Techniques d'apprentissage pour des problèmes de fragmentation

Victor Hoffmann

▶ To cite this version:

Victor Hoffmann. Techniques d'apprentissage pour des problèmes de fragmentation. Mathématiques [math]. 2023. hal-04142294

HAL Id: hal-04142294
https://inria.hal.science/hal-04142294

Submitted on 26 Jun 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

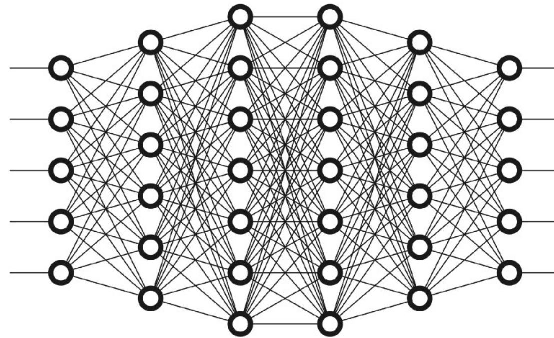
PROJET RECHERCHE
RAPPORT DE FIN D'ANNEE

TECHNIQUES D'APPRENTISSAGE POUR DES PROBLEMES DE FRAGMENTATION

VICTOR HOFFMANNN

ELEVE A L'ECOLE DES MINES DE NANCY

14 MAI 2023



ENCADRANTS :
MADALINA DEACONU & ANTOINE LEJAY.

INSTITUT ELIE CARTAN DE LORRAINE
INRIA NANCY GRAND EST

Table des matières

1 Introduction **5**

1.1 Qu'est-ce que la fragmentation ? . . . 5

1.1.1 Définition . . . 5

1.1.2 Fragmentation de roche . . . 5

1.2 Le modèle de Kuz-Ram . . . 6

1.3 Objectifs de ce projet . . . 7

2 Bases d'apprentissage automatique **8**

2.1 La tâche T . . . 8

2.2 La Mesure de Performance, P . . . 8

2.3 L'expérience, E . . . 9

2.4 Sur-apprentissage et sous-apprentissage . . . 10

2.5 Hyperparamètres et ensembles de validation . . . 12

2.6 Recherche des hyperparamètres optimaux . . . 13

2.6.1 GridSearch . . . 13

2.6.2 Random Search . . . 13

3 Les données et leur prétraitement **14**

3.1 Description des données . . . 14

3.1.1 Informations statistiques de base . . . 15

3.2 Visualisation des données . . . 16

3.2.1 L'algorithme SNE . . . 16

3.2.2 L'algorithme t-SNE . . . 19

3.2.3 Méthodes d'optimisation pour l'algorithme t-SNE . . . 21

3.2.4 t-SNE sur les données . . . 21

3.3 Prétraitement des données . . . 23

3.3.1 Suppression des *outliers* . . . 23
3.3.2 Transformation normale des données quantitatives : . . . 23
3.3.3 Normalisation des données . . . 26

4 Modèles d'apprentissage automatique utilisés **27**
4.1 Régression Linéaire et Polynomiale . . . 27
4.2 Régression Ridge . . . 28
4.3 Régression lasso et ElasticNet . . . 28
4.3.1 Lasso . . . 28
4.3.2 ElasticNet . . . 29
4.4 Réseaux de Neurones Profonds . . . 29
4.4.1 Les neurones . . . 29
4.4.2 Les couches . . . 29
4.4.3 Les poids . . . 30
4.4.4 La fonction de coût . . . 30
4.4.5 Epoque . . . 30
4.5 Régression à vecteurs suppots . . . 31
4.5.1 Machine à vecteurs supports . . . 31
4.5.2 Extension des machines à vecteurs supports à la régression . . . 33
4.6 Méthodes Ensemblistes . . . 34
4.6.1 Principe général, classifieurs fort et faible . . . 34
4.6.2 Le boosting . . . 35
4.6.3 Les modèles de stacking . . . 38

5 Mesure de l'incertitude des réseaux de neurones **39**
5.1 Définition . . . 39
5.1.1 La loi de Bayes . . . 40
5.2 Quel est l'intérêt de l'apprentissage profond bayésien ? . . . 41
5.3 Inférence Variatonnelle . . . 41
5.3.1 Introduction/Définitions . . . 41
5.4 Processus gaussien . . . 42
5.4.1 Exemples . . . 42
5.4.2 L'intérêt des processus gaussiens dans l'apprentissage profond bayésien . . . 42
5.4.3 Des processus gaussiens aux processus gaussiens profonds . . . 43
5.5 Monte Carlo Dropout . . . 43
5.5.1 Dropout . . . 43
5.5.2 Le Dropout, une approximation Bayésienne. . . . 44

6 Protocole et résultats **45**
6.1 Protocole 45
6.1.1 Gestion des données 45
6.1.2 La mesure de performance 46
6.1.3 Sélection de caractéristiques 46
6.1.4 Entraînement des modèles 47
6.2 Les résultats 48
6.2.1 Les scores des modèles 48
6.3 Analyse des résidus des modèles 49

7 Conclusion et travaux futurs **50**
7.1 Conclusion 50
7.2 Travaux futurs 51

Chapitre 1

Introduction

1.1 Qu'est-ce que la fragmentation ?

1.1.1 Définition

La **fragmentation** est la première étape d'une décomposition. C'est-à-dire la rupture du lien qui unissait les unités, pouvant aboutir à leur autonomie et à la disparition de l'ensemble décomposé.

La fragmentation joue un rôle central dans plusieurs domaines tels que les sciences naturelles ou l'industrie. On doit la plupart du temps décrire comment les amas présents dans l'expérience évoluent au cours du temps et plus précisément la manière dont ils se séparent en plus petits amas. C'est un sujet important dans les sciences physiques (moteurs), l'astrophysique (formation d'astéroïdes), en géosciences (éboulement, avalanches), etc [7].

1.1.2 Fragmentation de roche

La **fragmentation de roche** est le processus par lequel la roche est décomposée en plus petits fragments à l'aide d'outil mécaniques ou d'explosifs. La distribution des tailles de fragment peut être caractérisée par un histogramme montrant le pourcentage de taille de chaque particule. La première manière de fragmenter de la roche dans les mines est par le biais d'explosifs. Faire une explosion efficace permet d'économiser beaucoup d'argent qui aurait autrement servi à faire une deuxième explosion [1].

L'explosion dépend de plusieurs paramètres. Certains sont contrôlables, comme l'espacement entre chaque explosion, la quantité d'explosifs dans chaque trou etc. D'autres sont incontrôlables (le module d'Young de la roche par exemple). Il est ainsi nécessaire que les ingénieurs spécialisés en explosifs puissent trouver la bonne configuration des paramètres contrôlables pour avoir une explosion de qualité, créant des fragments de roche suffisamment petits pour être évacués. Cependant, ces fragments ne doivent pas non plus être trop petits, sous peine d'être inexploitables et d'être conduits dans des terrils, ce qui a un coût à la fois économique et environnemental.

1.2 Le modèle de Kuz-Ram

Plusieurs études ont tenté de prédire la distribution de la taille des fragments à partir des paramètres contrôlables et incontrôlables. Plusieurs modèles empiriques sont alors créés. Celui le plus utilisé est le modèle de **Kuz-Ram**.

Le modèle de Kuz-Ram [5] est constitué de trois parties : l'équation de Kuznetsov, l'équation de Rossin-Rammler et l'index d'uniformité de Cunningham.

Ici, nous donnerons une version modifiée de l'équation de Kuznetsov donnant un résultat plus précis que l'équation originale, mais demandant plus de paramètres.

La version modifiée de l'équation de Kuznetsov permet de prédire la taille médiane des fragments et s'écrit de la manière suivante :

$$X_{50} = AK^{-0.8}Q^{\frac{1}{6}}\left(\frac{115}{RWS}\right)^{\frac{19}{20}} \tag{1.1}$$

avec :

X_{50} la taille médiane du fragment en cm,

A le facteur de roche,

K le facteur de poudre en $\mathrm{kg\,m^{-3}}$, qui peut être vu comme la masse d'explosif par mètre cube de roche,

Q la masse en kg d'explosif dans le trou,

RWS la chaleur de la réaction par kg comparé à l'ANFO (qui est un explosif 25% plus puissant que la TNT).

L'équation de Rosin-Rammler cherche à estimer toute la distribution des tailles de fragment. Pour une grille composée d'ouvertures de taille X, l'équation est capable d'estimer le pourcentage de fragments retenus :

$$R_x = \exp\left(-0.693\left(\frac{X}{X_{50}}\right)^n\right) \tag{1.2}$$

avec R_x la proportion de fragments plus grands que l'ouverture X et n l'index d'uniformité de Cunningham.

L'index d'uniformité de Cunningham est donnée par la formule suivante :

$$n = \left(2.2 - \frac{14B}{d}\right)\sqrt{\left(\frac{1+\frac{S}{B}}{2}\right)}\left(1-\frac{W}{B}\right)\left(\left|\frac{BCL-CCL}{L}\right|+0.1\right)^{0.1}\frac{L}{H} \tag{1.3}$$

avec B le « *burden* », S l'espacement entre les trous en m, d le diamètre du trou en mm, W l'écart type de la précision du forage en m, L la longueur d'une charge d'explosif, BCL la hauteur du bas de la colonne d'explosifs, CCL la longueur de la colonne d'explosifs et H la profondeur du trou (ces paramètres seront décrits plus précisément dans le chapitre 3).

Ainsi, en connaissant l'index d'uniformité n et la taille médiane de fragments X_{50}, nous sommes capables d'avoir la distribution de la taille des fragments sous forme d'un pourcentage de fragments qui passent par rapport à une certaine ouverture de taille X.

L'avantage du modèle empirique de Kuz-Ram est sa facilité à être mis en place. En effet, on a besoin de relativement peu de paramètres qui sont faciles à obtenir. Néanmoins, ce modèle omet certains paramètres significatifs pour connaître la taille médiane des fragments de manière plus précise. L'équation modifiée de Kuznetsov donnée ci-dessus prend en compte certains de ces paramètres significatifs, toutefois sa précision n'est pas toujours suffisante.

1.3 Objectifs de ce projet

Le problème de l'équation de Kuznetsov permettant d'estimer la taille médiane des fragments X_{50} est qu'elle est dans certains contextes peu précise (notamment selon le facteur de roche du site sur lequel a lieu l'explosion). On pourrait alors se demander si des méthodes d'apprentissage automatique permettraient d'avoir des résultats plus précis. Evidemment, notre modèle de *machine learning* devra être tout autant facile d'utilisation que le modèle de Kuz-Ram. En d'autres termes, notre modèle devra prendre les mêmes paramètres ou des paramètres tout aussi faciles à obtenir que ce dernier.

Le travail de ce projet de recherche est donc d'appliquer des méthodes de *machine learning* et de *deep learning* à ce problème de fragmentation de roche afin de faciliter sa résolution. Pour ce faire, le travail a du se décomposer en plusieurs étapes :

1. Se renseigner sur plusieurs aspects théoriques du *machine learning* et du *deep learning*.
2. Identifier un problème de fragmentation et récolter des données sur ce-dernier.
3. Mettre au point différents modèles candidats à la résolution de ce problème.
4. Optimiser les modèles en ajustant leurs hyperparamètres.
5. Mesurer la performance des modèles.
6. Vérifier la validité du modèle.

Le problème de fragmentation entrepris est un problème de fragmentation de roche étudié par Richard Amoako, Ankit Jha et Shuo Zhong [1]. Le but est d'estimer la taille médiane de fragments X_{50} à partir des variables données. Il s'agit donc d'une tâche de régression. Plusieurs modèles ont été testés pour sa résolution : *ElasticNet*, réseau de neurones, modèles de *Gradient Boosting*, *Support Vector Regression*, modèle de *stacking*, ... Les hyperparamètres optimaux des modèles ont été trouvés à l'aide de la méthode *Random Search*. Puis les performances des modèles ont été mesurées avec l'erreur moyenne quadratique ainsi que le coefficient de détermination R^2. L'incertitude a pu être mesurée à l'aide d'une méthode nommée *Monte Carlo Dropout* sur le réseau de neurones profonds. Nous avons pu vérifier la validité des modèles en observant leurs résidus.

Chapitre 2

Bases d’apprentissage automatique

L’ensemble de ces concepts est présentée de manière plus exaustive dans [11]. Ici, nous détaillons quelques uns de ces concepts.

Un algorithme de *machine learning* est un algorithme qui est capable d’apprendre à partir de données. MITCHELL (1997) donne une définition de ceci :

> « Un algorithme dit apprendre de l’expérience E selon la classe de tâches T et de mesure de performance P, si ces performances aux tâches T mesurées par P, s’améliorent avec l’expérience E. »

2.1 La tâche T

Dans la définition relativement formelle du mot "tâche", le processus d’apprentissage en lui-même n’est pas la tâche. Apprendre est notre moyen d’atteindre la capacité de pouvoir exécuter la tâche.

Les tâches de *machine learning* sont souvent décrites en fonction de comment le système d’apprentissage automatique doit traiter **un exemple**. Un exemple est une collection de **caractéristiques** qui ont été mesurés quantitativement par un objet ou événement que nous voulons que le système d’apprentissage profond traite. Nous représentons typiquement un exemple comme un vecteur $\mathbf{x} \in \mathbb{R}^n$ où chaque composante x_i du vecteur est une autre caractéristique. Par exemple, les caractéristiques d’une image sont généralement la valeur des pixels de l’image.

Beaucoup de tâches différentes peuvent être résolues avec l’apprentissage profond. Ici, nous n’utiliserons que la tâche de régression :

• **Régression :** On demande au programme de prédire une valeur numérique à partir d’une certaine entrée. Pour résoudre cette tâche, on va demander à l’algorithme de sortir une fonction $f : \mathbb{R}^n \to \mathbb{R}$. Ce type de tâche est similaire à la classification, à l’exception près que le format de la sortie est différent. Un exemple de tâche de régression est la prédiction de prix.

2.2 La Mesure de Performance, P

Afin d’évaluer les capacités d’un algorithme de *machine learning*, nous devons façonner une mesure quantitative de sa performance. Habituellement, cette mesure de performance est spécifique à la

tâche T réalisée par le système.

La mesure de performance est souvent représentée par une fonction de perte, qui mesure l'erreur entre la vraie valeur de la sortie et la sortie prédite. Pour des tâches telles que la régression, nous utilisons souvent l'**erreur moyenne quadratique**(MSE) comme fonction de perte. Elle est représentée par :

$$MSE = \frac{1}{m} \sum_{i}^{m} (\hat{\mathbf{y}} - \mathbf{y})_i^2 \,, \tag{2.1}$$

avec $\mathbf{y}$ l'ensemble des sorties, $\hat{\mathbf{y}}$ l'ensemble des prédictions de la sortie et m le nombre de sorties.

Généralement nous sommes intéressés de voir comment est-ce que l'algorithme performe sur des données qu'il n'a encore jamais vues, étant donné que cela détermine la performance de l'algorithme sur le terrain. On évalue ainsi ces performances avec un **ensemble de données de test** séparé des données d'entraînement.

Le choix de la mesure de performance peut sembler clair et objectif, mais il est souvent difficile de choisir une mesure de performance qui correspond bien au comportement désiré du système.

Dans certains cas, c'est parce qu'il est diffcile de décider ce qui doit être mesuré. Par exemple, lors d'une tâche de régression, devrions nous plus pénaliser le système s'il fait souvent des erreurs moyennes ou s'il fait des grosses erreurs ? Ce type de choix dépend du contexte.

2.3 L'expérience, E

Les algorithmes de machine learning peuvent être globalement catégorisés comme **supervisés** ou **non supervisés** selon le type d'expérience qu'ils peuvent avoir lors de la phase d'apprentissage. Ici, nous n'utiliserons que des algorithmes supervisés, étant donnée que nous cherchons à résoudre un problème de régression.

La plupart des algorithmes cités ici peuvent avoir accès à tout un *dataset* ou ensemble de données. Un *dataset* est une collection de beaucoup d'exemples. Parfois on appellera les exemples des *data points* ou observations.

Les algorithmes non supervisés observent un jeu de données contenant beaucoup de caractéristiques, puis apprennent des propriétés utiles concernant la structure de ce jeu de données. Dans le contexte du *deep learning*, nous voulons habituellement apprendre l'entière distribution de probabilité ayant généré un ensemble de données, que ce soit de manière explicite avec les fonction de densité, ou implicite avec des tâches telles que la synthèse ou le dé-bruitage. D'autres algorithmes non supervisés ont d'autres rôles, comme par exemple le *clustering*, qui consiste à diviser le jeu de données en *clusters* (amas) d'exemples similaires.

Les algorithmes supervisés observent un jeu de données contenant des caractéristiques, mais chaque exemple est également associé avec un **label** ou une **cible**. Par exemple, le jeu de données *Iris* [1] contient les espèces de chaque plante iris. Un algorithme supervisé pourra étudier ce jeu de données pour apprendre à classifier les plantes iris dans trois différentes espèces selon leurs mesures.

Grossièrement, un algorithme non supervisé implique observer plusieurs exemples d'un vecteur aléatoire x et tenter d'apprendre implicitement ou explicitement la probabilité de distribution

1. Il s'agit d'un jeu de données classique qui accompagne de nombreuses bibliothèques.

$p(x)$, ou des propriétés intéressantes de cette distribution. Tandis que les algorithmes supervisés impliquent d'observer plusieurs exemples d'un vecteur aléatoire x et une valeur ou vecteur associé y et d'apprendre à prédire y à partir de x généralement en estimant $p(y|x)$. Le terme **apprentissage supervisé** provient de la vue de la cible y donné par un instructeur ou un enseignant qui montre au système quoi faire. Lors d'un apprentissage non supervisé, il n'y a pas d'instructeur ou d'enseignant et l'algorithme doit apprendre à donner du sens aux données sans ce guide. Néanmoins les notions d'algorithme supervisé et non supervisé ne sont pas proprement définies. La frontières entre ces deux notions est parfois floue. Beaucoup de technologies de *machine learning* peuvent être utilisées pour effectuer les deux tâches.

