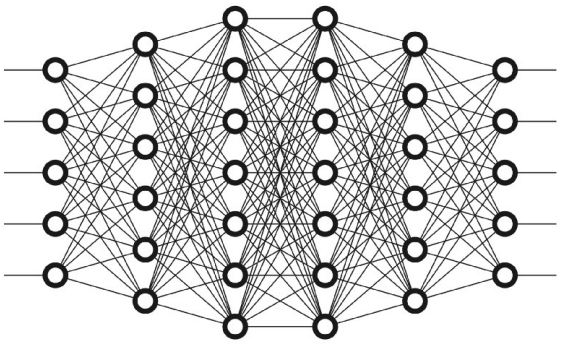
# PROJET RECHERCHE
## RAPPORT SEMESTRIEL

**APPLICATIONS DE L'APPRENTISSAGE AUTOMATIQUE A DES PROBLEMES DE FRAGMENTATION.**


## VICTOR HOFFMANNN

**ELEVE A L'ECOLE DES MINES DE NANCY.**


**12 DECEMBRE 2022**




**ENCADRANTS :**
MADALINA DEACONU & ANTOINE LEJAY.
**INSTITUT ELIE CARTAN DE LORRAINE**

# Table des matières

# 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 [6].

### 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 controllables, comme l'espacement entre chaque explosion, la quantité d'explosifs dans chaque trou etc. D'autres sont incontrollables, le module d'Young de la roche par exemple. Il est ainsi nécessaire que les ingénieurs spécialisés en explosifs puissent trouver la bonne configuration de ces paramètres controllables 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 inexploitable et d'être conduits dans des terrils, ce qui a un coût à la fois économique et environnemental.

## 1.2 Equation de fragmentation

Une équation à dérivées partielles permet de modéliser le processus de fragmentation. Soit un système à nombre de particules infini. Chaque particule est caractérisée par sa taille et peut se séparer en deux tout en conservant sa masse à des moments aléatoires. Soit $c(t,x)$ la concentration de particules de taille $x$ à l'instant $t$ dans le système. L'évolution de $c(t,x)$ au cours du temps est régie par l'équation suivante [2] :

$$\begin{cases} \frac{\partial}{\partial t}c(t,x) = \int_x^1 F(x,y-x)c(t,y)dy + \frac{1}{2}c(t,x)\int_{\mathbf{0}}^{\mathbf{x}} F(y,x_y)dy \ \ \forall t \geq 0,\ x \in [0,1] \\ c(0,x) = c_0(x) \ \ \forall x \in [0,1], \end{cases} \tag{1.1}$$

où $F$ est le noyau de fragmentation. La fonction $F : ]0,1]^2 \to \mathbb{R}_+$ est symétrique. $F(x,y)$ représente le taux de fragmentation de particule de taille $x+y$ en deux particules de taille $x$ et $y$. On suppose que la taille initiale de la particule est de 1.

Sur la permière ligne de l'équation, le premier terme du second membre compte la création de particules de taille $x$, due à la fragmentation de particules de tailles plus grande (disons $y$, avec $y \geq x$) en deux parties : $x$ et $y-x$. Le second terme compte la disparition des particules de taille $x$ après s'être séparés en deux plus petites particules de taille $y$ et $x-y$ (avec $y \leq x$ cette fois-ci).

Nous pouvons alors déterminer l'évolution des fragments au cours du temps. Toutefois, cette équation déterminste peut s'avérer très difficile à résoudre dans le cas de noyaux complexes.

Ainsi, d'autres méthodes peuvent être envisagées pour déterminer l'évolution des fragments. Dans ce rapport, nous allons nous concentrer sur des méthodes de machine learning et de deep learning permettant de résoudre des problèmes de fragmentation de roche.

## 1.3 Objectifs du travail de ce semestre

Le travail de ce semestre avait pour objectif d'appliquer des méthodes de machine learning et de deep learning à des problèmes de fragmentation afin de faciliter leur résolution. Pour ce faire, le travail a du se décomposer en plusieurs étapes :

1. Se renseigner sur l'aspect théorique 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 sur les données de test.
6. Mesurer l'incertitude des meilleurs modèles.

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 testé pour sa résolution : régression linéaire, régression Ridge polynomiale, réseau de neurones profonds et Support Vector Regression. Les hyperparamètres optimaux des modèles ont été trouvés

à l'aide des méthodes GridSearch et RandomizedSearch. 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.

# Chapitre 2

# Quelques bases d'apprentissage automatique :

L'ensemble de ces concepts est présentée de manière plus exaustive dans [9]. Ici, nous en 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 à aux tâches T mesurées par P, s'améliorent avec l'expérience E. »

## 2.1 La tâche T

Dans la définition relativement formelle du mot "tache", 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 example, 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. Ce genre de prédictions est également utilisé dans le trading.

## 2.2 La Mesure de Performance, P

Afin dévaluer les capaciter 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 tâches telles que la régression, nous utilisons souvent l'**erreur moyenne absolue (MAE)** comme fonction de perte. Elle est représentée par :

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

avec $y$ l'ensemble des sorties, $\hat{y}$ l'ensemble des prédictions de la sortie et m le nombre de sorties. On utilise également souvent **l'erreur quadratique moyenne (MSE)**

$$MSE = \frac{1}{m} \sum_{i}^{m} (\hat{\mathbf{y}} - \mathbf{y})_i^2 \,, \tag{2.2}$$

avec les mêmes notations.

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 claire et objective, 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 dépendent de l'application.

## 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 non supervisés, étant donnée que nous cherchons à résoudre un problème de régression.

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

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

**Les algorithmes supervisés** observent un dataset contenant des caractéristiques, mais chaque exemple est également associé avec un **label** ou une **cible**. Par exemple, le dataset Iris contient les espèces de chaque plante iris. Un algorithme supervisé pourra étudier ce dataset 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 $p(x)$, ou des propriétés intéressantes de cette distribution. Tandis que les algorithmes supervisés implquent 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.

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 Capicité, Overfitting et Underfitting

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 set d'entraînement, pouvons calculer une mesure de l'erreur sur cet ensemble d'entraînement et cherchons à la réduire, ce qui 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 du set de test si nous pouvons uniquement observer le set d'entraînement ? Le champ de **la théorie d'apprentissage statistique** nous donne des éléments de réponse. Si set d'entraînement et de test sont collectés aribitrairement, il y

a en effet peu de choses à faire. Toutefois, si nous pouvons émettre des hypothèses sur la manière dont le set 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 datasets 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 exemples dans chaque dataset sont **indépendants** entre eux, et que le set d'entraînement et de test sont **identiquement distribuées**, c'est à dire que les deux sets ont la même distribution de probabilité. Cette hypothèse nous permet de décrire le processus de génération de données avec une distribution de probabilité sur un seul exemple. La même distribution est utilisée pour générer chaque exemple 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 connection 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.

Evidemment, lorsque nous utilisons un algorithme de machine learning, nous ne pouvons pas fixer les paramètres à l'avance, puis prélever les deux sets de données. Il faut d'abord prélever le set d'entraînement et ensuite l'utiliser pour choisir les paramètres nécessaires à réduire l'erreur d'entraînement, puis on prélève le set 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-adaptation** et **l'underfitting ou sous-adaptation**. On observe de la sous-adaptation lorsque le modèle n'est pas capable d'obtenir une valeur assez basse sur l'ensemble d'entraînement. Quant à la sur-adaptation, 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 abilité à être ajusté avec un grand panel de fonctions. Les modèles avec une faible capacité peuvent avoir du mal à adapter le set d'entraînement. Les modèles à forte capacité peuvent donner une situation de sur-adapation en mémorisant les propriétés du set d'entraînement qui ne sont pas utiles sur le set 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. Faire ainsi accrôitre la capacité du modèle.

## 2.5 Exemple : Régression Linéaire et Quadratique

Voici un exemple concret : la régression linéaire. Comme le nom l'indique, la régression linéaire résout un problème de régression. En d'autres termes, le but est de construire un système qui peut prendre un vecteur $x \in \mathbb{R}^n$ commme entrée, et prédire un scalaire $y \in \mathbb{R}$ comme sortie. Soit $\hat{y}$ la valeur que prédit notre algorithme. Ainsi la valeur de la sortie est :

$$\hat{y} = w^\top x, \tag{2.3}$$

où $w \in \mathbb{R}^n$ est un vecteur de **paramètres**.

Les paramètres sont des valeurs qui contrôlent le comportement du système. Dans ce cas, $w_i$ est le coefficient que nous multiplions avec la caractéristique $x_i$ avant de sommer l'ensemble des contributions de toutes les caractéristiques. $w$ peut être pensé comme un ensemble de **poids** qui déterminent comment chaque caractéristique affecte la prédiction. Si $x_i$ reçoit un poids positif $w_i$, alors augmenter la valeur de cette caractéristique augmente la valeur de notre prédiction $\hat{y}$ et vice-versa si $w_i$ est négatif. Si le poids de la caractéristique est grand par rapport aux autres, alors il aura un grand effet sur la prédiction. Si ce poids est nul, il n'aura aucun effet sur cette dernière.

Nous avons ainsi une définition de notre tâche T : prédire y à partir de x, en sortant : $\hat{y}$. Maintenant, il nous faut une définition de notre mesure de performance P.

Supposons que nous avons une matrice représentant les m exemples d'entrée que nous utiliserons uniquement pour les tests. Nous avons aussi un vecteur nous donnant les bonnes valeurs de $y$ pour chacun de ces exemples. Parce que ce dataset sera uniquement utilisé pour l'évaluation, on va l'appeler **test set**. Soit $\mathbf{X}^{(test)}$ la matrice d'entrée et $\mathbf{y}^{(test)}$ le vecteur des bonnes valeurs de $y$ en sortie.

Une manière de mesurer la performance du module est de calculer **l'erreur quadratique moyenne** du modèle sur le test set. Si $\hat{\mathbf{y}}^{(test)}$ donne les prédictinos du modèle sur le test set, alors l'erreur quadratique moyenne est donnée par :

$$MSE_{test} = \frac{1}{m} \sum_{i}^{m} \left( \hat{\mathbf{y}}^{(test)} - \mathbf{y}^{(test)} \right)_i^2. \tag{2.4}$$

On peut donc voir intuitivement que l'erreur tend vers 0 quand $\hat{\mathbf{y}}^{(test)} = \mathbf{y}^{(test)}$ et que l'erreur augmente quand la distance euclidienne entre ces deux vecteurs augmente.

Pour faire un algorithme de machine learning, nous devons créer un algorithme qui va modifier les poids $\mathbf{w}$ de telle sorte que $MSE_{test}$ soit de plus en plus petit au fur et à mesure que l'algorithme gagne de l'expérience en observant un set d'entraînement $\left(\mathbf{X}^{(train)}, \mathbf{y}^{(train)}\right)$. Une manière intuitive de le faire serait à nouveau de minimiser l'erreur des moindres carrés, cette fois-ci sur le set d'entraînement : $MSE_{train}$.

