UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À

L'UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN MATHÉMATIQUES ET INFORMATIQUE

APPLIQUÉES

PAR

TETE Akpedje

Détection des composants critiques dans les systèmes orientés objet par l'apprentissage automatique profond et les métriques orientées objet

Novembre 2022

RÉSUMÉ

La rude concurrence dans le domaine du développement d'applications logicielles impose aux entreprises du domaine un rythme de travail accéléré, les empêchant ainsi de consacrer le temps nécessaire à chaque étape du développement logiciel. Par ailleurs, la complexité croissante des spécifications des clients affecte grandement la fiabilité et la qualité du produit logiciel livré. Ce manque de fiabilité et de qualité englobe, entre autres, la présence des fautes dans les systèmes logiciels. Certaines fautes causent des dysfonctionnements, qui ne peuvent être tolérés dans certains domaines tels que le transport et la médecine qui mettent en jeux des vies humaines. Le meilleur moyen pour identifier toutes ces fautes passe par les tests, une des étapes cruciales du cycle de développement logiciel. Dans l'impossibilité d'effectuer les tests sur l'ensemble des systèmes logiciels (vu leur taille), les compagnies de développement ont la lourde tâche de cibler les classes logicielles (dans le cas d'un système orienté objet) les plus à risque (les plus prioritaires) afin de réduire à un niveau raisonnable les ressources qui y sont consacrées. Nos travaux s'inscrivent dans cette logique d'aide au choix des classes prioritaires (détection des composants (classes) critiques dans les systèmes orientés objet) en s'appuyant sur les techniques d'apprentissage profond et les données des métriques orientées objet.

Pour ces travaux, nous avons conçu trois modèles d'apprentissage automatique; le premier est dédié à la prédiction du nombre de fautes, le deuxième à la prédiction de la présence ou non de fautes et le troisième à la prédiction de la fréquence de fautes (non fautive, faiblement fautive et fortement fautive). Nous avons comparé les performances du réseau de neurones artificiels (RNA) respectivement à la régression linéaire (RL), à la régression logistique binaire (Rlo) et à la régression logistique multi classes (RLoM). Nous avons utilisé les données de six versions du système POI, de la version 4.0.0 à la version 5.0.0, auxquelles nous avons appliqué les

techniques d'auto-encodage et d'ACP afin d'évaluer leur impact sur les performances de nos modèles. Certains de nos modèles ont fourni, en moyenne, des performances très significatives de 89%.

ABSTRACT

The tough competition in the software development field imposes on companies an accelerated pace of work, thus preventing them from devoting the time and resources to each stage of software development. Furthermore, the increasing complexity of customer specifications greatly affects the reliability and quality of the delivered software product. This lack of reliability and quality includes, among other things, the presence of faults in the software systems. Some faults cause malfunctions, which cannot be tolerated, especially in the fields of transport and medicine, putting human lives at stake. The best way to identify all these faults is through testing, one of the critical stages of the software development cycle. Unable to perform tests on the entire software system (given their size), software companies have the difficult task of targeting the riskiest software classes (in the case of an object-oriented system) in order to lower, to a reasonable level, the resources devoted to it. Our work is part of this logic of supporting the class prioritization, by detecting critical components (classes) in object-oriented systems, relying on deep learning techniques with OO metrics data.

During this work, we designed three machine learning models; the first is dedicated to the number of faults prediction, the second to the faults presence prediction and the third to the faults frequency prediction (non-faulty, weakly faulty and highly faulty). We compared the performance of ANN to the classic linear regression (RL), binary logistic regression (RL0) and multi-class logistic regression (RLoM) respectively. We used data from six versions of the POI system, from version 4.0.0 to 5.0.0, to which we applied the auto-encoder and the PCA techniques in order to evaluate their impact on the performance of our models. Some of our models provided, on average, as significant performances as 89%.

REMERCIEMENTS

Mes respects, ma considération et ma reconnaissance à mes directeurs de recherche, les professeurs Fadel Toure et Mourad Badri pour leur motivation, leur rigueur, leur disponibilité et leur enthousiasme afin que je puisse relever ce défi.

Mes sincères et vifs remerciements:

D'abord, aux membres du jury qui ont accepté de consacrer leur temps et énergie pour l'évaluation de ce modeste travail.

Ensuite, à mes parents pour leur soutien continu et pour leur éducation de la patience, de l'importance du travail, surtout du travail bien fait peu importe sa nature et, malgré les difficultés.

À mon conjoint et ma fille, pour leur patience et leur motivation, par leur voix et leur présence, malgré la distance.

À mes frères et sœurs, pour leurs encouragements et leurs soutiens indéfectibles.

Enfin, à mes amis et mes proches au Canada, pour leur accueil, leurs conseils et leurs efforts continus, quant à mon intégration depuis mon arrivée.

Que toute personne, m'ayant aidé de près ou de loin, trouve en ce mémoire, le fruit de leurs efforts dont je suis la porte-parole.

TABLE DES MATIÈRES

RÉSUMÉ.....I

ABSTRACT.....III

REMERCIEMENTS.....IV

TABLE DES MATIÈRES.....V

LISTE DES TABLEAUX.....VIII

LISTE DES FIGURES.....IX

CHAPITRE1. Introduction.....10

1.1 Contexte.....12

1.2 Problématique.....13

1.3 Approche.....14

1.4 Organisation du mémoire.....15

CHAPITRE2. État de l'art.....16

2.1 Métriques logicielles.....16

2.2 Les algorithmes d'apprentissage profond.....17

2.3 Prédiction des fautes logicielles.....19

2.3.1 Prédiction de la probabilité de fautes.....19

2.3.2 Prédiction du nombre de fautes logicielles.....20

2.3.3 Prédiction de la sévérité des fautes.....21

CHAPITRE3. Méthodologie de recherche.....23

3.1 Questions de recherche 24

3.2 Description des algorithmes d'apprentissage automatique utilisés 25

3.2.1 Le Réseau de Neurones artificiels (RNA) 26

3.2.1.1 Les composants d'un RNA 27

3.2.1.2 Les couches d 'un RNA 30

3.2.1.3 Fonctionnement d'un RNA 31

3.2.2 La régression linéaire 32

3.2.3 La régression logistique 33

3.2.4 Le système POI 33

3.2.5 Les métriques de Chidamber et Kemerer 34

3.2.6 Collecte, préparation et prétraitement de données 36

3.2.7 Statistiques descriptives 39

3.2.8 Entrainement des algorithmes d'apprentissage automatique 42

3.2.8.1 Le test Interversion (TIV) 44

3.2.8.2 Le test interversion avec ACP (TIVA) 45

3.2.8.3 Le test Inter-version avec Encodeur (TIVE) 49

3.2.9 Structure des réseaux de neurones profonds 52

3.2.9.1 Structure du Réseau de Neurones Artificiels (RNA) 52

3.2.9.2 Structure de l'auto encodeur 53

3.2.10 Mesures de performances des modèles 53

3.2.11 Outils logiciels de collecte et de prétraitements des données 53
3.2.11.1 GitHub 54
3.2.11.2 Apache Software Fondation (ASF) Bugzilla 54
3.2.11.3 Excel de la suite Microsoft Office 55
3.2.11.4 IntelliJ 55
3.2.11.5 Metrics Reloaded 55
3.2.11.6 Tanagra 55
CHAPITRE4. Résultats des expérimentations et interprétations 56
4.1 Prédiction du nombre de fautes. 56
4.2 Prédiction de la présence et de la fréquence de fautes 58
4.2.1 Prédiction de la présence de fautes 58
4.2.2 Prédiction de la fréquence de fautes 59
4.2.3 Réponses aux questions 3 et 4 de recherche 60
CHAPITRE5. MENACE DE VALIDITÉ ET CONCLUSION 61
5.1 Menace de validité 61
5.1.1 Menace de validité externe 61
5.1.2 Menace de validité conceptuelle 61
5.2 Conclusion 61

LISTE DES TABLEAUX

Tableau 1: Statistiques descriptives des six versions du système POI 41

Tableau 2: Pourcentage des classes fautives par rapport à la taille du système 41

Tableau 3: Pourcentage des différentes catégories de classes fautives 42

Tableau 4: Valeurs propres et valeurs cumulatives de l'ACP 48

Tableau 5: Corrélation des variables indépendantes avec les composantes principales. 48

Tableau 6: Résultat d'exécution des auto encodeurs 52

Tableau 7: Taille et fonction d'activation par type de modèle 52

Tableau 8: Prédiction du nombre de fautes 58

Tableau 9: Prédiction de la présence de fautes 59

Tableau 10: Prédiction de la fréquence de fautes logicielles 59

LISTE DES FIGURES

Figure 1: Composants d'un réseau de neurones [88] 27

Figure 2: Structure générale d'un neurone 27

Figure 3: Structure détaillée d'un neurone 28

Figure 4: Variation du nombre total de classes et de classes fautives 40

Figure 5: Test inter-version pour la classification 44

Figure 6: Test inter-version pour la régression 45

Figure 7: Test inter-version avec ACP pour la classification 45

Figure 8: Test inter-version avec ACP pour la régression 46

Figure 9: Test interversion avec encodeur pour la classification 49

Figure 10: Test interversion pour la régression 49

Figure 11: Structure de l'auto encodeur 51

CHAPITRE1. INTRODUCTION

La nature délicate des systèmes logiciels de nos jours, due à la sensibilité élevée et à la complexité croissante des spécifications des clients, rend plus complexe les responsabilités des développeurs [1, 2] à concevoir des systèmes logiciels de qualité. Un système logiciel de qualité est un système répondant aux spécifications établies [3], suivant un standard de développement [4] et qui est tolérant aux fautes [5]. Un composant logiciel dans le cadre d'un système orienté objet (OO) est un module, une classe ou une méthode [6, 7]. Un composant critique est un composant dans lequel la présence de fautes empêche le bon fonctionnement du système logiciel.

Afin de parvenir à un certain niveau de qualité logicielle, les compagnies de développement logiciel déploient un nombre considérable de ressources humaines et financières [8], en appliquant les activités d'assurance qualité [9] pour identifier les composants contenant des fautes. Une fois ces composants critiques identifiés, une grande partie des ressources de l'entreprise est consacrée à leurs corrections [8] afin de garantir des produits logiciels fiables qui répondent aux spécifications du client et qui respectent les délais de livraison. L'assurance qualité regroupe un ensemble d'activités [10], dont la vérification-validation (VV) [1, 11]. La VV est le processus par lequel les fonctionnalités du système logiciel sont comparées aux spécifications du client afin d'assurer leur conformité [1, 11]. La VV regroupe un ensemble de techniques dont les tests dans le domaine logiciel. Les tests logiciels sont constitués de méthodes statiques et dynamiques [12]. Les méthodes statiques renseignent sur la structure du programme tandis que les méthodes dynamiques se focalisent sur son exécution [12]. Assurer la qualité du produit logiciel revient, entre autres, à tester donc chaque composant du système et à identifier ceux qui sont fautifs et par extension, ceux qui sont critiques. Les tests exhaustifs sont réalisables pour des systèmes de petite taille, mais sont déraisonnables, difficiles et complexes pour des systèmes logiciels de taille importante à cause du temps et des ressources

qui y sont investis [13]. Comment identifier donc les composants critiques sans toutefois tester chaque composant du système logiciel ? Pour répondre à cette question, des recherches antérieures [14, 15] ont considéré la granularité des systèmes logiciels orientés objet (OO) au niveau de la classe.

Dans ces recherches antérieures, différentes approches ont été explorées : une première approche, par la classification binaire [16-18] ou multi classes ou multinomiale [19]. La classification binaire consiste à classer les classes logicielles en deux groupes (fautifs ou non fautifs), tandis que la classification multi classes consiste à les répartir en plusieurs groupes [19, 20]. Pour cette dernière, les groupes sont obtenus en se basant sur des critères préalablement définis. Ces critères tiennent compte des données et des objectifs de recherche, et dépendent du domaine de recherche. La sévérité des fautes [21, 22] a été le critère le plus souvent utilisé dans le domaine du logiciel. La sévérité d'une faute informe sur son niveau de gravité (faible, moyenne ou forte par exemple).

La classification binaire [16-18] ne fournit certes pas d'informations sur le nombre de fautes ni sur leur gravité. Néanmoins, elle contribue à identifier des classes fautives (composants fautifs). La classification multi classes [19] est plus utile et plus fine que la précédente, étant donné qu'elle fournit plus de possibilités de classification. Ces dernières donnent plus d'informations pour la priorisation des actions correctives au cours des tests, en tenant compte des caractéristiques des différents groupes. Cependant, elle ne renseigne pas sur le nombre de fautes contenues dans les classes logicielles des différents groupes.

D'autres recherches ont élaboré une approche de prédiction du nombre de fautes [23-25] des classes logicielles. Cette approche permet d'identifier certes les classes fautives et leur nombre de fautes, mais ne fournit pas d'informations sur le niveau de sévérité des fautes.

1.1 Contexte

Un logiciel est un ensemble de lignes de code écrites par des développeurs pour automatiser des fonctions manuellement ou semi manuellement réalisées. Ces fonctions constituent les spécifications auxquelles le logiciel doit répondre. Afin de s'assurer que le logiciel répond aux spécifications, les développeurs réalisent différents tests [26, 27] : test unitaire, test d'intégration, test de validation, etc. Les tests unitaires sont réalisés sur chaque entité qui constitue l'unité du système logiciel [26, 28] comme les classes, dans le cadre des systèmes (OO) [29]. Les tests d'intégration valident l'interaction et la collaboration entre les différentes classes logicielles [26]. Les tests de validation sont réalisés avec le client, interviennent après l'intégration des classes les unes aux autres et valident le fonctionnement du système dans son ensemble [26]. Les tests sont importants dans le cycle de développement d'un logiciel [30]. Cependant, ils requièrent d'importantes ressources pour leur conception et leur exécution [31] et consomment environ 40 à 50 % de l'effort de développement [32]. La plupart des compagnies de développement réduisent la quantité de tests à réaliser afin de limiter les ressources allouées à cette activité [33]. Cette réduction des ressources influence aussi bien la qualité que la fiabilité du produit logiciel [33]. Comment alors réduire à un niveau raisonnable les ressources allouées aux tests tout en produisant des logiciels fiables et de qualité ?

Pour répondre à cette question, il est important de disposer d'approches identifiant de façon fiable, efficace et automatique les composants potentiellement fautifs dans les systèmes logiciels. Avec ces approches, les tests ne vont cibler que les composants identifiés, réduisant ainsi le nombre de tests à effectuer. Cette réduction impacte à son tour le temps et les ressources allouées aux tests et vient répondre (ainsi) à l'un des objectifs des compagnies de développement : obtenir des logiciels de qualité, fiables à moindre coût [34, 35].

Notre travail s'inscrit dans ce contexte, dans l'identification et la détection des composants critiques dans les systèmes logiciels orientés objet en s'appuyant sur les techniques

d'apprentissage profond [36] utilisant les données des métriques OO [37]. Il s'agit d'une suite logique aux travaux effectués dans la thèse [38] qui a opté pour les modèles d'apprentissage classique : réseaux bayésiens [39], forêts aléatoires [40], KNN [41] et C4.5 [42]. Le but de nos travaux est de concevoir une approche identifiant automatiquement les classes critiques qui vont prioritairement nécessiter des activités de VV (les tests). Cette orientation des tests réduit les ressources qui leur sont allouées et contribuera à livrer un produit fiable et de qualité, conformément au planning et aux spécifications du client.

1.2 Problématique

Une faute logicielle est un défaut, le résultat d'une erreur (un manquement dans une des étapes du cycle de développement logiciel) [43, 44] qui cause la défaillance d'un système logiciel. La sensibilité et la précision des fonctionnalités dans certains domaines comme le transport [45] et la santé [46], mettant en jeu des vies humaines, requièrent des systèmes logiciels fiables exemptés (le plus possible) de toutes fautes. Comment pouvons-nous obtenir des produits logiciels fiables contenant le moins de fautes possible tout en prenant en compte le fait que les fautes font partie intégrante des systèmes logiciels (elles sont introduites au cours du cycle de vie)? Pour répondre à cette question, il faut d'abord des outils permettant de détecter et d'identifier les composants critiques qui contiennent potentiellement ces fautes et ensuite les corriger. Les tests sont les moyens les plus souvent utilisés pour identifier les fautes logicielles. Cependant, plus la taille du système est grande, plus les tests requièrent une quantité de ressources (humaines, et matérielles) importante. Comment retrouver donc les composants critiques tout en réduisant à la fois le nombre de composants logiciels à tester et les ressources qui y sont consacrées ? Une étape de priorisation (de préférence automatisée) est donc indispensable pour déceler les composants critiques avant les tests.

1.3 Approche

Afin de trouver une solution à la problématique posée, nous allons concevoir des modèles d'apprentissage profond [36], basés sur les réseaux de neurones artificiels (RNA)[47] et les métriques orientées objet [37], pour la détection et l'identification des composants critiques dans les systèmes logiciels (les classes logicielles critiques). Nous avons implémenté nos modèles suivant les 2 types de classification (binaire [16-18], multi classes [19]) et la régression. La classification binaire catégorise les classes logicielles en deux groupes: fautives et non fautives. Trois groupes de classification sont considérés lors de la classification multi-classes dont le groupe non fautif, le groupe faiblement fautif et le groupe fortement fautif afin d'évaluer la fréquence des fautes dans les classes logicielles. Ces trois groupes de classes sont obtenus en tenant compte de la moyenne du nombre de fautes contenues dans les différentes versions du système logiciel étudié. La régression prédit le nombre de fautes contenues dans les classes logicielles.

Nous avons aussi utilisé des modèles de régression et de classification basés sur les algorithmes d'apprentissage automatique traditionnels, afin de comparer leurs performances à celles du RNA. Suivant les différentes approches abordées, nous avons appliqué: la régression logistique binaire [48], la régression multi classes [49] et la régression linéaire [50, 51].

1.4 Organisation du mémoire

Cinq chapitres principaux nous permettent de présenter nos travaux de recherche:

Ce premier chapitre a introduit, de façon générale, l'importance de l'identification et de la détection des composants fautifs dans les systèmes logiciels. Il a expliqué aussi le contexte de nos travaux, a défini la problématique et a présenté notre approche pour répondre à cette problématique.

