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
MATCHA WYAO

IDENTIFICATION DES COMPOSANTS PRIORITAIRES POUR LES TESTS UNITAIRES DANS LES SYSTÈMES OO : UNE APPROCHE BASÉE SUR L'APPRENTISSAGE PROFOND

AVRIL 2020

CE MÉMOIRE A ÉTÉ ÉVALUÉ PAR UN JURY COMPOSÉ DE :

M. Fadel Toure, directeur de recherche
Département de mathématiques et d'informatique, UQTR

M. François Meunier, membre du jury
Département de mathématiques et d'informatique, UQTR

M. Dominic Rochon, membre du jury
Département de mathématiques et d'informatique, UQTR

REMERCIEMENTS

J'exprime mes sincères remerciements à mes directeurs de recherche Mr Fadel TOURE et Mr Mourad BADRI. Je vous remercie pour votre rigueur, votre patience à mon égard et vos encouragements durant ces deux dernières années de recherche. Merci d'avoir partagé votre expérience. Je vous serais, pour toujours, reconnaissant.

Je remercie également les membres de ma famille pour leurs prières, leurs encouragements et leur soutien.

RÉSUMÉ

Les systèmes logiciels actuels sont de grande taille, complexes et critiques. Le besoin en qualité exige beaucoup de tests. Souvent sous-évalué et parfois jugé peu digne d'intérêt par les développeurs, le test n'en est pas moins un maillon crucial de l'ingénierie logicielle, et un pilier fondamental sans lequel aucun logiciel ne peut être mis sur le marché avec la qualité nécessaire. Les tests unitaires sont une phase critique durant le développement d'un système logiciel. Un test unitaire est un moyen de tester un bloc de code pouvant être isolé logiquement dans un système. Dans la plupart des langages de programmation, il s'agit d'une fonction, d'un sous-programme, ou d'une classe. Dans un environnement orienté objet, le testeur développe une classe de test pour chaque classe logicielle à tester afin de détecter les fautes le plus tôt possible. Malheureusement, pour des contraintes de temps et de ressources, ces tests écrits ne couvrent souvent pas toutes les classes d'un logiciel, et encore moins au sein des classes, ils ne couvrent pas toutes les méthodes. Ainsi, les efforts de test sont souvent concentrés sur des classes particulières. Les critères liés aux choix de ces classes cibles ne sont généralement ni objectifs, ni explicites, et encore moins appliqués de manière systématique. Le testeur se base sur son expérience et sur sa connaissance du logiciel en cours de test. Pour nos travaux, nous proposons une approche basée sur l'apprentissage profond et sur l'historique du système logiciel avec pour objectif d'identifier les classes à tester en priorité. Pour cela, nous avons extrait des données logicielles de deux systèmes POI et ANT. Nous avons entraîné des modèles de réseaux de neurones sur ces données. Les classificateurs obtenus après entraînement ont été validés par différentes techniques de validation standard. Les résultats obtenus supportent fortement la viabilité de

l'approche basée sur l'apprentissage profond avec des taux d'exactitude supérieurs à soixante-dix pour cent (70 %).

ABSTRACT

Current software systems are complex and critical. The need for quality requires a lot of testing. Often undervalued by developers, the testing process is nonetheless a crucial step in software engineering, and a fundamental pillar without which no software can be released on the market. Unit testing is a critical phase during the software development. Unit testing is a way to test the smallest piece of code that can be logically isolated in a system. In most programming languages, it is a function, subroutine, or class in an object-oriented system. In an object-oriented system, the tester develops a test class for each software class to be tested in order to earlier detect faults earlier. Unfortunately, due to time and resources constrains, in many cases the developed test cases do not cover all classes. Thus, testing efforts are often focused on a subset of classes. The chosen criteria are generally neither objective nor explicit, let alone applied methodically. Based on his experiences and knowledge, a tester selects classes for which test classes will be written. During our research, we investigated an approach based on deep learning in order to suggest classes with a high priority to be tested based on the history of the software. To prove our point, we choose two Apache software systems POI and ANT on which we carried our experiments. We have trained a classification model on the data set collected from the software systems. The training phase will later on enable the classifier to predict classes to be tested or not. Once the phase of training is completed after some iterations, we validated the classifier by doing a series of tests. During the validation of our classifier, we were able to predict classes to be tested on each system with an average accuracy above seventy percent (70 %). Hence, the results we obtained strongly support the viability of our approach.