Pour minimiser $MSE_{train}$, on peut chercher la valeur de $\mathbf{w}$ pour laquelle son gradient s'annule :

$$\nabla_w MSE_{train} = 0, \tag{2.5}$$

$$\implies \mathbf{w} = \left(\mathbf{X}^{(train)\top}\mathbf{X}^{(train)}\right)^{-1}\mathbf{X}^{(train)\top}\mathbf{y}^{(train)}. \tag{2.6}$$

Le système d'équations dont la solution est donnée par l'équation (2.6) se nomme **équations normales**.

Il est important de noter que le terme de **régression linéaire** est souvent utilisée pour se réferer à un modèle légèrement plus sophistiqué avec un paramètre additonnel nommé *bias*. Dans ce modèle :

$$\hat{y} = w^\top x + b. \tag{2.7}$$

On a donc une fonction affine. A la place d'ajouter le paramètre de biais b, on peut toujours utiliser un modèle linéaire en rajoutant une entrée supplémentaire toujours égale à un. Son poids associé sera donc le paramètre de biais b. La terminologie du paramètre de **biais** vient du fait que la sortie de la transformation est biaisée vers b en l'absence d'entrée. ATTENTION : ce terme est différent des biais en statistiques, dans lesquels l'estimation statistique d'une quantité n'est pas égale à la vraie quantité.

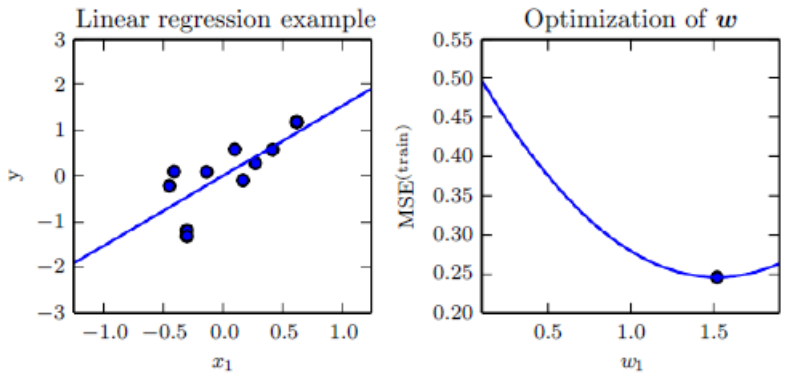



FIGURE 2.1 – Un problème de regression linéaire avec un set d'entraînement de 10 points, chacun contenant une caractéristique. Parce qu'il n'y a qu'une caractéristique, le vecteur de poids $\mathbf{w}$ ne contient qu'un seul paramètre à apprendre, $w_1$. A gauche, on remarque que la régression linéaire apprend à configurer $w_1$ de telle sorte que la ligne $y = w_1 x$ soit le plus proche de l'ensemble des points d'entrâinement. A droite, le point affiché indique la valeur de $w_1$ trouvé par les équations normal, qui minimise l'erreur quadratique moyenne sur le set d'entraînement.

En introduisant $x^2$ comme une autre caractéristique fournie par le modèle de la régression linéaire, nous pouvons apprendre un modèle qui est quadratique par rapport à x :

$$\hat{y} = b + w_1 x + w_2 x^2. \tag{2.8}$$

Même si le modèle implémente une fonction quadratique par rapport à l'entrée, la sortie est toujours une fonction linéaire par rapport aux *paramètres*. Nous pouvons donc toujours utiliser les équations normales pour entraîner le modèle. On pourrait continuer ainsi à ajouter des degrés de polynôme :

$$\hat{y} = b + \sum_{i=1}^{p} w_i x^i. \tag{2.9}$$

Les algorithmes de machine learning vont généralement être performants lorsque leur capacité est appropriée avec la véritable complexité de la tâche qu'ils doivent exécuter ainsi que la quantité de données d'entraînement qu'on leur fournit. Des modèles avec une capacité insuffisante sont incapables de résoudre des tâches complexes. Des modèles qui ont une grande capacité peuvent résoudre des tâches complexes, mais lorsque celle-ci est plus grande que nécessaire, ils peuvent être sur-adaptés (voir figure 1.).

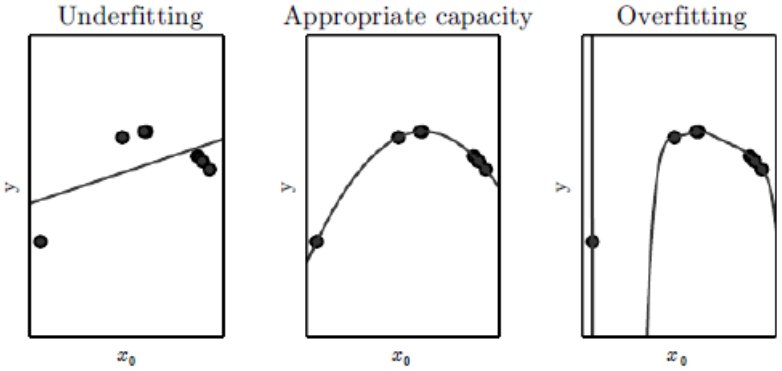



FIGURE 2.2 – Nous essayons d'adapter trois modèles à cet exemple de données d'entraînement. Cette dernière a été générée synthétiquement, en prenant des valeurs aléatoires de x et en choisissant y de manière déterminste en évaluant une fonction quadratique. A gauche, la fonction linéaire souffre de sous-adaptation - 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 d'underfitting ou d'overfitting. A droite, le polynôme de degré 9 souffre de sur-adaptation. 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.

## 2.6 Hyperparmètres et sets 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 nommées **hyperpamètres**. Les valeurs des hyperparamètres ne sont pas adaptés par l'algorithme lui-même.

Dans la régression polynomiale présentée précedemment (cf. *Projet Recherche Semaine 5*), il n'y avait qu'un seul hyperparamètre : le degré du polynôme, qui joue le rôle d'**hyperparamètre de capacité**. La valeur $\lambda$ utilisée pour controller la force du *weight decay* est un autre exemple d'hyperparamètre.

Parfois un paramètre est choisi pour être un hyperparamètre que l'algorithme d'apprentissage n'apprend pas 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 le set d'entrâinement.

Ceci s'applique pour tous les paramètres qui controllent la capacité du modèle. Si on les apprenait sur le set d'entrâinement, de tels hyperparamètres choisiront toujours le modèle avec la plus grande capacité, ce qui causera inéluctablement à de la sur-adaptation.

Pour résoudre ce problème, il nous faut un **set de validation** d'exemples que l'algorithme d'entrâinement n'observe pas.

Plus tôt, nous avions expliqué comment un set de test composé d'exemples venant de la même distribution que le set d'entraînement pouvait être utilisé pour estimer l'erreur de généralisation, une fois le processus d'apprentissage terminé. Il est primordial que le set de test ne soit pas utilisé pour faire des choix sur le modèle, les hyperparamètres compris. Pour cette raison, nous construisons toujours le set de validation à partir du set d'entraînement. Le set de validation va être utilisé pour guider la sélection des hyperparamètres. Comme ce dernier est utlisé pour "entrainer" les hyperparamètres, l'erreur du set 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 le set de test.

### Validation croisée

Diviser l'ensemble de données en un ensemble d'entrâinement 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'ensemle 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.

# Chapitre 3

# Les données

Dans ce chapitre, nous abordons la description du tableau utilisée pour la régression sur $X_{50}$ [11]. Le tableau se nomme `UnionData.csv`.

## 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) [11]. 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*.

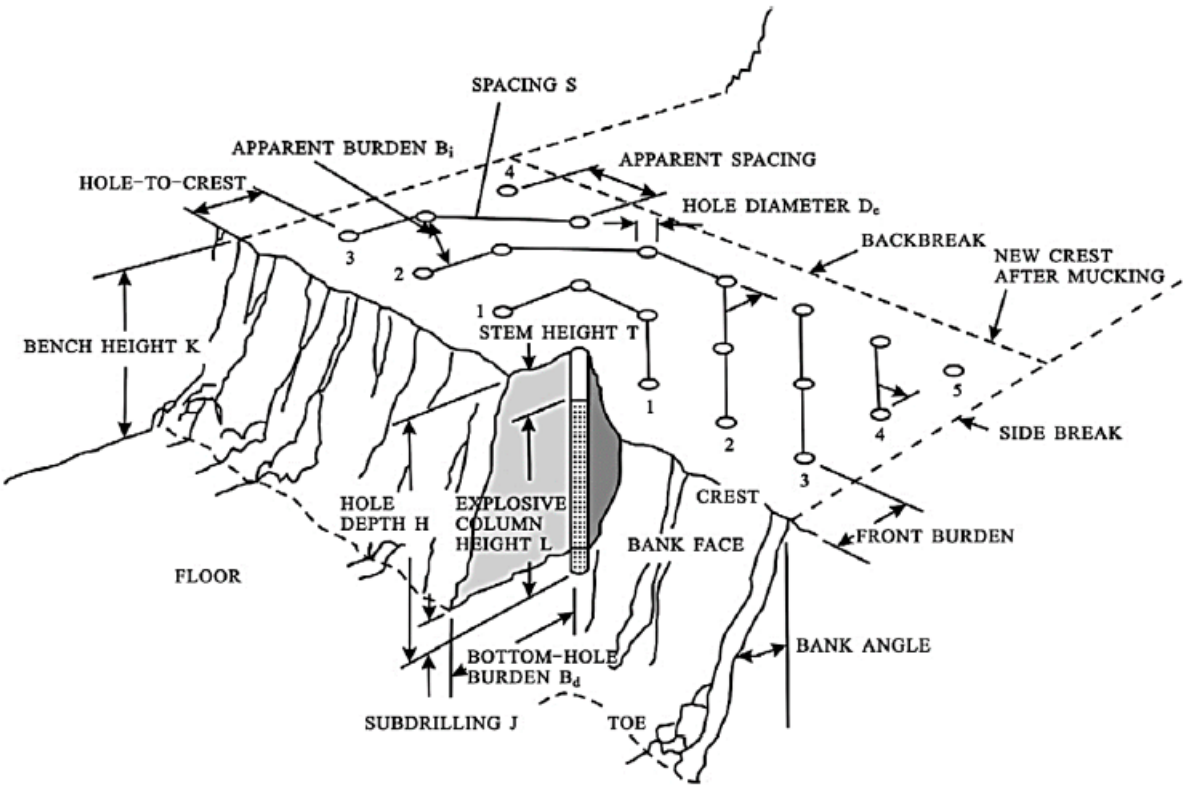



FIGURE 3.1 – Schéma représentant l'ensemble des paramètres géométriques propre à une explosion de roche. Dans le cadre de notre régression, nous n'utiliserons pas l'ensemble des paramètres de la figure, mais uniquement ceux énumérés précédemment. [1]

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

$T/B$ : 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.