Le deuxième chapitre mettra l'accent sur certains travaux antérieurs réalisés en lien avec notre sujet de recherche. Il présentera, dans un premier temps, les travaux basés sur les métriques logicielles orientées objet. Ensuite, il exposera les travaux qui ont utilisé les algorithmes d'apprentissage profond en génie logiciel. Enfin, il décrira les travaux traitant de la prédiction des fautes.

Le troisième chapitre, dédié à la méthodologie, présentera nos questions de recherche et décrira les algorithmes d'apprentissage automatique utilisés. Il présentera par ailleurs, les statistiques descriptives de notre ensemble de données et expliquera les différentes étapes de préparation, de traitement des données et de l'entraînement des algorithmes d'apprentissage. Le chapitre trois décrira les outils logiciels utilisés à cet effet.

Le chapitre quatre montrera les différentes expérimentations menées, les résultats obtenus et leur interprétation.

Enfin, le chapitre cinq présentera les critères de validité de nos travaux, une conclusion décrivant les contributions de cette recherche et les travaux futurs.

CHAPITRE2. ÉTAT DE L'ART

Dans ce chapitre, nous présentons un ensemble de travaux antérieurs en lien avec notre travail de recherche dont l'objectif est d'identifier et de détecter les composants critiques dans les systèmes logiciels orientés objet en utilisant les algorithmes d'apprentissage machine et les métriques orientées objet. Les composants critiques sont des composants fautifs qui causent les dysfonctionnements du système logiciel. Rathore et al. [52] ont exploré les différentes dimensions du processus de prédiction de fautes pour aider à comprendre les divers éléments associés à ce processus. Les auteurs ont identifié trois éléments dans ce processus, dont les métriques orientées objet, les algorithmes d'apprentissage automatique et les données.

2.1 Métriques logicielles

Une métrique logicielle est une mesure quantitative qui caractérise de manière objective les propriétés d'un système logiciel. Parmi ces propriétés, nous pouvons citer le nombre d'attributs, le nombre de lignes de code, le nombre de méthodes d'un composant ou encore, le nombre de classes dans les systèmes OO, le nombre d'interactions entre les classes, etc. Un système logiciel orienté objet est un ensemble de classes, dans lesquelles sont regroupés des méthodes et des attributs caractérisant un objet. Un objet est une instance d'une classe.

Les métriques de Chidamber et Kemerer (CK)[53] sont considérées comme les pionnières des métriques orientées objet et constituent la norme par défaut avec laquelle toute nouvelle métrique est comparée. Cette suite est constituée de métriques de couplage, de cohésion, d'héritage et de complexité.

Moudache et Badri [54] ont utilisé les métriques de CK [53] avec d'autres métriques pour la prédiction des classes sujettes aux fautes. Les auteurs ont combiné les métriques de CK soit entre elles, soit avec d'autres métriques pour évaluer le risque que représente une classe logicielle.

Kayarvizhy et al. [55] ont évalué la capacité de prédiction des fautes de la suite des métriques CK [53] et l'ont validée empiriquement comme pouvant améliorer la précision des algorithmes d'apprentissage automatique. Ils ont étudié la possibilité de remplacer la métrique de cohésion de la suite CK par une métrique de cohésion proposée: métrique de cohésion à haute précision. Rajkumar et al. [56] ont proposé une nouvelle approche empirique de prédiction de fautes logicielles qui se concentre sur l'amélioration de la qualité logicielle des systèmes orientés objet en n'utilisant que les métriques de CK.

D'autres métriques, telles que les métriques de changement, ont été aussi utilisées pour prédire les modules sujets aux fautes. Rhmann et al. [57] ont travaillé sur la conception de modèles d'apprentissage automatique et hybride basés sur les métriques de changement pour la prédiction des fautes logicielles.

Kumar, Sripada, et al. [58] ont présenté un outil efficace de prédiction des fautes logicielles en identifiant et en étudiant le pouvoir prédictif de plusieurs métriques logicielles bien connues et largement utilisées : CK [53], Martin [59], McCabe [60], Henderson-Sellers [61] et Tang et al. [62].

2.2 Les algorithmes d'apprentissage profond

Les Algorithmes d'Apprentissage Automatique (AAA) apprennent des caractéristiques d'un ensemble de données dans le but de reconnaître ces caractéristiques apprises dans de nouveaux ensembles de données, sans toutefois que des instructions claires et précises ne leur soient fournies [63]. Ils sont utilisés dans la reconnaissance d'images, d'objets, de voix, dans la traduction de langue et dans plusieurs autres domaines [64] dont le génie logiciel.

Les Apprentissages de Réseaux de Neurones Profonds (ARNP) sont des variantes plus complexes des algorithmes d'apprentissage automatique (AAA) [64, 65]. Ils sont conçus à l'image du cerveau humain [66, 67] pour apprendre des caractéristiques non linéaires [64] et plus abstraites des données. Le cerveau humain contient des milliards de neurones

interconnectés qui s'échangent des informations [66, 67]. Par analogie, les ARNP contiennent à moindre échelle, des milliers, voire des millions de neurones artificiels interconnectés. L'apprentissage automatique profond est donc réalisé en utilisant les réseaux de neurones artificiels.

Kaushik et al. [68], dans leurs travaux, ont utilisé deux Réseaux de Neurones Artificiels (RNA) pour l'amélioration de la précision dans l'estimation des coûts des systèmes logiciels dont le RNA à base radicale formé de trois couches (une d'entrée, une cachée et une de sortie) et le RNA de liaisons fonctionnelles constitué d'une couche d'entrée et d'une couche de sortie.

Manjubala et al. [69] ont présenté les différentes applications du RNA pour l'estimation et la prédiction des indicateurs de fiabilité logicielle comme: le nombre de pannes espérées en un temps donné, le temps espéré entre les différentes pannes, l'effort de développement et l'identification des modules sujets aux fautes.

Rajnish et al. [70] ont proposé un nouveau modèle de Réseau de Neurones Convolutifs (NRNC) pour la prédiction des classes logicielles sujettes aux fautes. Le Réseau de Neurones Convolutifs (RNC) est un RNA inspiré du processus de vision biologique. Il est utilisé le plus souvent dans la reconnaissance d'images. Les performances du NRNC sont comparées avec celles d'un RNA (constitué de plusieurs couches), de la forêt aléatoire, de l'Arbre de Décision et du Naïve Bayésien. Leurs résultats montrent que le NRNC surpasse les autres algorithmes.

De plus en plus de répertoires de codes sources logiciels émergent de nos jours et disposent d'une grande variété de données sur les logicielles. Reyes et al. [71] ont appliqué le Réseau de Neurones Récurrents (RNR : LSTM) pour la classification automatique des archives de codes sources en fonction du langage de développement. Le RNR est une variante du RNA capable de traiter les données en utilisant sa mémoire interne pour se rappeler leur position dans une

séquence. Les performances du RNR ont été comparées à celles du Naïves Bayésien (NB). Les résultats montrent qu'en général le RNR est plus performant que le NB.

Erturk et Sezer [72] ont présenté une première application du système d'inférence adaptatif Neuro Flou (ANFIS) pour la prédiction des classes sujettes aux fautes. Pour les auteurs, ce système est une combinaison de la logique floue et du RNA. La logique classique est une logique qui permet d'exprimer que deux états, vrais (1) ou faux (0), utilisés jusqu'à présent par les ordinateurs. La logique floue est plus proche de la logique humaine, et permet d'exprimer plusieurs états. Les résultats montrent que les performances de ce modèle sont meilleures que celles des RNA et du Support Vecteur Machine (SVM).

2.3 Prédiction des fautes logicielles

Dans la littérature, la prédiction des fautes a été abordée sous plusieurs angles. Nous mettons l'accent sur la prédiction de la probabilité de fautes, la prédiction du nombre de fautes et la prédiction de la sévérité des fautes.

2.3.1 Prédiction de la probabilité de fautes

Moudache et Badri [54] ont évalué la probabilité de fautes des classes des systèmes orientés objet en prenant en compte le risque qu'elles représentent. Dans leurs travaux, le risque d'une classe est représenté par le fait qu'une classe soit sujette aux fautes et l'impact de la faute sur le reste du système. Ces deux facteurs ont été capturés par des métriques orientées objet.

Le risque est obtenu en utilisant la distance euclidienne. Les auteurs ont réalisé une classification binaire des classes logicielles en « fautives » ou « non fautives ». Puis, une classification en trois catégories en fonction du nombre de fautes et de leur sévérité: non fautive, sévérité normale et sévérité élevée.

Neha et al. [15] ont travaillé sur la prédiction de la probabilité de fautes des classes en utilisant divers algorithmes d'apprentissage automatique. Avant de procéder à l'entraînement des algorithmes, les auteurs ont d'abord appliqué la technique de sélection des caractéristiques

basée sur la corrélation entre les données afin d'obtenir des métriques non corrélées. Ces métriques sont ensuite fournies aux algorithmes d'apprentissage pour l'entraînement et la classification.

Dallal [73] propose un modèle de prédiction des classes sujettes aux fautes à l'étape de déploiement du projet. Ce modèle se base sur les classes réutilisées, des versions antérieures immédiates (V1, V2, …, Vn-1) à celle en déploiement concernée (Vn). La méthodologie adoptée par l'auteur se décline en six étapes. Les trois premières étapes ont consisté à construire un modèle sur les données de fautes de chaque version antérieure (Mv1, Mv2, …, Mvn-1). Ces modèles sont ensuite appliqués aux classes de la version Vn pour identifier les classes nouvelles et les classes réutilisées avec leur probabilité de fautes. Au cours des trois dernières étapes, l'auteur a conçu un modèle entraîné sur l'ensemble des classes réutilisées (des versions antérieures dans la version Vn) obtenues précédemment. Ce modèle est ensuite appliqué sur l'ensemble des classes de la version Vn. Les résultats de ce modèle sont ensuite réajustés suivant des règles que l'auteur a préétablies.

2.3.2 Prédiction du nombre de fautes logicielles

Une faute logicielle, comme précédemment définie dans la problématique, est une conséquence d'erreurs introduites au cours des différentes phases du cycle de développement logiciel [43, 44, 74] et, est la cause des éventuelles défaillances. Rathore et Kumar [23] ont présenté une étude expérimentale visant à évaluer et à comparer la capacité de six algorithmes d'apprentissage automatique dans la prédiction du nombre de fautes en comparant leurs performances. Les résultats montrent que la régression binomiale négative (NBR) et la régression de Poisson (ZIP) sont les moins performantes parmi les techniques considérées**.**

Rathore et Kumar [75] ont amélioré les travaux précédents en proposant un ensemble de méthodes constitué d'algorithme d'apprentissages automatiques linéaires et non linéaires pour la prédiction du nombre de fautes. Les auteurs ont évalué les résultats en intra-version et en

inter-version en utilisant l'erreur absolue, l'erreur relative, la prédiction niveau l et la mesure de la complétude. En inter-version, les anciennes versions ont été utilisées pour (1) l'entraînement, (2) le test de validation, et la nouvelle version, pour la prédiction. En intra-version, chaque version est répartie en cinq parties : trois parties ont été utilisées pour l'entraînement, une partie pour le test de validation et la dernière partie pour la prédiction. Les résultats montrent que la validation en intra-version donne de meilleures performances.

2.3.3 Prédiction de la sévérité des fautes

Le niveau de sévérité des fautes dans les classes des systèmes logiciels est aussi l'un des facteurs permettant de prédire les fautes logicielles [17, 21, 22, 76, 77]. Le niveau de sévérité des fautes logicielles est le plus souvent classé en trois catégories [78, 79]: faible, moyen et élevé. Les fautes de sévérité élevée sont celles qui causent généralement le plus de défaillances [80], elles requièrent donc plus de ressources pour leur correction [81] afin de réduire ces défaillances en production.

Kumari et Rajnish [22] ont proposé un modèle réalisant la classification des modules en « fautifs » ou « non fautifs », ensuite pour ceux qui sont fautifs, le modèle les catégorise en trois niveaux de sévérité : faible, moyen et élevé. Les auteurs ont défini les niveaux de sévérité en prenant en compte le nombre de fautes contenues dans les modules. Ils préconisent que ce modèle soit utilisé durant le cycle de vie du logiciel, entre les étapes de développement et de test afin de prioriser les modules contenant plus de fautes lors de ces tests. Cependant, un nombre élevé de fautes n'est pas synonyme de fautes graves pouvant causer la défaillance du système.

Chen et al [21] ont exploré l'utilité des métriques réseau dans la prédiction de la probabilité de fautes de niveau élevé. Ces métriques réseau proviennent de l'analyse des réseaux sociaux où les modules représentent les nœuds, et leurs interdépendances représentent les branches du réseau. Les métriques réseau caractérisent donc le flux d'information et la topologie générale

du système logiciel. Leurs résultats révèlent que les métriques réseau sont significativement corrélées aux fautes de sévérité élevée.

Hamill et Goseva-Popstojanova [77] ont exploré la relation entre les différents types de fautes logicielles et les activités d'assurance qualité permettant de les identifier. Les auteurs ont considéré cinq types de fautes: les fautes de spécification, les fautes de conception, les fautes de codage, les fautes d'intégration-interface et une catégorie « autres fautes ». Leurs résultats montrent que le test est l'activité d'assurance qualité la plus susceptible de détecter les fautes critiques. Ils révèlent aussi que les fautes de codage sont responsables de la majorité des fautes critiques. Les fautes critiques sont les fautes de sévérité élevée.

D. Aggrawal et al. [82] ont travaillé sur un modèle de détection de fautes dont les nouvelles fautes de la version actuelle et les anciennes fautes restantes (non corrigées des versions précédentes). Ils estiment que plus la sévérité d'une faute est élevée, plus le temps entre son identification et sa correction est élevé, contrairement aux fautes de sévérité faible. Les auteurs ont évalué donc la sévérité en termes de temps entre la découverte et la correction de la faute. Ils ont considéré que les ressources nécessaires pour la correction d'une faute complexe diffèrent de celles nécessaires à une faute simple. Ce modèle prend en compte deux niveaux de sévérité des fautes : faible et élevé.

CHAPITRE3. MÉTHODOLOGIE DE RECHERCHE

L'objectif de nos travaux est d'identifier et de détecter les composants critiques dans les systèmes logiciels en utilisant les métriques orientées objet et les algorithmes d'apprentissage automatique profond. Nous considérons dans notre étude que les composants critiques sont les classes logicielles critiques. Nous définissons la criticité d'une classe logicielle par le fait qu'elle soit fautive ou non fautive, par le nombre et la fréquence de fautes qu'elle contient. Dans cette logique, une classe critique est une classe fautive ou une classe qui contient un grand nombre de fautes ou encore, une classe avec une fréquence élevée de fautes. Nous nous basons sur deux approches largement utilisées dans la littérature pour identifier les classes logicielles critiques: la classification et la régression.

La classification consiste à regrouper les classes logicielles en deux ou plusieurs groupes basés sur des critères préalablement définis. La classification en deux groupes est dite classification binaire. Dans le cadre de nos recherches, ces groupes correspondent aux classes « non fautives » et aux classes « fautives ». La classification en plusieurs groupes est dite classification multi classes. Pour cette dernière, nous utilisons trois groupes: les classes « non fautives », les classes « faiblement fautives » et les classes « fortement fautives ». Les classes non fautives sont celles dont leur nombre de fautes est égal à 0, les classes faiblement fautives sont les classes dont le nombre de fautes est inférieur ou égal à la moyenne (strictement supérieur à 0). Les classes fortement fautives sont celles dont le nombre de fautes est supérieur à la moyenne.

L'approche par la régression consiste à prédire, pour chaque classe du système logiciel, le nombre de fautes qu'elle contient.

Le grand nombre de classes logicielles étudiées et le nombre de paramètres à considérer (métriques logicielles) nous suggèrent l'utilisation d'algorithmes d'apprentissage automatique, afin d'automatiser les tâches, de réduire les ressources et le temps consacrés. Les algorithmes

d'apprentissage automatique s'entraînent sur les données qui leur sont fournies, afin de reconnaître les caractéristiques apprises de ces données dans de nouvelles données. Nous avons mis l'accent sur les algorithmes d'apprentissage automatique profond afin de déceler les caractéristiques profondes et abstraites des données. Ces caractéristiques sont mesurées par les métriques orientées objet. Nous avons comparé les performances de l'algorithme d'apprentissage automatique profond, dans les différentes approches, avec celle d'un algorithme d'apprentissage automatique. Dans la suite de ce chapitre, nous présentons les questions de recherche autour desquelles notre étude a été réalisée, nous présentons également les algorithmes d'apprentissage automatique utilisés. À la fin du chapitre, nous décrivons les données ainsi que les outils logiciels ayant permis de les collecter et de les traiter.

3.1 Questions de recherche

Notre étude a porté sur les questions de recherche suivantes :

- Question 1 : Peut-on construire un régresseur basé sur l'apprentissage automatique profond et sur les données des métriques OO, capable de prédire de manière satisfaisante le nombre de fautes dans les classes logicielles ?
- Question 2 : Comment se situent les performances d'un tel régresseur comparées à celles de la régression linéaire ?
- Question 3 : Peut-on construire un classificateur basé sur l'apprentissage automatique profond et sur les données des métriques logicielles, capable de prédire de manière satisfaisante la présence de fautes ou leur fréquence (de manière qualitative) dans les classes logicielles ?
- Question 4 : Comment se situent les performances d'un tel classificateur comparées à celles régression logistique binaire et multi classes ?

3.2 Description des algorithmes d'apprentissage automatique utilisés

L'apprentissage automatique est une branche de l'intelligence artificielle [83, 84] créée par Arthur Samuel [85] en 1959 qui donne la capacité aux ordinateurs d'apprendre à partir des données, des caractéristiques sans que celles-ci soient explicitement programmées [63, 84]. Cet apprentissage des caractéristiques des données est réalisé par des algorithmes d'apprentissage automatique. Ces algorithmes apprennent donc des fonctionnalités ou des caractéristiques implicites et explicites des données qui leur sont fournies. L'apprentissage automatique est effectué en trois étapes [86, 87]: une première étape dite « d'entraînement », durant laquelle l'algorithme apprend des données afin de réaliser la tâche à accomplir; une seconde étape appelée « test de validation » au cours de laquelle est évaluée la qualité de la phase précédente d'entraînement. Finalement, une troisième étape, dite de « test » lors de laquelle l'algorithme applique ce qu'il a appris sur des données différentes de celles de l'entraînement. Les deux premières étapes constituent la phase d'apprentissage.