TABLE DES MATIÈRES

REMERCIEMENTS ..... **III**

RÉSUMÉ ..... **IV**

ABSTRACT ..... **VI**

TABLE DES MATIÈRES ..... **VII**

LISTE DES TABLEAUX ..... **IX**

LISTE DES FIGURES ..... **X**

Chapitre 1 INTRODUCTION **11**

1.1 Introduction 11
1.2 Contexte 13
1.3 Problématique 14
1.4 Objectif 14

Chapitre 2 ÉTAT DE L'ART **18**

Chapitre 3 COLLECTE DE DONNÉES ET MÉTHODES D'ANALYSE **25**

3.1 Les Systèmes logiciels 25
3.1.1 Apache ANT 25
3.2 Les métriques 26
3.2.1 Métriques de couplage 26
3.2.2 Métriques d'héritage 27
3.2.3 Métrique de cohésion 28
3.2.4 Métriques de complexité 28
3.2.5 Métriques de taille 29
3.3 Outils de collecte de données 30
3.3.1 Granularité méthodes 31
3.3.2 Granularité lignes de codes 31
3.3.3 Couverture selon le profil opérationnel 32
3.4 Méthode de collecte des données 32
3.5 Statistiques descriptives 34
3.5.1 Système POI 34
3.5.2 Système ANT 35

Chapitre 4 MÉTHODE EXPÉRIMENTALE **38**

4.1 Le modèle d'apprentissage profond 38
4.1.1 Apprentissage supervisé 38
4.1.2 L'apprentissage non supervisé 39

4.1.3 L'apprentissage automatique par renforcement 40
4.2 Le Réseau de Neurones artificiels 40
4.2.1 Présentation générale 41
4.2.2 Les couches du perceptron 41
4.2.3 Le neurone 43
4.2.4 La règle delta 44
4.2.5 Le minimum global 46
4.3 Construction du Réseau de neurones 47
4.3.1 La fonction d'activation 48
4.3.2 L'optimiseur 48
4.3.3 La fonction perte 48
4.4 Évaluation et Validation du classificateur 48
4.4.1 Évaluation 48
4.4.2 Validation du classificateur 49

Chapitre 5 RÉSULTATS ET INTERPRÉTATIONS 52

5.1 CVV Validation 52
5.1.1 Validation sur le système POI 52
5.1.2 Validations CVV sur les versions du Système ANT 54
5.1.3 Conclusion partielle. 55
5.2 CPVV Validation 55
5.2.1 Validations sur le système POI 55
5.2.2 Validations CPVV sur les versions du Système ANT 57
5.2.3 Conclusion partielle 58
5.3 Validation CSPVV Validation 58
5.3.1 Validation sur les versions du système POI 59
5.3.2 Validations sur les versions du Système ANT 60
5.3.3 Conclusion partielle 61
5.4 LOSOV Validation 61
5.4.1 Résultats obtenus après validation sur le système ANT 62
5.4.2 Résultats obtenus après validation sur le système POI 63
5.4.3 Conclusion partielle 64
5.5 Limitations 64

Chapitre 6 CONCLUSION GÉNÉRALE 66

Chapitre 7 RÉFÉRENCES BIBLIOGRAPHIQUES 68

LISTE DES TABLEAUX

Tableau 1 : Statistiques descriptives des métriques de POI 35
Tableau 2 : Statistiques descriptives des métriques de ANT 36
Tableau 3 : Validation CVV sur le système POI.....53
Tableau 4 : Validation CVV sur le système ANT 54
Tableau 5 : Validation CPVV sur le système POI.....55
Tableau 6 : Validation CPVV sur le système ANT.....57
Tableau 7 : Validation CSPVV sur le système POI 59
Tableau 8 : Validation CSPVV sur le système ANT.....60
Tableau 9 : Validation LOSOV sur le système ANT 62
Tableau 10 : Validation LOSOV sur le système POI.....63

LISTE DES FIGURES

Figure 1 : Évolution du nombre de classes POI 35
Figure 2 : Évolution du nombre de classes du système ANT.....37
Figure 3 : Neurone biologique Vs Neurone artificiel [71] 41
Figure 4 : Réseau de neurones artificiel ou perceptron multi couche [73] 42
Figure 5 : Neurone biologique Vs Neurone artificiel [71] 43
Figure 6 : Architecture d'un neurone artificiel [74] 43
Figure 7: Illustration de la méthode de rétro propagation [78] 45
Figure 8 : Minimum local, minimum global et surface d'erreur [79] 46
Figure 9 : Illustration de la matrice de confusion.....49
Figure 10 : Cross Version Validation : CVV 50
Figure 11 : Combined Previous Version Validation CPVV.....50
Figure 12 : Combined System and Previous Version Validation. CSPVV 51
Figure 13 : Leave One System Out Validat
ion. LOSOV 51

Chapitre 1 INTRODUCTION

1.1 Introduction

Les systèmes logiciels modernes doivent répondre à un large éventail d'exigences en raison de leur criticité et de leur longévité induisant ainsi une grande complexité en matière de développement et de maintenance. Le cycle de développement d'un logiciel est un processus long et complexe, avec plusieurs étapes [1,2,3]. Les différentes phases du cycle de développement ont toutes leur importance. La phase d'analyse produit les spécifications et exigences du système, la phase de conception permet de construire le logiciel par rapport aux exigences recueillies durant la phase d'analyse. Finalement, après l'implémentation du système, une série de vérifications et validations sont effectuées durant la phase de test [1,2,3]. Le mot « tester » en industrie est souvent associé à tout type de techniques de vérification et de validation.