## 3.2 Informations statistiques de base

Le tableau comporte 97 observations, ce qui est relativement peu. 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 cojointe 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.

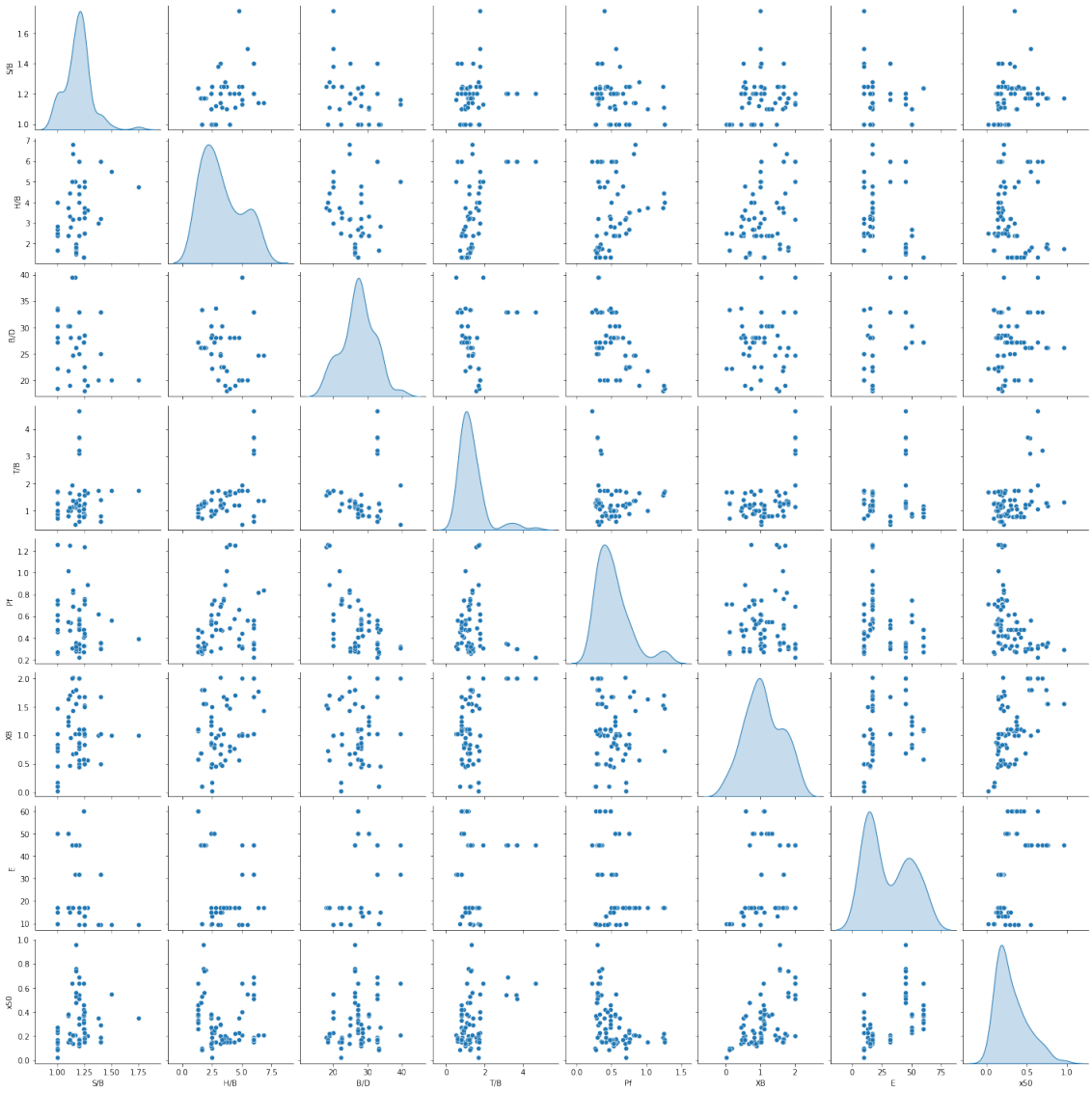



FIGURE 3.2 – Distribution cojointe des variables. Les figures sur la diagonale sont les estimations de densité de chacun des variables. On constate qu'il n'y a pas de corrélation évidente entre l'ensemble des variables.

# Chapitre 4

# Analyse de la sortie $X_{50}$

Une analyse des valeurs de la sortie (ici, $X_{50}$) peut s'avérer utile pour l'amélioration de nos modèles. En effet, certains datasets peuvent présenter des **outliers**, c'est-à-dire, des points qui sont margniaux par rapport à la tendance globale. Une manière de repérer ces outliers est d'estimer la densité de probabilité de $X_{50}$ et de l'identifier à une loi connue afin de repérer les quelques points qui écarteraient $X_{50}$ de sa loi de probabilité.

## 4.1 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 proabable.

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. Tout ce dont nous avons accès est un échantillon d'observations. 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 par exemple.

### 4.1.1 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 de **estimation de densité paramétrique.**

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 :

$$\hat{m} = \frac{1}{n}\sum_{i=1}^{n} x_i, \tag{4.1}$$

$$\hat{\sigma} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{m})^2}. \tag{4.2}$$

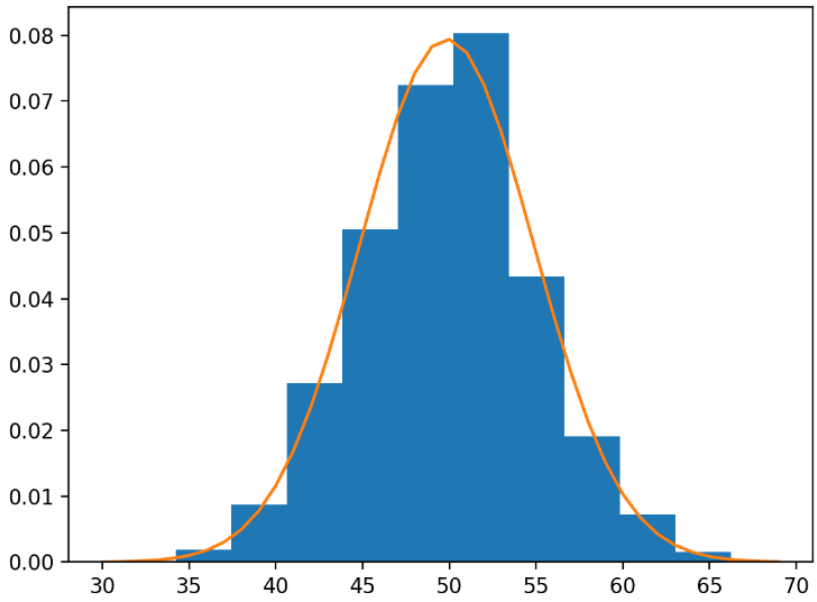



FIGURE 4.1 – Estimation de la densité à partir d'un histogramme. Ici, l'histogramme (en bleu) a été tracé en regroupant les valeurs des observations dans des intervalles de taille 5. L'abscisse correspond à la valeur des observations et l'ordonnée au nombre d'observations compris dans chaque intervalle divisé par le nombre total d'observations. L'histogramme prend clairement la forme d'une loi normale. A l'aide des observations, nous avons trouvé une estimation de la moyenne : $\hat{m} = 49.852$ et une estimation de l'écart-type : $\hat{\sigma} = 5.023$. On a donc en orange l'estimation de la densité des observations qui est une loi normale de paramètre (49.852, 5.023)

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, il se peut qu'il y ait des **observations aberrantes** dont les valeurs sont très loin de la moyenne. Les garder risquerait de donner une mauvaise estimation des paramètres de la distribution et ainsi donner une densité mal adaptée au modèle. Ces observations aberrantes doivent être supprimés avant d'estimer les paramètres de la distribution.

Un autre exemple est que 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 applicant 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'a ce que la distribution soit bien adaptée au modèle :

### 4.1.2 Estimation de densité non paramétrique

Dans certains cas, le dataset ne ressemble pas à une distribution de probabilité usuelle. Cela arrive souvent lorsque l'histogramme des données possède deux pics (distribution bimodale) voir plus (distribution multimodale).

Dans ce cas, l'estimation de densité paramétrique n'est pas applicable. Des méthodes alternatives, n'utilisant pas des distributions de probabilité usuelles, peuvent alors être utilisées. On va utiliser à la place un algorithme qui permet d'approcher la distribution de probabilité des données sans une distribution prédéfinie. Cette méthode se nomme **estimation de densité non paramétrique** [13]. En fait, les distributions auront toujours des paramètres, mais ceux-ci ne seront pas directement ajustable de la même manière qu'avec la méthode paramétrique.

L'approche non paramétrique la plus utilisée pour estimer la densité de probabilité d'une fonction d'un variable aléatoire continue se nomme **estimation de densité par noyau** (ou encore **méthode de Parzen-Rosenblatt**) (en anglais : kernel density estimation, KDE). Cette méthode non paramétrique estime les porbabilités des "nouveaux" points (ceux qui ne font pas partie de l'observation), en dehors des observations contenues dans le dataset.Voici sa définition :

Soit $(x_1, \ldots, x_n)$ l'ensemble des observations indépendantes et identiquement distribuées dont la véritable densité est $f$. Alors l'estimateur non-paramétrique $\hat{f}_h$ de $f$ par la méthode du noyau est :

$$\forall x \in \Omega, \hat{f}_h(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right) \tag{4.3}$$

où $K$ est le noyau et $h$ la fenêtre.

Cette méthode va donc dépendre de deux paramètres :

1. Le **noyau**. Un noyau est une fonction mathématique qui donne une probabilité pour une certaine valeur d'une variable aléatoire. Le noyau va "lisser" ou interpoler les probabilités sur l'ensemble de définition de la variable aléatoire tout en s'assurant que la somme des probabilités vaut 1. En d'autres termes : le noyau est la fonction choisie pour contrôler le poids des observations du dataset sur l'estimation des probabilités des nouveaux points.
2. Le **paramètre de lissage** ou **fenêtre**. Il va régir le degré de lissage de l'estimation. Ce paramètre définit la *largeur de la boîte* en chaque point $x$ où l'on va compter le nombre d'observations à l'intérieur de cette boîte. On va par la suite remplacer cette boîte par le noyau. Pour mieux comprendre cette notion, voir les exemples ci-dessous.

Si le choix du noyau est réputé comme peu influent sur l'estimateur, il n'en est pas de même pour le paramètre de lissage. Un paramètre trop faible provoque l'apparition de détails artificiels apparaissant sur le graphe de l'estimateur ; à l'inverse, pour une valeur de h trop grande, la majorité des caractéristiques est au contraire effacée. Le choix de h est donc une question centrale dans l'estimation de la densité. (cf. figure 4.2.)

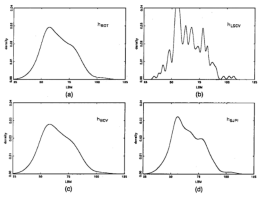


FIGURE 4.2 – Estimation par la méthode du noyau d'observations pour différentes valeurs de la fenêtre. [10]

Sur Python, la librairie de machine learning `scikit-learn` possède la classe `KernelDensity` qui permet d'implémenter une estimation par noyau.