L'apprentissage automatique existe en trois catégories [63, 84]: l'apprentissage par renforcement, l'apprentissage supervisé et l'apprentissage non supervisé.

Un algorithme d'apprentissage par renforcement apprend par un mécanisme d'essais-erreur. Cet algorithme est entraîné en plusieurs itérations [63]. À chaque itération, l'atteinte de l'objectif entraîne une récompense et l'échec une pénalité. Ainsi, l'algorithme met à jour ses stratégies d'apprentissage à chaque itération pour maximiser ses récompenses.

Un algorithme d'apprentissage supervisé s'entraîne sur des données étiquetées [63]. Les données étiquetées contiennent des variables d'entrées (variables indépendantes) et une ou plusieurs variables de sortie espérées (variables dépendantes). L'apprentissage supervisé est généralement utilisé aussi bien dans la prédiction que dans la classification.

Un algorithme d'apprentissage non supervisé est un algorithme à qui des données non étiquetées sont fournies [63]. Les données non étiquetées sont des données qui ne contiennent que des variables en entrée. L'algorithme se charge de comprendre les données afin de regrouper les observations ayant des caractéristiques semblables.

Le choix des algorithmes supervisés ou non supervisés est fonction de la problématique qui découle de la structure des données d'entraînement. Dans notre cas, l'ensemble de données est constitué de métriques orientées objet (variables indépendantes), et du « nombre de fautes » ou des différents groupes de classification (variables dépendantes) définis plus haut. Au regard de notre problématique et de la structure étiquetée de nos données, nous optons pour l'apprentissage automatique supervisé. Notre sujet de recherche étant basé sur les algorithmes d'apprentissage automatique profond, nous optons pour un RNA à plusieurs couches. Nous comparons les performances du RNA en termes de classification, à celles de la régression logistique (binaire et multi classes) et en termes de régression, à celles de la régression linéaire.

3.2.1 Le Réseau de Neurones artificiels (RNA)

Le RNA est un algorithme d'apprentissage automatique inspiré de l'architecture et du fonctionnement du cerveau biologique [66, 67]. Le cerveau est constitué de milliers de neurones interconnectés par des synapses, chargés d'échanger le flux d'information entre eux. Le réseau de neurones artificiels est un concept logique qui peut être constitué de plusieurs couches de neurones artificiels interconnectées. Le réseau de neurones artificiels, dépendamment de sa profondeur (nombre de couches cachées) et de la problématique, peut requérir une grande quantité de données pour son entraînement. Ces données sont constituées, dans le cadre de l'apprentissage supervisé, de variables en entrée appelées variables indépendantes et de variables en sortie appelées variables dépendantes.

3.2.1.1 Les composants d'un RNA

Un RNA est constitué des neurones organisés en couches interconnectées. Chaque neurone a un poids *w qui* détermine la force de sa connexion avec les neurones de la couche suivante.

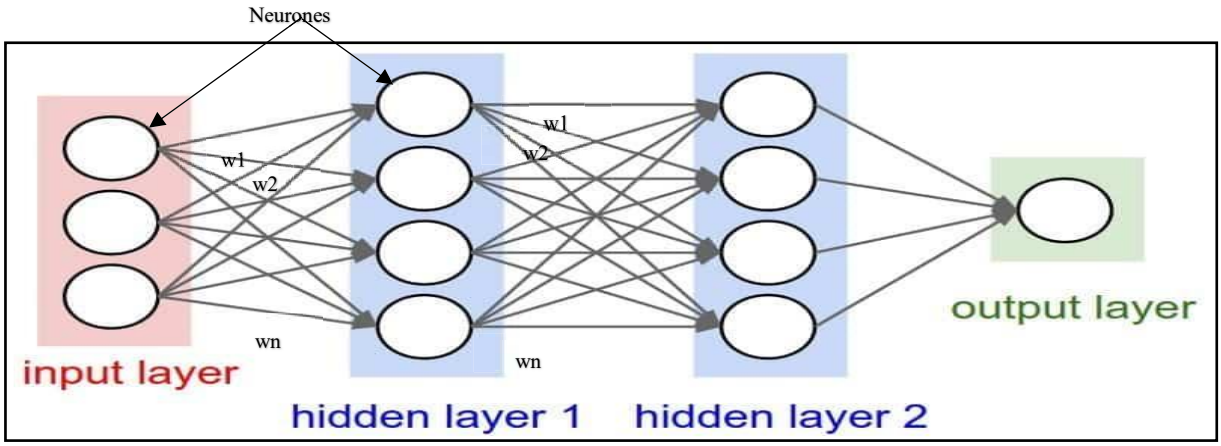

Figure 1: Composants d'un réseau de neurones [88]

a. Un neurone artificiel

Un neurone artificiel est une unité du réseau de neurones qui transmet l'information entre les différentes couches du réseau. Il prend en entrée des valeurs x, les analyse et produit une nouvelle valeur *y* en sortie.

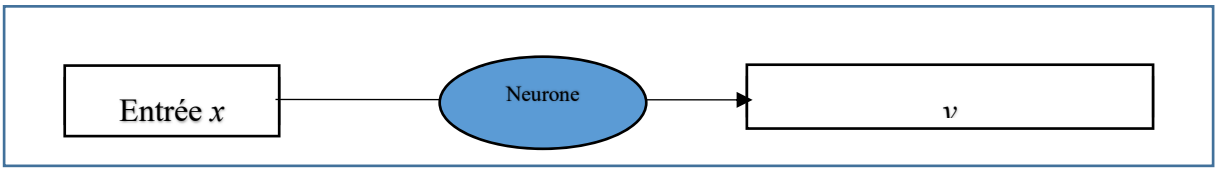

Figure 2: Structure générale d'un neurone

La valeur x en entrée est le résultat y en sortie d'un neurone de la couche précédente. Le nombre de neurones de la couche précédente correspond au nombre de valeurs x_i transmises aux neurones de la couche suivante où x est la valeur de l'information à transmettre et i le nombre de neurones.

La valeur de sortie y est obtenue par la somme pondérée des valeurs x de chaque neurone précédent et de leur poids respectif. Sur cette somme sont appliqués un biais, puis une fonction d'activation.

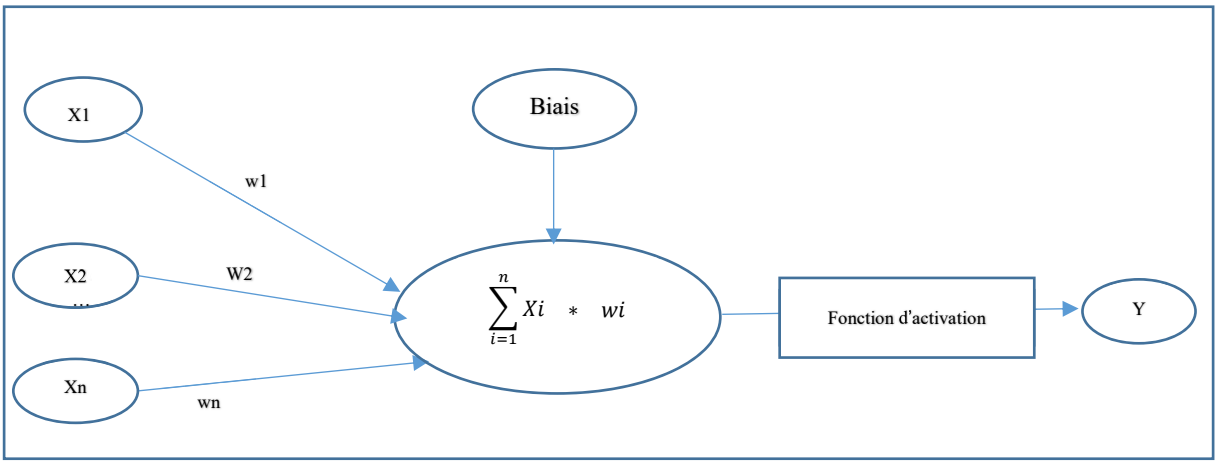

Figure 3: Structure détaillée d'un neurone

b. Le biais

Le biais est une valeur constante ajoutée à la somme des valeurs pondérées pour retarder l'activation du neurone. L'activation du neurone consiste à prendre en compte sa valeur dans la suite du traitement vers la couche suivante. Plus l'activation des neurones est retardée, plus le réseau augmente ses capacités d'apprentissage. Ainsi, le biais rend le réseau plus flexible et plus adaptable. Sa valeur est attribuée de manière aléatoire.

c. La fonction d'activation

La fonction d'activation est une fonction mathématique qui constitue une porte entre la somme des valeurs pondérées ajoutées au biais et la valeur de sortie du neurone. Elle permet de capturer les caractéristiques profondes des données. Elle décide de l'activation ou non d'un neurone après évaluation de la somme pondérée et du biais. Diverses fonctions d'activation regroupées en deux catégories sont disponibles selon la problématique traitée :

Les fonctions d'activation linéaires

Elles sont proportionnelles à la donnée en entrée de telle sorte que toutes les couches du réseau sont combinées et fonctionnent comme une seule couche. La sortie de cette fonction est dans l'intervalle [- ∞, +∞]. L'utilisation de cette fonction est déconseillée pour l'apprentissage sur les données complexes. Elle est définie par la formule : $F(\hat{x}) = a\hat{x}$.

Les fonctions d'activation non linéaires

Les fonctions d'activation non linéaires sont les plus utilisées. Elles capturent les caractéristiques profondes non linéaires des données et favorisent la rétro propagation. La rétro propagation consiste à mettre à jour le poids des neurones de la dernière couche à la première afin de corriger les erreurs lors de l'apprentissage de l'algorithme.

- ***La fonction Sigmoïde :*** La fonction sigmoïde prend en entrée des valeurs réelles et fournit en sortie des valeurs comprises dans l'intervalle [0,1]. Elle est utilisée dans les problématiques de prédiction de probabilité. Elle s'exprime par la formule suivante :

$$f(x) = 1/(1+e^{(-x)}).$$

- ***La fonction TanH :*** La fonction TanH est une fonction similaire à la fonction Sigmoïd. La différence réside au niveau des intervalles de valeurs. Les valeurs de la fonction Tanh se situent dans l'intervalle [-1, 1]. Elle est définie par la formule :

$$\tanh(x) = 2/(1+e^{(-2x)}) - 1.$$

- ***La fonction ReLU :*** La fonction ReLU prend en entrée des valeurs réelles et fournit en sortie des valeurs dans l'intervalle [0, ∞]. Elle est définie par la formule :

$$f(x) = \max(0, x).$$

- ***La fonction Softmax :*** La fonction softmax est une combinaison de plusieurs fonctions sigmoïdes. Elle est utilisée dans la classification multi classes. Elle prend en entrée des valeurs réelles et fournit en sortie plusieurs valeurs comprises dans l'intervalle [0,1]. Le nombre de valeurs produit équivaut aux nombres de classes à prédire. Elle affecte la valeur 1 à la plus grande valeur de probabilité et la valeur 0 aux restes des valeurs. La somme de toutes les valeurs est égale à 1 [89]. Elle est obtenue par la formule :

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

3.2.1.2 Les couches d'un RNA

Un RNA est constitué de trois types de couches : la couche d'entrée, les couches cachées et la couche de sortie.

a. La couche d'entrée

La couche d'entrée est la première couche du réseau. Elle prend les données en entrée et les transmet aux couches suivantes du réseau. Elle contient autant de neurones que de paramètres dans les données.

b. Les couches cachées

Les couches cachées sont chargées du traitement des données transmises par la couche d'entrée. Elles s'occupent du calcul de la valeur en sortie en prenant en compte la contribution de chacune des variables en entrée. Leur absence implique que chaque variable en entrée influence indépendamment la valeur en sortie. Cependant, dans la majorité des ensembles de données, les variables en entrée sont interdépendantes. Les couches cachées sont donc indispensables pour capturer cette interdépendance entre les variables. Le nombre de couches cachées détermine la profondeur du réseau. Cependant, un nombre élevé de couches cachées est l'une des causes du surapprentissage. De même le sous-apprentissage est l'une des conséquences d'un nombre trop faible de couches cachées. Le surapprentissage est un état dans lequel le réseau apprend très bien des données d'entraînement, mais peine à généraliser l'apprentissage sur de nouvelles données. En situation de sous-apprentissage, le réseau apprend peu sur les données qui lui sont soumises. Cela peut être dû à un sous-dimensionnement d'unités ou de couches, mais aussi au manque d'entraînement ou de données, ou à leur inconsistance ou encore à l'absence de liens entre les entrées et les sorties. Le choix d'un nombre judicieux de

couches, d'unités dans celles-ci, ainsi que la durée de l'entraînement, sont donc indispensables pour une meilleure performance du réseau.

c. La couche de sortie.

La couche de sortie fusionne et simplifie les sorties de la dernière couche cachée afin d'obtenir un résultat final. Le nombre de données en sortie de cette couche dépend de la problématique traitée, de la fonction d'activation utilisée et du nombre de neurones définis.

3.2.1.3 Fonctionnement d'un RNA

Un RNA prend les valeurs en entrée à travers la couche d'entrée. Ces valeurs sont ensuite transmises à la première couche cachée. Chaque couche cachée est un ensemble de neurones chargé du traitement des données de la couche précédente et de la transmission de ces données en sortie aux neurones de la couche suivante. Plusieurs paramètres rentrent dans le choix du nombre de neurones dans une couche cachée, ainsi que dans le choix du nombre de couches cachées dans un réseau: le type de données d'entraînement, le nombre de variables en entrée, etc. Cependant, à ce jour, il n'existe pas de règles pour obtenir un parfait nombre de neurones par couche ou un parfait nombre de couches par réseau selon les données et la problématique. L. Shafi et Al [90] ont testé différents nombres de neurones dans une ou plusieurs couches cachées. Leurs résultats ont montré qu'avec un nombre faible de neurones dans une couche cachée, celle-ci éprouve des difficultés à relayer la bonne information à la couche suivante. Ils ont remarqué aussi qu'avec un grand nombre de neurones, les performances du réseau ne s'améliorent pas, et que l'utilisation d'une seule couche cachée est appropriée aux ensembles de données non linéaires.

Chaque neurone possède un poids w qui est un élément essentiel dans l'activation du neurone. Un neurone est activé si seulement si sa valeur pondérée est supérieure à un certain seuil. La valeur pondérée d'un neurone est obtenue par le produit de la valeur d'entrée avec son

poids. Pour un neurone ayant plusieurs entrées, la valeur pondérée est la somme des valeurs pondérées de toutes les entrées.

Le biais est le seuil avec lequel la somme pondérée du neurone est comparée dans le cadre des RNA. Cette comparaison est réalisée par la fonction d'activation. La fonction d'activation compare la somme pondérée au biais (le seuil) et prend des décisions: si la somme pondérée est supérieure au biais, le neurone est activé et sa valeur est donc transmise aux neurones de la couche suivante, sinon le neurone est désactivé et sa valeur n'est pas communiquée à la couche suivante.

Ce processus est réalisé d'un neurone à un autre et d'une couche cachée à une autre. Les neurones de la dernière couche cachée transmettent les informations à la couche de sortie. Cette dernière a pour fonction de faire correspondre à chaque donnée en entrée, la sortie obtenue.
À la création du réseau de neurones, le poids des neurones et celui du biais sont attribués de façon aléatoire. Au cours du processus d'apprentissage, ces poids sont mis à jour pour minimiser la marge d'erreur entre la valeur de sortie obtenue et la valeur réelle libellée pour cette observation. L'écart (erreur) entre la valeur de sortie et le libellé est estimé et constitue la fonction coût à minimiser par une approche itérative de calcul de gradient grâce à laquelle les poids sont alors ajustés. Pour faciliter les calculs de gradient, ce processus de mise à jour des poids est réalisé de la couche de sortie vers la couche d'entrée en passant par les couches cachées. On parle alors de rétropropagation. Dans ce processus de rétropropagation, le réseau détermine l'impact du poids de chacun des neurones sur la marge d'erreur et effectue les modifications nécessaires pour minimiser cette marge d'erreur.

3.2.2 *La régression linéaire*

La régression linéaire est un algorithme d'apprentissage automatique qui établit une relation linéaire entre les variables indépendantes et la variable dépendante en utilisant le coefficient de régression linéaire [49, 50]. Sa variable dépendante est quantitative tandis ses variables

indépendantes sont quantitatives ou qualitatives. La régression linéaire a deux sous catégories : la régression linéaire simple et la régression linéaire multiple.

En régression linéaire simple, une seule variable indépendante explique la variable dépendante. En régression linéaire multiple, plusieurs variables indépendantes concourent à expliquer la variable dépendante. La régression linéaire simple a pour formule : $Y=\beta_0 + X\beta_1 + \varepsilon$ où Y est la variable dépendante, X la variable indépendante, β_0 la valeur de Y quand X est égale à 0, β_1 le coefficient de la pente et ε l'erreur.

3.2.3 La régression logistique

La régression logistique est un algorithme d'apprentissage automatique basée sur une méthode d'analyse statistique [47, 48]. Elle évalue la probabilité de réalisation d'un évènement. Elle prend en entrée des variables qualitatives ou quantitatives et en sortie des variables qualitatives. Elle est qualifiée de binaire si le nombre de variables dépendantes est égal à deux et de multi classes si ce nombre est supérieur à deux. Elle est généralement utilisée dans la classification et a pour formule (régression binaire) : $P(Y) = \frac{1}{1+e^{-(\beta_0+X\beta_1+\varepsilon)}}$ où P(Y) est la probabilité de réalisation de la variable dépendante Y. Les autres variables ont la même définition que celle dans la partie précédente (la régression linéaire).

Les algorithmes d'apprentissage automatique nécessitent une importante quantité de données pour leur apprentissage. En ce sens, nous avons collecté les classes logicielles de six versions du système logiciel POI [91].

3.2.4 *Le système POI*

Le système POI [91] pour *Poor Obfuscation Implementation* est un logiciel de la *Apache Software Fondation (ASF) [92]* implémenté en Java pour la création et la modification programmatique des fichiers de la suite Microsoft Office.

