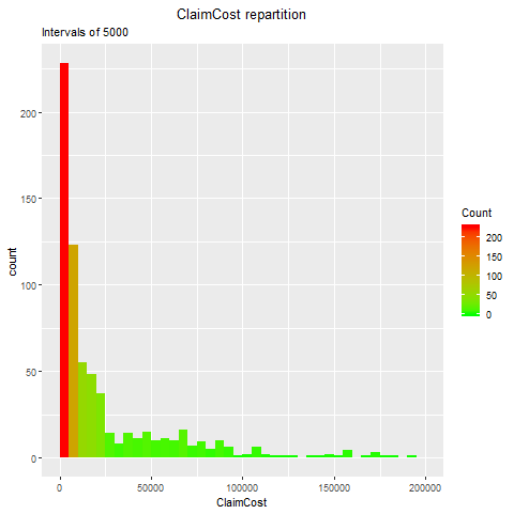


Figure 2.2.12: Distribution ClaimCost

La figure 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, cette tendance s'égalise passé ce cap. Enfin, la figure 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 déclaté de sinistres 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 certains peuvent monter à près de 200.000€.

2.3 Assurances

Dans cette section seront expliquées les particularités de l'analyse de données dans le cadre de l'assurance, et plus précisement dans le cas qui nous intéresse ici.

2.3.1 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 précisemment 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 du 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 facteur d'évaluations, les deux termes seront utilisés indifférement au cours de ce travail.

2.3.2 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[2] v.

$$Y = X/v$$

Il existe différents ratios mais ici ne sera présenté que celui utilisé dans l'analyse à venir : La fréquence des réclamations. Il s'agit du rapport entre le nombre de réclamations et la durée du contrat, nous obtenons ainsi une fréquence qui sera la variable prédite par nos modèles.

2.3.3 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 :

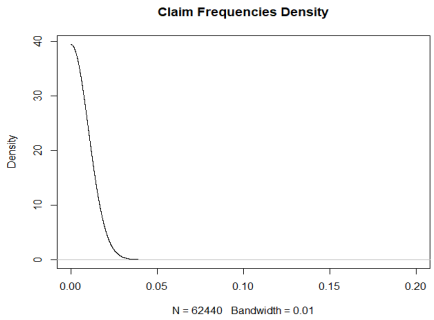
$$Y \sim Poisson(\lambda)$$

[2]L'exposition peut être le nombre de sinistres ou la durée du contrat.

$$E[Y] = \lambda$$

$$Var[Y] = \lambda$$

Afin de vérifier que la distribution de la fréquence des sinistres suit bien une loi de poisson celle-ci 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 précisemment 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 binomiale. Une variable aléatoire Y suit une dispersion exponentielle si sa fonction de distribution suit la forme suivante (Denuit, Hainaut, and Trufin 2019b) :

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

- θ_i: paramètre dépendant de i.
- ϕ: paramètre identique pour tous les i. C'est le paramètre de dispersion.
- $c(.)$ est indépendante du paramètre θ_i.

2.4 Preprocessing

Le preprocessing des données est une étape très importante lorsque nous travaillons avec ce type de données et celui-ci n'est pas à négliger. Il consiste a préparer les données pour les algorithmes, afin que ceux-ci puissent fonctionner de manière optimale. Dans ce travail, plusieurs méthodes de preprocessing ont été utilisées.

2.4.1 Variables binaires (Dummy variables)

Certaines de nos 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 sinon. Par exemple si x_{ij} représente la variable associée au client i et à la variable j, nous pouvons écrire :

$$x_{ij} = \begin{cases} 0 \text{ if } x_i \notin j \\ 1 \text{ if } x_i \in j \end{cases}$$

Les variables Gender, Zone, Class, et BonusClass peuvent être transformées de la sorte. Ce travail sera effectué avec le package caret[3] de R. La fonction incluse dans ce package crée n variables binaires, la dernière sera donc supprimée car celle-ci ne représente aucune information. Prenons pour exemple la variable Gender. Si $x_{iGender} = 1$ alors le client est une femme. Cependant nous ne devons pas créer une deuxième variable binaire puisque si $x_{iGender} = 0$ nous pouvons en déduire que le client est un homme. Ceci explique pourquoi nous n'avons besoin que de $n-1$ variables binaires pour représenter toute l'information de n catégories appartenant à une variable catégorielle.