### 4.1.3 Application sur R

On peut appliquer une estimation de densité par noyau sur un ensemble de données d'une variable avec la fonction suivante :

```
d = density(data, kernel = "...",  bw = "...")
```

Avec :

`data` : L'ensemble de données de la variable qui nous intéresse.

`kernel` : la fonction noyau.

`bw` : Le paramètre de lissage.

#### La fonction noyau sur R

Sur R, on peut utiliser plusieurs fonctions noyaux différentes [13] :

**Le noyau gaussien** : $ke(t) = \frac{1}{\sqrt{2}} exp(\frac{-t^2}{2})$,

**Le noyau uniforme** : $ke(t) = \frac{1}{2} I_{]-1,1[}(t)$,

**Le noyau triangulaire** : $ke(t) = (1 - |t|) I_{]-1,1[}$,

**Le noyau d'Epanechnikov** : $ke(t) = \frac{3}{4}(1 - t^2) I_{]-1,1[}$,

**Le noyau quartique (ou biweight)** : $ke(t) = \frac{15}{16}(1 - t^2) I_{]-1,1[}$,

textbfLe noyau circulaire (ou cosine) : $ke(t) = \frac{\pi}{4} cos(\frac{\pi}{2} t) I_{]-1,1[}$.

Toutefois, le choix de la fonction noyau a très peu d'influence sur l'estimation de la densité.

#### Le paramètre de lissage sur R

Contrairement à la fonction noyau, le paramètre de lissage a une grande influence sur la forme de la courbe. Voici les différents paramètres de lissage disponibles sur R :

**Silverman's Rule of Thumb :** $h = 0.9 \cdot \min(\hat{\sigma}, IQR/1.35) n^{-1/5}$ ,

**Scott's Rule of Thumb :** $h \approx 1.06 \cdot \hat{\sigma} n^{-1/5}$ .

Ces paramètres sont très efficaces pour des distributions unimodales [14]. Il y a encore beaucoup de paramètres de lissage différents, qui sont bien plus efficaces pour des distributions multimodales : *Unbiased Cross-Validation, Biased Crossed Validation, Sheater Jones....*

## 4.2 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 ne ressemble pas vraiment à une loi de probabilité usuelle. Nous allons donc procéder à une estimation de densité par noyau sur les valeurs de $X_{50}$.

Comme la distribution semble unimodale, nous allons utiliser le paramètre de lissage nommé **Silverman's Rule of Thumb** décrite précédemment. La fonction noyau choisie est le noyau gaussien. Ce choix est purement arbitraire, la fonction noyau ayant très peu d'influence sur le résultat. On observe sur la figure 4. que la distribution de $X_{50}$ ressemble à une loi gaussienne positive avec une queue allongée. $X_{50}$ pourrait donc suivre une loi log-normale. L'estimation de densité par noyau sur le logarithme de $X_{50}$ confirme cette hypothèse (cf. figure 4.5 ).

Ainsi, nous avons pu déterminer la distribution de $X_{50}$, qui semble suivre une loi log-normale. Comme le montre la figure 4. , il ne semble pas y avoir d'outliers notables. On va donc garder l'ensemble des observations pour nos modèles de régression. On pourrait penser qu'il serait judicieux d'exprimer notre modèle en fonction de logarithme de $X_{50}$ et non en fonction de $X_{50}$ lui-même, car ce premier suit une loi normale. Cela serait en effet vrai si nos modèles de régression supposeraient une distribution normale de nos variables. Cependant cela n'est pas le cas, les modèles de régression supposent uniquement une distribution normale sur les erreurs de prédictions autrement appelées les **résidus**. On tâchera donc de vérifier dans les résultats.

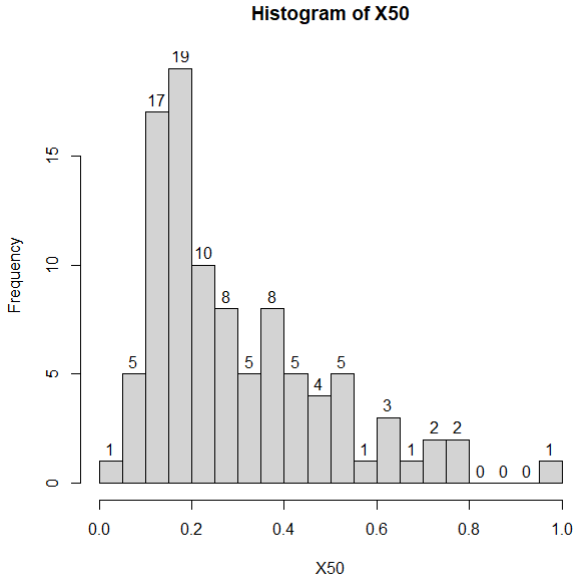



FIGURE 4.3 – Histogramme de $X_{50}$

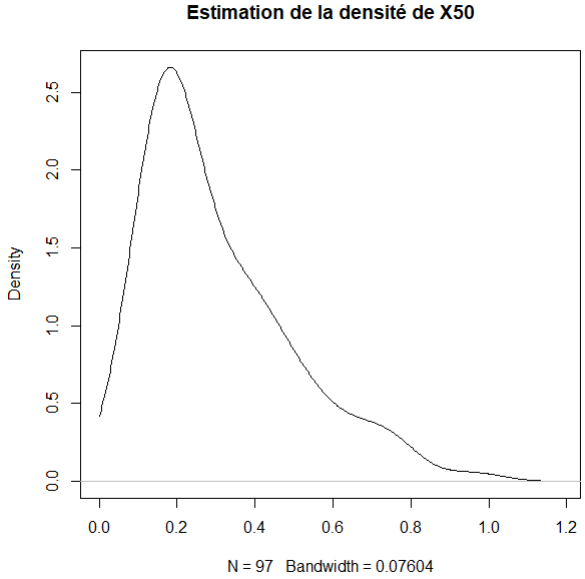



FIGURE 4.4 – Estimation de densité par noyau de $X_{50}$ avec un noyau gaussien et le paramètre de lissage *Silverman's Rule of Thumb*. La contrainte : $X_{50} \geq 0$ a été imposée

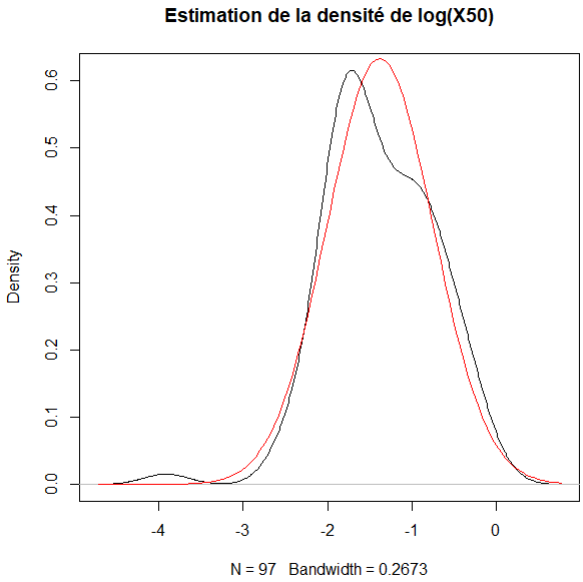



FIGURE 4.5 – Estimation de densité par noyau de $log(X_{50})$ en noir et loi normale de paramètres (moyenne($X_{50}$), écart-type($X_{50}$)) en rouge. On constate que la densité de probabilité de $log(X_{50})$ est proche d'une loi normale.

# Chapitre 5

# Modèles d'apprentissage automatique utilisés

## 5.1 Régression Linéaire

La régression linéaire a déja été expliquée dans la section 2.5.

## 5.2 Régression Ridge

On peut contrer le surapprentissage 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) :

$$MSE_{train} + \lambda \omega^\top \omega, \tag{5.1}$$

où $\lambda$ 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** [9].

Cette méthode est la régression **ridge**.

Ici nous allons utiliser la régression ridge comme une régression polynomiale (cf. section 2.5).

## 5.3 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 simple.

### 5.3.1 Les neurones

Il faut voir chaque **neurone** comme 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.

### 5.3.2 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 sous composants des valeurs associées aux variables. Ainsi, si un neurone dans une certaine couche cachée correspondant au sous-composant, ce dernier s'active. Plus on a de couches, plus les composants 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.

### 5.3.3 Les poids

Pour que notre réseau de neurone puisse apprendre au fur et à mesure des observations, on va utiliser des **poids** que l'on va assigner à chaque connexion que chaque neurone de la deuxième couche a avec la première couche. 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 $\sigma$, il vient que :

$$b_i = \sigma(\sum_{k=1}^{n} \omega_i^a a_i - bias), \tag{5.2}$$

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 $j$-ième 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 $j-1$-ième couche dépend elle même des poids, biais et activations de la $j-2$-ième 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'entrâinement au fur et à mesure des observations.

### 5.3.4 La fonction de coût

Pour que le modèle puisse évaluer la qualité de son estimation, on va introduire une fonction de coût, qui va mesurer la performance du modèle en comparant l'activation du neurone 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.

### 5.3.5 Epoch

Un epoch correspond à un apprentissage sur l'ensemble des observations. Lorsqu'un epoch est fini, le réseau de neurones a donc ajusté les poids les biais et les activations de chaque neurone en ayant balayé l'ensemble des obserations. Un epoch 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.

## 5.4 Support Vector Regression

Le modèle **Support Vector Regression** est un modèle de machine learning destiné à une tâche de régression. Il fait partie de la catégorie des **Support Vector Machines**. Afin de pouvoir expliquer comment les SVR fonctionnent, il est nécessaire d'expliquer au préalable la manière dont les Support Vector Machines ont été créé dans le cadre d'une classification simple. En effet, les Support Vector Machines sont dans l'ensemble une généralisation des classificateurs linéaires.

### 5.4.1 Support Vector Machine

Les Support Vector Machines sont expliquées plus en détail dans l'ouvrage [3].

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 $x_i$ associés à 1 d'un côté et les $x_i$ associés à $-1$ de l'autre.

Evidemment 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 :

$$h(x) = \langle w, x \rangle + w_0 = w^\top x + w_0 = 0, \tag{5.3}$$

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 :

$$\begin{cases} \min\limits_{w,w_0} \frac{1}{2}\|w\|^2, \\ y_i(w^\top x_i + w_0) \geq 1, i \in \{1, \ldots, n\}. \end{cases} \tag{5.4}$$

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 :

$$w^* = \sum_{i=1}^{n} \alpha_i y_i x_i, \ \alpha_i \in \mathbb{R}. \tag{5.5}$$

C'est à dire, un 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.

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 $\epsilon = (\epsilon_1, \ldots, \epsilon_n)$ telles que :

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

$epsilon_i > 1 \implies i$ est mal classé.