Les tests constituent l'une des principales méthodes utilisées en industrie pour évaluer la conformité du système développé ou en cours de développement avec ses spécifications.

Il peut également s'agir d'un processus permettant de vérifier et de valider qu'un système logiciel :

- Réponds aux exigences commerciales et techniques qui ont guidé sa conception et son développement ;
- Fonctionne comme prévu.

Durant le processus de développement [2,4], il est important de détecter les bogues logiciels, car, ceux-ci peuvent être coûteux voire problématiques si

découverts tardivement [4,5]. L'histoire nous démontre que ces coûts peuvent se mesurer en pertes de vies humaines et matérielles.

En 1985, la machine de radiothérapie canadienne Therac-25 a mal fonctionné en raison d'un virus informatique et a administré des doses létales de radiation à des patients, faisant trois morts et trois blessés.

L'Airbus A300 de China Airlines s'est écrasé le 26 avril 1994, à la suite d'un bogue informatique, faisant 264 morts.

En avril 1999, un bogue logiciel a provoqué l'échec du lancement d'un satellite militaire de 1,2 milliard de dollars, l'accident le plus coûteux de l'histoire.

En 2015, l'avion de combat F-35 a été victime d'un bogue logiciel qui l'a empêché de détecter correctement les cibles.

Généralement, les tests s'effectuent à plusieurs niveaux ou étapes de façon successive durant le développement d'un système logiciel [6].
Les niveaux de test les plus importants sont : les tests unitaires, les tests d'intégration, et les tests de validation [6].

- Les tests unitaires : Les tests unitaires comme nous l'avons défini plus haut permettent de tester si chaque composant pris individuellement fonctionne correctement [6,7].
- Tests d'intégration [6,7] : Ils consistent à tester la connectivité ou le transfert de données entre plusieurs composants testés unitairement. Il est subdivisé en approche de haut en bas, en approche de bas en haut et en approche de type sandwich.
- Tests de validation [6,7] : qui consistent à tester le logiciel avec des données réelles pour obtenir l'approbation du client afin que le logiciel puisse être livré et les paiements reçus. Les tests de validation sont appelés également test alpha, bêta et gamma. Cette

appellation dépend de la présence ou non du client et du type du système spécifique ou générique.

Les méthodes de test sont classées en deux types : les méthodes statiques et les méthodes dynamiques [6,7].

- En technique statique, les tests sont effectués sans exécution du programme [4,6,7]. Les méthodes de test statiques consistent en l'analyse textuelle du code du logiciel afin d'y détecter des erreurs. Les méthodes utilisées sont : la revue de code, l'analyse des types, l'analyse du domaine des variables, etc. Ces revues de code permettent l'examen détaillé d'une spécification, d'une conception ou d'une implémentation par une personne ou un groupe de personnes (lecture croisée), afin de déceler des fautes, des violations de normes de développement ou d'autres problèmes.
- Les méthodes de test dynamiques consistent en l'exécution du programme à valider à l'aide d'un jeu de tests. Elles visent à détecter des erreurs en confrontant les résultats obtenus à ceux attendus conformément aux spécifications [4,6].

Plus le système testé est complexe, plus l'effort de test est élevé. Des tests efficaces et complets améliorent la qualité du logiciel et réduisent les coûts de maintenance. Ils sont cependant très coûteux et difficiles à réaliser tenant compte des ressources et du temps [8,9] durant le développement d'un système logiciel d'envergure.

1.2 Contexte

La priorisation des tests est une approche permettant de hiérarchiser et de planifier les cas de test [8]. Cette technique est utilisée pour exécuter les cas de test de priorité élevée (mettre l'accent sur les composants les plus critiques) afin de minimiser le temps, les coûts et les efforts pendant la phase de test du logiciel.

Notre travail s'inscrit globalement dans le domaine de la priorisation et de l'orientation des tests dans un environnement OO. Il fait suite aux travaux de Toure et al. [11,12] faisant appel aux modèles d'apprentissage classique : Réseaux bayésiens, Forêts aléatoires, KNN et C4.5.

1.3 Problématique

Dans un environnement orienté objet (OO), il est conseillé de tester unitairement chaque composant d'un logiciel. Ces tests doivent être centrés (en particulier) sur les composants susceptibles d'échouer ou ayant une grande importance pour le fonctionnement global [8,9]. Écrire des classes de test pour chaque classe logicielle est un processus qui nécessite beaucoup de ressources et de temps. De ce fait, dans le cas d'un logiciel OO de grande taille, il est souvent difficile, voire irréaliste, de tester toutes les classes en raison de leur nombre versus la disponibilité des ressources et du temps [9]. Devant ces contraintes, un testeur est amené à faire des choix quant aux classes à tester. Ce choix se base d'habitude sur ses expériences et connaissances du logiciel en cours de test.