2.4 Sur-apprentissage et sous-apprentissage

Le défi central du *machine learning* est que nous devons être performant sur des entrées *nouvelles, encore non observées* - et pas seulement sur celles sur lesquelles notre modèle a été entraîné. Cette capacité se nomme **généralisation**.

Typiquement, lorsque nous entraînons un modèle de *machine learning*, nous avons accès à un ensemble d'entraînement, pouvons calculer une mesure de l'erreur sur cet ensemble d'entraînement et cherchons à la réduire, ce qui est un problème d'optimisation. Toutefois, ce qui différencie le *machine learning* de l'optimisation est que nous voulons que **l'erreur de généralisation**, c'est-à-dire **l'erreur de test** soit également basse. L'erreur de généralisation est définie comme la valeur de l'erreur sur une nouvelle entrée.

Comment peut-on alors affecter la performance de l'ensemble de test si nous pouvons uniquement observer les données d'entraînement ? Le champ de **la théorie d'apprentissage statistique** nous donne des éléments de réponse. Si l'ensemble d'entraînement et de test sont collectés arbitrairement, il y a en effet peu de choses à faire. Toutefois, si nous pouvons émettre des hypothèses sur la manière dont l'ensemble d'entraînement et de test sont collectés, alors on peut améliorer le modèle.

Les données de test et d'entraînement sont générés par une distribution de probabilité sur les ensembles de données nommé **processus de génération de données**. Nous faisons généralement un ensemble d'hypothèses connues sous le nom de **hypothèses i.i.d**. Ces hypothèses sont que les observations dans chaque ensemble de données sont **indépendantes** entre eux, et que l'ensemble d'entraînement et de test sont **identiquement distribuées**, c'est à dire que les deux ensembles ont la même distribution de probabilité. Cette hypothèse nous permet de décrire le processus de génération de données avec une distribution de probabilité sur une seule observation. La même distribution est utilisée pour générer chaque observation d'entraînement et de test. On appelle cette distribution la **distribution de génération de données** notée p_{data}. A l'aide de ces outils on peut étudier mathématiquement la relation entre l'erreur d'entraînement et de test.

Une connexion immédiate que l'on peut observer entre l'erreur de test et d'entraînement est que l'erreur d'entraînement d'un modèle aléatoire est égale à l'erreur de test de ce même modèle. On observe que pour une valeur fixée de w, l'erreur d'entraînement et de test sont exactement égales, parce que les deux prédictions sont formées en utilisant le même processus de prélèvement.

Évidemment, lorsque nous utilisons un algorithme de *machine learning*, nous ne pouvons pas fixer les paramètres à l'avance, puis prélever les deux ensembles de données. Il faut d'abord prélever l'ensemble d'entraînement et ensuite l'utiliser pour choisir les paramètres nécessaires à réduire l'erreur d'entraînement, puis on prélève l'ensemble de test. Sous ce processus, l'erreur de test

attendue est plus grande ou égale à l'erreur d'entraînement. Les facteurs déterminant comment un algorithme de *machine learning* traite les données sont ses capacités de :

1. Réduire l'erreur d'entraînement.
2. Réduire la différence entre l'erreur d'entraînement et de test.

Ces deux facteurs correspondent à deux défis centraux du *machine learning* : ***l'overfitting*** **ou sur-apprentissage** et ***l'underfitting*** **ou sous-apprentissage**. On observe du sous-apprentissage lorsque le modèle n'est pas capable d'obtenir une valeur assez basse sur l'ensemble d'entraînement. Quant au sur-apprentissage, il arrive lorsque la différence entre l'erreur d'entraînement et de test est trop grande.

On peut contrôler le fait qu'un modèle soit plutôt sur-adapté ou sous-adapté en modifiant sa **capacité**. Informellement, la capacité d'un modèle est son habilité à être ajusté avec un grand panel de fonctions. Les modèles avec une faible capacité peuvent avoir du mal à s'adapter sur le set d'entraînement. Les modèles à forte capacité peuvent donner une situation de sur-apprentissage en mémorisant les propriétés du set d'entraînement qui ne sont pas utiles sur l'ensemble de test.

Une manière de contrôler la capacité d'un algorithme de *machine learning* est de choisir son **espace d'hypothèse**, c'est-à-dire, l'ensemble de fonctions que l'algorithme peut prendre comme solutions. Par exemple, la régression linéaire a comme espace d'hypothèse l'ensemble de toutes les fonctions linéaires. Nous pouvons généraliser la régression linéaire de telle sorte à ce qu'elle inclut les polynômes dans son espace d'hypothèse, ce qui augmentera la capacité du modèle.

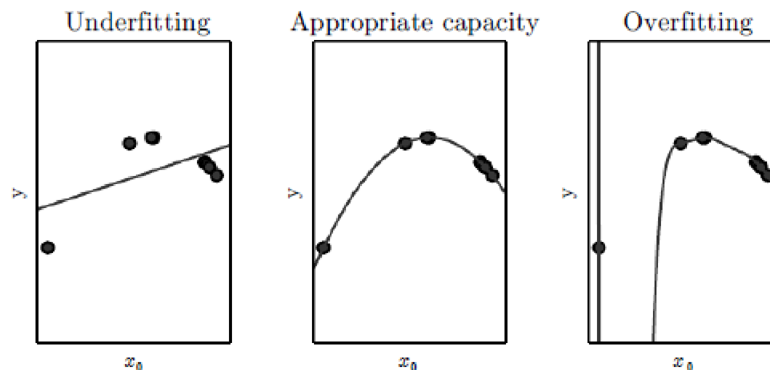


Figure 2.1 – Nous essayons d'adapter trois modèles à cet exemple de données d'entraînement. Cette dernière a été générée synthétiquement, en prenant des valeurs aléatoires de x et en choisissant y de manière déterministe en évaluant une fonction quadratique. A gauche, la fonction linéaire souffre de sous-apprentissage — elle ne peut pas modéliser la courbure présente dans les données. Au centre la fonction quadratique se généralise bien aux points « invisibles ». Elle ne souffre pas d'une quantité significative de sous-apprentissage ou de sur-apprentissage. A droite, le polynôme de degré 9 souffre de sur-apprentissage. En effet, cette fonction est capable de représenter la bonne fonction, mais elle peut également représenter un nombre infini d'autres fonctions passant par les mêmes points, parce que nous avons plus de paramètres que de points d'entraînement.

2.5 Hyperparamètres et ensembles de validation

La plupart des algorithmes de machine learning ont plusieurs paramètres que l'on peut utiliser afin de contrôler le comportement de l'algorithme d'apprentissage. Ces paramètres sont nommés **hyperparamètres**. Les valeurs des hyperparamètres ne sont pas adaptés par l'algorithme lui-même.

Par exemple, dans une régression polynomiale il n'y a qu'un seul hyperparamètre : le degré du polynôme, qui joue le rôle d'**hyperparamètre de capacité**.

Parfois un hyperparamètre n'est pas un paramètre en raison de sa difficulté à être optimisé. Plus fréquemment, le paramètre doit être un hyperparamètre car il n'est pas approprié d'apprendre ce paramètre sur l'ensemble d'entraînement. Ceci s'applique pour tous les paramètres qui contrôlent la capacité du modèle. Si on les apprenait sur l'ensemble d'entraînement, de tels hyperparamètres choisiront toujours le modèle avec la plus grande capacité, ce qui causera inéluctablement à du sur-apprentissage (une régression polynomiale avec un degré très grand représentera nécessairement mieux les données d'entraînement qu'une régression linéaire, sans être pour autant performant sur les données de test).

Pour résoudre ce problème, il nous faut un **ensemble de validation** d'observations que l'algorithme d'entraînement n'observe pas.

Plus tôt, nous avions expliqué comment un ensemble de test composé d'observations venant de la même distribution que l'ensemble d'entraînement pouvait être utilisé pour estimer l'erreur de généralisation, une fois le processus d'apprentissage terminé. Il est primordial que l'ensemble de test ne soit pas utilisé pour faire des choix sur le modèle, les hyperparamètres compris. Pour cette raison, nous construisons toujours l'ensemble de validation à partir de l'ensemble d'entraînement. Les données de validation vont être utilisées pour guider la sélection des hyperparamètres. Comme ce dernier est utilisé pour « entraîner » les hyperparamètres, l'erreur de l'ensemble de validation va sous-estimer l'erreur de généralisation. Une fois l'optimisation de l'ensemble des hyperparamètres faite, l'erreur de généralisation peut être estimée en utilisant l'ensemble de test.

Validation croisée

Diviser l'ensemble de données en un ensemble d'entraînement et un ensemble de test fixe peut être problématique si l'ensemble de test est petit. Un petit ensemble de test implique une incertitude statistique autour de l'erreur de test moyenne estimée, ce qui rend l'évaluation difficile. En effet, on aura plus de mal à dire qu'un algorithme A marcherait mieux qu'un algorithme B sur une tâche donnée.

Quand l'ensemble de données a 100 000 exemples ou plus, ce n'est pas un grand problème. Néanmoins, lorsque l'ensemble de données est petit, un échantillon de validation indépendant n'est pas toujours disponible. De plus, d'un échantillon de validation à un autre, la performance de validation du modèle peut varier. La validation croisée permet de tirer plusieurs ensembles de validation d'une même base de données et ainsi d'obtenir une estimation plus robuste, avec biais et variance, de la performance de validation du modèle.

On va utiliser la **validation croisée à k-blocs.** on divise l'échantillon original en k échantillons (ou « blocs »), puis on sélectionne un des k échantillons comme ensemble de validation pendant que les $k-1$ autres échantillons constituent l'ensemble d'apprentissage. Après apprentissage, on peut calculer une performance de validation. Puis on répète l'opération en sélectionnant un autre

échantillon de validation parmi les blocs prédéfinis. À l'issue de la procédure nous obtenons ainsi k scores de performances, un par bloc. La moyenne et l'écart type des k scores de performances peuvent être calculés pour estimer le biais et la variance de la performance de validation.

2.6 Recherche des hyperparamètres optimaux

L'optimisation des hyperparamètres est absolument fondamentale en apprentissage automatique. En effet des hyperparamètres mal ajustés peuvent amener le modèle à être en sous-apprentissage ou en sur-apprentissage. Plus un modèle est complexe, plus il aura d'hyperparmètres. Par exemple, le modèle de *gradient boosting XGBoost* possède plus d'une vingtaine d'hyperparamètres. Il serait donc extrêmement pénible de modifier l'ensemble des hyperparmètres à la main. Heureusement, plusieurs fonctions automatisent le processus de manière plus ou moins exhaustive.

Ces méthodes sont expliquées de manière plus détaillée dans [11]

2.6.1 GridSearch

Lorsqu'il y a moins de 4 hyperparamètres, il est commun d'effectuer ce qu'on appelle un *Grid Search*. Pour chaque hyperparmètre, l'utilisateur sélectionne une petite quantité finie de valeurs à explorer. L'algorithme de *Grid Search* va alors entraîner un modèle pour toutes les combinaisons des valeurs des hyperparamètres possibles. L'expérience qui donne le meilleur score (`R^2,val_accuracy etc.`) sur la base de validation pour une configuration d'hyperparamètre donnée est retenue.

Le problème évident de *Grid Search* est que son coût de calcule croît exponentiellement avec le nombre d'hyperparamètres. S'il y a m hyperparamètres prenant chacun n valeurs, alors sa complexité est en $O(n^m)$.

2.6.2 Random Search

Il y a une alternative à *Grid Search* qui est tout aussi simple à programmer et converge plus rapidement vers les bons hyperparamètres : *Random Search.*

Random Search procède de la manière suivante. D'abord nous définissons une distribution marginale pour chaque hyperparamètre (par exemple, une loi uniforme ou de Bernoulli pour des valeurs binaires ou discrètes, ou une exponentielle de gaussienne pour des valeurs positives continues). Par exemple :

$$\texttt{log_learning_rate} \sim \mathcal{U}(-1, -5), \tag{2.2}$$

où $\mathcal{U}(a, b)$ indique un échantillon d'une distribution uniforme dans l'intervalle (a, b).

Contrairement à *Grid Search*, on *ne discrétise pas* l'ensemble des valeurs des hyperparamètres. Il faut toutefois donner un nombre maximal de combinaisons à essayer pour que l'algorithme puisse terminer. Ceci permet d'explorer un plus grand ensemble de valeurs et n'augmente pas le coût du modèle. En fait, *Random Search* peut être exponentiellement plus efficace que *Grid Search*, quand il y a plusieurs hyperparamètres qui ne dépendent pas fortement de la mesure de performance.

Chapitre 3

Les données et leur prétraitement

Dans ce chapitre, nous abordons la description du tableau utilisé pour la régression sur X_{50} [13] ainsi que le prétraitement des données.

3.1 Description des données

Le tableau utilisé est un regroupement de deux tableaux donnant des informations sur la fragmentation de la roche en fonction de plusieurs paramètres, qui peuvent être catégorisés comme géométriques, explosifs et rocheux (cf. Figure 1) [13]. Les paramètres géométriques incluent :

B (m) *Burden* : La distance d'une rangée d'explosifs par rapport à la face de l'excavation et la distance entre l'excavation crée par une rangée par rapport à la suivante lorsque celles-ci sont déclenchées de manière séquentielle (ce qui est le cas ici).

S (m) *Spacing* : L'espacement entre les rangées.

H (m) *Hole Depth* : La profondeur des trous.

T (m) *Stemming* : La différence de hauteur entre la profondeur du trou et la hauteur de la colonne d'explosifs insérés dans le trou.

D (m) *Hole Diameter* : Le diamètre des trous.

Il n'y a qu'un seul paramètre d'explosion : P_f ($\mathrm{kg\,m^{-3}}$) : le facteur de poudre. Il montre la distribution des explosifs dans la roche.

Les paramètres de roche sont les suivants :

E (GPa) le module d'Young de la roche, représentant les propriétés de la roche.

X_b (m) : la taille du rocher *in situ*.

Nous remarquons que le tableau possède 8 variables quantitatives :

S/B : Le ratio de l'espacement par rapport au *burden*.

H/B : Le ratio de la profondeur des trous par rapport au *burden*.

B/D : Le ratio du *burden* par rapport au diamètre du trou.

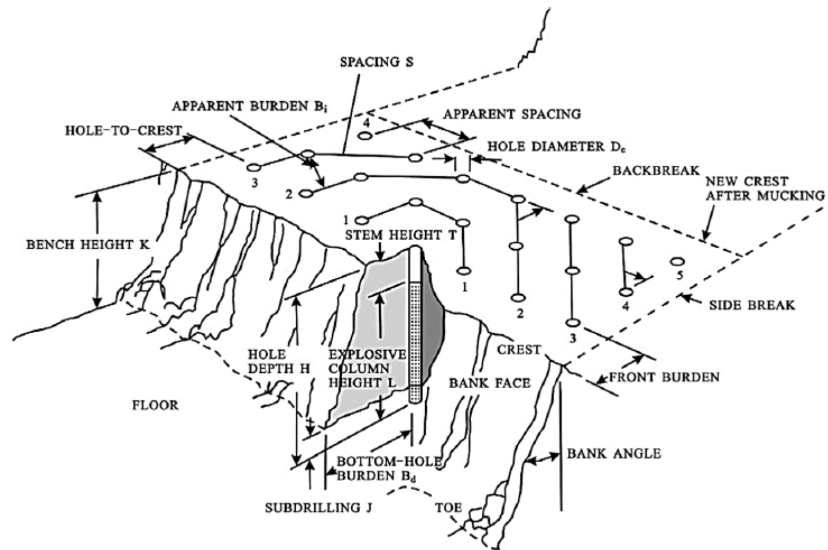


FIGURE 3.1 – Schéma représentant l'ensemble des paramètres géométriques propre à une explosion de roche. Dans le cadre de notre régression, nous n'utiliserons pas l'ensemble des paramètres de la figure, mais uniquement ceux énumérés précédemment [1].

T/B : Le ratio du *stemming* par rapport au *burden.*

P_f : Le facteur de poudre.

X_b : La taille du rocher *in situ.*

E : Le module d'Young de la roche.

X_{50} : la taille médiane des fragments

Ainsi, X_{50} constituera la sortie, et l'ensemble des autres variables constitueront l'entrée de notre problème de régression.

Par ailleurs, notre jeu de données possède également une variable qualitative, indiquant sur quel site a eu lieu l'explosion de l'observation. Nous évoquerons plus tard la manière dont cette variable qualitative a été encodée et si cette dernière s'est finalement avérée utile ou non.

3.1.1 Informations statistiques de base

Nos données comportent 104 observations. Cette faible quantité de données constitue un défi majeur de ce projet de recherche. Il est en effet difficile pour beaucoup de modèles de *machine learning* d'être précis avec aussi peu de données. Par ailleurs, l'influence des valeurs aberrantes (*outliers*) sera d'autant plus forte.

Nous donnons ici quelques informations statistiques élémentaires (moyenne, minimum, maximum, quartiles).