2.4.2 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, ce ne sera pas le cas dans le cadre de ce travail). Cependant cela ne signifie pas qu'aucune méthode de preprocessing ne doit être appliquée à ces données. La normalisation des données quantitatives aide certains algoithmes 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[4] utilisées. Celles-ci varient rapidement entre 0 et 1 (ou -1 et 1) et sans la

[3] Classification And REgression Tools : Package de machine learning pour R.
[4] Voir chapitre 3

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 nous ne décrirons que celle qui sera utilisée dans le cadre de ce travail : la normalisation Min Max (Denuit, Hainaut, and Trufin 2019b).
Elle consiste (pour une variable j) à convertir l'ensemble des données sur l'intervalle [0,1] en se servant de $Max(x_j)$[5] et $Min(x_j)$[6] 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.

2.4.3 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 nous pouvons soit garder la valeur si celle-ci semble ajouter de l'information au modèle ou la supprimer dans le but d'une meilleure généralisation par celui-ci. Dans la section 2.2 une donnée a particulièrement attiré notre attention comme outlier potentiel : une moto de 99 ans. Pour celle-ci nous pouvons entrevoir deux possibilités :

- 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 supporser 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.

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-aberantes seront donc conservées.

2.4.4 Durée de contrat

Pour le bon fonctionement 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. Les observations incriminées sont celles ayant une durée

[5] Valeur maximale prise par la variable j.
[6] Valeur minimale prise par la variable j.

de contrat nulle. Ces valeurs nulles de durée de contrat sont peu cohérentes (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.

Chapter 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 cette section, les différents algorithmes utilisés lors des analyses seront présentés à des fins de compréhension du travail. Pour commencer, des explications seront données sur le fonctionement 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 bases que sont les modèles linéaires généralisés et les réseaux de neurones artificiels seront introduit et leur fonctionement expliqué. Enfin, le boosting sera également étudié et plus particulièrement le "meta-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 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 notre 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, aucours 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 (après que celui-ci ait été validé, voir section 3.1.4). Les valeurs prédites par l'algorithmes 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 parmis 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'example 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, and 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 :

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

La déviance non mise à l'échelle peut ensuite être obtenue en multipliant cette formule par le paramètre de dispersion ϕ (voir section 2.3.3). 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 de nos modèles (Denuit, Hainaut, and Trufin 2019b).

- $a'(\theta) = e^\theta$

- $h(y) = ln(y)$

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

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

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

Cette formule est obtenue en remplaçant y_i par 0. Bien que trivial, ce développement est important car sans cette formule simplifiée, le cas $y_i = 0$ pose problème pour l'optimisation. En effet, le terme $y_i ln(y_i)$ vaut évidemment 0 dans ce cas précis, cependant $ln(0) = -\infty$ et ceci était source d'erreurs de convergence pour l'optimisation[4].

[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.

[4] Optimisation faite avec le package Keras, ceci était plus un problème informatique que mathématique.

3.1.3 Overfit, AIC et BIC

Il est important de comprendre le but derrière la création d'un tel modèle. Nous cherchons à approximer une fonction de réponse Y à l'aide de celui-ci. Afin que cette fonction sois 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 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 de variables explicatives ou une phase d'apprentissage mal configurée. Le modèle est alors inutilisable dans un scénario réél.
Afin de choisir des modèles limitant l'overfit, deux nouveaux critères seront introduits : AIC et BIC. Comme les autres critères vus, ce sont des mesures de la qualité d'un modèle statistique. Il est (presque) toujours possible d'améliorer les résultats d'un modèle sur les critères basiques en augmentant le nombre de variables explicatives. Cependant cette pratique augmente les risque d'overfit, c'est la raison pour laquelle dans la création d'un modèle la parcimonie[5] du modèle est importante. Ces deux critères permettent de prendre en compte le nombre de variables explicatives dans le calcul de la qualité de modèle (Denuit, Hainaut, and Trufin 2019b; Burnham and Anderson 2004).