L'une des questions de recherche que nous examinons dans ce travail est la suivante : peut-on suggérer des classes candidates au test en exploitant les caractéristiques et propriétés du système en cours de développement d'une part et l'expérience d'autres testeurs d'autre part ?

1.4 Objectif

Plusieurs travaux de recherche portant sur la priorisation des tests utilisant différentes métriques OO (Chidamber et Kemerer [13]) ont été proposés dans la littérature. Certaines de ces métriques, liées à différents attributs de classes logicielles, ont déjà été utilisées ces dernières années pour prédire la testabilité unitaire des classes dans des systèmes logiciels

OO (Gupta et al [14] ; Bruntink et al. [15,16] ; Badri et al. [11,12]). Ces études ont analysé divers systèmes logiciels écrits en Java, open source et dont les suites de tests unitaires JUnit sont disponibles dans des référentiels publics. L'une des observations faites dans ces études est que les tests unitaires ont été écrits uniquement pour un sous-ensemble de classes.

Le but de notre étude est de proposer une approche de sélection des classes candidates au test basée sur des techniques d'apprentissage profond (réseaux de neurones artificiels).
L'apprentissage profond [17,18] est issue de l'apprentissage automatique, qui est une branche de l'intelligence artificielle. Pour comprendre ce qu'est l'apprentissage profond, il convient donc de comprendre ce qu'est l'apprentissage automatique. L'intelligence artificielle [17] (IA) consiste à mettre en œuvre un certain nombre de techniques visant à permettre aux machines d'imiter une forme d'intelligence naturelle. L'IA se retrouve implémentée dans un nombre grandissant de domaines d'application [17].

La notion d'intelligence artificielle a vu le jour dans les années 1950 grâce au mathématicien Alan Turing [19]. Dans son livre [19], ce dernier soulève la question d'apporter aux machines une forme d'intelligence. Il décrit alors un test, aujourd'hui connu sous le nom de « Test de Turing » [20] dans lequel un sujet interagit à l'aveugle avec un autre humain, puis avec une machine programmée pour formuler des réponses sensées. Si le sujet n'est pas capable de faire la différence, alors la machine a réussi le test et, selon l'auteur, cette dernière peut véritablement être considérée comme « intelligente ». L'apprentissage machine a stagné durant de nombreuses années avant de reprendre de l'aile avec ce qu'on appelle l'apprentissage profond.

En 1988, Yann L. [22] a inventé une classe de réseaux de neurones dite convolutionelle pour effectuer une tâche de reconnaissance de caractère sur les chèques. Ensuite, plusieurs chercheurs [21,22] se sont rendu compte que les réseaux de neurones convolutionels étaient applicables à d'autres domaines que le traitement de l'image, et de manière générale à tout ce qui relève du traitement du signal. Par la suite, plus les chercheurs rajoutaient de couches aux réseaux (plus ils étaient profonds) et plus ils obtenaient de bons résultats en utilisant la même technique d'apprentissage et suffisamment de données. On peut en voir l'application concrète dans plusieurs domaines professionnels comme la reconnaissance faciale [23-26] et comportementale (sécurité et média sociaux), la recherche médicale, la reconnaissance vocale et de langage naturel, la prédiction de résultat (finance, moteur de recherche web), la classification de volume de données important et bien d'autres applications [27-31].

En génie logiciel, nous disposons d'une énorme banque de données logicielles disponibles et open source qui peuvent être utilisées dans l'analyse et la prévision de la qualité logicielle.
Voici quelques questions fondamentales qui nous ont guidés dans notre travail.

- Comment mettre à profit cette banque de données afin de supporter les testeurs dans la phase des tests unitaires afin de mieux orienter l'effort de test vers les composants les plus à risque.
- Est-il possible d'automatiser ce processus ?

Pour atteindre nos objectifs, nous avons travaillé sur deux systèmes logiciels d'Apache : POI et Ant. Pour chaque système, cinq versions différentes ont été considérées, à partir desquelles nous avons extrait certaines métriques de code source.

Ce mémoire sera divisé en plusieurs parties. Après avoir introduit la problématique au premier chapitre, s'ensuivra un état de l'art au chapitre deux. Au chapitre trois nous présenterons les données utilisées ainsi que la méthode d'analyse. Nous discuterons de la méthode expérimentale au chapitre quatre. Ensuite au chapitre cinq nous présenterons les résultats obtenus et discuterons de leur fiabilité, de leur précision ainsi que leurs limitations. Enfin au sixième chapitre, nous ferons une conclusion.

Chapitre 2 ÉTAT DE L'ART

La suggestion des composants candidats au test est un sujet qui a été exploré sous beaucoup d'angles et au cours de nombreuses études.