Var.	Moy.	Std.	min	25%	50%	75%	max
S/B	1.1890	0.1167	1.00	1.14	1.20	1.25	1.75
H/B	3.3452	1.6336	1.33	1.83	2.83	4.75	6.82
B/D	27.355	4.8385	17.98	24.72	27.27	30.30	39.47
T/B	1.2628	0.6738	0.50	0.83	1.14	1.40	4.67
P_f	0.5292	0.2354	0.22	0.35	0.48	0.66	1.26
X_b	1.1067	0.5322	0.02	0.73	1.03	1.56	2.35
E	29.461	17.878	9.57	15.00	16.90	45.00	60.00
X_{50}	0.3026	0.1883	0.02	0.16	0.23	0.40	0.96

L'ensemble des données tabulaires sont numériques. De plus, nous remarquons que les données sont relativement hétérogènes. On envisagera de les normaliser par la suite.

La figure 3.2 illustre la distribution conjointe des variables. On observe qu'il n'y a pas de corrélation linéaire particulière entre chacune des variables, elles semblent donc indépendantes. Ainsi, chaque variable semble avoir son importance dans l'estimation de X_{50}, une réduction de variables nous ferait perdre de l'information et donc de la précision sur l'estimation voulue. Par ailleurs, on constate que le lien entre notre sortie X_{50} est nos variables en entrée est hautement non linéaire.

3.2 Visualisation des données

Dans cette section nous essaierons de visualiser les données dans un plan afin d'observer si des *clusters* (amas) se manifestent. En effet, notre jeu de données contient des explosions provenant de plusieurs sites à travers le monde. On s'attendrait donc à voir des regroupements en fonction des sites.

Nos données quantitatives sont de dimension 8. Afin de pouvoir les afficher correctement dans un plan, il est donc nécessaire d'utiliser des méthodes de réduction de dimension. Nous avons donc envisagé d'utiliser l'algorithme de **t-SNE** (*t-Stochastic Neighbour Embedding*), qui est une méthode de réduction de dimension non-linéaire, contrairement à l'analyse des composantes principales. En effet, une méthode de réduction de dimension non-linéaire est nécessaire ici, au vu de la non-linéarité importante des données (cf. distribution conjointe des variables).

3.2.1 L'algorithme SNE

Avant de parler du t-SNE, décrivons ce qu'est une implantation stochastique de voisins (SNE ou *Stochastic Neighbour Embedding*) [12]. Un SNE commence par convertir la distance euclidienne entre deux points dans un espace à grande dimension en une probabilité conditionnelle qui présente des « similarités ». La « similarité » du point x_i par rapport au point x_j est modélisée par la probabilité conditionnelle $p_{j|i}$, que x_i accepterait x_j comme voisin (en supposant que les voisins sont choisis proportionnellement à la densité de probabilité d'une loi gaussienne centrée en x_i). Il vient alors que $p_{j|i}$ est élevé pour des points proches, et epsilonnesque pour des points éloignés. On pose :

$$p_{j|i} = \frac{\exp(-\|x_i - x_j\|^2/2\sigma_i^2)}{\sum_{k \neq i} \exp(-\|x_i - x_k\|^2/2\sigma_i^2)}. \tag{3.1}$$

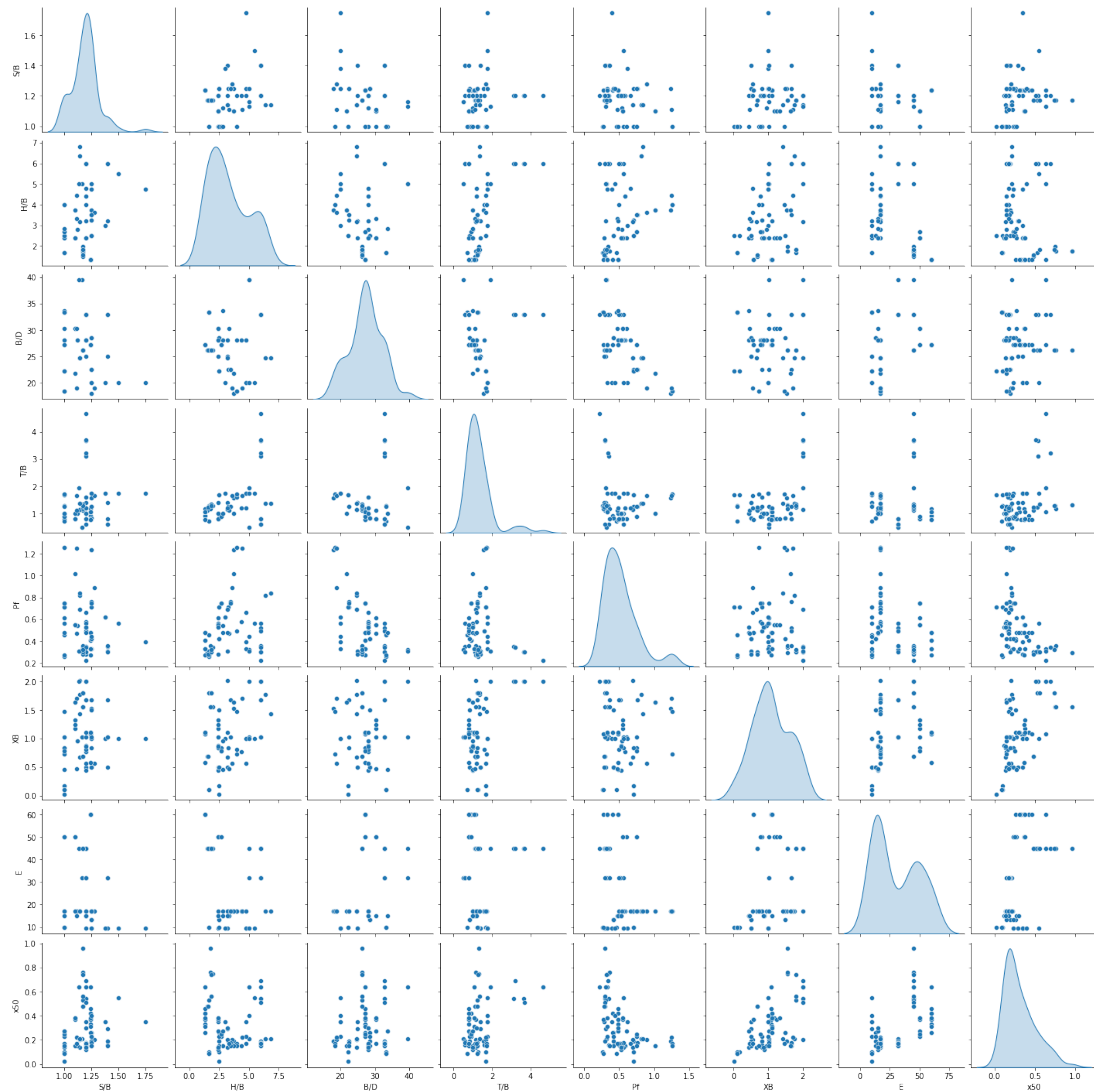


FIGURE 3.2 – Distribution conjointe des variables. Les figures sur la diagonale sont les estimations de densité de chacun des variables. On constate qu'il n'y a pas de corrélation évidente entre l'ensemble des variables.

Il s'agit en fait d'une distribution gaussienne sphérique normalisée, où σ_i est la variance de la gaussienne centrée en x_i. Comme nous ne sommes uniquement intéressés par les valeurs pour deux points distincts, nous posons $p_{i|i} = 0$.

Soit y_i et y_j les points représentant respectivement x_i et x_j dans l'espace à dimension faible. Nous pouvons de la même manière qu'avant définir une probabilité conditionnelle dans cet espace plus faible, notée $q_{i|j}$. En posant la variance de la gaussienne centrée en y_i égale à $\frac{1}{\sqrt{2}}$, nous pouvons ainsi modéliser la "similarité" du point y_j au point y_i par :

$$q_{j|i} = \frac{\exp(-\|y_i - y_j\|^2)}{\sum_{k \neq i} \exp(-\|y_i - y_k\|^2)} \tag{3.2}$$

qui s'agit encore une fois d'une distribution gaussienne sphérique, de variance $\frac{1}{\sqrt{2}}$. De la même manière qu'avant, nous posons $q_{i|i} = 0$ car nous ne sommes uniquement intéressés par les valeurs pour deux points distincts.

Si les points y_i et y_j représentent correctement les similarités entre les deux points x_i et x_j d'un espace à haute dimension, alors les probabilités conditionnelles $p_{i|j}$ et $q_{i|j}$ seront égales. L'algorithme SNE cherche alors à trouver un espace à faible dimension qui minimise les non-correspondances entre $p_{i|j}$ et $q_{i|j}$. Pour ce faire, on utilise la divergence de Kullback-Leibler.

Définition 1. *La divergence de Kullback-Leibler est une mesure de dissimilarité entre deux distributions de probabilités. Pour des distributions P et Q continues, on a :*

$$D_{\mathrm{KL}}(P\|Q) = \int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)}\, dx \tag{3.3}$$

où p et q sont les densités respectives de P et Q.

En d'autres termes, la divergence de Kullback-Leibler est l'espérance de la différence des logarithmes de P et Q, en prenant la probabilité P pour calculer l'espérance.

La SNE minimise alors la somme des divergences de Kullback-Leibler sur l'ensemble des points du jeu de données en utilisant une méthode de descente de gradient. La fonction de coût notée C est donnée par :

$$C = \sum_i KL(P_i || Q_i) = \sum_i \sum_j p_{j|i} \log \frac{p_{j|i}}{q_{j|i}} \tag{3.4}$$

dans laquelle P_i représente la distribution de la probabilité conditionnelle sur l'ensemble des points du jeu de données hormis x_i, sachant x_i. De la même manière, Q_i représente la distribution de la probabilité conditionnelle sur l'ensemble des points du jeu de données hormis y_i, sachant y_i.

Le paramètre restant à trouver est la variance σ_i de la loi gaussienne centrée en x_i. Il est peut probable qu'une seule valeur σ soit optimale pour tout i, car la densité du jeu de données peut varier. Dans des régions denses, une valeur basse de σ_i est généralement plus adéquate que dans des régions plus éparses. Chaque valeur de σ_i génère une distribution de probabilité P_i sur l'ensemble des données. Cette distribution possède une entropie croissante selon σ_i. L'algorithme SNE fait une recherche binaire pour la valeur de σ_i qui produit un P_i avec une *perplexité* fixe définie par l'utilisateur. La perplexité est définie comme :

$$\mathrm{Perp}(P_i) = 2^{H(P_i)}, \tag{3.5}$$

où $H(P_i)$ est l'entropie de Shannon de P_i mesurée en bits

$$H(P_i) = -\sum_j p_{j|i} \log_2 p_{j|i}. \tag{3.6}$$

La perplexité est un hyperparamètre de l'algorithme qui contrôle le nombre de voisins considérés pour chaque point lors de la construction de la représentation visuelle. En d'autres termes, la perplexité dans ce contexte détermine le nombre de voisins les plus proches que chaque point doit considérer pour déterminer sa position dans l'espace de visualisation.

Une perplexité élevée signifie que chaque point doit considérer un grand nombre de voisins, ce qui peut conduire à des regroupements plus lisses et globaux de points dans l'espace de visualisation. En revanche, une perplexité faible signifie que chaque point ne considère que quelques voisins proches, ce qui peut conduire à des regroupements plus fins et localisés. Le choix de la perplexité est donc crucial pour obtenir une représentation visuelle optimale de données avec l'algorithme de t-SNE.

La minimisation de la fonction de coût de l'équation précédente est calculée en utilisant une méthode de descente de gradient. Le gradient a la forme suivante ;

$$\frac{\partial C}{\partial y_i} = 2 \sum_j (p_{j|i} - q_{j|i} + p_{i|j} - q_{i|j})(y_i - y_j). \tag{3.7}$$

En faisant l'analogie avec la physique, le gradient peut être interprété comme la force résultante créée par un ensemble de ressorts entre le point y_i et tous les autres points de l'espace réduit y_j. Tous les ressorts exercent une force selon la direction $(y_i - y_j)$. Le ressort entre le point y_i et y_j repousse ou attire les points de l'espace réduit selon la distance entre ces derniers. La force exercée par le ressort entre y_i et y_j est également proportionnelle à son coefficient de rigidité, représenté par le décalage $(p_{j|i} - q_{j|i} + p_{i|j} - q_{i|j})$ entre les similarités de l'espace à haute dimension et de l'espace réduit.

3.2.2 L'algorithme t-SNE

Malgré le fait que le SNE construit de bonnes visualisations, ces dernières sont gênées par le fait que la fonction de coût est difficile à optimiser et par un problème nommé l'effet de couronnement. L'algorithme de t-SNE chercher à pallier ces deux problèmes.

L'algorithme SNE symétrique

Une alternative à la minimisation de la somme des divergences de Kullback-Leibler entre les probabilités conditionnelles $p_{j|i}$ et $q_{j|i}$ serait de minimiser une divergence de Kullback-Leibler entre une loi jointe P dans l'espace à haute dimension et une autre loi jointe Q dans l'espace à faible dimension :

$$C = KL(P||Q) = \sum_{i \neq j} p_{ij} \ln \frac{p_{ij}}{q_{ij}}. \tag{3.8}$$

où encore une fois, $p_{ii} = 0,\ q_{ii} = 0,\ \forall i$. Ce type de SNE est nommé SNE symétrique de par le fait que $p_{ij} = p_{ji}$ et $q_{ij} = q_{ji} \forall i, j$. On pose les similarités entre deux points dans l'espace à faible dimension q_{ij} comme étant égal à :

$$q_{ij} = \frac{\exp(-\|y_i - y_j\|^2}{\sum_{k \neq l} \exp(-\|y_k - y_l\|^2)}. \tag{3.9}$$

Et on va poser les similarités entre deux éléments dans l'espace à grande dimension comme étant égal à :

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2n}. \tag{3.10}$$

L'avantage principal du SNE symétrique est que son gradient peut s'écrire sous une forme plus simple, ce qui rend son calcul plus rapide. Son gradient est relativement similaire à celui du SNE :

$$\frac{\partial C}{\partial y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_{-} y_j). \tag{3.11}$$

Le SNE symmétrique produit des visualisations tout aussi bonnes voire meilleures que le SNE classique.

L'effet de couronnement

L'effet de couronnement est un phénomène qui peut se produire dans le cadre du t-SNE ou du SNE.

En résumé, l'effet de couronnement se produit lorsque la densité de points est élevée dans une région particulière de l'espace de données d'origine, ce qui entraîne une superposition de points dans l'espace de visualisation. Cela peut rendre difficile ou impossible la distinction entre les points individuels, car ils sont trop proches les uns des autres. Cela peut entraîner une distorsion de la structure des données d'origine, ce qui peut rendre difficile l'interprétation des résultats de la visualisation.

Le t-SNE tente de résoudre ces problèmes en ajoutant un faible terme de répulsion au « ressort » dans la fonction de coût en créant un modèle de fond uniforme avec une faible proportion de mélange notée ρ. Donc, peu importe à quel point deux points dans l'espace réduit sont éloignés, q_{ij} ne sera jamais inférieur à $\frac{2\rho}{n(n-1)}$ (car la distribution du fond uniforme est sur un ensemble de $n(n-1)/2$ paires). Ainsi, pour les points qui sont loin dans l'espace à haute dimension, q_{ij} sera toujours plus grand que p_{ij}, menant ainsi à une légère répulsion. On écrit donc maintenant q_{ij} sous la forme :

$$q_{ij} = \frac{(1-\rho)\exp(-\|y_i - y_j\|^2)}{\sum_{k<l} \exp(-\|y_k - y_l\|^2)} + \frac{2\rho}{n(n-1)}. \tag{3.12}$$

Cette technique est appelée UNI-SNE et bien qu'elle soit généralement plus performante que le SNE standard, l'optimisation de la fonction de coût UNI-SNE est fastidieuse.

Étant donné que le SNE symétrique correspond en réalité aux probabilités conjointes de paires de points de données dans les espaces de haute et de basse dimensions plutôt qu'à leurs distances, nous avons un moyen naturel de diminuer l'effet de couronnement qui fonctionne comme suit.

Dans l'espace de haute dimension, nous convertissons les distances en probabilités en utilisant une distribution gaussienne. Dans la carte de basse dimension, nous pouvons utiliser une distribution de probabilité qui a des queues beaucoup plus lourdes qu'une distribution gaussienne pour convertir les distances en probabilités. Cela permet à une distance modérée dans l'espace de haute dimension d'être fidèlement modélisée par une distance beaucoup plus grande dans la carte et, par conséquent, cela élimine les forces attractives indésirables entre les points de la carte qui représentent des points de données modérément dissemblables. Dans l'algorithme t-SNE, nous utilisons une distribution de Student avec un degré de liberté (qui est identique à une distribution de Cauchy) comme distribution à queues lourdes dans la carte de basse dimension. En utilisant cette distribution, les probabilités jointes q_{ij} sont définies comme :

$$q_{ij} = \frac{(1 + \|y_i - y_j\|^2)^{-1}}{\sum_{k \neq l}(1 + \|y_k - y_l\|^2)^{-1}}. \tag{3.13}$$

Une justification théorique pour le choix de la loi de Student est de stipuler qu'elle est étroitement reliée à la distribution gaussienne, étant donné que la loi de Student est un mélange de lois normales. Un avantage de la loi de Student est qu'il est bien plus rapide d'évaluer la densité d'un point sous une loi de Student que sous une loi normale car elle ne contient pas d'exponentielle (malgré le fait qu'elle est équivalente à un mélange infini de loi normales).