- $AIC = 2k - 2ln(L)$ [6]
  (Où k est le nombre de paramètres à estimer du modèle et L est le maximum de la fonction de vraisemblance du modèle)

- $BIC = k\,ln(N) - 2ln(L)$ [7]
  (Où N est le nombre d'observations dans l'échantillon)

3.1.4 Validation croisée (K-fold)

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, c'est pourquoi il est souvent préférable de travailler avec la base de donnée complète en évitant cette division

[5] Principe consistant à utiliser le moins de variables explicatives possible pour expliquer un phénomène afin de réduire le phénomène d'overfit.
[6] Akaike Information Criterion
[7] Bayesian Information Criterion

(Denuit, Hainaut, and Trufin 2019b). C'est pour palier à ce problème que la validation croisée sera utilisée.
Le principe de base est de diviser le jeu de données en K ensembles de taille égale. 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, and 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)

begin
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 $E(Q)$
end

Cette méthode permet de limiter l'overfit et de le détecter 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. Comme dit précédemment la base de donnée est constituée d'événements rares et cette méthode sera donc préférée à la traditionnelle décomposition entrainement/test.

3.2 GLM : modèle linéaire généralisé

3.2.1 Les bases du GLM

Les GLMs représentent une généralisation de l'OLS[8] dans deux directions :

- La distribution de probabilité : Dans l'OLS, nous supposons que les données suivent une loi gaussienne. Avec les GLM, nous travaillons avec une classe de distrution (les distributions EDM) dont la fonction de distribution générale a été présentée plus tôt dans ce travail. Plus particulièrement dans notre cas, cela va nous permettre de construire nos 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 confiances, ...

- 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 précisemment plus loin dans ce chapitre.

3.2.2 Hypothèses

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

[8] Ordinary Least Squares (régression linéaire)

Hypothèse 1 : Indépendance des polices d'assurances Considérons que nous avons n polices d'assurances, avec X_i correspondant à la réponse de la police i. Nous supposons que $X_1, ..., X_n$ sont indépendants.

Même si cette hypothèse est primordiale, il n'est pas difficile de trouver des cas ou 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 intéresseront pas dans le cadre de ce travail.

Hypothèse 2 : Indépendance temporelle Considérons 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 Considérons 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 : Il concernent soit une police différente soit un intervalle de temps différent.

3.2.3 Modèles à dispersion exponentielle (EDM)

Les modèles à dispersion exponentielles font parties 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 and Johansson 2010).

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

- 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 distributions 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. Considérons $N(t)$ comme le nombre de sinistres d'un indivitu au cours de la période $[0, t]$. Beard, Pentikäinen et Pesonen montrent que ce processus statistique suit une loi de poisson (Beard, Pentikainen, and Pesonen 1977). Si X_i représente le nombre de plaintes dans une classe de durée ω_i et prenons μ_i comme l'espérence lorsque $\omega_i = 1$. Alors X_i suit une distribution de Poisson et sa densité est :

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

La distribution du ratio $Y_i = X_i / w_i$ est aussi étudiée et tombe également sur une distribution de Poisson (Ohlsson and Johansson 2010)[9]:

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

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

Qui est la 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

Forme de tableau

Dans ce travail, nous avons jusqu'ici 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 ω_i ainsi que les facteurs d'évaluations correspondants. Cette forme représente la forme de liste, et est celle qui sera préférée lorsque nous travaillons 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 and 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

[9] 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

Fonction de lien

Maintenant que les notations utilisées dans cette parties sont expliquées, nous pouvons nous pencher sur la deuxième partie de la généralisation des GLMs, la fonction de lien. Considérons 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 nous allons considérer que la première variable comporte deux classes et la deuxième en comporte trois, ci dessous 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, ce qui nous servira à ajouter des contraintes, en particulier que sa moyenne μ_{ij} soit égale à γ_0. Si nous suivons notre exemple et 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 :

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

Nous réécrivons maintenant le modèle sous forme de liste et définissons :

- $\beta_1 := \gamma_0$
- $\beta_2 := \gamma_{12}$
- $\beta_3 := \gamma_{22}$
- $\beta_4 := \gamma_{23}$

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

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

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

$$\mu = X\beta$$

Généralisons maintenant 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.