Le système POI est à sa version 5.2.2 à ce jour. Pour nos travaux, nous nous sommes intéressés aux classes logicielles [93] de six versions : 4.0.0, 4.0.1, 4.1.0, 4.1.1, 4.1.2 et 5.0.0. Pour chaque classe collectée, nous avons calculé les métriques de Chidamber et Kemerer (CK).

3.2.5 *Les métriques de Chidamber et Kemerer*

Les nombreuses utilisations [54-57] des métriques de Chidamber et Kemerer [53] dans la prédiction de fautes logicielles nous motivent à les appliquer sur nos données étant donné que les objectifs de nos travaux s'inscrivent dans la même logique. À ces travaux, nous pouvons ajouter les conclusions des travaux de J.A. Dallal [94] qui montrent la capacité significative de ces métriques dans la prédiction de la propension aux fautes des classes logicielles orientées objet.

La suite des métriques de CK est constituée de six métriques de code source, dont la métrique de couplage CBO, les métriques d'héritages DIT et NOC, la métrique de manque de cohésion LCOM et les métriques de complexité RFC et WMC.

- ***La métrique de couplage CBO*** *:* Cette métrique mesure le nombre de classes avec lesquelles une classe est couplée. Deux classes sont couplées si elles dépendent l'une de l'autre par appel de méthodes ou d'attributs. Cette dépendance est dite entrante pour la classe appelée et sortante pour la classe appelante. La valeur de la métrique CBO est obtenue par la somme de ces deux dépendances. Une valeur élevée de cette métrique pour une classe logicielle orientée objet montre sa dépendance vis-à-vis d'autres classes du système. Cette dépendance réduit son indépendance ainsi que sa réutilisabilité vis-à-vis d'autres modules ou d'autres systèmes.

Les conclusions des travaux de M. Rizwan et Al. [95] ont montré que cette métrique a un impact sur la disposition aux fautes d'une classe logicielle.

- ***La métrique d'héritage DIT :*** La métrique DIT mesure le niveau de profondeur dans une relation d'héritage. Cette mesure commence à partir de la classe racine de l'héritage. Une

relation d'héritage diminue le niveau de complexité d'un système logiciel orienté objet. Cette métrique indique également le degré de réutilisation de la classe mère. Pour cette métrique, la classe racine d'un héritage à une valeur égale à 0 tandis que le DIT des classes filles dépend de leur niveau dans la hiérarchie de l'héritage : plus le niveau hiérarchique augmente, plus la valeur de cette métrique augmente.

- ***La métrique d'héritage NOC***: La métrique NOC mesure pour une classe mère, le nombre de ses classes filles immédiates dans une relation d'héritage. Elle mesure la largeur d'une relation d'héritage, contrairement au DIT qui mesure sa profondeur. Une valeur élevée de cette métrique indique pour une classe le nombre de classes filles directes auxquelles elle est connectée. Selon les travaux empiriques de M. Cartwright [96], la relation d'héritage a probablement une influence sur la densité du nombre de fautes dans une classe. Par ailleurs, les travaux de K. El Emam et al [97] et de L. C. Briand et al [98] démontrent sa corrélation avec la propension aux fautes d'une classe.

- ***La métrique de manque de Cohésion LCOM***: La métrique LCOM estime le niveau de cohésion d'une classe dans un système logiciel orienté objet par la différence entre le nombre de paires de méthodes n'ayant aucun attribut en commun et le nombre de paires de méthodes ayant au moins un attribut en commun. Plus sa valeur augmente, plus la cohésion de la classe diminue. Une séparation de la classe en plusieurs sous-classes est conseillée pour pallier ce manque de cohésion.

- ***Les métriques de complexité RFC:*** La métrique RFC mesure l'ensemble des réponses d'une classe à l'appel des méthodes de ses objets. Cette métrique évalue la communication de la classe avec les autres classes du système. Plus sa valeur pour une classe augmente, plus sa complexité augmente et plus sa compréhension diminue.

- ***Les métriques de complexité WMC:*** La métrique WMC est la somme pondérée des complexités cyclomatiques des méthodes définies dans une classe logicielle. La complexité cyclomatique (McCabe) d'une méthode est le nombre de possibilités ou de chemins possibles pour son exécution. Plus sa valeur augmente, plus le risque de fautes augmente. Plusieurs études montrent [72, 99] le lien entre la complexité cyclomatique et le nombre de fautes dans les classes logicielles.

3.2.6 *Collecte, préparation et prétraitement de données*

Nos données de recherche sont constituées de classes fautives et non fautives de six versions du système POI: de la version 4.0.0 à la version 5.0.0. Afin d'obtenir des ensembles de données sur lesquels nos algorithmes d'apprentissage seront capables de s'entraîner, nous avons réalisé pour chacune des versions, les étapes de préparation des données suivantes :

- **Étape1 :** Calcul des métriques de Chidamber et Kemerer pour l'ensemble des classes logicielles.

- **Etape2 :** Collecte et appariement des classes fautives avec leurs métriques respectives. Ce qui produit une liste de classes logicielles avec leurs métriques dans laquelle sont identifiées les classes fautives et non fautives

- **Etape3 :** Séparation de la liste obtenue précédemment en deux listes : une première liste contenant les classes fautives et une seconde liste contenant les classes non fautives.

- **Etape4 :** Insertion de la liste des classes fautives avec leurs métriques dans le logiciel TraiDon. TraiDon est un système de préparation de données (application web) que nous avons conçu, avec un backend, en SpringBoot [100], un frontend en Angular [101] utilisant une base de données Postgres [102]. L'objectif premier de TraiDon est de calculer plus facilement à l'aide des scripts SQL, pour chaque classe fautive, son nombre de fautes, groupé par sévérité. Cependant, l'analyse des données collectées a révélé l'incohérence des sévérités de fautes des

classes logicielles, des versions du système POI considérées: les fautes qualifiées d'amélioration et de régression sont prises en compte, d'après nos analyses, comme des niveaux de sévérité. D'après notre compréhension, une faute d'amélioration ou de régression est une faute qui survient au cours de la phase d'évolution ou de la maintenance du système. Les termes amélioration et régression font référence à l'étape du cycle de vie logiciel à laquelle la faute est introduite. Ces caractéristiques n'ont donc aucun rapport avec la sévérité de fautes. À la vue de ces analyses, nous avons utilisé TraiDon pour le calcul du nombre de fautes des classes fautives sans prendre en compte leur sévérité.

Le second objectif de TraiDon est de disposer d'un historique des données pour nos futurs travaux de recherche. Cet historique sera disponible une fois l'application hébergée, en tout temps et toute situation géographique disposant d'internet. TraiDon réalise les fonctionnalités suivantes:

- La gestion des systèmes logiciels (Ajout et Modification) ;
- La gestion des versions des systèmes logicielles (Ajout et Modification) ;
- La gestion des classes fautives (Ajout, Modification et Chargement des classes contenues dans des fichiers Excel) ;
- La gestion de la sévérité (Ajout et Modification) ;
- La gestion des fautes (Ajout et Modification) ;

La gestion de fautes consiste à associer à chaque faute d'une classe donnée, la classe (gérée au niveau de la gestion des classes fautives) combinée à la sévérité de la faute (insérée au niveau de la gestion des sévérités). Ainsi, nous obtenons pour chaque classe, le nombre de fautes et le nombre de fautes par sévérité. Le nombre de fautes correspond au nombre de fois que la classe est dupliquée. Le nombre de fautes par sévérité fait référence au nombre de fois que la classe est dupliquée par type de sévérité.

- **Étape5 :** Calcul du nombre de fautes des classes fautives dans TraiDon. Nous avons calculé à base des scripts SQL pour chaque classe fautive son nombre de fautes. Nous obtenons une liste des classes fautives avec leurs métriques et leur nombre de fautes.

- **Étape 6 :** Fusion de la liste des classes fautives obtenues (avec leurs métriques et leurs nombres de fautes) avec les classes non fautives de l'étape 3.

- **Étape 7 :** Mise à jour à 0 de la valeur de la colonne « nombre de fautes » des classes non fautives de la liste obtenue à l'étape 6. Nous obtenons une liste de classes fautives et non fautives avec leurs métriques et leur nombre de fautes. La liste obtenue à l'issue des sept étapes précédentes constitue l'ensemble de données pour les modèles d'apprentissage par la régression. Pour les modèles d'apprentissage par la classification, des étapes supplémentaires sont requises.

- **Étape 8 :** Pour la classification binaire, la colonne nombre de fautes est binarisée: cette colonne prend la valeur 0 pour les classes non fautives dont le nombre de fautes est égal à 0 et 1 pour les classes fautives dont le nombre de fautes est supérieur à 0. Nous obtenons donc pour la classification binaire, un ensemble de données constitué de classes fautives et de classes non fautives avec leurs métriques et la colonne nombre, renommée en fautive. Cette colonne contient les valeurs 0 ou 1 en fonction de la présence ou non de fautes.

En classification multi classes, cette étape consiste à calculer la moyenne du nombre de fautes pour l'ensemble des classes fautives de chaque version. Cette moyenne est le critère par lequel nous constituons les différents groupes de classification correspondant aux trois catégories qualificatives (non fautive, faiblement fautive et fortement fautive) pour la prédiction de la fréquence de fautes.

- **Étape 9:** Cette étape est spécifique à la classification multi classes. Elle consiste à rassembler en trois groupes les différentes valeurs de nombre de fautes. Ce regroupement utilise

la moyenne du nombre de fautes calculée à l'étape 8. Les classes non fautives dont le nombre de fautes est égal à 0 appartiennent au groupe 0, les classes fautives dont le nombre de fautes est inférieur ou égal à la moyenne appartiennent au groupe 1 (mais strictement > 0) et les classes fautives dont leur nombre de fautes est supérieur à la moyenne font partie du groupe 2. Ces trois groupes correspondent respectivement aux trois catégories qualificatives ordonnées suivantes: la catégorie des classes non fautives, la catégorie des classes faiblement fautives et la catégorie des classes fortement fautives. Cette étape conduit à l'obtention d'un ensemble de données constitué des classes fautives et non fautives avec leurs métriques et leur groupe d'appartenance.

3.2.7 *Statistiques descriptives*

Le tableau 1 présente les statistiques descriptives des six (6) versions du système POI. De ces statistiques descriptives présentées sur la figure 4, nous remarquons que:

La taille du système a augmenté de la version 4.0.0 à la 4.1.1, avant de diminuer de la version 4.1.1 à la version 5.0.0.

Le nombre de classes fautives a considérablement diminué de la version 4.0.0 à la version 4.0.1, avant d'augmenter de la version 4.0.1 à la version 4.1.1. Il a par la suite, connu une légère baisse à la version 4.1.2 puis une légère augmentation à la version 5.0.0.

Ces variations suggèrent que : les fautes introduites à la version 4.0.0 ont probablement été corrigées à la version 4.0.1 et l'ajout des fonctionnalités supplémentaires aurait éventuellement entraîné une augmentation de la taille du système; l'augmentation de la taille du système de la version 4.0.1 à 4.1.1 a peut-être introduit de nouvelles fautes. Ces fautes ont probablement été ensuite corrigées à la version 4.1.2. L'augmentation du nombre de fautes avec la diminution de la taille du système à la version 5.0.0 montre que les changements à la version 4.1.2 ont probablement introduit de nouvelles fautes à la version 5.0.0.

La version 4.0.0 demeure celle ayant la plus petite taille, mais qui contient 32 % de classes fautives. La version 4.0.1 a une taille légèrement plus grande que celle de la version 4.0.0 et c'est la version qui contient moins de classes fautives avec un pourcentage de 5 %.

Le tableau 1 montre également que les six versions du système POI comptabilisent un total de 18 802 classes logicielles, dont 1119 classes fautives.

Les moyennes des métriques indiquent de légères variations d'une version à une autre. Cette moyenne stable cache une grande disparité entre les classes. En effet, pour chaque version, la valeur de l'écart type de chaque métrique est supérieure à celle de la moyenne, à l'exception de celle de la métrique DIT. Cet écart montre une grande disparité des valeurs des métriques dans l'ensemble de données.

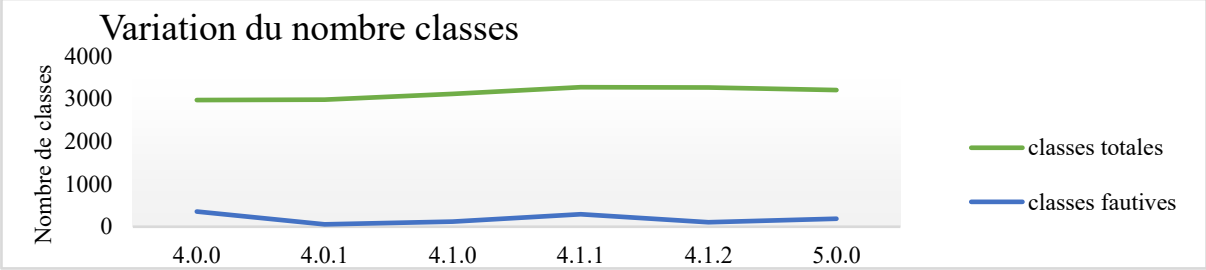

Figure 4: Variation du nombre total de classes et de classes fautives

Versions	Taille du système	Nombre de classes fautives	Statistique	CBO	DIT	NOC	LCOM	RFC	WMC
4.0.0	2970	356 (32 %)	***Minimum***	0	1	0	0	0	0
			Maximum	355	7	82	151	531	653
			Moyenne	13 654	1 862	2 304	0,493	22,34	18 392
			Écart type	23 991	1 189	3 997	3 591	34,25	34,21
4.0.1	2977	56 (5 %)	***Minimum***	0	1	0	0	0	0
			Maximum	356	7	82	151	539	648
			Moyenne	13 695	1,86	2 305	0,492	22,33	18 398
			Écart type	24 015	1 188	3 994	3 589	34,18	34 166
4.1.0	3112	121 (11 %)	***Minimum***	0	1	0	0	0	0
			Maximum	360	7	82	151	539	648
			Moyenne	13 597	1 863	2 292	0,499	22,01	17 912
			Écart type	23 536	1 176	3 906	3,52	33,78	33 602
4.1.1	3272	295 (26 %)	***Minimum***	0	1	0	0	0	0
			Maximum	361	7	82	151	546	929
			Moyenne	13,58	1,85	2 331	0,493	21,78	17 763
			Écart type	23 715	1 164	3 835	3 474	33,38	36,06
4.1.2	3264	106 (9 %)	***Minimum***	0	1	0	0	0	0
			Maximum	354	7	82	151	545	643
			Moyenne	13 692	1 842	2 475	0,5	22,1	17 864
			Écart type	23 742	1 147	3 895	3 495	33,66	32 133
5.0.0	3207	185 (17 %)	***Minimum***	0	1	0	0	0	0
			Maximum	440	7	81	150	534	644
			Moyenne	14 139	1 819	2,58	0,489	22,23	18 183
			Écart type	24 916	1 123	3 737	3 488	33,8	32 184
Total	18 802	1119							

Tableau 1: Statistiques descriptives des six versions du système POI

Le tableau 2 nous indique que le total des classes fautives des six versions représente 6 % de la taille du système. La version 4.0.0 contient plus de classes fautives constituant 12 % de sa taille tandis que la version 4.0.1 en contient le moins faisant 2 % de sa taille.

Version	Tailles du système	% classes fautives
4.0.0	2970	12 %
4.0.1	2977	1,9 %
4.1.0	3112	3,9 %
4.1.1	3272	9 %
4.1.2	3264	3,2 %
5.0.0	3207	5,9 %
Total	18 802	5,95 %

Tableau 2: Pourcentage des classes fautives par rapport à la taille du système

Du tableau 3 nous remarquons que l'ensemble des six versions du système contient plus de classes faiblement fautives que de classes fortement fautives. Ils représentent respectivement 5% et 1% de la taille de l'ensemble des versions.

Les versions 4.0.0 et 4.1.1 sont celles qui contiennent le plus de classes faiblement fautives (8% de leur taille) tandis que la version 4.0.1 en contient le moins (2% de sa taille).

Pour les classes fortement fautives, la version 4.0.0 en a le plus (4% de sa taille) alors que la version 4.1.2 en contient le moins (0,2% de sa taille).

Version	Taille	Pas fautives	Faiblement fautives	Fortement fautives
4.0.0	2970	12 %	8 %	4 %
4.0.1	2977	1,9 %	1,6 %	0,3 %
4.1.0	3112	3,9 %	2,6 %	1,3 %
4.1.1	3272	9 %	8 %	1 %
4.1.2	3264	3,2 %	3 %	0,2 %
5.0.0	3207	5,9 %	5,18 %	0,72 %
Total	18 802	5,95 %	4,86 %	1,09 %

Tableau 3: Pourcentage des différentes catégories de classes fautives

3.2.8 Entraînement des algorithmes d'apprentissage automatique

Au cours de nos travaux, plusieurs modèles d'apprentissage automatiques ont été implémentés en fonction de nos données et de nos objectifs. L'objectif principal de nos travaux est de détecter et d'identifier les composants critiques dans un système logiciel orienté objet en ayant recours à l'apprentissage automatique profond. Dans un système orienté objet, différents niveaux de granularité peuvent être considérés: le niveau de granularité package qui définit un regroupement logique d'un ensemble de classes; le niveau de granularité classe qui est un concept cohésif caractérisant les attributs et les méthodes communs à un ensemble d'objets ; et le niveau de granularité méthode qui définit des actions exécutées par les objets d'une classe. Pour nos travaux, nous avons considéré la granularité au niveau de la classe, car elle prend en compte toutes les propriétés définissant l'objet. Elle représente donc l'unité d'un système orienté objet. Par conséquent, les composants critiques à identifier et à détecter sont les classes critiques d'un système orienté objet.

Dans notre contexte, nous avons étudié la criticité d'une classe logicielle sous trois points de vue: (1) la présence ou non de fautes (2) la fréquence de fautes avec les catégories « non fautive », « faiblement fautive » et « fortement fautive », et (3) le nombre de fautes. Nous avons

ainsi construit trois modèles d'apprentissage automatique adaptés à chaque point de vue. Il s'agit respectivement (1) de la classification binaire, (2) de la classification multi classes et finalement (3) de la régression.