Le gradient de la divergence de Kullback-Leibler entre P et la probabilité jointe de Q est donnée par :

$$\frac{\partial C}{\partial y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j) \left(1 + \|y_i - y_j\|^2\right)^{-1} \tag{3.14}$$

En conclusion, le t-SNE cherche à modéliser des points dissemblables avec de grandes distances et des points semblables avec de petites distances. De plus, l'optimisation de la fonction de coût du t-SNE est bien plus facile que celle du SNE ou du UNI-SNE. Plus particulièrement le t-SNE utilise des « forces à longue portée » dans l'espace réduit permettant de rassembler deux points ou ensembles de points semblables qui auraient été séparés dans les premiers stades de l'optimisation tandis que le SNE et le UNI-SNE ne possèdent pas de telles « forces ». Ces dernières doivent donc avoir recours à un recuit simulé pour obtenir de bonnes solutions tandis que le t-SNE facilite l'identification d'optima locaux sans avoir recours à un recuit simulé.

3.2.3 Méthodes d'optimisation pour l'algorithme t-SNE

Une méthode simple du t-SNE est ici présentée. Elle utilise une méthode de descente de gradient afin d'optimiser la fonction de coût du t-SNE. Cette procédure simple utilise un terme d'inertie afin de réduire le nombre d'itérations requis. Cette méthode marche bien lorsque le terme d'inertie est petit et jusqu'à ce que les points de la carte deviennent relativement organisés. Voici un algorithme pseudo-code expliquant cette démarche.

Pour plus de détails concernant la t-SNE, vous pouvez lire [16] étant l'article d'origine.

3.2.4 t-SNE sur les données

Nous pouvons *in fine* représenter nos données dans un plan à deux dimensions à l'aide du t-SNE. La figure 3.3 montre la représentation de nos données dans le plan. On trouve alors bel et bien des clusters caractérisés par le site d'où proviennent les explosions.

Algorithm 1 t-SNE

Données Ensemble de points : $\mathcal{X} = \{x_1, x_2, \dots, x_n\}$,
Paramètres de la fonction de coût : perplexité notée Perp
Paramètres d'optimisation : nombre d'itérations T, pas η, inertie $\alpha(t)$
Résultat : Représentation en dimension réduite : $\mathcal{Y}^{(T)} = \{y_1, y_2, \dots, y_n\}$.
Début :
Calculer les « similarités » $p_{i|j}$ à l'aide de la perplexité Perp. Poser $p_{ij} = \frac{p_{i|j}+p_{j|i}}{2n}$.
Initialiser $\mathcal{Y}^{(0)} = \{y_1, y_2, \dots, y_n\}$ en échantillonnant dans $\mathcal{N}(0, 10^{-4}I)$
for $t = 1 : T$ **do** :
  Calculer les "similarités" q_{ij}
  Calculer le gradient $\frac{\partial C}{\partial \mathcal{Y}}$.
  Poser $\mathcal{Y}^{(t)} = \mathcal{Y}^{(t-1)} + \eta \frac{\partial C}{\partial \mathcal{Y}} + \alpha(t) \left(\mathcal{Y}^{(t-1)} - \mathcal{Y}^{(t-2)}\right)$
end for
Fin

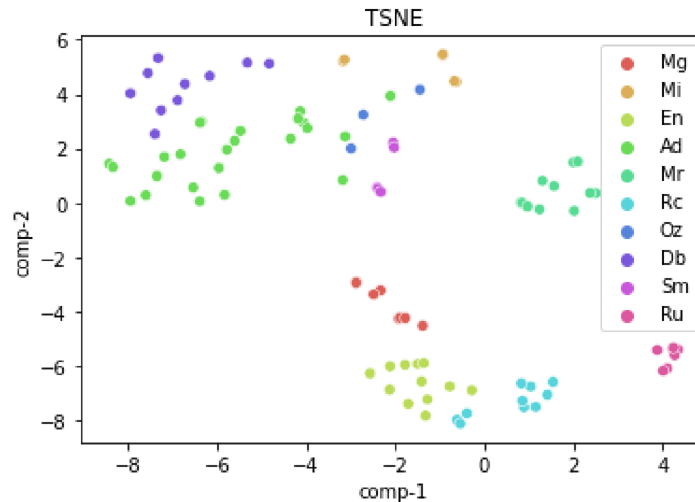


FIGURE 3.3 – Visualisation des données dans un plan à l'aide du t-SNE. La variable qualitative indiquant les sites d'où proviennent les explosions a permis de colorier les *clusters* en fonction de leur site (**mais cette variable qualitative n'a pas été prise en compte dans l'entrée du t-SNE**). On trouve alors bel et bien des *clusters* caractérisés par le site d'où proviennent les explosions.

3.3 Prétraitement des données

3.3.1 Suppression des *outliers*

Comme il était évoqué précédemment, les *outliers* peuvent avoir une influence majeure sur la construction de notre modèle. De plus, cette influence est d'autant plus grande dans le cas où nous avons peu de données. Nous appliquons ici une méthode simple pour supprimer des valeurs extrêmes à l'aide de diagrammes de boîte à moustache. Ainsi, pour chaque variable, si une observation est en dehors de la boîte à moustache (en d'autres termes, si sa valeur est supérieure au 3e quartile $+1,5$ fois l'écart interquartile ou inférieure au 3erquartile $-1,5$ fois l'écart interquartile) alors elle sera supprimée du jeu de données. De cette manière, nous avons pu supprimer 4 *outliers* de notre jeu de données, ce qui nous laisse donc 100 observations. De plus, cette suppression d'*outliers* n'a pas généré de nouveaux *outliers* ce qui nous évite de répéter le processus.

3.3.2 Transformation normale des données quantitatives :

Dans cette sous-section, nous allons voir des méthodes d'estimation de la densité de probabilité de nos variables, et nous allons les transformer de telle sorte à ce que chaque variable suit une loi normale. En effet, la plupart des modèles de *machine learning* sont plus performant lorsque nos variables suivent une distribution normale.

Estimation de densité

Dans le cas d'une variable aléatoire continue, nous sommes souvent intéressés par sa **densité de probabilité**. Par exemple, pour un échantillon aléatoire d'une variable, nous aimerions connaître la forme de la distribution de probabilité, ses valeurs mais aussi la manière dont les valeurs sont répartis etc. Connaître une distribution de probabilité pour une variable aléatoire peut être utile pour calculer la moyenne ou la variance, mais aussi pour déterminer si une observation est probable ou très peu probable.

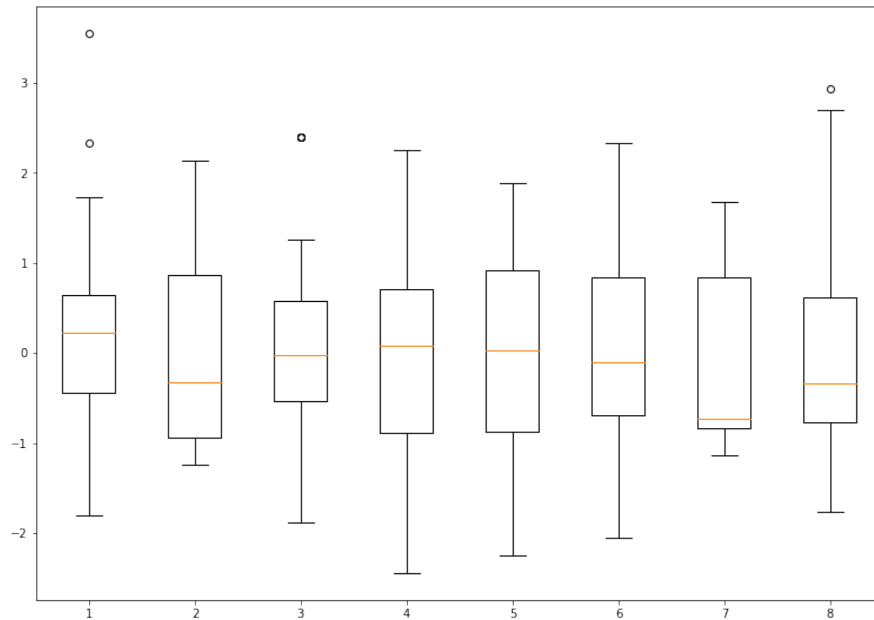
Le problème est que nous ne connaissons pas la distribution de probabilité d'une variable aléatoire, parce que nous n'avons pas accès à toutes ses valeurs possibles et ses probabilités associées. Seul un échantillon d'observations est connu. C'est pourquoi nous devons **sélectionner** une distribution de probabilité.

Ce problème se nomme **estimation de densité de probabilité** ou plus simplement estimation de densité, car nous utilisons les observations d'un ensemble d'exemples aléatoire pour estimer la densité globale de notre variable, au delà de seulement des exemples.

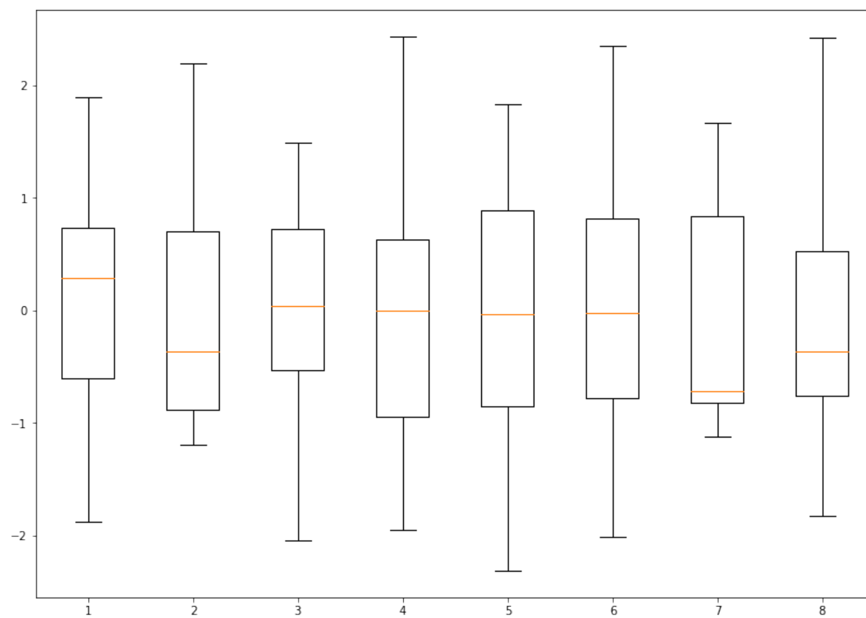
Généralement, le processus de l'estimation de densité se fait en plusieurs étapes. Tout d'abord, on regarde la densité des observations à l'aide d'un simple histogramme. A partir de l'histogramme, on pourrait peut être identifier une distribution de probabilité usuelle (loi normale, exponentielle, Poisson, etc.). Sinon, il va falloir utiliser d'autres méthodes, telles que l'estimation par noyau.

Estimation de densité paramétrique

Dans le cas où nous avons repéré à l'aide de l'histogramme une distribution de probabilité usuelle, on peut essayer d'adapter le modèle à la distribution de probabilité repérée en modifiant les paramètres de cette dernière. D'où le nom d'**estimation de densité paramétrique.**



(a) Boîtes à moustaches normalisées montrant la dispersion des valeurs pour chacune des variables quantitatives numérotées de 1 à 8. On constate que les variables 1, 3 et 8 (à savoir : B/D, S/B et X_{50}) possèdent des *outliers*, on va donc tenter de les supprimer.



(b) Boîtes à moustaches normalisées après suppression des *outliers*. On constate que les *outliers* des variables précédentes ont disparu.

Par exemple, dans le cas où l'on repère une loi normale à partir de notre histogramme, on sait que la loi normale dépend de deux paramètres : l'écart-type et la moyenne. On peut alors estimer ces paramètres à partir des observations. En posant alors $\hat{m}$ l'estimateur de la moyenne, $\hat{\sigma}$ l'estimateur de l'écart-type et $X = (x_1, \ldots, x_n)$ les n observations, on a :

$$\hat{m} = \frac{1}{n}\sum_{i=1}^{n} x_i, \tag{3.15}$$

$$\hat{\sigma} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{m})^2}. \tag{3.16}$$

Il est parfois possible que les données correspondent à une distribution de probabilité usuelle, mais doivent d'abord être transformées avant de faire une estimation.

par exemple, les données soient « déformées » par rapport à la densité de probabilité usuelle estimée. Dans ce cas, il serait peut-être nécessaire de transformer les données avant d'estimer les paramètres, en leur appliquant le logarithme, la racine carrée ou plus généralement une transformation de Box-Cox (qui optimise automatiquement la transformation des données auxquelles sont appliquées le logarithme ou la racine carrée).

Ces types de transformation ne sont pas toujours évidentes et une estimation de densité paramétrique performante peut nécessiter le processus itératif suivant :

1. Estimer les paramètres de la distribution,
2. Comparer la densité de probabilité obtenue avec les données,
3. Transformer les données de telle sorte à ce qu'ils s'adaptent mieux à la distribution,
4. Répéter ces opérations jusqu'à ce que la distribution soit bien adaptée au modèle.

Estimation de densité sur X_{50}

L'estimation de densité de X_{50} a été effectuée sur le logiciel `R`. Un histogramme de X_{50} a d'abord été réalisé afin d'observer sa tendance globale (cf. Figure 4.3). A l'aide de ce dernier, nous remarquons que la distribution de X_{50} semble être unimodale (il n'y a qu'une seule "bosse"). L'histogramme ressemble à peu près à une loi log-normale. Nous allons donc procéder à une estimation de densité sur les valeurs de $\ln(X_{50})$. La figure 3.3 montre que le logarithme de X_{50} est proche d'une loi normale.

Ainsi, nous allons transformer les données de X_{50} en lui appliquant le logarithme afin que notre variable suive une loi normale. Afin de faciliter les calculs en ayant uniquement des données positives, nous appliquerons plutôt à X_{50} la fonction $x \to \ln(x+1)$, qui suit également une loi log-normale.

Transformation des autres variables

Afin que les autres variables, qui constituent l'entrée, suivent également une loi normale, nous allons leur appliquer une transformation Box-Cox. Cette dernière arrive automatiquement à approcher la loi de chaque variable par une loi normale.

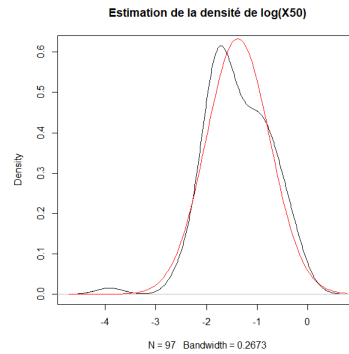


FIGURE 3.5 – Estimation de densité de $\ln(X_{50})$ en noir et loi normale de paramètres (moyenne(X_{50}), écart-type(X_{50})) en rouge. On constate que la densité de probabilité de $\ln(X_{50})$ est proche d'une loi normale.

3.3.3 Normalisation des données

La question de normalisation des données semble également légitime. En effet, certains modèles de *machine learning* nécessitent une normalisation des données pour assurer leur bon fonctionnement (régression linéaire, régression à vecteurs supports, réseaux de neurones, etc.), notamment pour résoudre des problèmes de gradient qui explosent dans l'optimisation de la fonction de coût. Néanmoins, il existe également d'autres modèles (arbres de décision par exemple) qui ne nécessitent pas de normalisation. Ainsi, en fonction du modèle, les données seront normalisées ou non.

Dans le cas où nous utiliserons des modèles qui nécessitent une normalisation des données, nous utiliserons la fonction `RobustScaler` pour effectuer cette dernière, car cette fonction est robuste aux *outliers*. `RobustScaler` consiste simplement à transformer les données de telle sorte à ce que la médiane soit égale à 0 et à ce que les 1eret 3^{e}quartiles soient respectivement égaux à -1 et 1. Ainsi, chacune de nos variables devrait se rapprocher d'une loi normale centrée réduite.

Chapitre 4

Modèles d'apprentissage automatique utilisés

4.1 Régression Linéaire et Polynomiale

Soit :
n le nombre d'observations,
p le nombre de variables
$X \in \mathcal{M}_{n,p}(\mathbb{R})$ le jeu de données en entrée,
$Y \in \mathcal{M}_n(\mathbb{R})$ la variable à déterminer (sortie).
Notons ici que les variables doivent toutes être **quantitatives**.

Le principe de la régression linéaire est alors de supposer que la sortie Y s'exprime comme une combinaison linéaire affine des variables de l'entrée X : en posant $X' = \begin{bmatrix} 1 & \dots & 1 \\ & X & \\ & & \end{bmatrix}$ alors,

$$Y = \omega X' + \varepsilon \tag{4.1}$$

avec $\omega \in \mathcal{M}_{p+1,p+1}$ la matrice des paramètres et $\varepsilon \in \mathcal{M}_{p+1}$ une matrice colonne représentant un bruit.

L'estimateur de Y est donc représenté par $\hat{Y} = \omega X$. On appelle **résidus** l'erreur entre l'estimateur et Y défini par $Y - \hat{Y}$.

Pour être un bon modèle, la régression linéaire doit respecter les critères d'**homoscédasticité**, c'est-à-dire que les résidus doivent être d'espérance nulle et de variance constante. De plus, dans le cadre de prévision (qui est notre cas ici), les résidus doivent également suivre une loi normale. Cela revient à dire que :

$$\varepsilon \sim \mathcal{N}(0, \sigma^2 I_p) \tag{4.2}$$

avec $\sigma \in \mathbb{R}$.

Pour trouver les valeurs optimales des paramètres (les coefficients de ω), on va chercher lors de l'entraînement à minimiser l'écart entre l'estimation de Y de notre modèle, et les vraies valeurs de Y, à savoir $\|Y - \omega X'\|_2^2$. On appelle ce processus de minimisation le **critère des moindres carrés** ou **mean squared error** (*MSE*) en anglais. On pourrait penser à utiliser d'autres critères, comme par exemple remplacer la norme L^2 par la norme L^1, mais ceci pose des problèmes de différentiabilité en 0 et pourrait ainsi poser problème lors du processus de minimisation, qui se fait souvent par descente de gradient.