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

On cherche donc maintenant à minmiser selon $w, w_0, \epsilon_i$

$$\begin{cases} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n} \epsilon_i, \\ y_i(w^\top x_i + w_0) \geq 1 - \epsilon_i, \\ \epsilon_i \geq 0, i \in \{1, \ldots, n\}, \end{cases} \tag{5.6}$$

avec $C$ à définir par l'utilisateur.

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

$$w^* = \sum_{i=1}^{n} \alpha_i y_i x_i, \ \alpha_i \in \mathbb{R}. \tag{5.7}$$

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

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

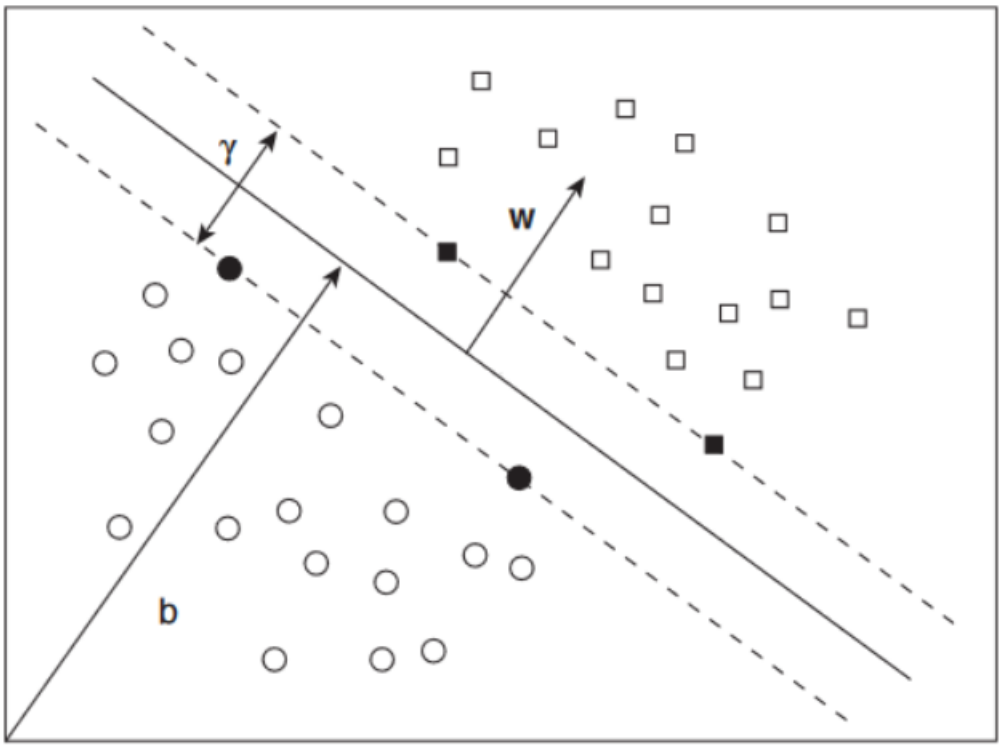



FIGURE 5.1 – Représente l'ensemble des observations $x_i$ associées soit à 1 (ici □) soit à $-1$ (ici ). L'hyperplan optimal est ici tracé en trait plein et les marges maximales de chaque côté en pointillés. Les vecteurs supports sont les échantillons en noir sur la marge. [12]

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

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

On a depuis le début supposé que les observations peuvent être linéairement séparables. Evidemment, 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 :

$$h(x) = \sum_{i=1}^{n} \alpha_i y_i x_i^\top x + w_0 \tag{5.8}$$

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

$$h(x) = \sum_{i=1}^{n} \alpha_i y_i K(x_i, x) + w_0 \tag{5.9}$$

Il existe moult fonctions noyau 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 (ou radial basis function).

### 5.4.2 Extension des Support Vector Machines à la régression

Posons :

$p \in \mathbb{N}$ le nombre de variables

$n \in \mathbb{N}$ le nombre d'individus

$x \in \mathbb{R}^{np}$ l'ensemble des observations

$\forall i \in \mathbb{N}_n^*,\ x_i \in \mathbb{R}^p$ une observation

$y \in \mathbb{R}^n$ 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

$C, \varepsilon$ 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 :

$$|\langle w, x_i \rangle + w_0 - y_i| \leq \varepsilon \; i \in \{1, \ldots, n\}, \tag{5.10}$$

avec $\varepsilon$ à définir par l'utilisateur.

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

$$\begin{cases} \min(\frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}(\xi_i + \xi_i^*)), \\ y_i - \langle w, x_i\rangle + w_0 \leq \varepsilon + \xi_i, \\ -y_i + \langle w, x_i\rangle + w_0 \leq \varepsilon + \xi_i, \\ \xi_i \geq 0, \xi_i^* \geq 0, \\ i \in \{1, \ldots, n\}. \end{cases} \tag{5.11}$$

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

$$w^* = \sum_{i=1}^{n}(\alpha_i^* - \alpha_i)x_i,\ \alpha, \alpha^* \in \mathbb{R}^n. \tag{5.12}$$

Les vecteurs supports sont donc les vecteurs $x_i$ tels que $(\alpha_i^* - \alpha_i)$ ne soit pas nul. On pourra donc à partir de ces résultats utiliser l'astuce de noyau montrer précedemment.

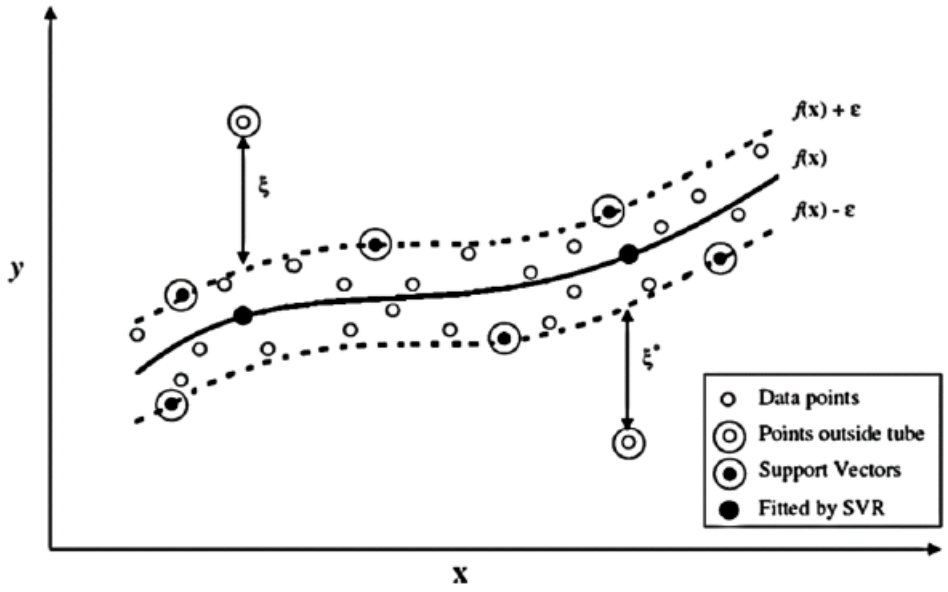



FIGURE 5.2 – Représentation graphique d'un Support Vector Regression. [1]

# Chapitre 6

# Recherche des hyperparamètres optimaux

L'optimisation des hyperparamètres est fondamentale à la fois dans le machine learning et dans le deep learning. En effet des modèles mal ajustées peuvent dans le cas du deep learning converger beaucoup plus lentement vers l'erreur de validation la plus basse voire rendre le modèle carrément sur- ou sousadapté. Plus un modèle est complexe, plus il aura d'hyperparmètres. Il serait donc extrêmement pénible voire impossible 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 [9]

## 6.1 GridSearch

Lorsqu'il y a moins de 4 hyperparamètres, il est commun d'effectuer ce qu'on appelle un **Grid Seach.** Pour chaque hyperparmètre, l'utilisateur sélectionner un petite quantité finie de valeurs à explorer. L'algorithme de grid search va alors entraîner un modèle pour chaque spécification de valeurs d'hyperparmètres dans le produit cartésien de l'ensemble des valeurs pour chaque hyperparamètre pris indiviuellement. L'expérience qui donne le meilleur score (`R^2,val_accuracy etc.`) pour une configuration d'hyperparamètre donnée est retenue.

Grid search performe souvent bien lorsqu'il est exécuté de manière répétitive. Par exemple, supposons que nous lançons un grid search pour un hyperparamètre $\alpha$ avec les valeurs $(-1, 0, 1)$. Si la meilleure valeur est 1, alors nous avons sous-estimé l'ensemble dans lequel le meilleur $\alpha$ se trouvait. On peut alors réexécuter le Grid Search avec valeurs supérieures pour $\alpha$.

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 $O(n^m)$.

## 6.2 RandomizedSearch

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

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

$$\texttt{log\_learning\_rate} \sim u(-1, -5), \tag{6.1}$$

où $u(a, b)$ indique un échantillon d'une distribution uniforme dans l'intervalle (a,b).

Contrairement à Grid Search, on *ne devrait pas discrétiser* l'ensemble des valeurs des hyperparamètres. Ceci permet d'explorer un plus grand ensemble de valeurs et n'augmente pas le coût du modèle. En fait, Randomized Search peut être exponentiellement plus efficace que Grid Search, quand il y a plusieurs hyperparamètres qui ne dépendent pas fortemenent de la mesure de performance.

Une des raisons principales pour lesquelles Randomized Search trouve plus rapidement des bonnes solutions que Grid Search est qu'il n'y a pas de "runs" expérimentaux inutiles, contrairement à Grid Search, lorsque deux valeurs d'un hyperparamètre donnent la même valeur. En effet, si Randomized Search remarque que deux valeurs différentes d'un même hyperparamètre donnent des réultats similaires, ce dernier va chercher à explorer d'autres hyperparamètres. De même qu'avec Grid Search, il est préférable de lancer plusieurs fois Random Search pour réajuster les valeurs pour chaque hyperparamètre choisies.

# Chapitre 7

# Mesure de l'incertitude : Dropout de Monte Carlo

## 7.1 Bayesian Neural Network

Posons :

$n$ le nombre d'observations,

$m$ le nombre de variables,

$p$ le nombre de paramètres,

$S$ l'ensemble dans lequel se trouve chacune des étiquettes.

$\mathcal{D}_{tr} = (x_i, y_i),\ i \in \{1, \ldots, n\}$ l'ensemble d'entraînement, on suppose les $(x_i, y_i)$ indépendant deux à deux et $x_i \in \mathbb{R}^m, y_i \in S$

$\Theta \in \mathbb{R}^p$ le vecteur contenant l'ensemble des paramètres du modèle,

$F_\Theta : \mathbb{R}^m \to S$ la fonction représentant le modèle,

$\mathcal{L}()$ la fonction de perte.