Un modèle d'apprentissage automatique (classificateur, regresseur, …) résulte de la formation d'un algorithme d'apprentissage automatique sur un ensemble de données. Cette formation s'effectue en 3 étapes: une étape d'entraînement (E), suivie d'une étape de test de validation de l'entraînement (TVE) et enfin d'une étape de Test sur de Nouveaux Ensembles de données (TNE). Avant la formation du modèle d'apprentissage automatique, une étape de prétraitement des données est réalisée afin d'organiser et de structurer les données en fonction du type de modèle à former. Dans notre cas, cette étape de prétraitement consiste en l'équilibrage et en la normalisation de l'ensemble de données. Nous avons appliqué la normalisation aux modèles de classification et de régression. De plus, en classification et afin de pallier le problème de données déséquilibrées le plus souvent rencontré, nos ensembles de données ont été équilibrés.

L'équilibrage est une technique de redimensionnement de l'ensemble de données de manière à avoir une répartition uniforme des observations suivant les classes de sortie. Il est réalisé soit par augmentation des observations de l'attribut minoritaire ou soit par la réduction de celles de l'attribut majoritaire. Les méthodes par augmentation et par réduction sont appelées, respectivement, le suréchantillonnage ou le sous-échantillonnage. Les deux méthodes impactent aussi bien la taille de l'attribut concerné que la taille de l'ensemble de données. Le sous-échantillonnage est déconseillé, car il peut supprimer certaines caractéristiques importantes de l'ensemble de données. Nous avons, donc, opté naturellement pour le suréchantillonnage dans le traitement de nos données.

La normalisation est une technique par laquelle les valeurs d'un ensemble de données sont rapportées à une échelle commune. Elle réduit la variance entre les valeurs des caractéristiques

des observations. Ainsi, elle réduit le risque d'explosion des calculs dû à l'usage intensif des fonctions exponentielles, dans les fonctions coût et activation, pour les algorithmes de descente de gradient. Nous avons utilisé cette technique aussi bien au cours de l'entraînement qu'au cours du TNE. Nous avons appliqué la formule de normalisation (standardisation) suivante pour cette technique:

$$X' = (X - \bar{X})/\sigma(X)$$

Dans la formule ci-dessous, X représente l'ensemble de données à normaliser, $\bar{X}$ la moyenne et $\sigma(X)$ l'écart type.

Au cours du TNE, nous avons utilisé trois différentes stratégies de test afin d'évaluer les performances de nos modèles sous différents aspects. Les trois stratégies concernent: le Test Inter-Version (TIV), le Test Inter-Version avec ACP (Analyse en Composantes Principales : TIVA) et le Test Inter-Version avec Encodeur (TIVE).

3.2.8.1 Le test Interversion (TIV)

Le test inter-version consiste à entraîner le modèle sur un ensemble de données d'une version J, puis d'effectuer le TNE sur les données de la version J+1 du système considéré. Cette stratégie de test constitue la base pour la stratégie de test inter-version avec ACP et du test inter-version avec encodeur.

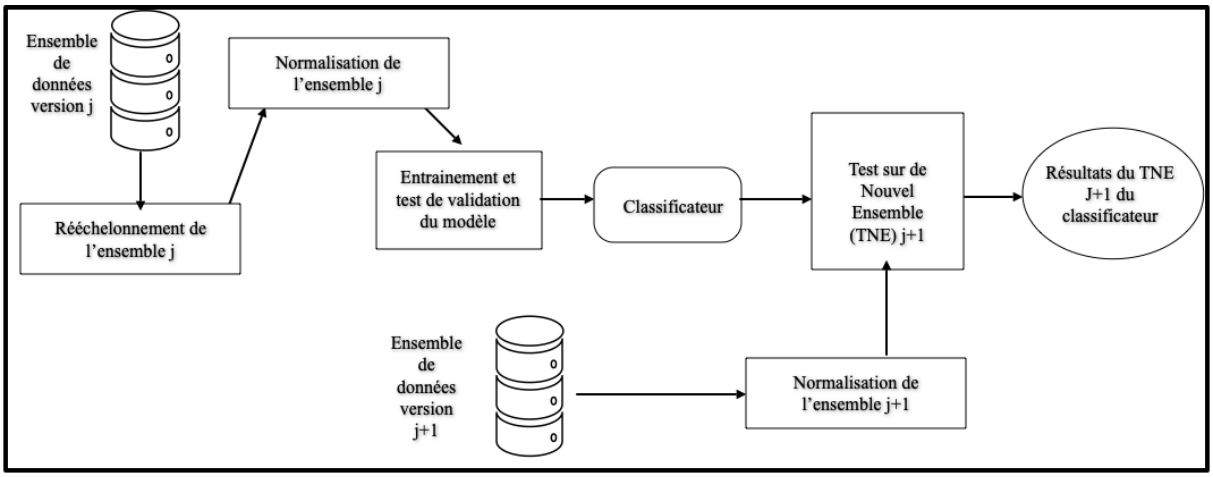

Figure 5: Test inter-version pour la classification

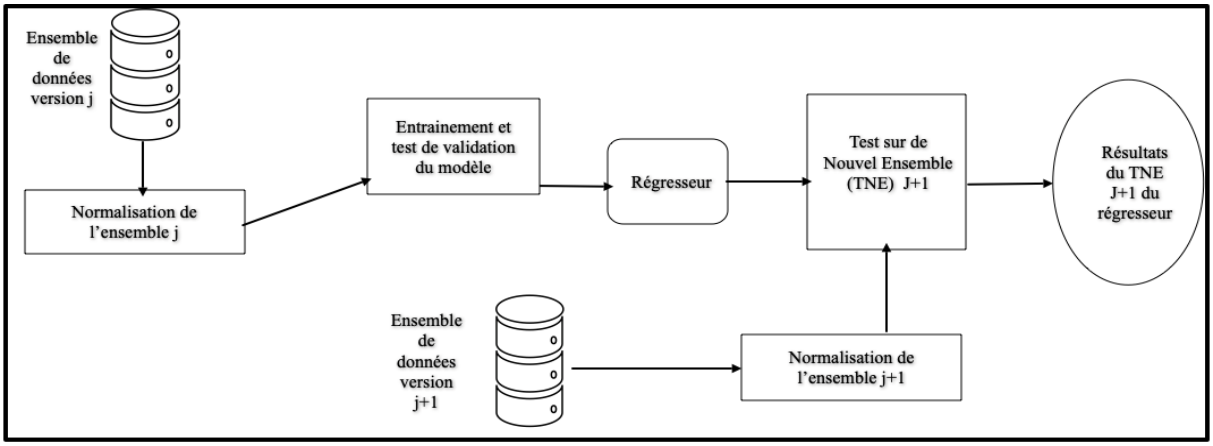

Figure 6: Test inter-version pour la régression

3.2.8.2 Le test interversion avec ACP (TIVA)

La stratégie de test interversion avec ACP est pratiquement comparable à la stratégie de test interversion. La différence réside au niveau du traitement des données. Au cours de ce test, les données normalisées dans le cas de la régression, et équilibrées en plus dans le cas de la classification, subissent une ACP avant d'être fournies à l'algorithme d'apprentissage automatique, pour l'entraînement, et au régresseur ou au classificateur pour le TNE.

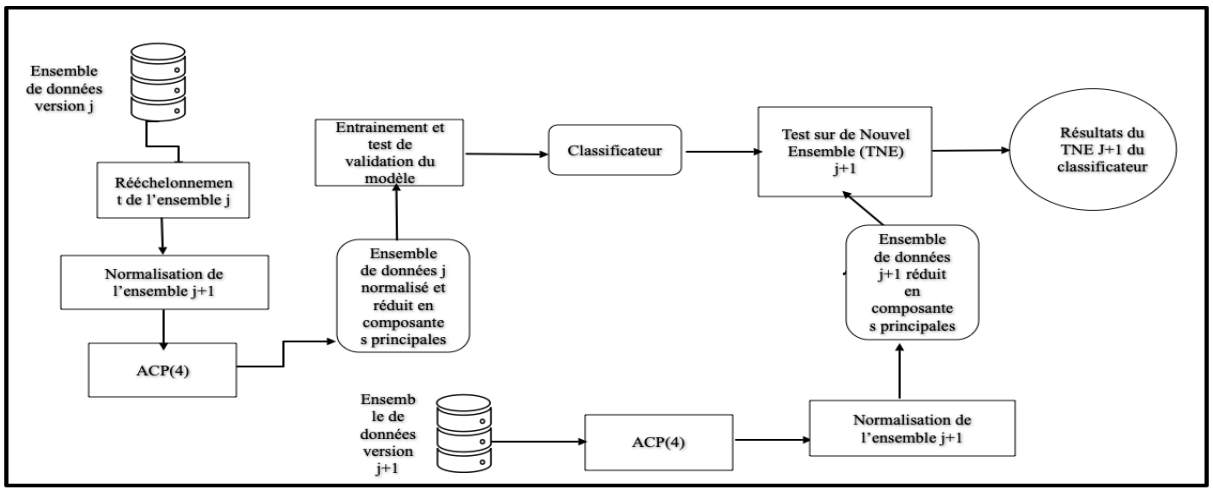

Figure 7: Test inter-version avec ACP pour la classification

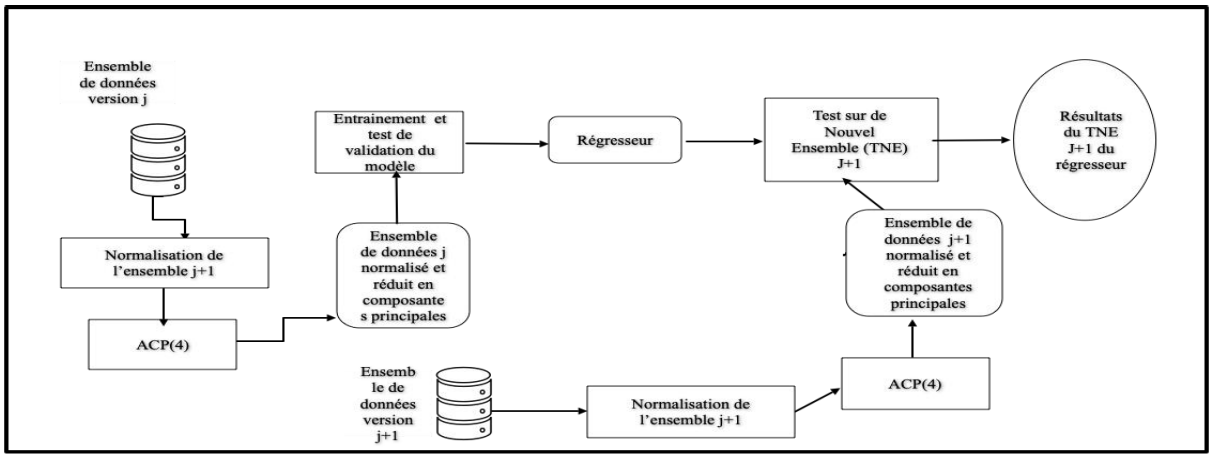

Figure 8: Test inter-version avec ACP pour la régression

L'ACP

L'Analyse en Composantes Principales est une technique de sélection, de filtrage et de réduction des variables indépendantes de l'ensemble de données [103], dont le but est d'obtenir un ensemble de données exemptées de toutes informations redondantes [104]. Le nombre de composantes principales retenu influence les performances du modèle d'apprentissage automatique. Un petit nombre de composantes risque de ne contenir qu'une partie de l'information de l'ensemble de données initial. Cependant, un trop grand nombre risque également d'inclure des informations superflues. Un nombre de composantes principales adéquat est indispensable pour une amélioration des performances des modèles. Afin de guider dans le choix du nombre de composantes principales à prendre en compte, plusieurs règles ont été mises en place dont la règle de la proportion d'inertie expliquée [105] et la règle de Kaiser [106].

La règle de la proportion d'inertie expliquée stipule que la valeur cumulative des composantes principales à prendre en compte doit atteindre un certain seuil défini préalablement. Dans notre cas, nous avons fixé le seuil à 90%.

La règle de Kaiser suggère de considérer uniquement les composantes principales dont les valeurs propres sont supérieures ou égales à 1.

Du tableau 4, nous remarquons :

- Qu'en considérant la règle de la proportion d'inertie, la valeur cumulative des composantes atteint 91 % avec les quatre premières composantes.
- Qu'avec la règle de Kaiser, les valeurs propres des quatre premières composantes sont sensiblement supérieures ou égales à 1.

Les deux règles nous suggèrent donc de retenir les quatre premières composantes.

Le tableau 5 nous indique la corrélation entre nos variables indépendantes et les composantes principales. Cette corrélation permet d'identifier les variables indépendantes que représentent les composantes principales.

De l'analyse du tableau 6, nous remarquons que:

- La composante principale 1 (axe1) représente les métriques de complexité RFC, WMC et de couplage CBO
- La composante principale 2 (axe2) représente la métrique d'héritage NOC
- La composante principale 3 (axe3) représente la métrique d'héritage DIT
- La composante principale 4 (axe4) représente la métrique de cohésion LCOM
- La composante principale 5 (axe5) représente une partie de la métrique de couplage CBO non capturée dans la première composante.
- La composante principale 6 (axe6) ne représente explicitement aucune des variables.

Nous concluons donc que les quatre premières composantes principales représentent valablement les informations de l'ensemble de données initiales.

Axes	Valeurs propres	Proportion (%)	Valeurs cumulatives (%)
1	**2,53**	**42,1**	**42,1**
2	**1,05**	**17,51**	**59,62**
3	**1,002**	**16,71**	**76,33**
4	**0,856**	**14,29**	**90,62**
5	0,443	7,4	98,01
6	0,119	1,99	100
Total	6		

Tableau 4: Valeurs propres et valeurs cumulatives de l'ACP.

Attribute	Axis1		Axis2		Axis3		Axis4		Axis5		Axis6	
	Corr.	% (Tot.%)	Corr.	% (Tot.%)	Corr.	% (Tot.%)	Corr.	%(Tot.%)	Corr.	%(Tot.%)	Corr.	%(Tot.%)
RFC	**0,933**	**87% (87%)**	-0,111	*1% (88%)*	0,025	*0% (89%)*	0,128	*2% (90%)*	-0,182	*3% (93%)*	-0,256	*7% (100%)*
WMC	**0,909**	**83% (83%)**	-0,124	*2% (84%)*	-0,033	*0% (84%)*	0,131	*2% (86%)*	-0,298	*9% (95%)*	0,228	*5% (100%)*
CBO	**0,769**	**59% (59%)**	0,279	*8% (67%)*	-0,015	*0% (67%)*	0,244	*6% (73%)*	**0,521**	***27% (100%)***	0,036	*0% (100%)*
NOC	0,128	*2% (2%)*	**0,937**	***88% (89%)***	-0,049	*0% (90%)*	-0,266	*7% (97%)*	-0,179	*3% (100%)*	-0,009	*0% (100%)*
DIT	-0,054	*0% (0%)*	0,090	*1% (1%)*	**0,986**	***97% (98%)***	0,129	*2% (100%)*	-0,029	*0% (100%)*	0,012	*0% (100%)*
LCOM	0,469	*22% (22%)*	-0,243	*6% (28%)*	0,165	*3% (31%)*	-0,822	*68% (98%)*	0,132	*2% (100%)*	0,012	*0% (100%)*
Var. Expl.	2,526	*42% (42%)*	1,051	*18% (60%)*	1,003	*17% (76%)*	0,857	*14% (91%)*	0,444	*7% (98%)*	0,119	*2% (100%)*

Tableau 5: Corrélation des variables indépendantes avec les composantes principales.

3.2.8.3 Le test Inter-version avec Encodeur (TIVE)

La stratégie de test interversion avec encodeur consiste à passer l'ensemble de données de la version J dans un encodeur de la version J, avant de le soumettre à l'algorithme pour l'entraînement. L'ensemble de données J+1 est ensuite passé dans un encodeur de la version J+1 avant d'être transmis au classificateur ou au régresseur pour le TNE.

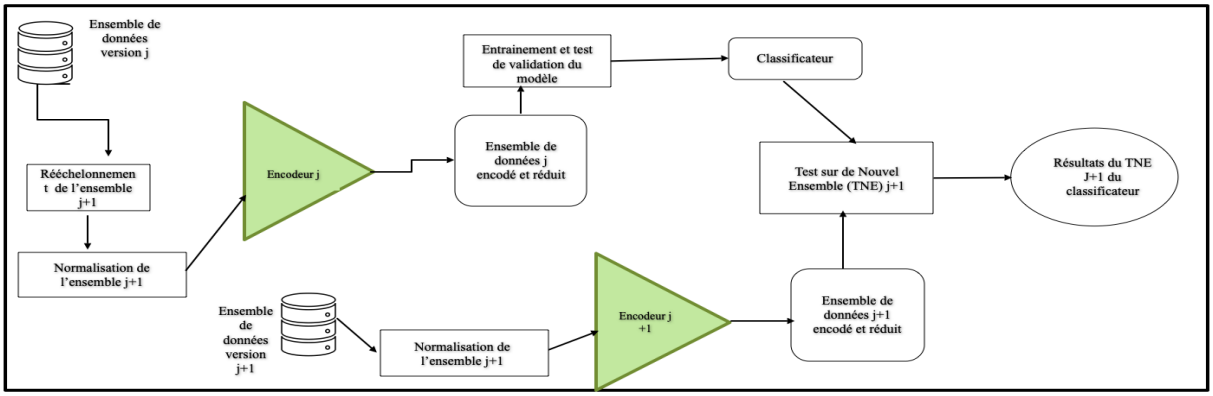

Figure 9: Test interversion avec encodeur pour la classification

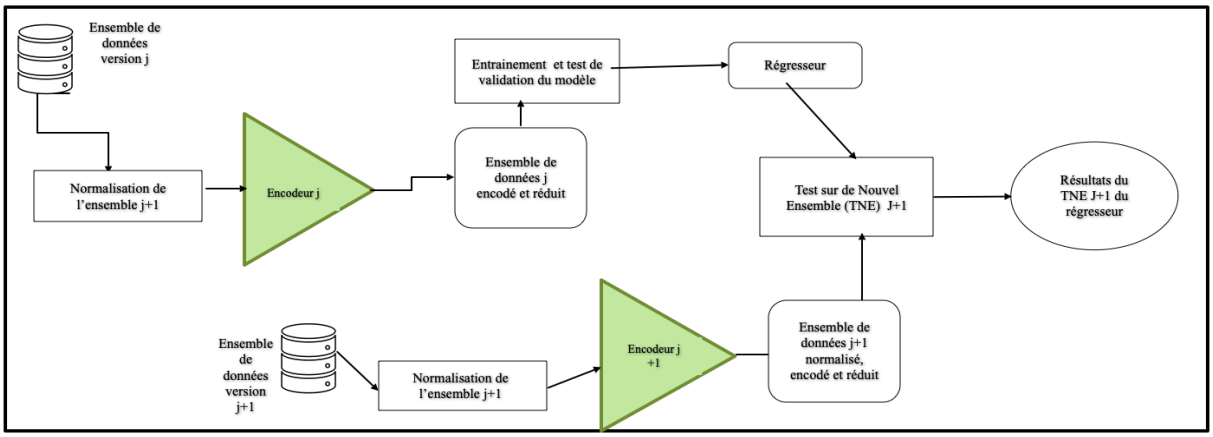

Figure 10: Test interversion pour la régression

L'encodeur

L'encodeur est un constituant de l'auto encodeur. L'auto encodeur est un modèle de RNA en forme de sablier pour l'encodage et le décodage des informations contenues dans un ensemble de données [107, 108]. Il est composé d'une couche d'entrée, de plusieurs couches

cachées et d'une couche de sortie. Les couches cachées sont constituées de l'encodeur, du goulot d'étranglement et du décodeur.