On peut par ailleurs généraliser la définition de la régression linéaire à un polynôme. Avec les notations précédentes :

$$Y = \sum_{i=0}^{k} \omega_i X'^i + \varepsilon \tag{4.3}$$

avec $k \in \mathbb{N}$, $\omega_i \in \mathcal{M}_{p+1,p+1} \forall i \in \{0, \ldots, k\}$ les matrices de paramètres, X'^i la matrice X' dont chaque coefficient est élevé à la puissance i et $\varepsilon \in \mathcal{M}_{p+1}$ la matrice colonne représentant le bruit qui respecte encore une fois les conditions d'homoscédasticité.

Cette fois-ci, le critère des moindres carrés s'écrira : $\|Y - \sum_{i=0}^{k} \omega_i X'^i\|_2^2$. Pour simplifier dans la suite de ce chapitre, nous noterons l'erreur des moindres carrés MSE_{train}.

4.2 Régression Ridge

Il arrive parfois que notre modèle soit en sur-apprentissage, c'est-à-dire que notre modèle apprend par coeur l'ensemble des points de la base d'entraînement au lieu de comprendre le lien entre notre entrée et notre sortie. Cela se traduit par une erreur sur la base d'entraînement très basse, et une erreur sur la base de test (ou base de validation) élevée. On peut contrer le sur-apprentissage en contraignant les paramètres du modèle à ne pas prendre des valeurs trop élevées. En d'autres termes, on cherche à régulariser le modèle [15].

Dans le cadre de la régression, au lieu d'estimer les paramètres y_i en minimisant l'erreur quadratique moyenne, on peut chercher à minimiser (en reprenant les mêmes notations que dans la section 2.5) :

$$MSE_{train} + \lambda \|\omega\|_2^2, \tag{4.4}$$

où λ est un hyperparamètre positif.

On voit apparaître un compromis entre l'adéquation aux données (mesurée par la *MSE*, premier terme de l'expression) et la valeur des poids. On nomme le second terme dégradation de poids ou *weight decay* [11].

Cette méthode se nomme régression *ridge*.

4.3 Régression lasso et ElasticNet

4.3.1 Lasso

Dans la régression *ridge*, nous avons ajouté un terme de régularisation dans L^2 à la fonction de coût. La régression lasso, elle, rajoute un terme de pénalisation dans L^1. On va donc chercher à

minimiser :

$$MSE_{train} + \lambda\|\omega\|_1 \tag{4.5}$$

avec λ un hyperparamètre.

Le fait que l'on utilise la norme L^1 peut poser des problèmes lors de la minimisation, car la valeur absolue est non différentiable.

4.3.2 ElasticNet

La régression *ElasticNet* consiste à combiner les deux approches de régularisation *ridge* et lasso. On va donc minimiser lors de l'entraînement :

$$MSE_{train} + \lambda_1\|\omega\|_2^2 + \lambda_2\|\omega\|_1 \tag{4.6}$$

avec λ_1, λ_2 des hyperparamètres.

4.4 Réseaux de Neurones Profonds

Un **réseau de neurones profond** est un algorithme représenté sous la forme d'un réseau, contenant des couches. Chacune de ces couches contient un certain nombre de neurones. Afin d'éviter de surcharger le rapport, l'explication des réseaux de neurones restera basique.

4.4.1 Les neurones

Un **neurone** est une fonction qui retient un nombre réel. On appelle ce nombre l'activation. En fonction de la valeur de ce-dernier, un neurone va décider de transmettre ou de ne pas transmettre l'information aux neurones de la couche suivante.

Par exemple dans notre contexte, les neurones de la couche d'entrée vont contenir les nombres de chaque variable pour une observation donnée. Il n'y a qu'un seul neurone en couche de sortie, car la tâche de notre modèle est la régression. En effet, on veut que notre algorithme ait en sortie un scalaire, donc un seul neurone.

4.4.2 Les couches

Les couches sont divisées en trois sous-catégories. Il y a la couche d'entrée, la couche de sortie et les couches cachées, contenant chacune des neurones. Les couches cachées vont essayer de déterminer des compositions des valeurs associées aux variables. Plus on a de couches, plus les compositions sont divisés en sous catégories. La première couche cachée contient alors les éléments primaires. A partir de ces derniers, la couche cachée suivante va essayer d'assembler ces éléments primaires pour former des catégories. La couche suivante va former des catégories de ces catégories en les assemblant etc.

4.4.3 Les poids

Pour que notre réseau de neurones puisse apprendre au fur et à mesure des observations, on va utiliser des **poids** que l'on va assigner à chaque connexion de chaque neurone de la deuxième couche a avec la première couche. On caractérise ce réseau de neurones comme dense. Ces poids sont des nombres positifs comme négatifs. On va donc multiplier à chaque activation leur poids et sommer le tout. Toutefois, on veut parfois qu'un neurone s'active uniquement à partir d'un certain seuil. C'est pourquoi on peut soustraire à cette somme une constante qui se nomme **biais**. On va également appliquer à cette somme une certaine fonction, afin de pouvoir mieux contrôler l'activation de chacun des neurones. Cette fonction se nomme **optimiseur**. Ainsi pour un neurone b_i appartenant à une certaine couche, supposant que la couche précédente possède n neurones dont leur activation est notée $(a_1, \ldots, a_n)$, que la constante de biais est notée *bias* et que la fonction appliquée se nomme σ, il vient que :

$$b_i = \sigma(\sum_{k=1}^{n} \omega_i^a a_i - bias), \quad (4.7)$$

avec $\omega^a \in \mathbb{R}^n$ le vecteur poids correspondant à aux neurones de la couche précédant celle de b_i. Ainsi, chaque neurone de la j^{e}couche a des poids différents associés à sa connexion avec les neurones de la $j-1$-ème couche et un biais différent. On remarque alors que l'on a un processus récursif, car l'activation des neurones de la $j-1^{\text{e}}$couche dépend elle même des poids, biais et activations de la $j-2^{\text{e}}$couche. On va donc parcourir l'ensemble du réseau de neurones à l'envers. On nomme ce processus la **rétropropagation**.

Ainsi, lorsque l'algorithme « apprend » à travers les données que l'on lui fournit, il modifie l'ensemble des poids et des biais du réseau. Ainsi, plus il y a de données, mieux l'algorithme sera entraîné. Cependant, il faut conserver une partie des données pour tester le modèle et évaluer sa performance. Avant l'entraînement, le modèle met en place des poids totalement aléatoires et va améliorer sa performance sur les données d'entraînement au fur et à mesure des observations.

4.4.4 La fonction de coût

Pour que le modèle puisse évaluer la qualité de sa prédiction, on va introduire une fonction de coût, qui va mesurer la performance du modèle en comparant l'activation du neurone de la couche de sortie avec la vraie valeur à trouver. Ainsi, plus la fonction de coût est grande, plus le modèle est imprécis et vice-versa. La fonction de coût aura une valeur différente pour chaque observation de test. On va donc chercher à minimiser la moyenne des fonctions de coût. Pour ce faire, l'algorithme va chercher un minimum local par descente de gradient.

4.4.5 Epoque

Une époque (*epoch*) correspond à un apprentissage sur l'ensemble des observations. Lorsqu'une époque est finie, le réseau de neurones a donc ajusté les poids, les biais et les activations de chaque neurone en ayant balayé l'ensemble des observations. Une époque ne suffit généralement pas à avoir un modèle performant. On peut ainsi demander au modèle de repasser plusieurs fois sur les observations pour continuer à réajuster l'ensemble des paramètres.

4.5 Régression à vecteurs suppots

La régression à vecteurs supports (SVR) est un modèle de *machine learning* destiné à une tâche de régression. Il fait partie de la catégorie des machines à vecteurs supports (SVM). Afin de pouvoir expliquer comment les SVR fonctionnent, il est nécessaire d'expliquer au préalable la manière dont les machines à vecteurs supports ont été créés dans le cadre d'une classification simple. En effet, les machines à vecteurs supports sont dans l'ensemble une généralisation des classificateurs linéaires.

4.5.1 Machine à vecteurs supports

Les machines à vecteurs supports sont expliquées plus en détail dans l'ouvrage [2].

Prenons le cas d'une tâche de classification simple : prenons des observations $x_i \in \mathbb{R}^p,\ \forall i \in [1, n]$ et une sortie associée à chacune des observations $y_i \in \{-1, 1\},\ \forall i \in [1, n]$. On cherche donc à prédire à laquelle des deux classes x_i appartient. Le SVM cherche un hyperplan qui permet de séparer les x_i associées à 1 d'un côté et les x_i associées à -1 de l'autre.

Évidemment il n'existe pas toujours un hyperplan (voire même quasiment jamais dans des cas pratiques) qui permet de séparer les y_i de cette manière. **Supposons dans un premier temps que cet hyperplan existe bel et bien**. L'hyperplan va se caractériser de la façon suivante :

$$h(x) = \langle w, x \rangle + w_0 = w^\top x + w_0 = 0, \tag{4.8}$$

avec :

$w = (w_1, w_2, \ldots, w_p)^\top \in \mathbb{R}^p$ le vecteur poids,

$x \in \mathbb{R}^p$ l'observation,

$w_0 \in \mathbb{R}$ le vecteur poids à l'origine.

Ainsi, si $h(x_i) > 0$, on suppose que x_i est associée à la classe 1 et si $h(x_i) < 0$, on suppose que x_i est associée à la classe -1. **L'hyperplan optimal** est celui qui va réussir à maximiser **la marge.** On définit la marge maximale par cette équation :

$$\begin{cases} \min\limits_{w, w_0} \frac{1}{2}\|w\|^2, \\ y_i(w^\top x_i + w_0) \geq 1, i \in \{1, \ldots, n\}. \end{cases} \tag{4.9}$$

La maximisation de cette marge se résout à l'aide du théorème de Karush-Kuhn-Tucker. On obtient finalement que le vecteur de poids optimal w^* s'écrit comme :

$$w^* = \sum_{i=1}^{n} \alpha_i y_i x_i,\ \alpha_i \in \mathbb{R}. \tag{4.10}$$

C'est-à-dire, une combinaison linéaire de points se trouvant sur la marge $(y(w^\top x + w_0) = 1$. On nomme ces points les **vecteurs supports**. Ces derniers sont les seuls points qui participent à la caractérisation de l'hyperplan optimal.

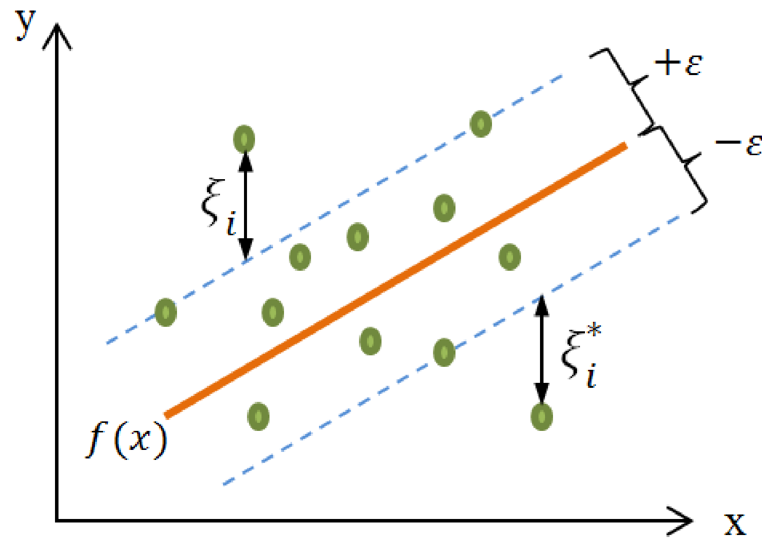


FIGURE 4.1 – Représente l'ensemble des observations x_i associées soit à 1 (ici □) soit à -1 (ici o). L'hyperplan optimal est ici tracé en trait plein et les marges maximales de chaque côté en pointillés. Les vecteurs supports sont les échantillons en noir sur la marge [14].

Examinons maintenant le cas le plus fréquent où il n'existe pas d'hyperplans permettant de séparer en deux les observations de cette manière. On va donc alors introduire des **variables ressorts** ou *slacks variables*. Qui vont permettre à notre modèle de dire que certains points sont mal classés ou bien classés mais à l'intérieur de la marge. Posons les variables ressorts $\varepsilon = (\varepsilon_1, \ldots, \varepsilon_n)$ telles que :

$\varepsilon_i \in]0, 1] \implies i$ est bien classée mais dans la marge.

$\varepsilon_i > 1 \implies i$ est mal classée.

$\varepsilon_i = 0 \implies i$ est bien classée, sur la marge ou au dessus.

On cherche donc maintenant à minimiser selon w, w_0, ε_i

$$\begin{cases} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n} \varepsilon_i, \\ y_i(w^\top x_i + w_0) \geq 1 - \varepsilon_i, \\ \varepsilon_i \geq 0, i \in \{1, \ldots, n\}, \end{cases} \tag{4.11}$$

avec C à définir par l'utilisateur.

Encore une fois, le poids w^* solution de ce problème d'optimisation s'écrit comme :

$$w^* = \sum_{i=1}^{n} \alpha_i y_i x_i, \; \alpha_i \in \mathbb{R}. \tag{4.12}$$

Désormais, les vecteurs supports qui sont les vecteurs solutions du minimum sont soit sur la marge $(\varepsilon_i) = 0$ ou en-dessous de la marge ($\varepsilon_i > 0$ et $\alpha_i = C$).

L'hyperparamètre C influence grandement l'efficacité du modèle [15] :

Lorsque C diminue, la marge augmente et donc les ε_i augmentent. Ce qui fait que beaucoup d'observations ne sont pas bien classées.

Lorsque C augmente, les ε_i diminuent et la marge également, ce qui fait que plus de points seront bien classés sur les données d'entraînement mais on risque un sur-apprentissage.

On a depuis le début supposé que les observations peuvent être linéairement séparables. Évidemment, ce n'est que très rarement le cas sur le terrain. On utilise alors souvent une astuce nommée *kernel trick* ou astuce du noyau. Cette dernière consiste à changer l'espace de représentation des données, et d'appliquer une machine de vecteurs supports linéaire dans cet espace. En reprenant notre poids optimal et en l'insérant dans l'équation de l'hyperplan, on a :

$$h(x) = \sum_{i=1}^{n} \alpha_i y_i x_i^\top x + w_0. \tag{4.13}$$

L'astuce du noyau consiste à prendre une fonction noyau $K : \mathbb{R}^{2p} \to \mathbb{R}$ telle que $K(x, y) = \phi(x)^\top \phi(y)$ avec$\phi : \mathbb{R}^p \to \mathbb{R}$ et $x, y \in \mathbb{R}^p$ et à modifier le produit scalaire de x_i et x dans l'équation de l'hyperplan :

$$h(x) = \sum_{i=1}^{n} \alpha_i y_i K(x_i, x) + w_0. \tag{4.14}$$

Il existe beaucoup de fonctions noyaux permettant de modifier l'espace de représentation :

$K(x, y) = x^\top y$ le noyau linéaire, qui ne modifie pas l'espace,

$K(x, y) = (x^\top y + 1)^d$ le noyau polynomial,

$K(x, y) = \exp\left(-\frac{\|x-y\|^2}{2\sigma^2}\right)$ le noyau gaussien.

4.5.2 Extension des machines à vecteurs supports à la régression

Posons $p \in \mathbb{N}$ le nombre de variables, $n \in \mathbb{N}$ le nombre d'individus, $(x_i)_{i \in \mathbb{N}_n^*} \in \mathbb{R}^{np}$ l'ensemble des observations, $y \in \mathbb{R}^n$ les vraies valeurs de la sortie X_{50}, $w \in \mathbb{R}^p$ les poids associés aux variables, $w_0 \in \mathbb{R}$ le poids à l'origine, $\xi, \xi^* \in \mathbb{R}$ les variables ressorts et C, ε des hyperparamètres.

Dans le cadre d'une régression ($y_i \in \mathbb{R}$), le contexte de séparation des données par un hyperplan devient flou. On va plutôt chercher un hyperplan qui est le plus proche possible des y_i. C'est-à-dire trouver w, w_0 tels que :

$$|\langle w, x_i \rangle + w_0 - y_i| \leq \varepsilon,\ \forall i \in \{1, \ldots, n\}, \tag{4.15}$$

avec ε à définir par l'utilisateur.

En d'autres termes, les observations doivent être dans une marge de taille 2ε. Pour éviter de prendre un ε trop grand et de voir sa précision dégringoler, on introduit encore une fois des variables ressorts (*slack variables*) autorisant certaines observations à être mal classées. On va donc résoudre le problème de minimisation suivant selon (w, w_0, ξ, ξ^*) :

$$\begin{cases} \min(\frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}(\xi_i + \xi_i^*)), \\ y_i - \langle w, x_i \rangle + w_0 \leq \varepsilon + \xi_i, \\ -y_i + \langle w, x_i \rangle + w_0 \leq \varepsilon + \xi_i, \\ \xi_i \geq 0, \xi_i^* \geq 0, \\ i \in \{1, \ldots, n\}. \end{cases} \tag{4.16}$$

On trouve alors que le vecteur poids minimisant ce problème w^* s'écrit comme :

$$w^* = \sum_{i=1}^{n}(\alpha_i^* - \alpha_i)x_i,\ \alpha, \alpha^* \in \mathbb{R}^n. \tag{4.17}$$

Les vecteurs supports sont les seuls vecteurs x_i tels que $(\alpha_i^* - \alpha_i)$ ne soit pas nul. Ainsi, le poids optimal w^* est une combinaison linéaire des vecteurs supports. On pourra donc à partir de ces résultats utiliser l'astuce de noyau montrée précédemment.