Lors de l'entraînement dans un réseau de neurones usuel, on va chercher à trouver les paramètres qui minimisent notre fonction de perte. En notant $\Theta^*$ l'ensemble de paramètres satisfaisant cette condition, on a :

$$\Theta^* = arg\min_\Theta \frac{1}{n} \sum_{i=1}^{n} L(F_\Theta(x_i), y_i) \tag{7.1}$$

Dans un réseau de neurones classiques, 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 échantillonées sur leur distribution. L'objectif n'est alors plus de trouver la meilleure valeur des poids, mais la meilleure distribution pour chaque poids.

### 7.1.1 Limite des réseaux de neurones profonds

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 à prendre en compte :

1. Demandent beaucoup de données pour être efficaces
2. Demandent beaucoup de calculs pour être entraîné.
3. Ils n'arrivent pas a représenter l'incertitude de ses prédictions.
4. Ils peuvent facilement être dupés par certains examples "adversaires"
5. Ce sont souvent des boîtes noires, qui manquent donc cruellement de transparance. Ce qui peut nous donnner une difficulter à faire confiance à de tels modèles.

Evidemment, 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**.

### 7.1.2 Définition

**L'apprentissage profond Bayésien** ou **Bayesian Neural Networks** 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 seul combinaison de paramètres $\Theta$, 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**.

### 7.1.3 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, dans 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 $\Omega$. Alors :

$$\begin{cases} \mathbb{P}(X) = \int_{y \in \Omega} \mathbb{P}(X, y) dy \\ \mathbb{P}(X, Y) = \mathbb{P}(X \mid Y)\mathbb{P}(Y) \end{cases} \tag{7.2}$$

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 **priore**. 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 :

$$\mathbb{P}(\Theta \mid \mathcal{D}, F_\Theta) = \frac{\mathbb{P}(\Theta \mid F_\Theta)\mathbb{P}(\mathcal{D} \mid \Theta, F_\Theta)}{\mathbb{P}(\mathcal{D} \mid F_\Theta)}, \tag{7.3}$$

où :

$\mathbb{P}(\Theta \mid F_\Theta)$ est la priore,
$\mathbb{P}(\mathcal{D} \mid \Theta, F_\Theta)$ est la vraisemblance,
$\mathbb{P}(\Theta \mid \mathcal{D}, F_\Theta)$ est la postériore.
Cette expression est en fait caractéristique de l'entraînement d'un modèle avec une approche bayésienne : on donne une priore 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 priore à l'aide des observations que l'on a fait. On a donc une postériore, qui est en fait la priore du prochain paquet, etc. On peut ainsi voir l'entraîenement 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 :

$$\mathbb{P}(y \mid \mathcal{D}_{tr}, x, F_\Theta) = \int \mathbb{P}(y \mid \Theta, \mathcal{D}_{tr}, x, F_\Theta)\mathbb{P}(\Theta \mid \mathcal{D}_{tr}, x, F_\Theta)d\Theta \tag{7.4}$$

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.

Finalement, 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 Bayéienne 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**.

### 7.1.4 Quel est l'intérêt du Bayesian Deep Learning ?

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

Le Bayesian Deep Learning 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 savent lorsqu'ils ne savent pas. 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.

## 7.2 Inférence Variatonnelle

### 7.2.1 Introduction/Définitions

Revenons à l'expression (3) étant caractéristique de l'entraînement d'un modèle avec une approche bayésienne :

$$\mathbb{P}(\Theta \mid \mathcal{D}, F_\Theta) = \frac{\mathbb{P}(\Theta \mid F_\Theta)\mathbb{P}(\mathcal{D} \mid \Theta, F_\Theta)}{\mathbb{P}(\mathcal{D} \mid F_\Theta)}, \tag{7.5}$$

où :

$\mathbb{P}(\Theta \mid F_\Theta)$ est la priore,

$\mathbb{P}(\mathcal{D} \mid \Theta, F_\Theta)$ est la vraisemblance,

$\mathbb{P}(\Theta \mid \mathcal{D}, F_\Theta)$ est la postériore.

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 fait ce que l'on appelle de **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 $\Theta$, où $\nu$ est un **paramètre variationnel**. Notre objectif est de trouver la meilleure valeur de $\nu$ 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**.

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

$$D_{\mathrm{KL}}(P\|Q) = \int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)}\, dx \tag{7.6}$$

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.

Nous avons donc le système d'optimisation suivant :

$$\nu^* = arg \min_\nu D_{\mathrm{KL}}(q(\Theta, \nu)\|\mathbb{P}(\Theta \mid \mathcal{D}, F_\Theta)) \tag{7.7}$$

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 tel :

$$\mathcal{L}_{VI}(\nu) := \int q(\Theta, \nu)\, log(\mathbb{P}(\mathcal{D} \mid \Theta, F_\Theta)) d\Theta - D_{\mathrm{KL}}(q(\Theta, \nu)\|\mathbb{P}(\Theta \mid F_\Theta)). \tag{7.8}$$

On remarque que l'ELBO ne dépend pas de la postérieure de $\Theta$ (notamment, la divergence de Kullback Leibler se fait entre $q$ et la priore de $\Theta$), ce qui la rend donc facilement calculable.

Minimiser (selon $\nu$) la divergence de Kullback-Leibler équivaut à maximiser (selon $\nu$) la *log evidence lower bound*.

*In fine* notre système d'optimisation se réduit à :

$$\nu^* = arg \max_\nu \mathcal{L}_{VI}(\nu) \tag{7.9}$$

## 7.3 Processus Gaussien

### 7.3.1 Définition

Définissons ce qu'est un processus gaussien.

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 infine. 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 $K(s,t) = Cov(X_s, X_t)$. 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 loies finies dimensionnelles de $X$ et donc aussi sa loi [4].

### 7.3.2 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 spécifirons 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.

### 7.3.3 L'intérêt des processus gaussiens dans le Bayesian Deep Learning

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 paramètrique est un modèle dépendant d'un ensemble fini de paramètres, qui vont être optimisés par l'apprentissage d'observation. Un modèle non paramétrique 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 $D_{obs} = (x_{obsi}, y_{obsi})_{i\in\{1,\dots,m\}}$ l'ensemble des données observées et $D_{pred} = (x_{predi}, y_{predi})_{i\in\{1,\dots,n\}}$ 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 :

$$\exists a, b \in \mathbb{R},\ A \in \mathcal{M}_{m,m},\ C \in \mathcal{M}_{n,m},\ B \in \mathcal{M}_{m,n} \mid\ \mathcal{L}(y_{obs}, y_{pred}) = \mathcal{N}\left(\begin{bmatrix} a \\ b \end{bmatrix}, \begin{bmatrix} A & B \\ B^\top & C \end{bmatrix}\right) \quad (7.10)$$

La loi de Bayes nous donnne que :

$$\mathbb{P}(y_{pred} \mid y_{obs}) = \frac{\mathbb{P}(y_{pred}, y_{obs})}{\mathbb{P}(y_{obs})} \quad (7.11)$$

Il vient donc que :

$$\mathcal{L}(y_{pred} \mid y_{obs}) = \mathcal{N}(a + BC^{-1}(y_{obs} - b), A - BC^{-1}B^\top) \quad (7.12)$$

### 7.3.4 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 $\varepsilon$ multiplié par une variance $\sigma_y$. 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 $y(x) = f(g(x)) + \varepsilon\sigma_y$ [5]. On peut donc généraliser cette définition à un processus gaussien à $n$ couches.

## 7.4 Monte Carlo Dropout

### 7.4.1 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. [7]

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 entre $x$ :

$$\hat{y} = \sigma(xW_1 + b)W_2. \quad (7.13)$$

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 $pi \in [0, 1]$ pour $i = 1, 2$. Ainsi, $z_{1,q} \sim Bernoulli(p_1)$ pour $q = 1, \dots, Q$ et $z_{2,k} \sim Bernoulli(p_2)$ pour $k = 1, \dots, K$. Etant donnée une entrée $x$, la proportion $1 - p_1$ d'éléments de l'entrée sont réduit à zéro : $x \circ z_1$ avec $\circ$ le produit de Hadamard. Il vient alors que : $\hat{y} = ((\sigma((x \circ z_1)W_1 + b)) \circ z_2)W_2$.

**La fonction de perte**

Soit $\hat{y}$ la sortie d'un réseau de neurones à $L$ couches et $E(.,.)$ une fonction de perte telle que softmax ou l'erreur quadratique moyenne. Notons $W_i \in \mathcal{M}_{K_i,K_{i-1}}$ et les matrices représentant les poids de notre réseau de neurones sur la couche $i \in \{1,\dots,L\}$ et $b_i \in \mathbb{R}^{K_i}$ les vecteurs de de biais de cette même couche. Notons $y_i$ la donnée observée sortie correspondant à l'entrée $x_i$ pour $i \in \{1,\dots,N\}$ et définissons les ensemble d'entrée et de sortie comme respectivement : $X$ et $Y$. Lors de l'optimisation d'un réseau de neurones, un terle de régularisation est souvent ajouté. Nous utilisons souvent une régularisation $L_2$, dont l'importance est quantifiée par un coefficient $\lambda$, ce qui nous donne l'objectif de minimisation suivant :

$$\mathcal{L}_{dropout} = \min_{W_i,b_i} \frac{1}{N}\sum_{i=1}^{N} E(y_i,\hat{y_i}) + \lambda \sum_{i=1}^{L}(\|W_i\|_2^2 + \|b_i\|_2^2). \tag{7.14}$$

Avant de présenter les liens entre processus Gaussiens profonds et les réseaux de neurones avec Dropout, il faut d'abord brièvement introduire la méthode d'intégration de Monte Carlo.

### 7.4.2 Intégration numérique : méthode de Monte Carlo

Supposons que l'on a une fonction $g : \Omega \to S$ que l'on désire intégrer sur l'espace $I \subset \Omega$ :

$$G = \int_I g(x)dx \tag{7.15}$$

Il peut parfois s'avérer que le calcul pratique de cette intégrale est très difficile. Des méthodes déterministes, telles que la méthode des trapèzes, des rectangles ou encore de Simpson peuvent être efficaces pour approximer cette intégrale. Toutefois, ces méthodes ont des limites dans des cas où l'on intègre sur un espace multi-dimensionnel.

Supposons qu'il existe une variable aléatoire $X$ et qu'il existe une expression de l'espérance de $g(X)$ telle que :

$$G = \mathbb{E}(g(X)) = \int_\Omega g(x) f_X(x)dx, \tag{7.16}$$

Avec $f_X$ la fonction densité de probabilité de X.

Produisons un échantillon i.i.d de X, $(x_1, x_2, \dots, x_N)$ d'après la densité $f_X$ sur $I$. Calculons un estimateur de $G$ à partir de cet échantillon.

Pour ce faire, nous pouvons utiliser la moyenne empirique, comme le présume la loi des grands nombres, qui est ici applicable de par le caractère i.i.d de notre échantillon. Nommons $\hat{G}$ cet estimateur de $G$, il vient que :

$$\hat{G} = \frac{1}{N}\sum_{i=1}^{N} g(x_i). \tag{7.17}$$

Cet estimateur est par ailleurs non biaisé : $\mathbb{E}(\hat{G}) = G$.