Dans leur article, Toure et al. [11,12] ont étudié une approche basée sur l'historique des informations logicielles pour supporter la priorisation des classes à tester. Pour atteindre cet objectif, ils ont d'abord analysé différents attributs de dix systèmes logiciels open source, écrits en Java, pour lesquels des scénarios de tests unitaires JUnit ont été développés pour plusieurs classes. Premièrement, par une analyse de la moyenne et la régression logistique, ils ont identifié les classes pour lesquelles des classes de test JUnit ont été développées par les testeurs. Deuxièmement, ils ont utilisé deux classificateurs entraînés sur les valeurs de métriques et les informations de tests unitaires collectées à partir des systèmes sélectionnés. Ces classificateurs fournissent, pour chaque logiciel, un ensemble de classes sur lesquelles les efforts de tests unitaires doivent être concentrés. Les ensembles obtenus ont été comparés aux ensembles de classes pour lesquels des classes de test JUnit ont été développées par les testeurs. Les résultats montrent que : (1) les valeurs moyennes des métriques des classes testées identifiées sont très différentes des valeurs moyennes des métriques des autres classes, (2) les attributs d'une classe influencent la décision de développer ou non une classe de test JUnit dédiée à celle-ci, et (3) les ensembles de classes suggérées par les classificateurs reflètent assez correctement la sélection des testeurs.

Yu et al. [32] ont proposé une technique de priorisation des cas de test basée sur les fautes. Cette technique qui utilise directement les connaissances théoriques sur leur capacité à détecter les fautes. La

technique est basée sur les relations entre les cas de test et les fautes logicielles.

Lazić L. et al. [33] ont utilisé des algorithmes Naïve Bayes, et des algorithmes génétiques à différents niveaux de granularité pour mettre en œuvre leurs approches de priorisation. Les résultats ont montré que l'utilisation de ces algorithmes permet une amélioration du taux de détection de fautes.

Rothermel et al. [34] ont comparé neuf techniques de hiérarchisation des cas de test basées sur la hiérarchisation aléatoire, la hiérarchisation de la couverture et la détection des défaillances. Les résultats obtenus donnent un aperçu des compromis parmi diverses techniques de hiérarchisation des cas de test.

Dans la comparaison des stratégies de priorisation, Rothermel et al. [34] ont utilisé plusieurs techniques de priorisation comme mentionnées plus haut. Parmi lesquelles, quatre techniques ciblent le taux de couverture à différents niveaux du programme: (1) la couverture totale des branches est une technique de priorisation qui consiste à trier les cas de test (après instrumentation du code) dans un ordre qui maximise le nombre total de branches couvertes, (2) la couverture incrémentale des branches qui consiste à chaque itération du tri, à sélectionner le cas de test qui couvre une nouvelle branche du flux, (3) la couverture totale des contrôles, et (4) la couverture incrémentale des contrôles. Les deux dernières techniques sont similaires aux deux précédentes stratégies à la différence qu'elles ciblent les contrôles et non les branches dans le programme. Après avoir été appliquées sur huit systèmes, les quatre techniques démontrent des performances similaires aux techniques ciblant directement l'optimisation du taux de fautes.

Dans les techniques basées sur la couverture, l'objectif principal est d'exécuter des suites de test couvrant la plupart des artefacts logiciels modifiés lors des tests de régression. Cet objectif conduit à une amélioration du taux de détection des fautes.

Des algorithmes plus complexes ont été proposés par Mirarab et al. [35]. Les auteurs ont utilisé les réseaux bayésiens pour créer un modèle unifié basé sur les informations fournies par les métriques de CK [13], les changements et les taux de couverture. L'approche ainsi définie, optimise la couverture et améliore le taux de détection des fautes par rapport à la planification aléatoire des scénarios de test.

Pour optimiser les tests de régression, Wong et al. [36] suggèrent une des premières stratégies de priorisation dans la suite de tests de régression. Elle optimise le facteur « coût par taux de couverture (des branches ayant subi le changement) ». Les auteurs se focalisent sur la priorisation d'un sous-ensemble de cas de test d'une suite de tests par la technique du *Safe Regression Selection*. Les cas de test qui touchent les zones modifiées sont sélectionnés et placés en tête de file, les autres cas sont placés à la suite dans l'ordre d'exécution. L'objectif est d'optimiser le rapport entre le coût et le taux de couverture, mais aussi vise globalement à couvrir les zones ayant subi des changements en priorité. Cette technique est un simple algorithme gourmand et itératif qui cherche à couvrir le plus de branches de code ayant subi le plus de changements.