L'encodeur se charge d'encoder les données transmises par la couche d'entrée. Le goulot d'étranglement a pour fonction de réduire les informations encodées. Il constitue le filtre de l'auto encodeur et sa taille est souvent plus petite que celle de l'encodeur et du décodeur. La taille du filtre influence la performance de l'auto encodeur. Une taille de filtre trop petite risque de bloquer des informations importantes et une taille trop grande risque de laisser passer des informations superflues. Une taille adéquate de filtre (goulot d'étranglement) est donc requise. Le décodeur se charge de reconstituer l'ensemble de données initial à partir des données réduites par le goulot d'étranglement. À la fin du processus l'ensemble de données d'entrée A, est comparé à l'ensemble de données en sortie B. Plus la divergence entre ces deux ensembles est grande et plus la marge d'erreur est grande, et vice versa. Dans le cas d'une petite marge d'erreur, l'ensemble de données AR, obtenu après l'étape de réduction par le goulot d'étranglement, représente valablement l'ensemble de données initiales A. Le rôle de l'auto-encodeur, dans ce cas, est de comparer les données initiales (A) aux données réduites (AR), afin de confirmer leur ressemblance ou leur disparité.

En fin de compte, l'auto encodeur est constitué de deux modèles dont un modèle d'encodage composé de la couche d'entrée, de l'encodeur et du goulot d'étranglement et un second modèle de décodage constitué du décodeur et de la couche de sortie. Seul le modèle d'encodage est sauvegardé, et est appliqué à notre ensemble de données pour la réduction de la taille des données.

L'encodeur est une ACP dynamique basée sur un algorithme d'apprentissage profond. Il révèle les caractéristiques abstraites des données que l'ACP n'est pas capable de détecter.

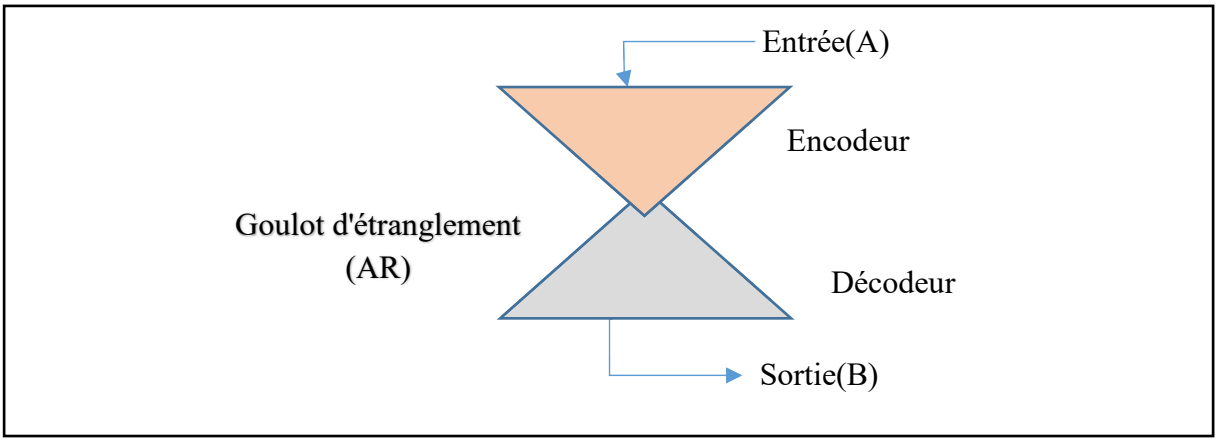

Figure 11: Structure de l'auto encodeur

Nous disposons de six versions du système POI, et pour chacune des versions nous avons conçu un encodeur spécifique par entraînement. Nous avons donc six modèles d'encodage correspondant à ces six versions concernées.

Les auto-encodeurs ont été exécutés avec de différentes tailles de goulot d'étranglement (GoE) afin d'identifier la taille minimale qui fournit une erreur négligeable à la sortie. Les performances des auto-encodeurs ont été évaluées avec la mesure de l'exactitude selon la formule:

$$Exactitude = 1 - erreur.$$

Du tableau 6, nous remarquons que les performances de l'auto-encodeur varient de 17 à 73 % avec une taille du GoE fixé à 2, de 72 à 81 % avec une taille de GoE à 3 et de 60 à 83 % avec une taille de GoE à 4. Nous avons utilisé le critère du plus faible écart entre l'exactitude minimum et l'exactitude maximum des performances de l'auto encodeur pour le choix du plus performant. Nous remarquons alors (dans le tableau 6) que l'auto-encodeur avec la taille du GoE à 3 est le plus performant. Nous avons construit six encodeurs avec le GoE fixé à 3 pour chacune des six versions du système POI.

Versions	Exactitude en %		
	GoE2	**GoE3**	**GoE4**
4.0.0	68,82	***81,41***	76,46
4.0.1	53,9	***72,23***	76,61
4.1.0	16,9	***78,69***	77,08
4.1.1	62,27	***81,01***	82,67
4.1.2	72,87	***79,3***	59,95
5.0.0	56,15	***77,56***	78,46

Tableau 6: Résultat d'exécution des auto encodeurs

3.2.9 *Structure des réseaux de neurones profonds*

3.2.9.1 Structure du Réseau de Neurones Artificiels (RNA)

Après plusieurs tentatives d'exécution, nos algorithmes d'apprentissage automatique retenus pour l'entraînement des modèles sont constitués de:

- Une couche d'entrée : sa taille varie de 3 (pour les modèles qui utilisent l'encodeur) à 6 (pour les modèles qui prennent toutes nos variables indépendantes en entrée). L'encodeur réduit la taille de ces variables de 6 à 3 afin de réduire les informations redondantes. La couche d'entrée contient 144 neurones pour un total d'entrées variant de 442 à 864.
- 3 couches cachées avec chacune 144 neurones. La couche d'entrée et les couches cachées utilisent la fonction d'activation ReLU.
- Une couche de sortie dont la taille et la fonction d'activation diffèrent selon le type du modèle. Le tableau 7 présente ces paramètres selon les modèles

Modèles	Couche de sortie	
	Taille	**Fonction d'activation**
Prédiction du nombre de fautes	1	Linéaire
Classification binaire	2	Sigmoïde
Classification multi classe	3	Softmax

Tableau 7: Taille et fonction d'activation par type de modèle

- Un optimiseur Adam avec un pas d'apprentissage de 0,005. Un optimiseur est une fonction qui réduit le taux d'erreur de la fonction de perte d'un algorithme d'apprentissage automatique. Cette réduction augmente les performances de l'algorithme d'apprentissage.

- Un nombre d'époques définit le nombre de fois que l'algorithme parcourt l'ensemble de données.
- Un batch size de 50. Le batch size est le nombre d'observations utilisées, simultanément dans l'algorithme de la descente de gradient, à chaque itération pendant une époque. Le nombre d'itérations est donc proportionnel au batch size.

3.2.9.2 Structure de l'auto encodeur

L'auto encodeur est composé de:

- Un encodeur constitué d'une couche d'entrée de taille 6, de cinq couches cachées de taille respectivement décroissante : 75, 60, 45, 30 et 15
- Un goulot d'étranglement composé d'une couche cachée de taille 3.
- Un décodeur qui contient 5 couches cachées de tailles respectivement croissantes : 15, 30, 45, 60 et 75. Elle comprend aussi une couche de sortie de taille 6 avec une fonction d'activation ReLU. La taille de la couche de sortie correspond à celle de la couche d'entrée de l'encodeur. La structure du décodeur est la symétrie exacte de celle de l'encodeur.

3.2.10 *Mesures de performances des modèles*

Nous avons évalué la performance de nos modèles en nous basant sur la fonction de perte MSE (Mean Squared Error) qui est l'erreur quadratique moyenne. Elle mesure le carré de la moyenne de l'écart entre la sortie réelle et la sortie prédite. Plus sa valeur est petite, plus le modèle est performant.

3.2.11 *Outils logiciels de collecte et de prétraitements des données*

L'ASF (*Apache Software Fondation*) utilise les plateformes *GitHub [109]* et ASF Bugzilla [110] pour gérer l'historique des erreurs du système POI. Nous avons collecté les classes fautives sur ces deux plateformes. Nous avons utilisé pour le prétraitement des données Excel[111] de la suite Office de Microsoft, l'environnement de développement *IntelliJ [112]*, le plug-in *Metrics Reloaded [113]* et le logiciel Tanagra *[114]*.

3.2.11.1 **GitHub**

GitHub est une plateforme commerciale propriété de *Microsoft.* Elle héberge des projets logiciels en ligne, indépendamment du langage de développement. Basée sur le système de gestion de version de *Git*, elle permet d'avoir un historique des contributions de chacun des membres de l'équipe projet, d'une version à une autre. Elle est lancée depuis avril 2008 et est écrite en JavaScript, Golang, Ruby, C et autres.

De *GitHub*, nous avons pu collecter certaines des classes fautives du système POI de la version 4.0.0 à la version 5.0.0. *Github* est alimenté par les commits des développeurs. Ces commits contiennent, entre autres, des classes logicielles. En général, ces classes logicielles sont soit de nouvelles classes introduites, soit des classes existantes, mise à jour pour corriger une faute ou pour ajouter de nouvelles fonctionnalités au système. Les classes fautives dans notre cas sont toutes ces classes logicielles existantes mises à jour pour la correction des fautes lors de l'utilisation du système POI.

3.2.11.2 Apache Software Fondation (ASF) Bugzilla

ASF Bugzilla est une interface web de suivi des erreurs développée par Mozilla [115], utilisée par ASF pour la gestion de la majorité des erreurs de POI. Bugzilla présente les caractéristiques (statut, importance, sévérité) de la faute à corriger contrairement à GitHub, et une référence vers l'ensemble des fichiers impactés par la faute. Ces fichiers sont accessibles via *ViewVC* [116]. *ViewVc* est un logiciel open source qui permet de visualiser les fichiers des dépôts *CSV* ou *SVN* via un navigateur. *ViewVC* présente le lien vers les fichiers impactés ainsi que les actions menées sur chaque fichier concerné. Les actions rassemblent la modification, la surpression et l'ajout effectués sur une classe logicielle.

3.2.11.3 Excel de la suite Microsoft Office

Excel [111] a permis de réaliser la binarisation de la colonne « nombre de fautes » dans le cadre de la classification binaire et le calcul de la moyenne du nombre de fautes pour la création des trois groupes concernant la classification multi classes.

3.2.11.4 IntelliJ

IntélliJ [112] est un environnement de développement multiplateforme écrit en Java avec une interface en Swing supportant les langages comme Java, Kotlin, Groovy, Scala, JavaScript, TypeScript, HTML, CSS et le langage de requête structuré. Il est développé par JetBrains. *IntelliJ* a servi à calculer, pour les classes de chaque version considérée, les métriques de CK [53] à l'aide du plug-in *Metrics Reloaded.*

3.2.11.5 Metrics Reloaded

Metrics Reloaded [117] est un plug-in de l'IDE IntelliJ dédié au calcul des métriques de code source des systèmes logiciels. Il calcule automatiquement plusieurs métriques entre autres celles de CK. Ainsi, Metrics Reloaded calcule également le nombre de classes, la dépendance, la complexité, le nombre d'assertions et de méthodes de test *JUnit*, la couverture des tests, le nombre de lignes de code et plus encore. Ces métriques sont calculées selon différentes granularités (niveau global système, niveau module, niveau fichier actuel …).

3.2.11.6 Tanagra

Tanagra [114] est une application gratuite d'analyse de données utilisée dans l'enseignement et dans la recherche. Elle intègre plusieurs méthodes de fouille de données, de statistiques et d'apprentissage automatique. Nous l'avons utilisé pour réaliser l'Analyse en Composantes Principales (ACP) de nos données.

CHAPITRE4. RÉSULTATS DES EXPÉRIMENTATIONS ET INTERPRÉTATIONS

Les expérimentations réalisées au cours de ces travaux concernent trois types de modèles d'apprentissage automatique: la régression pour la prédiction du nombre de fautes, la classification binaire pour la prédiction de la présence de faute (classes fautives ou non fautives) et la classification multi classes pour la prédiction de la fréquence de fautes (non fautive, faiblement fautive et fortement fautive).

Pour chaque type de modèle, nous avons appliqué trois stratégies de test sur de nouveaux ensembles (TNE) : le test inter-version (TIV), le test inter-version avec ACP (TIVA) et le test inter-version avec encodeur (TIVE). Dans chacune des stratégies de test, les performances du réseau de neurones artificiels sont comparées à celles de la régression linéaire dans le cadre de la régression, de la régression logistique pour la classification binaire ou de la régression multi classes pour la classification multi classes. Les résultats des modèles sont indiqués en exactitude afin de mieux appréhender leurs performances. L'exactitude est obtenue par la formule :

$$Exactitude = 1 - MSE.$$

4.1 Prédiction du nombre de fautes.

Nous avons réalisé ces expérimentations dans le but de répondre aux questions 1 et 2 de recherche :

Question 1 : Peut-on construire un régresseur basé sur l'apprentissage automatique profond et sur les données des métriques OO, capable de prédire de manière satisfaisante le nombre de fautes dans les classes logicielles ?

Question 2 : Comment se situent les performances d'un tel régresseur comparées à celles de la régression linéaire ?

Le tableau 8 présente l'ensemble des résultats obtenus pour cette série d'expérimentations.

La colonne TIV du tableau 8 nous indique que le taux d'exactitude le plus élevé pour le RNA est de 94% tandis que celui de la RL est de 95%. En moyenne, l'exactitude du RNA est de 89% contre 91% pour la RL.

De la colonne TIVE du tableau 8, nous remarquons que le RNA présente un taux d'exactitude élevé de 99% contre 96% pour la RL. L'exactitude moyenne est de 93 % pour le RNA contre 92 % pour la RL.

La colonne TIVA du tableau 8 nous indique que le RNA et la LR ont tous une meilleure performance de 95%. La valeur moyenne de l'exactitude est de 89% pour le RNA et de 90% pour la RL.

En réponse à la question 1 de recherche, le RNA a une capacité de prédiction moyenne qui varie de 89% en TIV à 93% en TIVE, pour la prédiction du nombre de fautes. L'application de l'encodeur, pour la réduction des informations redondantes, a légèrement amélioré les performances du RNA. Les performances du RNA en TIVA indiquent que l'analyse en composante principale des données n'a pas eu d'impact sur sa performance.

À la question 2 de recherche, le taux d'exactitude de la RL est légèrement supérieur ou égal à celui du RNA, dans toutes les stratégies de TNE, à l'exception de la stratégie TIVE où celui du RNA est légèrement supérieur. Cependant, au niveau du modèle 2 (ligne 3 du tableau 8), les performances du RNA sont supérieures ou égales à celles du RL dans toutes les stratégies de TNE.

En TIV et en TIVA, la RL est sensiblement plus performante que le RNA. Les résultats moyens en TIVE montrent que l'encodeur impacte plus le RNA que la RL. Les performances en TIVA indiquent que l'ACP influence légèrement la RL, mais n'a pratiquement aucun effet sur le RNA.

Régresseurs	Taux Exactitude en % des Tests Inter-versions					
	TIV		TIVE		TIVA	
	RNA	RL	RNA	RL	RNA	RL
4.0.0->4.0.1 (1)	***85,8***	93,16	***87,71***	**94,75**	87,44	92,93
4.0.1->4.1.0 (2)	**94,25**	90,86	92,57	90,52	**94,69**	90,85
4.1.0->4.1.1 (3)	*86,6*	***86***	*98,77*	***86,2***	*86,68*	***86,07***
4.1.1->4.1.2 (4)	87,47	**94,9**	**99,06**	95,14	***84,01***	**95.41**
4.1.2->5.0.0 (5)	91,11	91,6	88,13	91,8	92,34	91,51

Tableau 8: Prédiction du nombre de fautes

4.2 Prédiction de la présence et de la fréquence de fautes

Ces expérimentations nous permettent de répondre aux questions de recherches 3 et 4:

Question 3 : Peut-on construire un classificateur basé sur l'apprentissage automatique profond et sur les données des métriques logicielles capable de prédire de manière satisfaisante la présence ou la fréquence de fautes (de manière qualitative) dans les classes logicielles ?

Question 4 : Comment se situent les performances d'un tel classificateur comparées à celles de la régression logistique (RLo) et de la régression logistique multi classes (RLoM) ?

4.2.1 *Prédiction de la présence de fautes*

Le tableau 9 présente les différents résultats des expériences de prédiction de la présence de fautes dans les classes logicielles.

En TIV, le taux d'exactitude le plus élevé pour le RNA est de 91% contre 82% pour la RLo. Le taux d'exactitude moyen pour le RNA pour l'ensemble des modèles est de 85% tandis que celui de la RLo est de 77%.