### 7.4.3 Le Dropout, une approximation Bayésienne.

Le but de cette partie est de prouver qu'un réseau de neurones dans lequel on applique un dropout à chaque couche cachée est une approximation d'un processus gaussien. [8]. Elle se base essentiellement sur l'article [7].

Soit :

$\mathcal{D}_{obs} = (X, Y) \in \mathbb{R}^{N*Q} \times \mathbb{R}^N$ l'ensemble des données observés,

$X = (x_1, \dots, x_N),\ Y = (y_1, \dots, y_N)$,

les $(x_i, y_i)_{i \in \{1,\dots,N\}}$ indépendants deux à deux.

Commençons par définir un opérateur de covariance. Posons $\sigma(.)$ une fonction non-linéaire telle que $ReLU$ ou $TanH$. Nous définissons l'opérateur de covariance $K(x, y)$ par :

$$K(x, y) = \int \mathbb{P}(w)\mathbb{P}(b)\sigma(w^\top x + b)\sigma(w^\top y + b)dwdb,\ \forall (x, y) \in \mathbb{R}^2, \tag{7.18}$$

avec $\mathbb{P}(w)$ une distribution gaussienne multivariée centrée en 0 de dimension $Q$, et une certaine distribution $\mathbb{P}(b)$. Il est vraisemblablement trivial de montrer que cet opérateur de covariance est valide.

Montrons brièvement qu'un processus Gaussien profond à $L$ couches et de fonction de covariance $K(x, y)$ peut être approché par une distribution variationnelle sur chaque composante de la décomposition spectrale notre opérateur de covariance. Cette décompostion spectrale permet d'associer chaque couche du processus Gaussien à une couche de neurones cachés.

Posons $W_i$ une matrice aléatoire de dimension $K_{ii-1}$ pour chaque couche $i$, et définissons $\omega = \{W_i\}_i i = 1^L$. Nous donnons à chacune de ces matrices $W_i$ une distribution $p(w)$ *à priori*. De plus, posons un vecteur $m_i$ de dimension $K_i$ pour chaque couche $i$ de notre processus gaussien profond. La probabilité de prédiction de notre processus gaussien profond est la suivante :

$$p(y \mid x, X, Y) = \int p(y \mid x, \omega)p(\omega \mid X, Y)d\omega \tag{7.19}$$

avec : $x, y \in \mathbb{R}^Q \times \mathbb{R}$ un couple entrée sortie de prédiction.

On sait par ailleurs que :

$$p(y \mid x, \omega) = \mathcal{N}(y, \hat{y}(x, \omega), \tau^{-1}) \tag{7.20}$$

$$\hat{y}(x, \omega = \{W_1, \dots, W_L\} = K_L^{-\frac{1}{2}} W_L \sigma(\dots K_1^{-\frac{1}{2}} W_2 \sigma(W_1 x + m_1) \dots), \tag{7.21}$$

Avec $\tau > 0$ un paramètre de précision. La postérieure $p(\omega \mid X, Y)$ est très difficile à calculer. Nous alons donc chercher à l'apporximer par une méthode d'inférence variationnelle. Pour ce faire, utilisons une distribution $q(\omega)$ sur des matrices, dont les colonnes sont aléatoirement nulles :

$$W_i = M_i.diag([z_{i,j}]_{j=1i}^K), \tag{7.22}$$

$$z_{i,j} \sim Bernoulli(p_i),\ i = 1, \dots, L,\ j = 1, \dots, K_{i-1}, \tag{7.23}$$

où $(p_i)$ sont des probabilités et $M_i \in \mathcal{M}_{K_i,L}$ sont des matrices faisant office de paramètres variationnels. La variable binaire $z_{i,j} = 0$ correspond au fait que l'activation du neurone $j$ de la

couche $i$ vaut 0.

Maintenant que nous avons notre distribution variationnelle $q(\omega)$, minimisons la divergence de Kullback-Leibler entre l'approximation de la postérieure $q(\omega)$ et la postérieure $p(\omega \mid X, Y)$. Comme évoqué précédemment, minimiser la divergence de Kullback-Leibler équivaut à maximiser la *log evidence lower bound*. D'où notre objectif de minimisation (de l'opposé du log ELBO) :

$$\mathcal{L}_{GP-MC} = \min_{p_i, M_i,\ i \in \{1,\dots,L\}} - \int q(\omega) log p(Y \mid X, \omega) d\omega + KL(q(\omega) \| p(\omega)). \tag{7.24}$$

Comme nous savons que les $(x_i, y_i)$ sont indépendant deux à deux, nous pouvons réécrire le premier terme comme une somme :

$$\int q(\omega) log p(Y \mid X, \omega) d\omega = -\sum_{n=1}^{N} \int q(\omega) log(p(y_n \mid x_n, \omega)) d\omega \tag{7.25}$$

et nous pouvons approcher chaque terme dans la somme par une intégration de Monte Carlo avec un unique échantillon $\hat{\omega_n} \sim q(\omega)$ afin d'avoir un estimateur non biaisé : $-log(p(y_n \mid x_n, \hat{\omega_n}))$. Nous approchons également le second terme de l'équation :

$$KL(q(\omega) \| p(\omega)) \approx \sum_{i=1}^{L} \left( \frac{p_i l^2}{2} \| M_i \|_2^2 + \frac{l^2}{2} \| m_i \|_2^2 \right), \tag{7.26}$$

avec $l$ une certaine échelle de longueur définie à *priori*. Etant donné la précision $\tau$, nous multiplions le résultat par la constante $\frac{1}{\tau N}$ pour obtenir l'objectif :

$$\mathcal{L}_{GP-MC} \propto \frac{1}{N} \sum_{n=1}^{N} \frac{-log(p(y_n \mid x_n, \hat{\omega_n}))}{\tau} + \sum_{i=1}^{L} \left( \frac{p_i l^2}{2\tau N} \| M_i \|_2^2 + \frac{l^2}{2\tau N} \| m_i \|_2^2 \right). \tag{7.27}$$

En posant :

$$E(y_n, \hat{y}(x_n, \hat{\omega_n})) = -log(p(y_n \mid x_n, \hat{w_n}))/\tau, \tag{7.28}$$

On retrouve finalement $\mathcal{L}_{dropout}$ en utilisant une bonne valeur de $\tau$ et de $l$.

# Chapitre 8

# Les résultats

## 8.1 Mise en place des algorithmes

Les algorithmes ont été programmés sur `Python` à l'aide de plusieurs librairies :

`Pandas` : pour importer et manipuler les données sous forme de tableau.

`Matplotlib.pyplot` : pour créer des graphiques

`Keras (Tensorflow)` : pour utiliser des modèles de deep learning (réseaux de neurones).

`Scikit-learn` : pour utiliser des modèles de machine learning (régression linéaire, ridge, Support Vector Regression) ainsi que des fonctions d'optimisation d'hyperparamètres (`GridSearchCV`, `RandomizedSearchCV`).

`Scikeras` : pour convertir les modèles Keras en modèles Scikit-learn afin d'optimiser les hyperparamètres.

Les données [11] ont tout d'abord été importées à l'aide de la librairie `Pandas`. Elles ont par la suite été mélangées afin de garantir l'homogénéité entre le set d'entraînement et le set de test. Le chapitre 3 avait souligné précédemment que les variables avaient des plages de valeurs différentes. Elles ont donc toutes été normalisées entre 0 et 1. Les données de test constituent 20% de l'ensemble des données. Les 80% vont par conséquent constituer les données d'entraînement.

### 8.1.1 Modèles de machine learning

Pour les modèles de machine learning suivants : Ridge, Support Vector Regression, les hyperparamètres optimaux ont été recherchés à l'aide de la fonction `GridSearchCV` (`Scikit-learn`) avec une validation croisée à 5 plis sur le set d'entraînement. Voici les hyperparamètres optimaux trouvés (avec les notations précédentes) :

**Régression Ridge :**

Degré du polynôme : 1 (recherché dans `linspace(1,10, 10)`)

$\lambda = 0.1$ (recherché dans `logspace(-5, 5, 11)`)

**Support Vector Regression :**

Noyau : Gaussien (recherché dans [Linéaire, Polynomial, Sigmoïdal, Gaussien])

$C = 0.744$, (recherché dans `linspace(0.5,10,40)`)

$\epsilon = 0.03461$, (recherché dans `linspace(10**(-6),0.3, 27)`)

### 8.1.2 Réseau de neurones profond

Le réseau de neurones profond constitue une couche d'entrée égale à 7 (car il y a 7 variables en entrée) et une couche de sortie égale à 1 (car il s'agit d'un problème de régression). L'optimiseur du modèle s'agit de *Adam*. Les hyperparamètres optimaux ont été recherchés à l'aide de la fonction `RandomizedSearchCV` (`Scikit-learn`) avec une validation croisée à 5 plis sur le set d'entrainement. Le nombre maximal d'itérations est fixé à 5000. La libraire `Scikeras` a donc du être utilisée afin d'utiliser les fonctionnalités de `Scikit-learn` sur des modèles de deep learning. Les hyperparamètres suivants ont été cherchés :

Le nombre de couches cachées : entre 1 et 6.

Le nombre de neurones par couche cachée : ici régie par une loi uniforme entre 32 et 320

Le taux d'apprentissage de l'optimisateur : recherché dans `logspace(-7, -1, 7)`

Le nombre d'époques est réglé à 2000. Voici les hyperparamètres optimaux obtenus pour le réseau de neurones :

**Nombre de couches cachées :** 4

**Couche cachée 1 :** 115 neurones

**Couche cachée 2 :** 40 neurones

**Couche cachée 3 :** 180 neurones

**Couche cachée 4 :** 35 neurones

**Taux d'apprentissage de l'optimisateur** : $10^{-5}$

La figure 8.1 nous permet d'affirmer que nos hyperparamètres sont bien adaptés au modèle, car l'erreur d'entraînement tend vers 0 et l'écart entre l'erreur de validation et l'erreur d'entraînement est très faible.

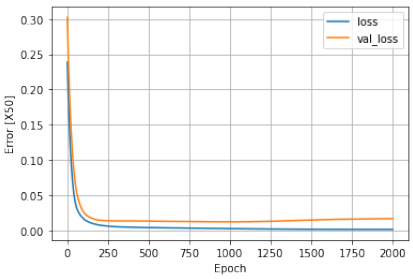



FIGURE 8.1 – Erreur d'entraînement (orange) et erreur de validation (bleue) en fonctions des époques.

## 8.2 Erreur moyenne quadratique et coefficient de détermination des modèles :

Pour estimer l'erreur moyenne quadratique, chaque modèle a été entrâiné sur 80% du set d'entraînement, les 20% restants sont utilisés comme un set de validation.