Dans le cadre des tests fonctionnels, Bryce et Memon [37] proposent une stratégie de priorisation des suites de test visant à maximiser la couverture de l'ensemble des interactions possibles au niveau de l'interface utilisateur. Après avoir passé en revue toutes les combinaisons de scénarios de test possibles (permutations), grâce aux spécifications et à l'analyse de

l'interface utilisateur, leur algorithme (de type gourmand « gready ») va maximiser le taux de couverture avec le moins de permutations. À des fins de comparaison, différents critères sont introduits dans cet algorithme en plus du taux de couverture. Finalement, les auteurs appliquent la stratégie à quatre systèmes et déterminent les performances des techniques grâce à la métrique APFD [35]. Les résultats montrent que cibler la couverture en se basant sur leur technique donne le meilleur taux de détection (APFD) de fautes.

Certains auteurs utilisent le taux de couverture pour minimiser (réduire) la taille d'une suite de test. La réduction des suites est souvent basée sur la couverture [35, 38]. Il s'agit de trouver un sous-ensemble réduit de la suite de tests originale qui maximise le taux de couverture (code/composants/spécifications). Notons que, d'une part, l'utilisation itérative de la minimisation des cas de test (en sélectionnant les cas de test retenus à chaque étape) permet de prioriser les cas de test et que, d'autre part, l'utilisation de seuils pour la métrique APFD dans la priorisation (tronquer le résultat à partir d'une certaine valeur satisfaisante de la métrique coût APFD) permet de réduire la suite de tests. Les stratégies mises en place par différents chercheurs pour réduire ou prioriser les tests sont équivalentes dans le sens où chaque type de stratégie permet à la fois de minimiser et de prioriser les suites de tests.

La priorisation basée sur l'historique utilise à la fois les informations des tests de régression précédents du même système logiciel et les informations de modifications en cours afin de hiérarchiser les nouvelles suites de tests.

Kim et Porter [38] ont utilisé les données d'exécution historiques pour hiérarchiser les cas de test pour les tests de régression. Tandis que Lin

et al. [39] examinent le poids des informations utilisées entre deux versions en ayant recours aux techniques de hiérarchisation basées sur l'historique. Les différents résultats montrent que la hiérarchisation basée sur l'historique fournit un meilleur taux de détection de défauts.

Pour optimiser le taux de détection de fautes lors des tests, Carlson et al. [40] ont utilisé le clustering sur des données issues de la complexité cyclomatique basé sur l'historique du taux de couverture des tests ainsi que des fautes logicielles. Ces données ont été utilisées seules ou en combinaison entre elles. La stratégie a été appliquée à un système industriel de grande taille (700K lignes de code réparties sur 827 classes) sur trois versions. Les résultats montrent une nette amélioration du taux de prédiction de fautes par rapport à la méthode (sans priorisation formelle) utilisée par le contrôle technique de l'industriel.

Elbaum et al. [41] conçoivent une méta-stratégie de sélection de la technique la plus efficace en fonction du type de logiciel en cours de développement. Pour cela, ils appliquent un ensemble de quatre techniques de priorisation à huit programmes et analysent les valeurs de la métrique APFD. Ils constatent que les performances de ces techniques varient dépendamment des spécificités des programmes. Ils réalisent alors une méta-classification (utilisant un arbre de classification) au niveau des différentes stratégies de priorisation des tests et des jeux de données issus d'un programme, pour déterminer la meilleure technique de priorisation pour ce programme. Le résultat des analyses donne des indices intéressants sur les types de techniques de priorisation - appropriées ou non à un logiciel donné selon ses spécificités.

Qu et al. [42] présentent un processus général de priorisation des cas de tests de régression, lorsque le code source n'est pas disponible. Les

auteurs ont implémenté un algorithme basé sur l'historique des tests. L'idée de base consiste à regrouper les cas de test (réutilisés) en fonction du type et du nombre de fautes révélées durant les exécutions précédentes. Pour ensuite, prioriser et réajuster la sélection en fonction des fautes révélées dans l'exécution courante. Après avoir introduit deux métriques d'évaluation de performances inspirées de la suite de métrique APFD et spécifiques aux tests en boîte noire, les auteurs mènent l'expérimentation sur deux systèmes logiciels. Ils évaluent l'efficacité de leur algorithme en le comparant aux performances du modèle aléatoire. Les résultats montrent que l'algorithme peut aider à améliorer les performances de test.

Dans [11,12], Touré et al. ont proposé de hiérarchiser les classes candidates au test unitaire pour les systèmes logiciels OO. Pour sélectionner les composants à tester, ils ont supposé que les testeurs s'appuient généralement sur les caractéristiques des classes capturées par les métriques de code source. Ils ont ainsi, proposé une approche tirant partie des différentes expériences des testeurs de logiciels et des attributs des classes logiciels, afin de prioriser les classes (candidates) à tester.

Sélectionner les cas de tests les plus prometteurs pour détecter les bogues est difficile s'il y a des incertitudes sur l'impact des modifications dans le code ou si des liens de traçabilité entre le code et les tests ne sont pas disponibles.