En TIVE, les meilleurs classificateurs du RNA et de la RLo ont pratiquement le même taux d'exactitude respectivement 94% et 93%. Le taux d'exactitude moyen est de 83% pour le RNA et de 71% pour la Rlo.

En TIVA, en moyenne les taux d'exactitude pour le RNA et la RLo sont respectivement de 78% et 61%. Le RNA a une meilleure performance de 83% contre 69% pour la RLo.

Classificateurs	Exactitude en % des Tests inter-verion					
	TIV		TIVE		TIVA	
	RNA	RLo	RNA	RLo	RNA	RLo
4.0.0->4.0.1 (1)	85,67	78,8	**94,21**	90,23	75,74	65,55
4.0.1->4.1.0 (2)	**91,43**	**81,56**	81,97	**93,4**	**74,68**	56
4.1.0->4.1.1 (3)	83,67	73,11	81,41	70,95	76,99	66,43
4.1.1->4.1.2 (4)	**83,51**	**71,4**	**71**	**16,18**	78,23	**68,57**
4.1.2->5.0.0 (5)	86,91	79,83	87,8	82,22	**83,06**	**48,88**

Tableau 9: Prédiction de la présence de fautes

4.2.2 *Prédiction de la fréquence de fautes*

Le tableau 10 présente les résultats des expérimentations sur la prédiction de la fréquence de fautes dans les classes logicielles. Les résultats en TIV indiquent que le RNA a une meilleure performance (91%) contre 81 % pour la RLoM. Les performances moyennes varient de 67% pour le RLoM à 83% pour le RNA.

En TIVE, nous observons en moyenne le taux d'exactitude de 64% pour le RloM contre 72% pour le RNA. Les classificateurs les plus performants présentent des taux d'exactitude sensiblement égaux de 89% pour le RNA et 88% pour le RloM.

Concernant le TIVA, les meilleurs résultats obtenus sont de 68% pour le RNA à 51% pour le RloM. Nous constatons qu'en moyenne le taux d'exactitude des modèles donne respectivement 78% pour le RNA et 61% pour le RloM.

Classificateurs	Exactitude en % des Tests inter-version					
	TIV		TIVE		TIVA	
	RNA	RLoM	RNA	RLoM	RNA	RLoM
4.0.0->4.0.1 (1)	***76,81***	69,79	***31,22***	**87,77**	***44,21***	34,52
4.0.1->4.1.0 (2)	**91,1**	**80,68**	70,37	71,58	**68,24**	12,52
4.1.0->4.1.1 (3)	81,23	***44,15***	85,47	67,51	64,11	37,25
4.1.1->4.1.2 (4)	81,53	65,15	84,56	***10,79***	62,98	**51,22**
4.1.2->5.0.0 (5)	85,6	75,83	**88,76**	80,79	49,31	***3,93***

Tableau 10: Prédiction de la fréquence de fautes logicielles

4.2.3 *Réponses aux questions 3 et 4 de recherche*

Pour la question de recherche 3, nous notons que le RNA a une capacité moyenne de 85%, 83% et 78% pour la prédiction de la présence de fautes, respectivement en TIV, TIVE et en TIVA.

Pour la prédiction de la fréquence de fautes, il a une capacité moyenne de 83% en TIV, de 72% en TIVE et de 78% en TIVA. Nous remarquons que l'application de l'encodeur et de l'ACP sur l'ensemble de données a eu un impact négatif sur les performances des classificateurs avec une baisse de performances passant de 85% en TIV à 78% en TIVA pour les classificateurs de prédiction de la présence de fautes, et de 83% en TIV à 72% en TIVE pour les classificateurs de prédiction de la fréquence de fautes.

En réponse à la question 4 de recherche, la comparaison des taux d'exactitude moyens montre que le RNA fournit de meilleurs résultats que la Rlo dans la prédiction de la présence de fautes dans toutes les stratégies de test, mais plus en TIV qu'en TIVE et TIVA. Nous constatons que l'application de l'encodeur et de l'ACP à nos données n'a pas d'impact positif sur les performances ni de la Rlo, ni sur celles du RNA.

Pour la prédiction de la fréquence de fautes, le RNA est plus performant que la RloM dans tous les TNE avec un taux d'exactitude moyen de 83% contre 67% en TIV, de 72% contre 64% en TIVE et de 78% contre 61% au TIVA. Nous remarquons que la réduction des informations redondantes de l'ensemble des données par l'encodeur et l'ACP n'a amélioré ni les performances du RNA, ni celles de la RLoM.

CHAPITRE5. MENACE DE VALIDITÉ ET CONCLUSION

5.1 Menace de validité

Nous avons identifié des points qui pourraient influencer les résultats de nos travaux :

5.1.1 *Menace de validité externe*

Nous avons entraîné et testé nos modèles d'apprentissage automatique sur les classes logicielles des six versions du système POI. Étant donné que ces classes logicielles sont conçues en Java, les valeurs des métriques OO calculées et utilisées pourraient ne prendre en compte que les caractéristiques propres à ce langage de programmation [118]. Un test de nos modèles sur les classes logicielles d'autres systèmes OO, conçus dans d'autres langages de programmation autres que Java, est donc indispensable pour la généralisation de nos résultats.

5.1.2 *Menace de validité conceptuelle*

Nos modèles d'apprentissage automatique ont été entraînés sur six versions du système POI de la version 4.0.0 à la version 5.0.0. Certaines classes logicielles sont conservées (avec ou sans modification) d'une version à une autre. Ces classes logicielles sont donc communes à toutes les versions. Par conséquent, une interdépendance entre les différentes versions existe dans nos données de recherche. Cette interdépendance pourrait avoir eu un impact sur les résultats de nos modèles. Nous avons utilisé ces données afin de permettre à nos modèles de s'entraîner sur une grande quantité de données (nous n'avons que ces données de recherche à notre disposition) et d'évaluer leur capacité de prédiction sur différentes versions du même système logiciel. L'entraînement de nos modèles sur différents systèmes logiciels (au lieu de différentes versions du même système logiciel) pourrait leur permettre de fournir de meilleurs résultats.

5.2 Conclusion

La problématique traitée dans ce mémoire concerne la détection des composants critiques dans les systèmes orientés objet par l'utilisation de l'apprentissage automatique profond et les métriques orientées objet. Un composant critique est un composant dans lequel la présence de

fautes empêche le bon fonctionnement du système logiciel. Nous avons considéré que les composants critiques à détecter sont les classes logicielles critiques [6, 7], étant donné que la classe logicielle est un concept cohésif regroupant des données et des fonctions dans un système orienté objet.

Nous avons étudié la criticité d'une classe sous les points de vue de la présence/absence de fautes, de la fréquence de fautes et du nombre de fautes en son sein. Par conséquent, nous avons mis en place, en fonction de chacun de ces points de vue, des modèles d'apprentissage automatiques pour l'identification des classes logicielles critiques dans les systèmes orientés objet, afin de guider les testeurs dans la priorisation de ces classes critiques lors des tests.

Les travaux réalisés dans ce contexte consistent à concevoir d'abord un régresseur capable de prédire le nombre de fautes, ensuite un classificateur efficace capable de prédire la présence ou non de fautes et enfin, un second classificateur performant pour la prédiction de la fréquence de fautes dans les classes des systèmes logiciels, par l'utilisation du RNA et des métriques OO. Nous avons défini la fréquence de fautes dans une classe avec trois catégories qualificatives: « non fautive », « faiblement fautive » et « fortement fautive ».

Pour les différents modèles, nous avons comparé les performances du RNA respectivement avec celles de la régression linéaire, de la régression logistique et de la régression multi classes. Au cours de l'entraînement des modèles, nous avons adopté trois stratégies de test sur de nouveaux ensembles de données (TNE) : le test inter-version (TIV), le test inter-version avec encodeur (TIVE) et le test inter-version avec ACP (TIVA). Les modèles ont été entraînés sur les classes logicielles de six versions (4.0.0-5.0.0) du système POI.

Les résultats de nos expérimentations nous montrent que les régresseurs obtenus de l'algorithme d'apprentissage automatique profond (RNA) et des métriques OO ont une capacité satisfaisante (variant entre 94 à 99%) de prédiction du nombre de fautes dans les classes logicielles des systèmes OO. Les performances de ces régresseurs sont sensiblement égales à

celles de la régression linéaire (95%) dans toutes les stratégies de validation à l'exception du TIVE où celles du RNA sont meilleures (99%).

Concernant les classificateurs obtenus de l'algorithme d'apprentissage automatique profond (RNA) et des métriques OO, ils ont une capacité de 94% pour prédire la présence de fautes et de 91% pour prédire la fréquence de fautes dans les classes logicielles orientées objet. Pour la prédiction de la présence de fautes, les classificateurs du RNA sont meilleurs que ceux de la RLo dans toutes les stratégies de validation à l'exception du TIVE où leurs performances sont sensiblement égales. Pour la prédiction de la fréquence de fautes, les performances des classificateurs du RNA (68%-91%) sont meilleures que celles de la RLoM (51%-88%) dans toutes les stratégies de validation.

Au cours de nos travaux, en plus des questions de recherche nous avons aussi étudié: (1) le pouvoir prédictif de nos modèles d'apprentissage automatique profond entraînés et validés sur des ensembles de données d'une version j, à prédire les fautes dans la version j+1 d'un système logiciel, (2) l'impact de l'encodeur et de l'ACP sur les performances du réseau de neurones artificiels, de la régression linéaire et de la régression logistique. Les résultats obtenus montrent qu'en moyenne, le RNA a une capacité de 89%, de 85% et de 83%, pour prédire respectivement le nombre, la présence et la fréquence de fautes dans une version j+1 d'un système logiciel.

L'application de l'encodeur a en moyenne un impact positif sur les performances du RNA qui sont passés de 89 à 93% dans la prédiction du nombre de fautes, tandis que celles de la RL n'ont pratiquement pas évolué (91 à 92%). Par contre, il a un impact négatif sur les performances moyennes du RNA (85 à 83%, 83 à 72%) ainsi que sur celles de la RLo (77 à 71%) et de la RLoM (67 à 64%) respectivement dans la prédiction de la présence et de la fréquence de fautes. Nous pouvons conclure suivant ces résultats que l'encodeur a un impact positif sur le RNA et n'a aucun impact sur la RL en prédiction de nombre de fautes. Ces résultats pourraient s'expliquer par le fait que l'encodeur basé sur le RNA soit prédisposé à capturer les

caractéristiques non linéaires [119] des données, contrairement à la RL qui est plus propice aux caractéristiques linéaires[120]. En termes de prédiction du nombre et de la fréquence de fautes, l'encodeur contribue à la baisse des performances aussi bien pour le RNA que pour la RLo et la RLoM.

La réduction des données avec l'ACP n'a en moyenne aucun impact sur les performances RNA (89%), ni sur celles du RL (90%) en prédiction du nombre de fautes. En termes de prédiction de la présence et de la fréquence de fautes, il conduit (comme l'encodeur) à la baisse des performances de tous les modèles : RNA (85 à 78 %, 83 à 78 %), RLo (77 à 61 %) et RLoM (67 à 61 %).

Nos futurs travaux consisteront à explorer la capacité d'un algorithme d'apprentissage automatique profond, le RNA dans la prédiction du nombre de fautes des classes logicielles regroupées par sévérité. Cette approche apportera une précision sur les fautes nécessitant plus de ressources pour leur correction.

Nous envisageons travailler aussi sur un modèle fusionnant les résultats de plusieurs modèles de RNA entraîné sur différents ensembles de données (différentes versions d'un système ou différents systèmes orientés objet) dans la prédiction de la présence de fautes ou du nombre de fautes ou de la fréquence de fautes. Avec cette approche, nous pourrons améliorer les performances de nos modèles [121].

Références

[1] M. A. Jamil, M. Arif, N. S. A. Abubakar, and A. Ahmad, "Software Testing Techniques: A Literature Review," in *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*, 22-24 Nov. 2016 2016, pp. 177-182, doi: 10.1109/ICT4M.2016.045.

[2] K. Aggarwal, *Software engineering*. New Age International, 2005.

[3] G. G. Schulmeyer, I. C. Society, Ed. *Handbook of software quality assurance*. Artech House, Inc., 2007.

[4] B. Dugalic and A. Mishev, "ISO Software Quality Standards and Certification," in *BCI (Local)*, 2012: Citeseer, pp. 113-116.

[5] S. Kaur, "Software Quality," *International Journal of Computers & Technology,* vol. 3, no. 1, 2012.

[6] V. Gupta and J. Kumar Chhabra, "Package coupling measurement in object-oriented software," *Journal of computer science and technology,* vol. 24, no. 2, pp. 273-283, 2009.

[7] O. Nierstrasz, "A Survey of Object-Oriented Concepts," ed, 1989.

[8] B. Kitchenham and S. L. Pfleeger, "Software quality: the elusive target [special issues section]," *IEEE software,* vol. 13, no. 1, pp. 12-21, 1996.

[9] J. G. Cooper and K. A. Pauley, "Healthcare software assurance," in *AMIA Annual Symposium Proceedings*, 2006, vol. 2006: American Medical Informatics Association, p. 166.

[10] M. Khanna, "A Systematic Review of Ensemble Techniques for Software Defect and Change Prediction," *E-Informatica Software Engineering Journal,* vol. 16, no. 1, 2022, Art no. 220105, doi: 10.37190/e-Inf220105.

[11] D. R. Wallace and R. U. Fujii, "Software verification and validation: an overview," *Ieee Software,* vol. 6, no. 3, pp. 10-17, 1989.

[12] R. E. Fairley, "Tutorial: Static analysis and dynamic testing of computer software," *Computer,* vol. 11, no. 4, pp. 14-23, 1978.

[13] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[14] D. Sharma and P. Chandra, "Software Fault Prediction Using Machine-Learning Techniques," in *Smart Computing and Informatics*, Singapore, S. C. Satapathy, V. Bhateja, and S. Das, Eds., 2018// 2018: Springer Singapore, pp. 541-549.

[15] Neha, A. Jaiswal, and A. Tandon, "Object Oriented Fault Prediction Analysis Using Machine Learning Algorithms," in *ICDSMLA 2019*, Singapore, A. Kumar, M. Paprzycki, and V. K. Gunjan, Eds., 2020// 2020: Springer Singapore, pp. 886-892.

[16] P. Yu, T. Systa, and H. Muller, "Predicting fault-proneness using OO metrics. An industrial case study," in *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, 2002, pp. 99-107, doi: 10.1109/CSMR.2002.995794. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84884708074&doi=10.1109%2fCSMR.2002.995794&partnerID=40&md5=b63af136213017845d581e88e615cfed

[17] Z. Yuming and L. Hareton, "Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults," *IEEE Transactions on Software Engineering,* vol. 32, no. 10, pp. 771-789, 2006, doi: 10.1109/TSE.2006.102.

[18] A. Majd, M. Vahidi-Asl, A. Khalilian, P. Poorsarvi-Tehrani, and H. Haghighi, "SLDeep: Statement-level software defect prediction using deep-learning model on static code features," *Expert Systems With Applications,* vol. 147, 2020, doi: 10.1016/j.eswa.2019.113156.

[19] S. Ray, "A Quick Review of Machine Learning Algorithms," in *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*, 14-16 Feb. 2019 2019, pp. 35-39, doi: 10.1109/COMITCon.2019.8862451.

[20] D. Kumari and K. Rajnish, "Comparing efficiency of software fault prediction models developed through binary and multinomial logistic regression techniques," in *Information Systems Design and Intelligent Applications*: Springer, 2015, pp. 187-197.

[21] L. Chen *et al.*, "Empirical analysis of network measures for predicting high severity software faults," *Science China Information Sciences,* vol. 59, no. 12, p. 122901, 2016/11/07 2016, doi: 10.1007/s11432-015-5426-3.

[22] D. Kumari and K. Rajnish, "A Systematic Approach Towards Development of Universal Software Fault Prediction Model Using Object-Oriented Design Measurement," in *Nanoelectronics, Circuits and Communication Systems*, Singapore, V. Nath and J. K. Mandal, Eds., 2019// 2019: Springer Singapore, pp. 515-526.

[23] S. S. Rathore and S. Kumar, "An empirical study of some software fault prediction techniques for the number of faults prediction," *Soft Computing,* vol. 21, no. 24, pp. 7417-7434, 2017/12/01 2017, doi: 10.1007/s00500-016-2284-x.

[24] O. A. Qasem, M. Akour, and M. Alenezi, "The Influence of Deep Learning Algorithms Factors in Software Fault Prediction," *IEEE Access,* vol. 8, pp. 63945-63960, 2020, doi: 10.1109/ACCESS.2020.2985290.

[25] L. Qiao, X. Li, Q. Umer, and P. Guo, "Deep learning based software defect prediction," *Neurocomputing,* vol. 385, pp. 100-110, 2020/04/14/ 2020, doi: https://doi.org/10.1016/j.neucom.2019.11.067.

[26] K. Sneha and G. M. Malle, "Research on software testing techniques and software automation testing tools," in *2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*, 2017: IEEE, pp. 77-81.

[27] M. A. Umar, "Comprehensive study of software testing: Categories, levels, techniques, and types," *International Journal of Advance Research, Ideas and Innovations in Technology,* vol. 5, no. 6, pp. 32-40, 2019.

[28] M. Ellims, J. Bridges, and D. C. Ince, "The economics of unit testing," *Empirical Software Engineering,* vol. 11, no. 1, pp. 5-31, 2006.

[29] P. Oladimeji, M. Roggenbach, and H. Schlingloff, "Levels of testing," *Advance Topics in Computer Science,* 2007.

[30] M. E. Khan and F. Khan, "Importance of software testing in software development life cycle," *International Journal of Computer Science Issues (IJCSI),* vol. 11, no. 2, p. 120, 2014.

[31] J. Horgan and A. Mathur, "Software testing and reliability," *The Handbook of Software Reliability Engineering,* pp. 531-565, 1996.

[32] M. Tuteja and G. Dubey, "A research study on importance of testing and quality assurance in software development life cycle (SDLC) models," *International Journal of Soft Computing and Engineering (IJSCE),* vol. 2, no. 3, pp. 251-257, 2012.