Le coefficient de détermination est régie par la formule suivante :

$$R^2 = 1 - \frac{\sum_i^n (y_i - \hat{y_i})^2}{\sum_i^n (y_i - y_{moy})^2}, \tag{8.1}$$

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.

Les coefficients de détermination $R^2$ et les écart-types des modèles ont été estimés à l'aide de la fonction `cross_val_score (scikit-learn)` sur l'ensemble du jeu de données avec une validation croisée à 5 plis. Comme il s'agit d'une fonction `scikit-learn`, il était une nouvelle fois nécessaire de convertir notre réseaux de neurones à l'aide de la libairie `Scikeras`. Les résultats sont affichés dans le tableau suivant (ici, le réseau de neurones profond est nommé "DNN", pour *Deep Neural Network*).

| Modèles | MSE | $R^2$ | Std |
|---|---|---|---|
| *Rég.Lin.* | 0.024832 | 0.478421 | 0.212597 |
| *Ridge* | 0.024155 | 0.483206 | 0.199264 |
| *SVR* | 0.012641 | 0.728896 | 0.125059 |
| *DNN* | 0.007832 | 0.689225 | 0.108839 |

Comme nous pouvons le voir, le réseau de neurones profond possède l'erreur moyenne quadratique et l'écart type les plus faibles. La Support Vector Regression est le modèle avec le coefficient de détermination le plus proche de 1 et le deuxième modèle le plus performant selon le critère MSE. Ces deux modèles peuvent donc être considéres comme les meilleurs pour notre problème.

## 8.3 Analyse des résidus de la SVR et du réseau de neurones

Il était expliqué dans le chapitre 4 que les résidus de nos modèles doivent suivre une loi normale centrée en 0. Vérifions cela en traçant l'estimation de densité par noyau de nos deux meilleurs modèles (Figure 8.2).

Comme nous pouvons le voir, nos deux modèles forment bel et bien une gaussienne à peu près centrée en 0. Nos modèles semblent donc bien adapté au problème.

## 8.4 Esimtation de l'incertitude pour le réseau de neurones profond et la SVR

Concernant le réseau de neurones, nous pouvons utiliser la méthode de *Monte Carlo Dropout* pour estimer l'incertitude de chacune de nos prédictions en appliquant une couche de *Dropout* entre

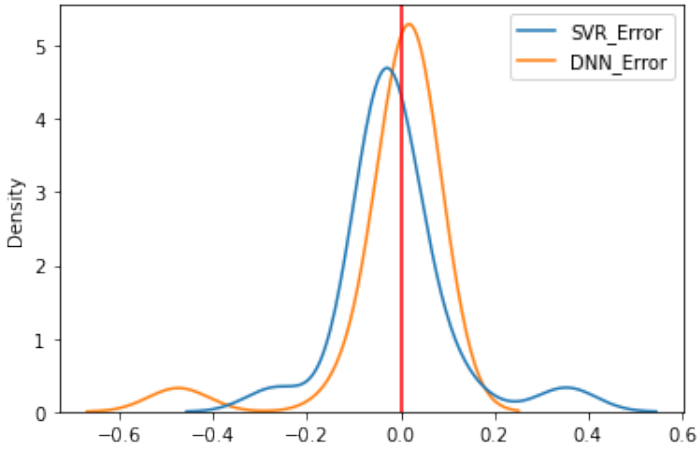



FIGURE 8.2 – Estimation de densité par noyau des résidus du réseau de neurones profond (orange) et de la SVR (en bleu). Le trait rouge marque la droite $x = 0$.

chacune de nos couches à l'entraînement et à la phase de test. La figure 8.3 montre l'incertitude du modèle pour la première valeur de test de $x_{50}$ Nous constatons que le modèle est plutôt 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 $x_{50}$.

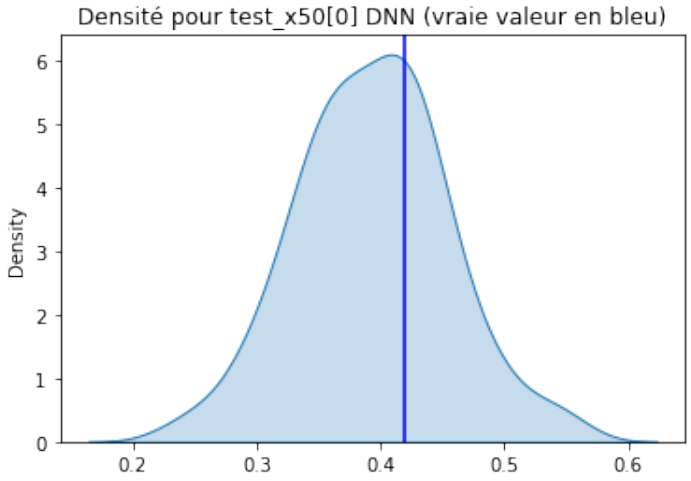



FIGURE 8.3 – Densité de probabilité du réseau de neurones pour estimer la première valeur de test de $x_{50}$

La Support Vector Regression n'étant pas un réseau de neurones profond, 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. Nous pouvons cependant comparer l'estimation de densité de sa prédiction de $x_{50}$ par rapport à l'estimation de densité de $x_{50}$.

Analysons maintenant l'estimation de densité des prédictions de nos modèles par rapport à la vraie estimation de densité de la sortie. Le réseau de neurones semble trop prévoir aux alentours de $x = 0.2$ et pas assez dans les autres zones (figure 8.4). La SVR semble avoir le même problème sa densité de probabilité est tout de même plus proche de celle de $x_{50}$ (figure 8.5).

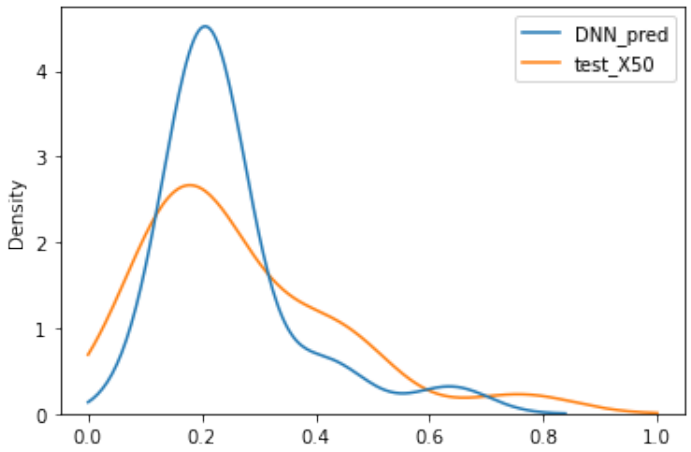



FIGURE 8.4 – Estimation de densité des prédictions de $x_{50}$ du réseau de neurones (bleu) et estimation de densité de $x_{50}$ (orange).

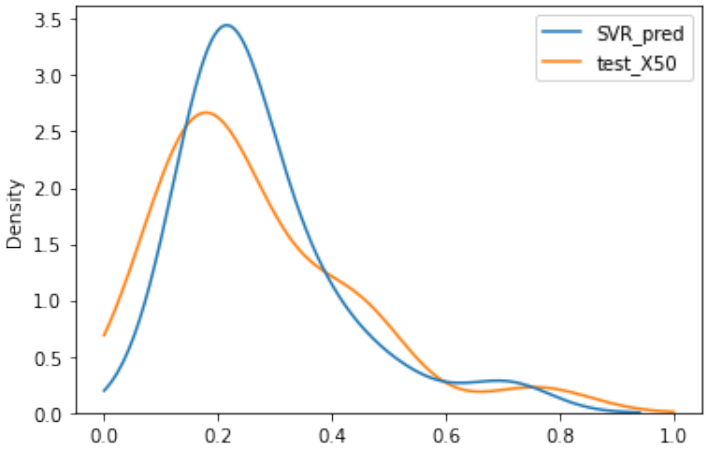



FIGURE 8.5 – Estimation de densité des prédictions de $x_{50}$ de la SVR (bleu) et estimation de densité de $x_{50}$ (orange).

# Chapitre 9

# Conclusions et travail à faire pour le prochain semestre

## 9.1 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 à l'aide de modèles d'apprentissage automatique. Parmi les modèles mis au point, le réseau de neurones profond et la Support Vector Regression montrent les meilleures performances. Toutefois, l'incertitude estimée sur ces modèles montrent que ces derniers sont encore perfectibles.

## 9.2 Travail à faire pour le prochain semestre

Le travail à faire pour le prochain semestre peut s'orienter dans plusieurs directions. Nous pourrions nous orienter vers l'amélioration des performances de ces modèles, notamment en procédant à de la *data augmentation*, c'est-à-dire, augmenter le nombre de données. Certains algorithmes de deep learning comme l'autoencodeur variationel (VAE) permettent d'agrandir le nombre de données sans en fournir d'avantage. On pourrait également chercher à fournir des données supplémentaires aux observations déjà existantes, comme par exemple des images satellites des lieux où a lieu la fragmentation avec des localisations des trous dans lesquels sont insérés les explosifs. On pourrait alors procéder à un réseau de neurones qui traiterait en parallèle les données tabulaires avec un *Multilayer Perceptron* et les images avec un *Réseau de Neurones Convolutionnel*.

Une autre piste serait d'appliquer le savoir-faire acquis dans ce problème à d'autres problèmes de fragmentation, qui seraient peut-être plus en lien avec les travaux de Madalina Deaconu et Antoine Lejay. Cependant cette application reste encore à définir.

# 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] Lucian Beznea, Madalina Deaconu, and Oana Lupaşcu. Branching processes for the fragmentation equation. *Stochastic Processes and their Applications*, 125(5) :1861–1885, May 2015. doi:10.1016/j.spa.2014.11.016.

[3] 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.

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

[5] Andreas C. Damianou and Neil D. Lawrence. Deep gaussian processes, 2012. URL : https://arxiv.org/abs/1211.0358, doi:10.48550/ARXIV.1211.0358.

[6] Madalina Deaconu and Antoine Lejay. Probabilistic representations of fragmentation equations. December 2021. Working paper or preprint. URL : https://hal.inria.fr/hal-03483448.

[7] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation : Appendix, 2015. URL : https://arxiv.org/abs/1506.02157, doi:10.48550/ARXIV.1506.02157.

[8] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation : Representing model uncertainty in deep learning, 2015. URL : https://arxiv.org/abs/1506.02142, doi:10.48550/ARXIV.1506.02142.

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

[10] M Chris Jones, James S Marron, and Simon J Sheather. A brief survey of bandwidth selection for density estimation. *Journal of the American statistical association*, 91(433) :401–407, 1996.

[11] 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.

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

[13] Emanuel Parzen. On estimation of a probability density function and mode. *The annals of mathematical statistics*, 33(3) :1065–1076, 1962.

[14] Bernard W Silverman. *Density estimation for statistics and data analysis*. Routledge, 2018.

[15] Frederic Sur. Introduction à l'apprentissage automatique. page 172.