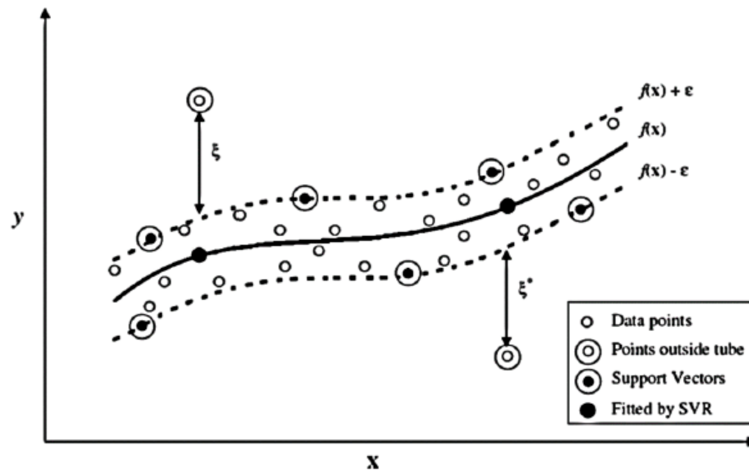


FIGURE 4.2 – Représentation graphique d'une régression à vecteurs supports. [1]

4.6 Méthodes Ensemblistes

Pour plus de détails sur cette section, voir le chapitre 7 du cours d'introduction à l'apprentissage automatique de Frédéric Sur [15].

La théorie des méthodes ensemblistes sera expliquée le plus souvent dans le cas de la classification supervisée. Le cas de la régression et de la classification des méthodes ensemblistes sont très similaires. Dans la mesure du possible, un équivalent pour le cas de la régression sera donnée.

4.6.1 Principe général, classifieurs fort et faible

Imaginons un cas de figure où nous avons à résoudre un problème de classification en deux catégories distinctes. Nous disposons de N classifieurs qui nous fournissent des prédictions indépendantes. Chaque classifieur présente une probabilité d'erreur p inférieure à $1/2$. Lorsque cette probabilité est légèrement inférieure à $1/2$, le classifieur est à peine meilleur que le hasard. On parle alors de **classifieur faible**.

Il semble alors naturel d'utiliser un système de "vote" pour prédire la classe de chacune des observations : la classe majoritaire parmi les N classifieurs est la classe prédite. Nous pouvons, par un calcul rapide, observer si cela augmente la qualité de notre prédiction.

Un simple dénombrement sous hypothèse d'indépendance des prédictions nous dit que la probabilité que la moitié des classifieurs parmi les N prédisent la mauvaise classe est :

$$\sum_{k=[N/2]}^{N} C_N^k p^k (1-p)^{N-k}. \tag{4.18}$$

A l'aide de l'inégalité de Höffding, on peut en déduire la majoration suivante :

$$\sum_{k=[N/2]}^{N} C_N^k p^k (1-p)^{N-k} \leq e^{-2N\left(p-\frac{1}{2}\right)^2}. \tag{4.19}$$

Le terme de droite décroît en $O(e^{-N})$ lorsque N croît et que $p < \frac{1}{2}$.

Il est donc avantageux de combiner des classifieurs faibles pour créer un classifieur fort. Toutefois, il est important de noter que le classifieur fort ainsi créé en choisissant la classe majoritaire parmi les classes proposées par les classifieurs faibles ne sera performant que si les réponses des classifieurs faibles sont indépendantes les unes des autres. Dans le cas contraire, notre méthode ne sera plus valide.

L'association des réponses de classifieurs faibles pour créer un classifieur fort est l'objet des "méthodes ensemblistes". Nous pouvons distinguer deux grandes familles de méthodes ensemblistes : les approches de type *bagging* et les approches de type *boosting*. Ici, nous ne parlerons uniquement de l'approche de *boosting*.

Dans le cadre de la régression : les notions de classifieur faible et fort ne sont pas directement transposables dans le contexte de la régression. Cependant, on peut trouver des notions similaires dans certains types de modèles de régression. Par exemple, les modèles linéaires simples, tels que la régression linéaire, peuvent être considérés comme des modèles de régression faibles, car ils ont une précision limitée dans la prédiction de données complexes. En revanche, les modèles de régression plus complexes, tels que les arbres de décision, les réseaux de neurones, ou les méthodes d'ensemble, peuvent être considérés comme des modèles de régression forts, car ils sont capables de prédire avec précision une grande variété de données d'entrée complexes.

4.6.2 Le boosting

Le *boosting* est une technique d'apprentissage automatique où l'on construit séquentiellement plusieurs classifieurs faibles sur l'ensemble de la base de données. À chaque étape, un nouveau classifieur est construit en se basant sur les classifieurs précédents. Les observations qui ont été mal prédites par les classifieurs précédents sont pondérées de manière plus importante pour que les futurs classifieurs puissent mieux les traiter. Cette stratégie permet de concentrer les efforts sur les observations les plus difficiles à classer, améliorant ainsi la performance globale du modèle de classification.

Exemple de modèle : Adaboost

Supposons que notre base d'apprentissage contient N observations labellisées notées $(x_i, y_i)_{1\leq i\leq N}$ avec y_i valant -1 ou 1 selon la classe. On note h_m le classifieur faible sélectionné à l'étape m.

Notons K_m la fonction discriminante du classifieur fort construit à la m^eétape, combinant les classifieurs faibles précédents telle que :

$$\begin{cases} K_0 \text{ est le classifieur de } \mathcal{H} \text{ minimisant le nombre d'erreurs pour des observations non pondérées,} \\ K_m = K_{m-1} + \alpha_m h_m, \end{cases} \tag{4.20}$$

avec $\alpha_m > 0$. Nous cherchons à trouver les poids α_m et les classifieurs h_m à chaque étape, de manière à obtenir un classifieur fort intéressant après quelques étapes en minimisant un risque empirique calculé sur la base d'apprentissage. Le risque empirique du classifieur K_m est une fonction de perte exponentielle :

$$E_m = \sum_{i=1}^{N} e^{-y_i K_m(x_i)}. \tag{4.21}$$

Ainsi,

$$E_m = \sum_{i=1}^{N} e^{-y_i(K_{m-1}(x_i)+\alpha_m h_m(x_i))} = \sum_{i=1}^{N} w_i^m e^{-y_i \alpha_m h_m(x_i))} \tag{4.22}$$

avec $w_i^m = e^{-y_i K_{m-1}(x_i)}$.

Comme $y_i h_m(x_i) = 1$ si $h_m(x_i) = y_i$ et $y_i h_m(x_i) = -1$ si $h_m(x_i) \neq y_i$ on peut écrire :

$$E_m = \sum_{i \ t.q. \ h_m(x_i)=y_i} w_i^m e^{-\alpha_m} + \sum_{i \ t.q. \ h_m(x_i)\neq y_i} w_i^m e^{\alpha_m}, \tag{4.23}$$

$$E_m = e^{-\alpha_m} \left(\sum_{i=1}^{N} w_i^m + \sum_{i \ t.q. \ h_m(x_i)\neq y_i} w_i^m \left(e^{2\alpha_m} - 1\right) \right). \tag{4.24}$$

Minimiser E_m sur $\mathcal{H}$ revient à minimiser la seconde somme car elle seule dépend de h_m. Donc on a :

$$h_m^* = \underset{h_m}{\arg\max} \sum_{i \ t.q. \ h_m(x_i)\neq y_i} w_i^m \tag{4.25}$$

car on verra plus tard que $e^{2\alpha_m} - 1 > 0$. En d'autres termes, h_m^* est le classifieur faible qui minimise la somme des coûts w_i^m sur les observations mal classées.

Désormais, trouvons le poids optimal α_m^* minimisant E_m. On sait que α_m^* est bien un minimum si :

$$\frac{\partial E_m}{\partial \alpha_m} = 0. \tag{4.26}$$

Ce qui équivaut à :

$$\alpha_m^* = \frac{1}{2} \ln \left(\frac{\sum_{i \ t.q. \ h_m(x_i)=y_i} w_i^m}{\sum_{i \ t.q. \ h_m(x_i)\neq y_i} w_i^m} \right). \tag{4.27}$$

Et par définition de w_i^m :

$$w_i^{m+1} = e^{-y_i K_m(x_i)} = w_i^m e^{-\alpha_m y_i h_m(x_i)}. \tag{4.28}$$

D'où la règle de mise à jour :

$$\begin{cases} \text{Si } h_m(x_i) \neq y_i : \ w_i^{m+1} = w_i^m e^{\alpha_m} \\ \text{Si } h_m(x_i) = y_i : \ w_i^{m+1} = w_i^m e^{-\alpha_m}. \end{cases} \tag{4.29}$$

De plus, les poids sont normalisés afin d'éviter des problèmes d'instabilité : $\sum_{i=1}^N w_i^m = 1$.

En notant $\varepsilon_m = \sum_{i \ t.q. \ h_m(x_i) \neq y_i} w_i^m$ le coût des erreurs, on a donc : $1 - \varepsilon_m = \sum_{i \ t.q. \ h_m(x_i) = y_i} w_i^m$. Ainsi :

$$\alpha_m = \frac{1 - \varepsilon_m}{2\varepsilon_m}. \tag{4.30}$$

Comme promis précédemment, on trouve que : $e^{2\alpha_m} - 1 = e^{\frac{1-\varepsilon_m}{\varepsilon_m}} - 1 > 0$.

On peut ainsi écrire l'algorithme d'Adaboost, avec M un hyperparamètre :

Algorithm 2 AdaBoost

Initialisation : $\forall i \in \{1, \ldots, N\}$: $w_i^0 = \frac{1}{N}$.
Pour $m = 0$ à M :
1. Sélectionner le classifieur faible h_m dans $\mathcal{H}$ minimisant ε_m.
2. En posant : $\varepsilon_m^* = \min_{h_m} \epsilon_m$, on pose : $\alpha_m = \frac{1-\varepsilon_m^*}{2\varepsilon_m^*}$.
3. Mettre à jour les poids des observations, $\forall i \in \{1, \ldots, N\}$:

$$\begin{cases} \text{Si } h_m(x_i) \neq y_i : \ w_i^{m+1} = w_i^m e^{\alpha_m} \\ \text{Si } h_m(x_i) = y_i : \ w_i^{m+1} = w_i^m e^{-\alpha_m} \end{cases} \tag{4.31}$$

et les normaliser tels que $\sum_{i=1}^N w_i^{m+1} = 1$.

Le Gradient Boosting

Dans l'algorithme d'AdaBoost, on cherchait à minimiser le risque empirique E_m du classifieur K_m défini par :

$$E_m = \sum_{i=1}^N l(y_i, K_m(x_i)). \tag{4.32}$$

Avec l la fonction de perte exponentielle et $K_m = K_{m-1} + \alpha_m h_m$.

Le *gradient boosting* utilise une technique de généralisation pour appliquer cette approche à d'autres fonctions de perte qui sont supposées être différentiables [8]. Le but est de minimiser E_m en utilisant la méthode de la plus forte pente. Si on développe E_m jusqu'à l'ordre 1, on obtient :

$$\begin{aligned} E_m &= \sum_{i=1}^{N} l(y_i, K_{m-1}(x_i) + \alpha_m h_m(x_i)), \\ &\approx \sum_{i=1}^{N} l(y_i, K_{m-1}(x_i)) + \alpha_m \sum_{i=1}^{N} \frac{\partial l}{\partial y}(y_i, K_{m-1}(x_i))h_m(x_i), \\ &= E_{m-1} + \alpha_m \sum_{i=1}^{N} \frac{\partial l}{\partial y}(y_i, K_{m-1}(x_i))h_m(x_i) \end{aligned} \tag{4.33}$$

avec $\frac{\partial l}{\partial y}$ la dérivée partielle de l par rapport à sa deuxième composante.

Il vient alors que, pour minimiser E_m en suivant la pente la plus forte, il faut :

- choisir h_m dans $\mathcal{H}$ tel que $h_m = \arg\min_{h_m \in \mathcal{H}} - \sum_{i=1}^{N} \frac{\partial l}{\partial y}(y_i, K_{m-1}(x_i))h_m(x_i)$,
- choisir $\alpha_m > 0$ minimisant E_m.

Les modèles ensemblistes que nous utiliserons dans ce projet (`XGBoost` et `Light GBM`) sont des algorithmes de *gradient boosting*, où l'approximation de l'erreur empirique est cette fois-ci d'ordre 2 et un terme de régularisation L^1/L^2 est utilisé afin d'éviter le sur-apprentissage.

4.6.3 Les modèles de stacking

Lorsque l'on développe des modèles d'apprentissage automatique et que l'on teste leur performance, il arrive que certains modèles soient très performants sur certaines régions de nos données, mais moins performants sur d'autres régions. De même, on peut parfois trouver des modèles plus performants sur certaines régions de nos données, là où d'autres modèles pourraient être moins performants, bien qu'ils soient plus performants en moyenne sur l'ensemble des données. Par ailleurs, il arrive que certains modèles soient quasiment autant performants les uns que les autres. Quel modèle devrait-on alors choisir ?

Avec la méthode de *stacking*, la question ne se pose plus. On va choisir **tous les modèles**. Dans le domaine de la régression, le *stacking* fonctionne de la manière suivante : on choisit un ensemble de modèles (déjà optimisés au préalable), ainsi qu'un **méta-régresseur**. Un méta-regresseur va être un modèle de régression (souvent simple, comme une régression linéaire par exemple) qui va prendre en entrée les sorties de chaque modèle de notre ensemble. C'est donc les poids de ce méta-régresseur qui vont être optimisés lors de la phase d'entraînement. Dans le cas où le méta-régresseur est une régression linéaire, la sortie du méta-régresseur va être une moyenne pondérée de la sortie des modèles de notre ensemble [3].

Pour que la méthode de *stacking* soit efficace, il serait avantageux que l'ensemble de modèles que l'on donne en entrée soient de natures différentes (par exemple un modèle de *Gradient Boosting*, un SVR, un réseau de neurones...) De toute manière, un modèle de *stacking* sera toujours au moins aussi performant que le meilleur modèle de l'ensemble donné en entrée.

La contrepartie d'une telle méthode est bien entendu sa complexité, car on devra entraîner l'ensemble des modèles données en entrée ainsi que le méta-régresseur.

Chapitre 5

Mesure de l'incertitude des réseaux de neurones

Les réseaux de neurones et les systèmes de *deep learning* donnent des très bonnes performances dans beaucoup de tâches différentes, mais ils présentent généralement beaucoup de désavantages : ces derniers demandent beaucoup de données pour être efficaces, beaucoup de calculs pour être entraînés, ils n'arrivent pas a représenter l'incertitude de leurs prédictions, peuvent facilement être dupés par certains exemples "adversaires". Par ailleurs, ce sont souvent des boîtes noires, qui manquent donc cruellement de transparence. qui peut nous donner une difficulté à faire confiance à de tels modèles.

Évidemment, l'ensemble de ces points sont des sujets de recherche. Ici, nous nous focaliserons sur l'ensemble de solutions apportées pour la représentation de l'incertitude des modèles à l'aide de **méthodes bayésiennes pour le *deep learning***.

Dans ce chapitre nous décrirons l'état de l'art de la mesure de l'incertitude des réseaux de neurones et plus en particulier de la méthode du *Dropout* de Monte-Carlo pour estimer la probabilité postérieure de notre sortie sachant notre entrée et notre modèle entraîné. L'entièreté des théories de l'incertitude des réseaux de neurones sont basées sur des méthodes bayésiennes, d'où le nom de *Bayesian Deep Learning*.

Dans un réseau de neurones classique, les poids sont des scalaires que l'on optimise. Or dans un réseau de neurones bayésien, chaque poids suit une loi probabiliste. Ainsi, après chaque *forward pass*, les poids sont échantillonnés sur leur distribution. L'objectif n'est alors plus de trouver la meilleure valeur des poids, mais la meilleure distribution pour chaque poids.

5.1 Définition

L'apprentissage profond bayésien est le traitement des sources d'incertitudes des paramètres (les poids) et de la structure (nombre de couches, nombre de neurones par couche, fonction d'activation, etc.) d'un réseau de neurones.

Une approche bayésienne des modèles d'apprentissage automatique repose sur la **marginalisation** au lieu de **l'optimisation**. Au lieu d'utiliser une seule combinaison de paramètres Θ, nous utilisons

l'ensemble des combinaisons de paramètres possibles tout en pondérant chaque combinaison de paramètres par leurs probabilités postérieures dans un **modèle bayésien moyen**.

5.1.1 La loi de Bayes

La loi de Bayes est une manière de mettre à jour nos croyances sur les hypothèses, c'est-à-dire, des quantités incertaines, sachant que l'on a observé certaines données, c'est-à-dire, des quantités certaines.