Dans le domaine de l'IA, spécialement en Apprentissage profond (Confer Chapitre 4), Spieker et al. [43] présentent Retecs (Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration). Retecs est une nouvelle méthode de sélection et priorisation des cas de test basée sur l'apprentissage automatique. Elle est utilisée dans les cas d'intégration continue avec pour objectif de minimiser

le délai d'aller-retour entre les validations de code et les commentaires des développeurs en cas d'échec des tests. La méthode Retecs utilise l'apprentissage par renforcement pour sélectionner et prioriser les cas de test. Dans un environnement où de nouveaux cas de test sont créés et des cas de test obsolètes sont supprimés, la méthode Retecs apprend à hiérarchiser les tests sujets aux erreurs grâce à une fonction de récompense et en se basant sur les cycles précédents de l'intégration. En appliquant Retecs sur les données extraites de trois études de cas industrielles, Spieker et al. [43], ont montré que l'apprentissage par renforcement permet une meilleure sélection et une hiérarchisation automatique dans les tests d'intégration et de régression.

Dans le prochain chapitre, nous discuterons des données utilisées pour l'expérimentation, la collecte de ces données ainsi que la méthodologie utilisée pour l'analyse de ces données collectées.

Chapitre 3 COLLECTE DE DONNÉES ET MÉTHODES D'ANALYSE

Dans notre méthodologie et pour atteindre nos objectifs de recherche, nous avons fait nos expérimentations sur deux systèmes logiciels, collecté différentes métriques de code source, fait une analyse descriptive sur les données recueillies et ensuite utilisé une technique d'apprentissage automatique (Apprentissage profond). L'apprentissage profond (Confer chapitre quatre (4)) est une fonction d'intelligence artificielle qui imite le fonctionnement du cerveau humain dans le traitement des données et la création de modèles à utiliser dans la prise de décision.

3.1 Les Systèmes logiciels

Les systèmes POI et ANT considérés sont des logiciels de Apache Software Apache POI Fundation [44].

Apache POI est un logiciel développé en Java permettant de manipuler divers formats de fichiers basés sur les standards Office Open XML (OOXML). Le logiciel POI permet de lire et de créer des fichiers MS Word, MS Excel, et MS PowerPoint à l'aide du langage Java. Pour notre travail, nous nous sommes intéressés aux versions 13, 14, 15, 16 et 17.

3.1.1 Apache ANT

Apache ANT est un outil en ligne de commande, utilisé pour piloter les processus d'exécution décrits dans les fichiers XML (build files) interdépendants. Il est similaire à Make « Make build Tool of Unix », mais est implémenté à l'aide du langage Java. Ant fournit un certain nombre de tâches intégrées permettant de compiler, d'assembler, de tester et d'exécuter des applications Java. Apache Ant peut également être utilisé

efficacement pour gérer des projets d'application dans d'autres langages comme C ou C++. Nous avons considéré les versions 1.3, 1.4, 1.5, 1.6 et 1.7.

3.2 Les métriques

Nous avons retenu six métriques de code source logiciel proposées par Chidamber et Kemerer en 1991 [13], ainsi que la métrique SLOC. Les métriques choisies ont suscité un intérêt considérable dans la littérature et constituent actuellement la suite de métriques orientées objet la plus utilisée. La suite CK [13] comprend six métriques qui évaluent différents attributs de bas niveau du code source du logiciel orienté objet. Ces métriques capturent les attributs de couplage, d'héritage, de cohésion, de complexité et de taille. Nous avons complété cet ensemble par la métrique SLOC.

3.2.1 Métriques de couplage

CBO (Coupling Between Objects) : le couplage entre objets. Cette métrique appartient à la suite CK [13]. Pour une classe logicielle donnée, elle détermine le nombre de classes auxquelles elle est couplée et vice versa. Il s'agit d'une sommation des couplages entrants et sortants. Cette métrique compte le nombre de classes ou d'interfaces avec lesquelles chaque classe est "couplée". Une classe est dite couplée à une autre s'il existe une dépendance de l'une vers l'autre. Briand et al. [45] ont montré qu'un couplage excessif entre les classes nuit à la modularité et constitue un obstacle à la réutilisation. Plus une classe est indépendante, plus il est facile de la réutiliser. Afin d'améliorer la modularité et de favoriser l'encapsulation, le couplage des classes doit être le plus faible possible. Plus le nombre de couples est élevé, plus les différentes parties du système sont sensibles aux modifications et plus la maintenance est difficile. Par

ailleurs, la mesure du couplage s'avère utile pour prévoir le niveau de complexité des tests des différentes parties du système [45, 46].