Nous introduisons maintenant la fonction de lien $g(.)$.

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

Ici $g(.)$ représente la fonction de lien est 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) nous cherchons à avoir $\mu_{ij} = \gamma_o\gamma_{1i}\gamma_{2j}$, ce qui peut être obtenu grâce au logarithmes :

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

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

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

La fonction de lien est donc le logarithme dans le cas des modèles multiplicatifs, bien souvent utilisés dans les problèmes d'assurances (Ohlsson and 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 :

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

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 :

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

- $v(.)$: fonction de variance

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

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

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

3.2.6 Modélisation de Poisson

Nous allons maintenant discuter du cas où Y suit une loi de poisson, ce qui est le cas pour notre base de données. Supposons :

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

Alors les propritétés de la loi de poisson donnent :

$$E(Y_i) = \lambda_i$$

$$var(Y_i) = \lambda_i$$

Donc la fonction de variance sera :

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

Et la fonction de lien utilisée dans ce cas-ci sera le logarithme (qui nous 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 peut l'indiquer, 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", et lorsqu'un neurone reçoit un certain nombre de signaux en même temps, celui-ci va déclencher son propre signal (Denuit, Hainaut, and Trufin 2019b).

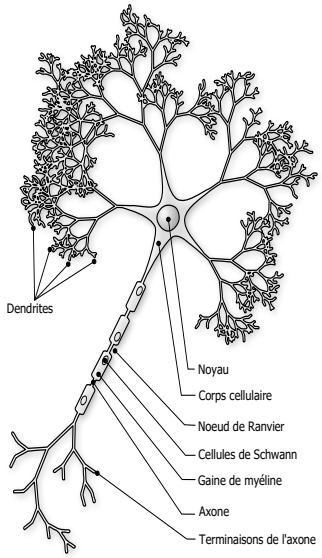


Figure 3.3.1: Schéma d'un neurone biologique.

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 très 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 nos algorithmes car non différentiable, ce qui est un problème pour la calibration, voir sections

suivantes), c'est d'ailleurs cette fonction qui a été utilisée en premier pour les perceptrons (Géron 2017).

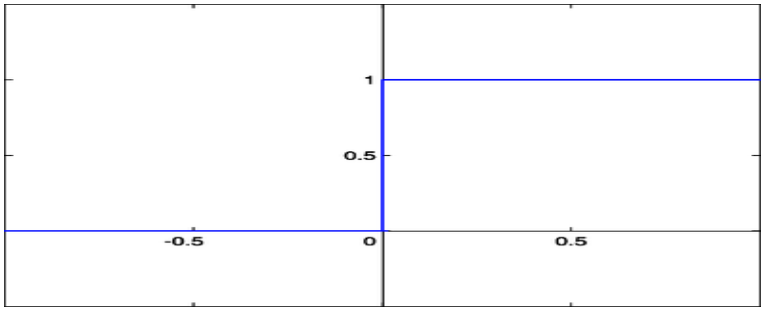


Figure 3.3.2: Exemple de fonction d'activation : Step function

Pour en revenir au perceptron, notons que les données d'un individu sont considérées comme un vecteur X_i et que 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'intéragit 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 variable explicatives utilisées. Un perceptron peut être représenté comme ceci :