Rappelons d'abord la règle de la somme et du produit en probabilité : soient X et Y deux variables aléatoires définies dans l'espace Ω. Alors :

$$\begin{cases} \mathbb{P}(X) = \int_{y \in \Omega} \mathbb{P}(X, y) dy, \\ \mathbb{P}(X, Y) = \mathbb{P}(X \mid Y)\mathbb{P}(Y). \end{cases} \tag{5.1}$$

Avant d'observer les données, nous devons exprimer nos connaissances sur les hypothèses à partir d'une distribution de probabilité, représentant l'incertitude de nos hypothèses. Cette distribution de probabilité se nomme la loi *à priori*. La vraisemblance est finalement la probabilité d'obtenir l'ensemble des observations si on admet le modèle de réseaux de neurones et les paramètres choisis. C'est donc la tentative de résumer la réalité (dont les observations font partie) à des modèles mathématiques. Ainsi, la loi de Bayes nous donne que :

$$\mathbb{P}(\Theta \mid \mathcal{D}, F_\Theta) = \frac{\mathbb{P}(\Theta \mid F_\Theta)\mathbb{P}(\mathcal{D} \mid \Theta, F_\Theta)}{\mathbb{P}(\mathcal{D} \mid F_\Theta)}, \tag{5.2}$$

où :
$\mathbb{P}(\Theta \mid F_\Theta)$ est la loi *à priori*,
$\mathbb{P}(\mathcal{D} \mid \Theta, F_\Theta)$ est la vraisemblance,
$\mathbb{P}(\Theta \mid \mathcal{D}, F_\Theta)$ est la loi *à posteriori*.
Cette expression est en fait caractéristique de l'entraînement d'un modèle avec une approche bayésienne : on donne une loi *à priori* sur la distribution des paramètres, puis à chaque fois que le modèle s'entraîne sur un certain paquet de données, on met à jour cette loi *à priori* à l'aide des observations que l'on a fait. On a donc une loi *à posteriori*, qui est en fait la loi *à priori* du prochain paquet, etc. On peut ainsi voir l'entraînement d'un modèle d'apprentissage automatique comme une forme d'inférence.

Concernant la prédiction supposons que nous observons le nouvel événement (x, y), la loi de la somme et du produit en probabilité nous donnent que :

$$\mathbb{P}(y \mid \mathcal{D}_{tr}, x, F_\Theta) = \int \mathbb{P}(y \mid \Theta, \mathcal{D}_{tr}, x, F_\Theta)\mathbb{P}(\Theta \mid \mathcal{D}_{tr}, x, F_\Theta) d\Theta. \tag{5.3}$$

En d'autres termes, cette équation stipule que, pour connaître la distribution des prédictions sachant nos données et notre modèle, il faut utiliser toutes les combinaisons possibles de paramètres pondérés par leurs probabilités à posteriori.

L'entraînement d'un réseau de neurones classique est un cas particulier, où l'on suppose que $\hat{\Theta} = \Theta^*$ et que la postérieure est égale à $\delta(\Theta = \Theta^*)$.

Finalement, l'approche bayésienne nous permet de faire une inférence sur la distribution des paramètres au lieu de simplement appliquer la distribution des paramètres. Nous remarquons ici que nos expressions ne dépendent pas du fait que notre modèle soit un réseau de neurones. En effet, **ces méthodes bayésiennes s'appliquent pour tout modèle d'apprentissage automatique**.

5.2 Quel est l'intérêt de l'apprentissage profond bayésien ?

Les réseaux de neurones classiques n'arrivent pas à estimer leur incertitude. En effet, ce dernier est parfaitement déterministe une fois entraîné.

L'apprentissage profond bayésien est fondamental pour calibrer non seulement le modèle mais également l'incertitude de nos prédictions. En d'autres termes, on aimerait avoir des modèles qui soient conscients de leur incertitude de prédiction. En effet, une meilleure représentation de l'incertitude aura un impact absolument crucial sur les décisions que l'on prendra à partir des résultats de nos modèles. Par ailleurs, les réseaux de neurones bayésiens ont des meilleures prédictions que les réseaux de neurones standards. En d'autres termes, faire une moyenne des N prédictions pour une entrée donnée d'un réseau de neurone bayésien sera plus précis que la prédiction de cette même entrée par un réseau de neurones standard [10].

5.3 Inférence Variatonnelle

5.3.1 Introduction/Définitions

Revenons à l'expression (5.3) qui est caractéristique de l'entraînement d'un modèle avec une approche bayésienne :

$$\mathbb{P}(\Theta \mid \mathcal{D}, F_\Theta) = \frac{\mathbb{P}(\Theta \mid F_\Theta)\mathbb{P}(\mathcal{D} \mid \Theta, F_\Theta)}{\mathbb{P}(\mathcal{D} \mid F_\Theta)}, \tag{5.4}$$

où :
$\mathbb{P}(\Theta \mid F_\Theta)$ est la loi *à priori*,
$\mathbb{P}(\mathcal{D} \mid \Theta, F_\Theta)$ est la vraisemblance,
$\mathbb{P}(\Theta \mid \mathcal{D}, F_\Theta)$ est la loi *à posteriori*.
En pratique, il s'avère que le dénominateur : $\mathbb{P}(\mathcal{D} \mid F_\Theta)$ est souvent très difficile à calculer. Ainsi, le calcul de la postérieure est également difficile. Pour résoudre ce problème on utilise **l'inférence variationnelle**.

L'inférence variationnelle va transformer un problème **d'inférence** en un problème **d'optimisation**. Posons $(q(\Theta; \nu))_\nu$ une famille de distributions sur la variable latente Θ, où ν est un **paramètre variationnelle**. Notre objectif est de trouver la meilleure valeur de ν telle que $q(\Theta, \nu)$ et $\mathbb{P}(\Theta \mid \mathcal{D}, F_\Theta)$ soient les plus proches possibles en terme de **divergence de Kullback-Leibler** définie dans le chapitre 3. Nous avons donc le système d'optimisation suivant :

$$\nu^* = \arg\min_\nu D_{\mathrm{KL}}(q(\Theta, \nu) \| \mathbb{P}(\Theta \mid \mathcal{D}, F_\Theta)). \tag{5.5}$$

Toutefois, nous avons encore le même problème qu'avant, à savoir que nous devons calculer la postérieure $\mathbb{P}(\Theta \mid \mathcal{D}, F_\Theta)$. Pour résoudre ce problème, introduisons la borne inférieure variationnelle logarithmique ou *log evidence lower bound*, plus communément appelée **ELBO**, que l'on note : $\mathcal{L}_{VI}$ et définie comme :

$$\mathcal{L}_{VI}(\nu) := \int q(\Theta, \nu)\, log(\mathbb{P}(\mathcal{D} \mid \Theta, F_\Theta))d\Theta - D_{\mathrm{KL}}(q(\Theta, \nu) \| \mathbb{P}(\Theta \mid F_\Theta)). \tag{5.6}$$

On remarque que l'ELBO ne dépend pas de la postérieure de Θ (notamment, la divergence de Kullback Leibler se fait entre q et la loi *à priori* de Θ), ce qui la rend donc facilement calculable.

Lemme 1. *Minimiser (selon ν) la divergence de Kullback-Leibler équivaut à maximiser (selon ν) la log evidence lower bound.*

In fine notre système d'optimisation se réduit à :

$$\nu^* = \arg\max_{\nu} \mathcal{L}_{VI}(\nu) \tag{5.7}$$

5.4 Processus gaussien

Définissons ce qu'est un processus gaussien.

Définition 2. *Soit X un processus stochastique et E un ensemble fini de sites. X est dit gaussien sur E si, pour toute partie finie $A \subset E$ et toute suite réelle (a) sur A, $\sum_{s \in A} a_s X(s)$ est une variable gaussienne.*

Il s'agit donc en quelque sorte d'une généralisation des vecteurs gaussiens à la dimension infinie. De la même manière que pour les vecteurs gaussiens, la loi d'un processus gaussien est déterminée par sa fonction moyenne $a(t) = \mathbb{E}[X_t]$ et l'opérateur de covariance $K(s,t) = \mathrm{Cov}(X_s, X_t)$. A ce moment là, la loi finie dimensionnelle de $(X_{t_1}, \ldots, X_{t_n})$ est alors la loi normale de dimension n $\mathcal{N}(a_n, K_n)$ avec $a_n = (a(t_1), \ldots, a(t_n))$ et $K_n = (K(t_i, t_j))_{1 \leq i,j \leq n}$. Les fonctions a et K définissent donc toutes les lois finies dimensionnelles de X et donc aussi sa loi [4].

5.4.1 Exemples

Nous donnons ici quelques exemples de processus gaussiens fréquemment utilisés. Comme dit précédemment, il suffit de donner la fonction moyenne et l'opérateur de covariance de notre fonction pour déterminer entièrement notre processus gaussien. En terme d'analyse de variation du processus gaussien, la fonction moyenne est peu utile, c'est pourquoi nous spécifierons ici uniquement les opérateurs de covariance :

constant : $K_C(x, x') = C$,
bruit gaussien blanc : $K_{GN}(x, x') = \sigma^2 \delta_{x,x'}$,
le mouvement brownien : $K_{MB}(x, x') = \min(x, x')$,
Ornstein–Uhlenbeck : $K_{OU}(x, x') = \exp\left(-\frac{|x-x'|}{\ell}\right)$.

En apprentissage automatique, les hyperparamètres des processus gaussiens sont les paramètres dans l'opérateur de covariance choisi.

5.4.2 L'intérêt des processus gaussiens dans l'apprentissage profond bayésien

Les processus gaussiens peuvent-être utilisés pour résoudre des problèmes de *machine learning*. En effet, ces-derniers peuvent résoudre des tâches très variées, notamment la tâche de régression. De plus, ces modèles sont **non-paramétriques**.

Prenons par exemple une tâche de régression. Un modèle paramétrique est un modèle dépendant d'un ensemble fini de paramètres, qui vont être optimisés par l'apprentissage d'observations. Les modèle non-paramétriques tels que les processus gaussiens ont en fait un nombre infini de paramètres (qui correspondent à l'ensemble des possibilités de positionnement des points). Par ailleurs, le caractère probabiliste des processus gaussiens nous permet d'estimer très facilement son incertitude en tout point.

Comment peut-on faire des prédictions avec les processus gaussiens ? Soit $D_{obs} = (x_{obs_i}, y_{obs_i})_{i\in\{1,...,m\}}$ l'ensemble des données observées et $D_{pred} = (x_{pred_i}, y_{pred_i})_{i\in\{1,...,n\}}$ l'ensemble des données de prédiction (ou de test). Notons $y(x)$ la variable de sortie et x les variables d'entrées. En supposant que $y(x)$ est un processus gaussien, il vient que :

$$\exists a, b \in \mathbb{R},\ A \in \mathcal{M}_{m,m},\ C \in \mathcal{M}_{n,m},\ B \in \mathcal{M}_{m,n} \mid \mathcal{L}(y_{obs}, y_{pred}) = \mathcal{N}\left(\begin{bmatrix} a \\ b \end{bmatrix}, \begin{bmatrix} A & B \\ B^\top & C \end{bmatrix}\right). \quad (5.8)$$

La loi de Bayes nous donne que :

$$\mathbb{P}(y_{pred} \mid y_{obs}) = \frac{\mathbb{P}(y_{pred}, y_{obs})}{\mathbb{P}(y_{obs})}. \quad (5.9)$$

Il vient donc que :

$$\mathcal{L}(y_{pred} \mid y_{obs}) = \mathcal{N}(a + BC^{-1}(y_{obs} - b), A - BC^{-1}B^\top). \quad (5.10)$$

5.4.3 Des processus gaussiens aux processus gaussiens profonds

Nous pouvons modéliser un processus gaussien de la manière suivante. Un processus gaussien est un modèle dont la sortie y est égale à une variable aléatoire $f(x)$ avec l'ajout d'un bruit gaussien centré réduit ε multiplié par une variance σ_y. On peut donc écrire que : $y(x) = f(x) + \varepsilon\sigma_y$ avec $\varepsilon \sim \mathcal{N}(0, 1)$. On a : $f(x) \sim \mathcal{GP}(0, K_f)$ avec K_f l'opérateur de covariance de $f(x)$.

Posons maintenant : $g(x) \sim \mathcal{GP}(0, K_g)$ avec K_g l'opérateur de covariance de $g(x)$. Alors, **un processus gaussien profond à 2 couches** est un modèle tel que $y(x) = f(g(x)) + \varepsilon\sigma_y$ [6]. On peut donc généraliser cette définition à un processus gaussien à n couches.

5.5 Monte Carlo Dropout

5.5.1 Dropout

Revoyons formellement la définition d'un réseau de neurones à *dropout* pour le cas d'un réseau de neurones *à une seule couche cachée*, en raison de la facilité des notations. Par ailleurs la généralisation à plusieurs couches cachées est facile une fois le cas à une seule couche cachée fait [9].

Supposons que l'entrée est à Q dimensions, que la sortie est à D dimensions et que nous avons K neurones dans la couche cachée. Notons $W_1, W_2 \in (Q \times K) \times (K \times D)$ les matrices des poids connectant respectivement la couche d'entrée à la couche cachée et la couche cachée à la couche de sortie. Ces poids transforment linéairement les entrées des couches avant d'appliquer une fonction

d'activation $\sigma(.)$ non linéaire. Notons $b \in \mathbb{R}^K$ le vecteur des biais. Alors, un réseau de neurones standard donnerait la sortie $\hat{y}$ suivante, étant donnée une entrée x :

$$\hat{y} = \sigma(xW_1 + b)W_2. \tag{5.11}$$

On applique le *dropout* en utilisant deux vecteurs binaires : $z_1 \in \{0,1\}^Q$ et $z_2 \in \{0,1\}^K$. Les éléments des vecteurs sont distribués selon une distribution de Bernoulli avec un paramètre $p_i \in [0,1]$ pour $i = 1, 2$. Ainsi, $z_{1,q} \sim Bernoulli(p_1)$ pour $q = 1, \ldots, Q$ et $z_{2,k} \sim Bernoulli(p_2)$ pour $k = 1, \ldots, K$. Étant donnée une entrée x, la proportion $1 - p_1$ d'éléments de l'entrée sont réduits à zéro : $x \circ z_1$ avec $\circ$ le produit de Hadamard. Il vient alors que : $\hat{y} = ((\sigma((x \circ z_1)W_1 + b)) \circ z_2)W_2$.

5.5.2 Le Dropout, une approximation Bayésienne.

Nous pouvons alors montrer qu'un réseau de neurones dans lequel on applique un *dropout* à chaque couche cachée est une approximation d'un processus gaussien [10]. Pour la démonstration, voir [9].

Ainsi, un réseau de neurones auquel on applique une couche de *dropout* entre chaque couche (étant effective à la fois lors de l'entraînement et à la fois lors de la prédiction) nous permet d'approcher un processus gaussien profond, qui nous permet alors d'approcher la postérieure. Donc pour chaque observation de test, nous sommes capables de tracer la densité de probabilité correspondant à la loi postérieure. Par exemple, la figure 5.1 montre l'estimation de densité de la loi de la première valeur de test sachant les données d'entraînement et le modèle. Nous constatons que notre modèle est relativement imprécis (il ne faut pas oublier que X_{50} se situe entre 0 et 1). Toutefois, le maximum de la densité de probabilité est atteint aux alentours de la vraie valeur de X_{50}.

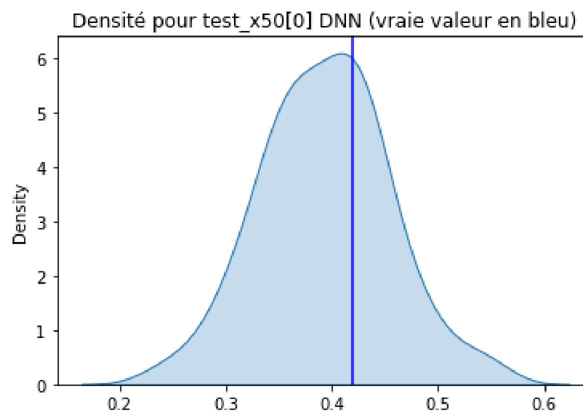


FIGURE 5.1 – Estimation de densité de la loi de la première valeur de test sachant les données d'entraînement et le modèle à l'aide de la méthode du *Monte-Carlo Dropout*.

Les autres modèles utilisés n'étant pas des réseau de neurones, nous ne pouvons pas appliquer la méthode de *Monte Carlo Dropout*. Pour tenter de trouver une estimation de l'incertitude sur ce dernier, aucun article de recherche pertinent n'a été trouvé à ce jour.

Chapitre 6

Protocole et résultats

Pour rappel, nous nous sommes focalisés sur les données provenant de [13]. Notre objectif est de prédire la taille médiane des fragments X_{50} à partir des données. L'équation de Kuznetsov, utilisée dans le modèle de Kuz-Ram, est largement utilisée dans le domaine de la fragmentation de roche pour prédire la taille médiane des fragments. Nous voulons ici voir si des modèles d'apprentissage automatiques sont capables d'avoir une plus grande précision que cette équation pour estimer la taille des fragments.

L'article [1] a déjà testé les performances de deux modèles : un réseau de neurones et une régression à vecteurs supports. Voici les résultats obtenus dans cet article [1] :

	MSE	R^2
Rés. de neurones	0.0031	0.87
SVR	0.0044	0.81
Eq. de Kuznetsov	0.0121	0.58

Les résultats semblent très satisfaisants. Cependant aucun écart-type n'a été fourni. Par ailleurs, les scores ont été établis sur une base de test fixe et n'ont donc pas été évalués à l'aide d'une validation croisée (cf. section 6.1.2).

6.1 Protocole

6.1.1 Gestion des données

Après importation, les données ont été transformées et les *outliers* ont été supprimés (cf. chapitre 3). Nous nous retrouvons alors avec 100 observations. Une simple séparation en données d'entraînement et en données de test fixes ne donnerait pas des résultats pertinents, car ayant trop peu de données de test, ces dernières ne seraient pas représentatives de l'ensemble des données. Nous allons donc analyser les performances des modèles en faisant une validation croisée à cinq plis et en prenant la moyenne et l'écart-type des scores sur les cinq plis.