[33] S. Planning, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology,* p. 1, 2002.
[34] D. Šmite, C. Wohlin, T. Gorschek, and R. Feldt, "Empirical evidence in global software engineering: a systematic review," *Empirical software engineering,* vol. 15, no. 1, pp. 91-118, 2010.
[35] H. I. Aljamaan and M. O. Elish, "An empirical study of bagging and boosting ensembles for identifying faulty classes in object-oriented software," in *2009 IEEE Symposium on Computational Intelligence and Data Mining*, 2009: IEEE, pp. 187-194.
[36] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature,* vol. 521, no. 7553, pp. 436-444, 2015.
[37] M.-H. Tang, M.-H. Kao, and M.-H. Chen, "An empirical study on object-oriented metrics," in *Proceedings sixth international software metrics symposium (Cat. No. PR00403)*, 1999: IEEE, pp. 242-249.
[38] F. Toure. "Orientation de l'effort des tests unitaires dans les systèmes orientés objet : une approche basée sur les métriques logicielles." 2018. http://hdl.handle.net/20.500.11794/27081 (accessed 24 avril 2018.
[39] S. H. Chen and C. A. Pollino, "Good practice in Bayesian network modelling," *Environmental Modelling & Software,* vol. 37, pp. 134-145, 2012.
[40] S. J. Rigatti, "Random forest," *Journal of Insurance Medicine,* vol. 47, no. 1, pp. 31-39, 2017.
[41] G. Guo, H. Wang, D. Bell, Y. Bi, and K. Greer, "KNN model-based approach in classification," in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, 2003: Springer, pp. 986-996.
[42] S. Ruggieri, "Efficient C4. 5 [classification algorithm]," *IEEE transactions on knowledge and data engineering,* vol. 14, no. 2, pp. 438-444, 2002.
[43] I. Burnstein, *Practical software testing: a process-oriented approach*. Springer Science & Business Media, 2006.
[44] A. J. Ko and B. A. Myers, "Development and evaluation of a model of programming errors," in *IEEE Symposium on Human Centric Computing Languages and Environments, 2003. Proceedings. 2003*, 31-31 Oct. 2003 2003, pp. 7-14, doi: 10.1109/HCC.2003.1260196.
[45] C. H. Fleming and N. G. Leveson, "Early concept development and safety analysis of future transportation systems," *IEEE Transactions on Intelligent Transportation Systems,* vol. 17, no. 12, pp. 3512-3523, 2016.
[46] T. Wizemann, "Trustworthy medical device software," in *Public Health Effectiveness of the FDA 510 (k) Clearance Process: Measuring Postmarket Performance and Other Select Topics: Workshop Report*, 2011: National Academies Press (US).
[47] S. Agatonovic-Kustrin and R. Beresford, "Basic concepts of artificial neural network (ANN) modeling and its application in pharmaceutical research," *Journal of pharmaceutical and biomedical analysis,* vol. 22, no. 5, pp. 717-727, 2000.
[48] A. Christmann and P. J. Rousseeuw, "Measuring overlap in binary regression," *Computational Statistics & Data Analysis,* vol. 37, no. 1, pp. 65-75, 2001.
[49] A. M. El-Habil, "An application on multinomial logistic regression model," *Pakistan journal of statistics and operation research,* pp. 271-291, 2012.
[50] G. A. Seber and A. J. Lee, *Linear regression analysis*. John Wiley & Sons, 2012.
[51] D. C. Montgomery, E. A. Peck, and G. G. Vining, *Introduction to linear regression analysis*. John Wiley & Sons, 2021.

[52] S. S. Rathore and S. Kumar, "A study on software fault prediction techniques," *Artificial Intelligence Review,* vol. 51, no. 2, pp. 255-327, 2019/02/01 2019, doi: 10.1007/s10462-017-9563-5.
[53] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering,* vol. 20, no. 6, pp. 476-493, 1994, doi: 10.1109/32.295895.
[54] S. Moudache and M. Badri, "Software Fault Prediction Based on Fault Probability and Impact," in *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*, 16-19 Dec. 2019 2019, pp. 1178-1185, doi: 10.1109/ICMLA.2019.00195.
[55] N. Kayarvizhy, S. Kanmani, and V. R. Uthariaraj, "Enhancing the fault prediction accuracy of CK metrics using high precision cohesion metric," *International Journal of Computer Applications in Technology,* vol. 54, no. 4, pp. 290-296, 2016/01/01 2016, doi: 10.1504/IJCAT.2016.080493.
[56] Rajkumar, Viji, and S. Duraisamy, "An empirical approach for complexity reduction and fault prediction for software quality attribute," *International Journal of Business Intelligence and Data Mining,* vol. 13, no. 1-3, pp. 177-187, 2018/01/01 2017, doi: 10.1504/IJBIDM.2018.088429.
[57] W. Rhmann, B. Pandey, G. Ansari, and D. K. Pandey, "Software fault prediction based on change metrics using hybrid algorithms: An empirical study," *Journal of King Saud University - Computer and Information Sciences,* vol. 32, no. 4, pp. 419-424, 2020/05/01/ 2020, doi: https://doi.org/10.1016/j.jksuci.2019.03.006.
[58] L. Kumar, S. K. Sripada, A. Sureka, and S. K. Rath, "Effective fault prediction model developed using Least Square Support Vector Machine (LSSVM)," *Journal of Systems and Software,* vol. 137, pp. 686-712, 2018/03/01/ 2018, doi: https://doi.org/10.1016/j.jss.2017.04.016.
[59] R. Martin, "Oo design quality metrics," *An Analysis of Dependencies,* vol. 12, pp. 151-170, 1994.
[60] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering,* Article vol. SE-2, no. 4, pp. 308-320, 1976, doi: 10.1109/TSE.1976.233837.
[61] B. Henderson-Sellers, "The mathematical validity of software metrics," *SIGSOFT Softw. Eng. Notes,* vol. 21, no. 5, pp. 89–94, 1996, doi: 10.1145/235969.235994.
[62] M.-H. Tang, M.-H. Kao, and M.-H. Chen, "Empirical study on object-oriented metrics," in *International Software Metrics Symposium, Proceedings*, 1999, pp. 242-249. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-0033344924&partnerID=40&md5=6a6d1dd3bbf17d7c33cd8e4cccdedba6. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-0033344924&partnerID=40&md5=6a6d1dd3bbf17d7c33cd8e4cccdedba6
[63] B. Mahesh, "Machine learning algorithms-a review," *International Journal of Science and Research (IJSR).[Internet],* vol. 9, pp. 381-386, 2020.
[64] E. Charniak, *Introduction au deep learning*. Dunod, 2021.
[65] J. D. Kelleher, *Deep learning*. MIT press, 2019.
[66] A. Krogh, "What are artificial neural networks?," *Nature biotechnology,* vol. 26, no. 2, pp. 195-197, 2008.
[67] S.-C. Wang, "Artificial neural network," in *Interdisciplinary computing in java programming*: Springer, 2003, pp. 81-100.
[68] A. Kaushik, D. K. Tayal, K. Yadav, and A. Kaur, "Integrating firefly algorithm in artificial neural network models for accurate software cost predictions," *Journal of*

Software: Evolution and Process, https://doi.org/10.1002/smr.1792 vol. 28, no. 8, pp. 665-688, 2016/08/01 2016, doi: https://doi.org/10.1002/smr.1792.

[69] N. K. G. Manjubala Bisi, *Artificial neural network applications for software reliability prediction* (Artificial Neural Network for Software Reliability Prediction). 2017, pp. 73-102.

[70] K. Rajnish, V. Bhattacharjee, and M. Gupta, "A Novel Convolutional Neural Network Model to Predict Software Defects," *Fundamentals and Methods of Machine and Deep Learning,* https://doi.org/10.1002/9781119821908.ch9 pp. 211-235, 2022/02/24 2022, doi: https://doi.org/10.1002/9781119821908.ch9.

[71] J. Reyes, D. Ramírez, and J. Paciello, "Automatic Classification of Source Code Archives by Programming Language: A Deep Learning Approach," in *2016 International Conference on Computational Science and Computational Intelligence (CSCI)*, 15-17 Dec. 2016 2016, pp. 514-519, doi: 10.1109/CSCI.2016.0103.

[72] E. Erturk and E. A. Sezer, "A comparison of some soft computing methods for software fault prediction," *Expert Systems with Applications,* vol. 42, no. 4, pp. 1872-1879, 2015/03/01/ 2015, doi: https://doi.org/10.1016/j.eswa.2014.10.025.

[73] J. A. Dallal, "Categorisation-based approach for predicting the fault-proneness of object-oriented classes in software post-releases," *IET Software*, vol. 14, no. 5*,* pp. 525-534. [Online]. Available: https://digital-library.theiet.org/content/journals/10.1049/iet-sen.2019.0326

[74] V. P. Nelson, "Fault-tolerant computing: Fundamental concepts," *Computer,* vol. 23, no. 7, pp. 19-25, 1990.

[75] S. S. Rathore and S. Kumar, "Linear and non-linear heterogeneous ensemble methods to predict the number of faults in software systems," *Knowledge-Based Systems,* vol. 119, pp. 232-256, 2017/03/01/ 2017, doi: https://doi.org/10.1016/j.knosys.2016.12.017.

[76] A. Ebru and S. Parvinder, "A soft computing approach for modeling of severity of faults in software systems," *International Journal of Physical Sciences,* vol. 5, no. 2, pp. 74-85, 2010.

[77] M. Hamill and K. Goseva-Popstojanova, "Exploring fault types, detection activities, and failure severity in an evolving safety-critical software system," *Software Quality Journal,* vol. 23, no. 2, pp. 229-265, 2015/06/01 2015, doi: 10.1007/s11219-014-9235-5.

[78] R. Malhotra, A. Kaur, and Y. Singh, "Empirical validation of object-oriented metrics for predicting fault proneness at different severity levels using support vector machines," *International Journal of System Assurance Engineering and Management,* vol. 1, no. 3, pp. 269-281, 2010/09/01 2010, doi: 10.1007/s13198-011-0048-7.

[79] R. Jindal, R. Malhotra, and A. Jain, "Prediction of defect severity by mining software project reports," *International Journal of System Assurance Engineering and Management,* vol. 8, no. 2, pp. 334-351, 2017/06/01 2017, doi: 10.1007/s13198-016-0438-y.

[80] R. Chillarege, S. Biyani, and J. Rosenthal, "Measurement of failure rate in widely distributed software," in *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*, 1995: IEEE, pp. 424-433.

[81] L. Lazic and N. Mastorakis, "Cost effective software test metrics," *WSEAS Transactions on Computers,* vol. 7, no. 6, pp. 599-619, 2008.

[82] O. Singh, D. Aggrawal, A. Anand, and P. K. Kapur, "Fault severity based multi-release SRGM with testing resources," *International Journal of System Assurance Engineering and Management,* vol. 6, no. 1, pp. 36-43, 2015/03/01 2015, doi: 10.1007/s13198-014-0241-6.

[83] D. Dhall, R. Kaur, and M. Juneja, "Machine Learning: A Review of the Algorithms and Its Applications," in *Proceedings of ICRIC 2019*, Cham, P. K. Singh, A. K. Kar, Y. Singh, M. H. Kolekar, and S. Tanwar, Eds., 2020// 2020: Springer International Publishing, pp. 47-63.

[84] D. Sharma and N. Kumar, "A review on machine learning algorithms, tasks and applications," *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET),* vol. 6, no. 10, pp. 2278-1323, 2017.

[85] E. A. Weiss, "Biographies: Eloge: Arthur Lee Samuel (1901-90)," *IEEE Annals of the History of Computing,* vol. 14, no. 3, pp. 55-69, 1992, doi: 10.1109/85.150082.

[86] D. I. Broadhurst and D. B. Kell, "Statistical strategies for avoiding false discoveries in metabolomics and related experiments," *Metabolomics,* vol. 2, no. 4, pp. 171-196, 2006/12/01 2006, doi: 10.1007/s11306-006-0037-z.

[87] K. R. Foster, R. Koprowski, and J. D. Skufca, "Machine learning, medical diagnosis, and biomedical engineering research - commentary," *BioMedical Engineering OnLine,* vol. 13, no. 1, p. 94, 2014/07/05 2014, doi: 10.1186/1475-925X-13-94.

[88] B. L. "Réseau de neurones artificiels : qu'est-ce que c'est et à quoi ça sert." LeBigData.fr. https://www.lebigdata.fr/reseau-de-neurones-artificiels-definition (accessed 5 avril 2019.

[89] D. Gupta. "Fundamentals of Deep Learning – Activation Functions and When to Use Them?" https://www.analyticsvidhya.com/blog/2020/01/fundamentals-deep-learning-activation-functions-when-to-use-them/ (accessed.

[90] I. Shafi, J. Ahmad, S. I. Shah, and F. M. Kashif, "Impact of Varying Neurons and Hidden Layers in Neural Network Architecture for a Time Frequency Application," in *2006 IEEE International Multitopic Conference*, 23-24 Dec. 2006 2006, pp. 188-193, doi: 10.1109/INMIC.2006.358160.

[91] POI. "POI." https://poi.apache.org/changes.html (accessed.

[92] A. S. Foundation, "The Apache Software Foundation," 1999. [Online]. Available: https://www.apache.org/.

[93] A. S. Foundation. "POI faute." https://poi.apache.org/changes.html (accessed 26 janvier 2006.

[94] Jehad Al Dallal, "Predicting Object-Oriented Class Fault-Proneness: A Replication Study " *Journal of Software,* vol. vol. 13 no. 5, pp. 269-276, 2018. [Online]. Available: http://www.jsoftware.us/index.php?m=content&c=index&a=show&catid=194&id=2866.

[95] M. Rizwan, A. Nadeem, and M. A. Sindhu, "Empirical Evaluation of Coupling Metrics in Software Fault Prediction," in *2020 17th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, 14-18 Jan. 2020 2020, pp. 434-440, doi: 10.1109/IBCAST47879.2020.9044489.

[96] M. Cartwright, "An empirical view of inheritance," *Information and Software Technology,* vol. 40, no. 14, pp. 795-799, 1998/12/01/ 1998, doi: https://doi.org/10.1016/S0950-5849(98)00105-0.

[97] K. El Emam, W. Melo, and J. C. Machado, "The prediction of faulty classes using object-oriented design metrics," *Journal of Systems and Software,* vol. 56, no. 1, pp. 63-75, 2001/02/01/ 2001, doi: https://doi.org/10.1016/S0164-1212(00)00086-8.

[98] L. C. Briand, W. L. Melo, and J. Wust, "Assessing the applicability of fault-proneness models across object-oriented software projects," *IEEE Transactions on Software Engineering,* vol. 28, no. 7, pp. 706-720, 2002, doi: 10.1109/TSE.2002.1019484.

[99] I. Shuman and A. Edwin, "Cyclomatic complexity as a utility for predicting software faults," NAVAL POSTGRADUATE SCHOOL MONTEREY CA, 1990.

[100] Pivotal. "SpringBoot." https://spring.io/projects/spring-boot (accessed.
[101] G. e. l. c. Angular. "Angular." https://angular.io/ (accessed.
[102] P. G. D. Group. " PostgreSQL." https://www.postgresql.org/ (accessed.
[103] H. Abdi and L. J. Williams, "Principal component analysis," *Wiley interdisciplinary reviews: computational statistics,* vol. 2, no. 4, pp. 433-459, 2010.
[104] G. Destefanis, M. T. Barge, A. Brugiapaglia, and S. Tassone, "The use of principal component analysis (PCA) to characterize beef," *Meat science,* vol. 56, no. 3, pp. 255-259, 2000.
[105] S. Camiz and V. D. Pillar, "Identifying the informational/signal dimension in principal component analysis," *Mathematics,* vol. 6, no. 11, p. 269, 2018.
[106] L. Ferré, "Selection of components in principal component analysis: a comparison of methods," *Computational Statistics & Data Analysis,* vol. 19, no. 6, pp. 669-682, 1995.
[107] M. H. Nguyen, P. L. Nguyen, K. Nguyen, V. A. Le, T. H. Nguyen, and Y. Ji, "PM2.5 Prediction Using Genetic Algorithm-Based Feature Selection and Encoder-Decoder Model," *IEEE Access,* vol. 9, pp. 57338-57350, 2021, doi: 10.1109/ACCESS.2021.3072280.
[108] K. Cho *et al.*, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078,* 2014.
[109] P. J. H. Tom Preston-Werner, Scott Chacon, Chris Wanstrath. "Github." https://github.com/ (accessed.
[110] Mozilla. "ASF Bugzilla." https://bz.apache.org/bugzilla/ (accessed.
[111] Microsoft. "Microsoft Excel." https://www.microsoft.com/fr-ca/microsoft-365/excel (accessed.
[112] J. Brains. "Intelli J IDEA." https://www.jetbrains.com/fr-fr/idea/ (accessed.
[113] B. L. S. a. R. R. Software. "MetricsReloaded." https://plugins.jetbrains.com/plugin/93-metricsreloaded (accessed.
[114] U. Lumière-Lyon-II. "Tanagra." https://www.statisticalconsultants.co.nz/blog/tanagra-a-free-data-mining-program.html (accessed.
[115] D. H. Mozilla Foundation, Joe Hewitt (en) et Blake Ross. "Mozilla." https://mozilla.design/ (accessed.
[116] B. Fenner. "ViewVC." https://viewvc.org/ (accessed.
[117] J. Brains. "Calculate metrics." https://plugins.jetbrains.com/plugin/93-metricsreloaded (accessed.
[118] S. Moudache, "Prédiction du risque logiciel, une approche basée sur la probabilité et l'impact des fautes: évaluation empirique," Université du Québec à Trois-Rivières, 2018.
[119] Y. Shi, M. Lei, R. Ma, and L. Niu, "Learning Robust Auto-Encoders With Regularizer for Linearity and Sparsity," *IEEE Access,* vol. 7, pp. 17195-17206, 2019, doi: 10.1109/ACCESS.2019.2895884.
[120] X. Su, X. Yan, and C. L. Tsai, "Linear regression," *Wiley Interdisciplinary Reviews: Computational Statistics,* vol. 4, no. 3, pp. 275-294, 2012.
[121] X. Feng, R. Subbu, and P. Bonissone, "Locally Weighted Fusion of Multiple Predictive Models," in *The 2006 IEEE International Joint Conference on Neural Network Proceedings*, 16-21 July 2006 2006, pp. 2137-2143, doi: 10.1109/IJCNN.2006.246985.