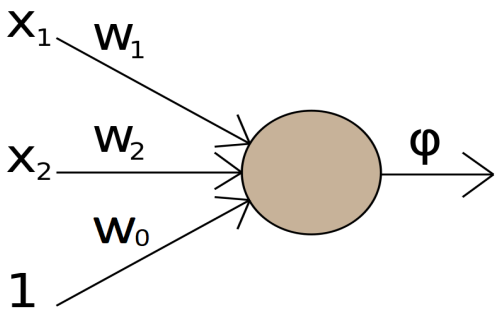


Figure 3.3.3: Représentation d'un perceptron

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 fonctionement 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 signifie 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 nous utilisons une règle de mise à jour des poids, afin de donner plus d'importance aux entrées qui auraient rendu la prédiction correcte. Ce procédé sera expliqué plus en détail dans la section suivante, mais nous pouvons 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 nous devons modifier les poids 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[10], insolvable par le perceptron et mentionné par Minsky et Papert dans leur papier intitulé "Perceptrons" qui met en avant un certaines faiblesses du perceptron. Suite à ces découvertes la recherche dans le domaine va être 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

3.3.2.0.1 Fonctionnement Le perceptron multicouche, plus communément appelé MLP[11] est une structure composée de couches, elles-mêmes formées par plusieurs perceptrons. Pendant longtemps, les chercheurs ont tenté de trouver une manière efficace

[10]"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.
[11]Multilayer perceptron

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 :

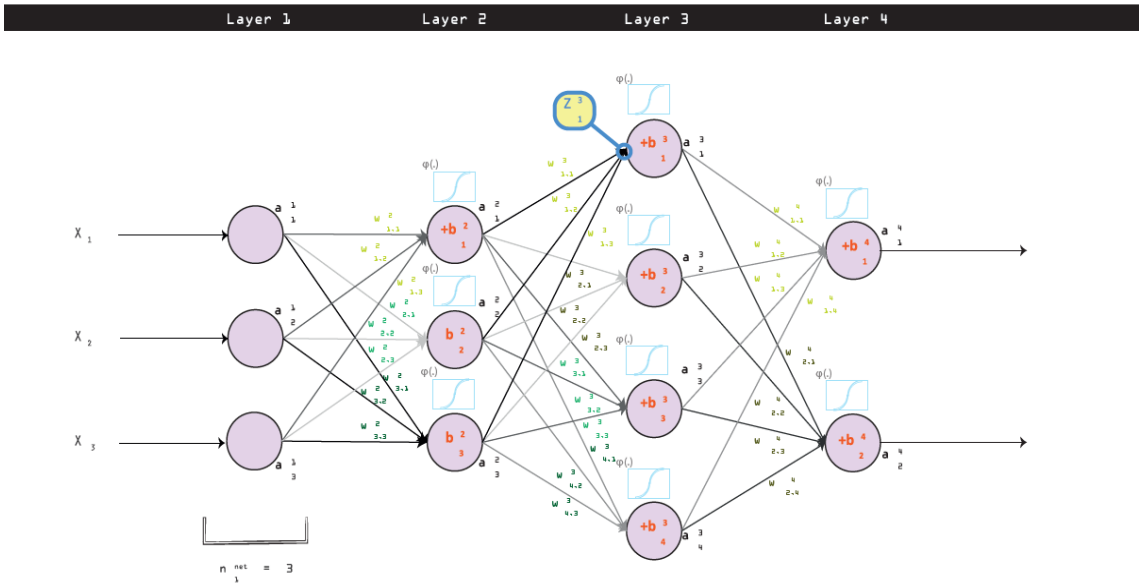


Figure 3.3.4: Représentation d'un MLP "feed-forward".

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[12]. Comme observable sur le graphique, l'information circule dans un seul sens sur notre schéma. Il faut savoir qu'il existe différent 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 seront donc pas vus en détails, 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, et le modèle nous renvoie une réponse $\hat{y}_i$ en sortie.
Un réseau tel que celui-ci est composé de plusieurs couches à travers 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[13] et enfin le couche de sortie renvoyant la réponse du réseau. Le nombre de neurones par couche et le nombre de couches cachées

[12]Computer vision.