Par ailleurs, nous devons traiter l'encodage de notre variable qualitative, nous donnant pour chaque observation le site où a eu lieu l'explosion. On compte en tout 10 sites différents. Nous

allons procéder à du *one hot encoding* pour numériser cette variable. Pour chaque site, nous allons créer une nouvelle colonne et dans chaque colonne i, nous allons indiquer par un 1 ou un 0 si l'observation provient bien du site i. Nous avons donc maintenant en tout 17 variables, ce qui est plutôt élevé pour 100 observations. Dans la section 6.1.3, nous allons tenter de réduire ce nombre de variables, car un trop grand nombre de variables par rapport à nos observations rendront nos modèles peu performants.

6.1.2 La mesure de performance

Nous utiliserons comme mesure de performance l'erreur moyenne quadratique entre la vraie sortie X_{50} et la prédiction de cette dernière. Néanmoins il ne faut pas oublier que nous avons transformé notre sortie pour qu'elle puisse suivre une loi normale (nous lui avons appliqué la fonction $x \to \ln(1+x)$). Nous devons ainsi créer une nouvelle mesure de performance nous permettant de retransformer la sortie de notre modèle en prédiction de X_{50}, et non pas en prédiction de $\ln(1+X_{50})$. Ainsi, notre mesure de performance sera la suivante :

$$\frac{1}{n}\sum_{i=1}^{n}(\exp(y_i) - \exp(\hat{y_i}))^2 \tag{6.1}$$

avec n le nombre d'observations, y_i la i-ème valeur de $\ln(1+X_{50})$ et $\hat{y_i}$ la prédiction de la i-ème valeur de $\ln(1+X_{50})$. Nous avons donc simplement appliqué à notre sortie transformée la fonction $x \to \exp(x) - 1$.

Dans les résultats on peut aussi voir le score du coefficient de détermination R^2 régie par $R^2 = 1 - \frac{\sum_i^n (y_i - \hat{y_i})^2}{\sum_i^n (y_i - y_{moy})^2}$ avec : $y \in \mathbb{R}^n$ les vraies valeurs de x_{50}, $\hat{y} \in \mathbb{R}^n$ les valeurs prédites de X_{50} et $y_{moy} \in \mathbb{R}$ la valeur moyenne de y. Plus R^2 se rapproche de 1, plus notre modèle est adapté au problème. Cette mesure est donnée à titre d'indication car il s'agit usuellement de la mesure de référence pour tester la validité d'un modèle. Ce n'est pas la mesure sur laquelle les modèles ont été entraînés.

6.1.3 Sélection de caractéristiques

Notre nombre de variables (17) semble légèrement élevé pour notre nombre d'observations (100). On craint ainsi une malédiction de la dimension, rendant notre modèle moins performant. Nous allons donc vérifier que l'ensemble de nos variables sont bel et bien importantes pour l'efficacité de notre modèle. Si certaines variables s'avèrent inutiles ou peu importantes, nous les supprimerons. Ce procédé se nomme **sélection de caractéristiques**. Comment peut-on connaître l'importance de nos variables pour la prédiction de notre sortie ? Les modèles de type arbre de décision peuvent nous venir en aide. Suite à l'entraînement de tels modèles, nous pouvons mesurer l'importance de chaque variable, d'où leur surnom de "boîte blanche" en opposition aux réseaux de neurones. Nous avons donc entraîné un modèle d'arbres de décision de type *Gradient Boosting* à savoir *XGBoost* (car il s'agit de l'arbre de décision le plus performant sur nos données) dont les hyperparamètres ont été optimisés à l'aide d'un algorithme de *Random Search.* Nous pouvons alors tracer l'importance de chaque variable pour le modèle *XGBoost.* (cf. figure 6.1).

On constate alors que l'ensemble des colonnes encodant notre variable qualitative n'ont aucune importance pour notre modèle *XGBoost* hormis le site noté `Ad`. On peut comprendre par là que l'ensemble des informations propres au site sont déjà contenues dans les variables représentant les

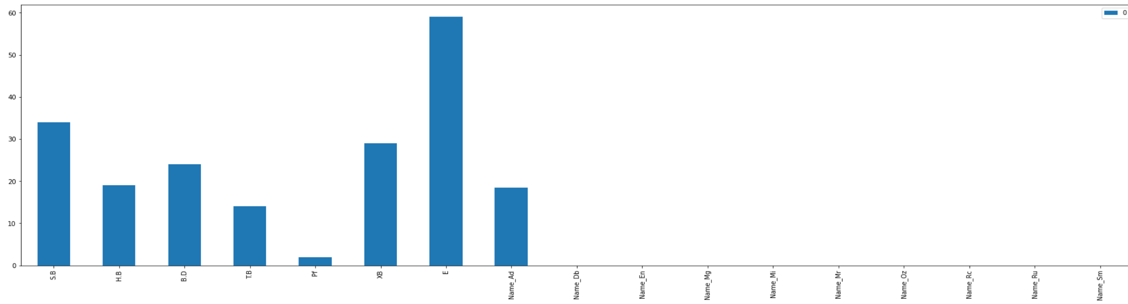


Figure 6.1 – Histogramme montrant l'importance de chacune des variables pour le modèle *XGBoost*. L'ensemble des variables dont le nom commence par **'Name'** sont les colonnes de la variable qualitative que nous avons encodé.

caractéristiques de la roche (le module d'Young de la roche E et la taille de la roche *in situ* X_b) hormis pour le site `Ad`. Nous allons donc supprimer l'ensemble des colonnes dont l'importance est nulle pour ce modèle. Ainsi ils nous restent uniquement les données quantitatives que nous avions au départ ainsi que la colonne propre au site `Ad`. On pourrait également supprimer le facteur de poudre P_f, de par son importance très faible. Néanmoins garder cette variable pourrait justement légèrement améliorer notre performance.

6.1.4 Entraînement des modèles

Une fois nos variables sélectionnées, nous pouvons entraîner nos modèles pour observer leur performance. Les modèles entraînés sont les suivants : *XGBoost* (*Gradient Boosting*), LigthGBM (*Gradient Boosting*), une régression à vecteurs supports, *ElasticNet*, un réseau de neurones ainsi qu'un modèle de *stacking*. Pour chaque modèle, les hyperparamètres ont été optimisés à l'aide d'un algorithme de *Random Search*. On peut par exemple observer sur la figure 6.2 que notre réseau de neurones est bien adapté aux données : son erreur d'entraînement diminue drastiquement au fur et à mesure des époques et son erreur de validation et très proche de l'erreur d'entraînement.

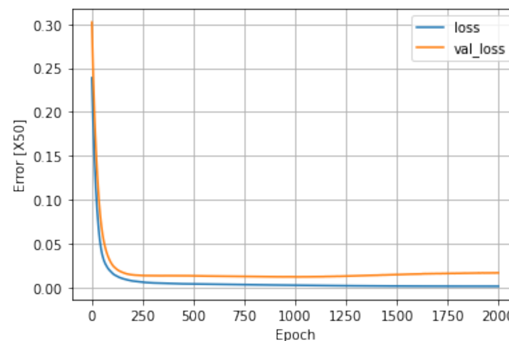


Figure 6.2 – Mesure de performance du réseau de neurones en fonction du nombre d'époques. Parmi les données d'entraînement, 20% constituent la base de validation.

Concernant le modèle de *stacking*, l'ensemble des *régresseurs* donnés en entrée du modèle sont tous les modèles précédents, à savoir *XGBoost*, *LightGBM*, la régression à vecteurs supports,

ElasticNet et le réseau de neurones. Nous avons choisi comme méta-régresseur une autre régression à vecteurs supports dont les hyperparamètres ont été optimisés de la même manière qu'avant. En tout, la phase d'optimisation des hyperparamètres pour la totalité des modèles a duré une vingtaine d'heures (avec pour chaque modèle un *Random Search* à validation croisée à 5 plis et 5000 itérations).

Certains modèles ont besoin d'avoir en entrée des données normalisées tandis que d'autres sont plus performants en gardant l'échelle initiale des données. Les modèles d'arbres de décision tels que *XGBoost* et *LightGBM* appartiennent à cette dernière catégorie. On va donc normaliser les données en entrée pour l'ensemble des modèles sauf ces deux derniers à l'aide de la fonction `RobustScaler` présentée dans le chapitre 3.

6.2 Les résultats

6.2.1 Les scores des modèles

La mesure de performance et le coefficient de détermination (R^2) de chaque modèle ont été déterminés par validation croisée à cinq plis comme expliqué précédemment dans ce chapitre. Voici ci-dessous le tableau représentant les mesures de performance ainsi que le coefficient de détermination pour chaque modèle, en comparaison avec le modèle de l'équation de Kuznetsov.

Modèles	Mesure de perf.	Ecart-type mesure	R^2	Ecart-type R^2
Rés. de neurones	0.0108	0.0041	0.6596	0.0515
XGBoost	0.0123	0.0059	0.5769	0.1659
LightGBM	0.0122	0.0060	0.5838	0.1629
SVR	0.0077	0.0041	0.7497	0.1084
ElasticNet	0.0084	0.0044	0.7226	0.1156
Stacking	0.0074	0.0021	0.7518	0.0433

	Mesure de perf.	R^2
Eq. de Kuznetsov	0.0121	0.58

La mesure de performance, le coefficient de détermination et leurs écarts-types respectifs nous montrent le modèle de *stacking* est légèrement meilleur que la régression à vecteurs supports et s'avère être le seul significativement meilleur que l'équation de Kuznetsov (selon le critère de la mesure de performance). Le modèle de *stacking* est donc vraisemblablement un bon candidat.

Une autre observation est que nos résultats semblent inférieurs aux résultats trouvés dans l'article [1] (notre meilleur modèle a une performance deux fois inférieure au réseau de neurones utilisé dans l'article). Néanmoins, ces derniers ont utilisé une base de test fixe alors que nous avons utilisé une validation croisée à cinq plis. Il se peut alors que l'article ait choisi une base de test favorable à la prédiction de leurs modèles. Au vu de l'écart-type de nos modèles les plus performants, on peut constater que leur mesure de performance est proche des mesures de performances trouvées par l'article [1] pour les bases de test les plus faciles à prédire.

6.3 Analyse des résidus des modèles

Pour que nos modèles soient valables, leurs résidus doivent suivre une loi normale centrée en 0 de variance constante (critère d'homoscédasticité). Vérifions cela en traçant l'estimation de densité des résidus de nos modèles (à savoir, la densité de $\mathbf{Y} - \hat{\mathbf{Y}}$ avec $\mathbf{Y}$ la variable aléatoire représentant les vraies valeurs de X_{50} et $\hat{\mathbf{Y}}$ la prédiction de cette dernière par notre modèle) (Figure 6.3).

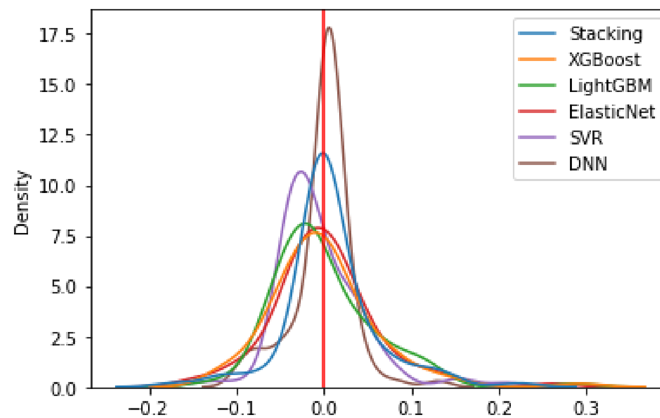


Figure 6.3 – Estimation de densité des résidus de nos modèles. Le trait rouge marque la droite $x = 0$.

Nous constatons sur la figure 6.3 que les résidus de notre modèle de *stacking* semblent être ceux les plus proches d'une loi normale centrée en 0 de variance constante. Le SVR quant à lui ne semble pas centré en 0, ce qui contredit l'hypothèse d'homoscédasticité. Le modèle de *stacking* peut donc être considéré comme un modèle valable et plus performant que l'équation de Kuznetsov pour estimer la taille médiane des fragments.

Chapitre 7

Conclusion et travaux futurs

7.1 Conclusion

La fragmentation est un sujet important dans de nombreux domaines. Ici, nous nous sommes concentrés sur la fragmentation de roche et avons apporté une solution à un problème concret, qu'est l'estimation de la taille des fragments à l'aide de modèles d'apprentissage automatique. Parmi les modèles mis au point, le modèle de *stacking* montre des performances supérieures à l'équation de Kuznetsov traditionnellement utilisée pour estimer la taille médiane des fragments. De plus, ce modèle de *machine learning* peut facilement être implémenté dans des logiciels d'analyse de fragmentation et pourra un jour aider les experts à planifier des explosions de roche plus sûres et plus efficaces. Bien que nos résultats semblent à première vue moins bons que les résultats de [1], nous pensons néanmoins que l'utilisation d'une validation croisée à cinq plis pour prédire les scores et les écarts-types sont plus pertinents que les scores (sans leurs écarts-types) sur une base de test fixe. Par ailleurs, malgré le fait que le modèle de *stacking* nécessite du temps à être optimisé (car nous devons optimiser l'ensemble des modèles donnés en entrée du *stacking* ainsi que le méta-régresseur), ceci ne pose pas vraiment de problème dans le contexte de la fragmentation de roche où les explosions sont prévues plusieurs mois voire plusieurs années à l'avance.

Pendant un an, j'ai su développer un ensemble de compétences dans plusieurs domaines scientifiques. J'ai manipulé des concepts mathématiques nouveaux tels que la divergence de Kullback-Leibler ou l'inférence variationnelle. J'ai également beaucoup appris en apprentissage automatique, domaine dans lequel mes connaissances étaient quasiment nulles au début de ce projet. J'ai pu me renseigner sur son aspect théorique et mettre en pratique l'ensemble des articles que j'ai lu, que ce soit sur des méthodes de visualisation des données, sur certains modèles d'apprentissage en particulier ou encore sur l'incertitude des réseaux de neurones. Par ailleurs j'ai grandement développé mes compétences de programmation en `Python` en manipulant des librairies telles que `Scikit-learn`,`Tensorflow`, `Seaborn` ou encore `Scipy`. Certains projets personnels annexes à ce projet de recherche, tels que la participation à des *Data Challenges* m'ont également permis d'élargir ma vision globale du problème et d'améliorer mon approche de résolution, que ce soit sur le prétraitement des données ou sur l'optimisation des modèles. Je tiens à remercier Madalina Deaconu (IECL & INRIA) et Antoine Lejay (IECL & INRIA) pour m'avoir aidé et accompagné tout le long de ce projet.

7.2 Travaux futurs

Pour améliorer nos résultats, la piste d'amélioration la plus évidente serait d'augmenter notre nombre de données. Nous pouvons espérer à l'avenir que de nouvelles mesures sur de nouveaux sites soient publiées.

Une autre piste d'amélioration serait de travailler d'avantage sur le *feat ure engineering*, c'est-à-dire, sur la gestion des variables données en entrée. En effet, peut-être que l'ajout de variables étant des combinaisons non linéaires de nos variables d'origines pourraient améliorer nos résultats. Nous avons déjà fait des tests en remplaçant nos variables par des combinaisons linéaires des variables d'origine à l'aide d'une analyse en composantes principales, mais les résultats n'étaient pas concluants.

Enfin, une autre voie d'amélioration plus ambitieuse serait d'ajouter des images à nos observations pour avoir des informations supplémentaires sur la géométrie de la roche et éventuellement procéder à de la segmentation d'images à l'aide de réseaux de neurones convolutifs.

Bibliographie

[1] Richard Amoako, Ankit Jha, and Shuo Zhong. Rock Fragmentation Prediction Using an Artificial Neural Network and Support Vector Regression Hybrid Approach. *Mining*, 2(2) :233–247, April 2022. doi:10.3390/mining2020013.

[2] Bernhard Boser, Isabelle Guyon, and Vladimir Vapnik. A Training Algorithm for Optimal Margin Classifier. *Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory*, 5, August 1996. doi:10.1145/130385.130401.

[3] Leo Breiman. Stacked regressions. *Machine learning*, 24 :49–64, 1996.

[4] Jean-Christophe Breton. *Processus Gaussiens*. Master IMA 2ème année, 2006. Notes de cours.

[5] CVB Cunningham. The kuz-ram fragmentation model–20 years on. In *Brighton conference proceedings*, volume 2005, pages 201–210. European Federation of Explosives Engineers, England, 2005.

[6] Andreas Damianou and Neil D Lawrence. Deep gaussian processes. In *Artificial intelligence and statistics*, pages 207–215. PMLR, 2013.

[7] Madalina Deaconu and Antoine Lejay. Probabilistic representations of fragmentation equations. *Probability Surveys*, 20 :226–290, 2023. doi:10.1214/23-PS14.

[8] Jerome H Friedman. Greedy function approximation : a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[9] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation : Appendix, 2015. URL : https://arxiv.org/abs/1506.02157.

[10] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation : Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059. PMLR, 2016. URL : https://arxiv.org/abs/1506.02142.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[12] Geoffrey E Hinton and Sam Roweis. Stochastic neighbor embedding. *Advances in neural information processing systems*, 15, 2002.

[13] P.H.S.W. Kulatilake, Wu Qiong, T. Hudaverdi, and C. Kuzu. Mean particle size prediction in rock blast fragmentation using neural networks. *Engineering Geology*, 114(3) :298–311, August 2010. doi:10.1016/j.enggeo.2010.05.008.

[14] Alessia Mammone, Marco Turchi, and Nello Cristianini. Support vector machines. *Wiley Interdisciplinary Reviews : Computational Statistics*, 1(3) :283–289, 2009.

[15] Frederic Sur. *Introduction à l'apprentissage automatique*. École des Mines de Nancy, 2022. Notes de cours.

[16] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.