[13]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).

sont tout deux des hyperparamètres[14] à définir.
Le fonctionement de ce réseau est similaire à celui du perceptron, cependant son architecture 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, le 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 nous cherchons par exemple à estimer à quelle classe une instance appartient nous pouvons représenter chaque classe par un neurone dans cette couche.

Le fonctionement 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 neurone serait inefficace. Il existe diverses méthodes permettant d'optimiser un réseau de neurone 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, and Trufin 2019b; Géron 2017).

[14] 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.

3.3.2.0.2 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_{jk}^i). Cette méthode présente cependant un inconvenient de taille : l'inversion numérique de la matrice Hessienne est très long 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_{jk}^i 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[15] t. Le vecteur Ω_t est modifié d'un petit pas dans la direction opposée à celle du gradient à chaque époque 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, and 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**

[15] Époque : une période ou moment particulier dans l'exécution de l'algorithme.

3.4 Boosting

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 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 de 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 de 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 raisonement 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 échantillonage 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.

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-algortithme.

Ces méthodes se basent sur l'entrainement d'apprenants faibles[16] combinés afin de créer un ensemble de prédicteurs considéré comme "fort". Elles se basent souvent sur une règle de décision 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 ceux-ci. 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"[17], il en existe plusieurs versions à la fois pour la classification et la régression et c'est ce méta-algorithme qui sera utilisé.

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 and 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 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 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)[18](Drucker 1997).

AdaBoost suppose qu'un apprenant de base (apprenant faible) peut être utilisé pour crée 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, puisque nous somme 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 :

[16] 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".

[17] Boosting adaptatif.

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

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

Afin d'améliorer le 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 produires des hypothèses de bases différentes et variées, ce qui au final permet d'obtenir de biens meilleur 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 :

Algorithm 3.3 AdaBoost (Freund and 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 and 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 permets une hypothèse h_{t+1} efficace sur les examples 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 :

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

Qui peut être scindée en deux cas (si l'exemple i est correctement classé ou non) comme indiqué dans l'algorithme 3 (Schapire and 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 prenant leur décision 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 réél 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 le plus confiants quant à leur décisions 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 parmis 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 section 3.4.6.

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 suivant de se concentrer sur les exemple mal prédits.

- Aggrégation 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 examples sont ensuites 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'additioner (en suivant l'odre 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.

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 de rasoir d'Ockham est un principe qui vient de la philosophie, mais dont le raisonement 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" (réf wiki). 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 nous allons le voir dans cette section. (boosting margin ref).

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[19] 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 recquiert pas que la variance soit élevée afin de délivrer de bonnes performances.

TODO : Faire graphique learning curves pour montrer + conclusions.

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 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 que 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 rééchantillonages).

3.4.6 Erreur naturelle

En suivant cette méthode 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 est possible de le démontrer (Drucker 1997) :

- N_1: le nombre d'observations du jeu d'entraînement.

[19] Décision basée sur l'aggrégation du vote de plusieurs modèles, comme dans le boosting.

- 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) :

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

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

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.

Bibliography

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

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

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

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

Denuit, Michel, Donatien Hainaut, and Julien Trufin (2019a). *Effective Statistical Learning Methods for Actuaries I, GLMs and extensions*. Ed. by Donatien Hainaut. Springer.

— (2019b). *Effective Statistical Learning Methods for Actuaries III, Neural Networks and extensions*. Ed. by Donatien Hainaut. Springer.

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

Drucker, Harris (1997). "Improving regressors using boosting techniques". In: *ICML*. Vol. 97, pp. 107–115.

Freund, Yoav and 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, pp. 23–37.

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

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

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

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

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

Schapire, Robert E and 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, pp. 1651–1686.

Schwenk, Holger and Yoshua Bengio (2000). "Boosting neural networks". In: *Neural computation* 12.8, pp. 1869–1887.

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

Shrestha, Durga L and Dimitri P Solomatine (2006). "Experiments with AdaBoost. RT, an improved boosting scheme for regression". In: *Neural computation* 18.7, pp. 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].