3.2.2 Métriques d'héritage

DIT : (Depth of Inheritance Tree) Calcule la profondeur de l'héritage. Cette métrique de la suite CK [13] compte le nombre de classes qu'il y a entre la classe mesurée et la racine de sa hiérarchie d'héritage. Elle évalue la complexité de la classe mesurée, mais aussi la complexité de la conception de la classe. En effet, le comportement d'une classe étant plus difficile à comprendre quand le nombre de méthodes héritées augmente. Avec plus de classes héritées, la conception et la redéfinition des comportements deviennent plus délicates. Par ailleurs, cette métrique mesure aussi le niveau de réutilisation du code des classes dans la hiérarchie d'héritage. Plus une classe est profonde dans la hiérarchie, moins elle est générique.

NOC : (Number of Children) Le nombre d'enfants ou de descendants d'une classe. Calcule le nombre de sous-classes qui héritent de la classe mesurée. Cette métrique appartient à la suite CK [13]. Elle compte le nombre de classes immédiatement dérivées de la classe concernée. NOC reflète (théoriquement) l'impact potentiel d'une classe sur ses descendants. NOC évalue aussi le niveau de réutilisabilité de la classe dans la mesure où une classe ayant plusieurs enfants (classes dérivées) est souvent très générique. Cependant, un très grand nombre de dérivations directes est signe de mauvaise conception et d'une abstraction impropre [13]. Théoriquement, le nombre de dérivations directes impacte fortement la hiérarchie des classes et peut nécessiter un effort de test particulier pour la classe en question.

3.2.3 Métrique de cohésion

LCOM : (Lack of Cohesion Of Methods) : Manque de cohésion entre méthodes d'une classe. La métrique LCOM est issue de la suite de CK [13] et évalue théoriquement le manque de cohésion dans une classe logicielle. Nous avons utilisé une variante de la métrique LCOM conçue par Hitz et Montazeri [46], plus appropriée pour Java. LCOM donne la différence entre le nombre de paires de méthodes ne partageant pas de variables d'instances et le nombre de paires de méthodes partageant des variables d'instances dans une classe logicielle. La métrique indique que deux méthodes d'une classe sont liées si elles ont accès et utilisent une variable, ou qu'une méthode en appelle une autre. Si cette différence est négative, LCOM est fixée à zéro. LCOM est censée renseigner la qualité de la structure de la classe. Une faible cohésion indique éventuellement que la classe a été investie de responsabilités disparates. La cohésion renseigne aussi l'encapsulation des données et des fonctionnalités dans une classe et son niveau de réutilisabilité. Sa faiblesse est souvent un signe de défaut de conception. Une classe peu cohésive devant sans doute être éclatée en plusieurs classes plus cohésives (remaniement).

3.2.4 Métriques de complexité

WMC : (Weighted Method Complexity) : Somme des complexités cyclomatiques des méthodes dans chaque classe. Cette métrique est issue de la suite CK [13]. Elle somme les complexités cyclomatiques de toutes les méthodes de la classe concernée. La complexité cyclomatique d'une méthode est donnée par la formulation de McCabe [47] comme étant le nombre de chemins linéairement indépendants qu'elle contient plus un. WMC reflète donc la complexité globale d'une classe. Elle est liée à l'effort nécessaire au développement et à la maintenance d'une classe

logicielle. Une valeur élevée est synonyme d'un risque élevé de présence de fautes dans la classe, mais aussi d'une faible compréhensibilité du code [46]. En tenant compte du nombre de méthodes, la complexité cyclomatique est aussi liée à une autre perspective, en l'occurrence la taille des classes logicielles.

RFC : (Response For Classes). Issue de la suite de CK [13], cette métrique est définie comme le nombre de méthodes possibles pouvant être appelées en réponse à l'invocation par une méthode de la classe. En pratique, il s'agit de la somme du nombre de méthodes y compris les constructeurs que la classe peut appeler directement. La métrique RFC reflète le niveau de communication potentiel entre une classe et les autres dont elle utilise les services. Cette métrique est théoriquement liée à la complexité et à la facilité de test d'une classe logicielle dans la mesure où plus une classe invoque des méthodes de diverses origines, plus ses fonctionnalités seront compliquées à vérifier. Une classe qui appelle un plus grand nombre de méthodes est considérée comme plus complexe et nécessitant plus d'effort de test.

3.2.5 Métriques de taille

SLOC : (System Lines Of Codes) : Calcule le nombre de lignes de code dans chaque classe. Cette métrique compte le nombre de lignes d'instructions dans la classe mesurée. SLOC évalue la taille d'une classe. La taille est fortement liée à la complexité. Une classe de grande taille est souvent synonyme de responsabilités disparates et présente une forte probabilité de fautes. Cette classe doit donc être restructurée.

Pour nos travaux, nous avons investigué la capacité de ces métriques à prédire les classes prioritaires au test unitaire.

3.3 Outils de collecte de données

Les outils et logiciels qui nous ont permis de collecter les données sont : le référentiel GitHub, l'environnement de développement IntelliJ Idea, le plugin Code Mr ainsi que le Framework JUNIT.

GitHub [48] fournit un hébergement pour le contrôle de versions de développement de logiciel à l'aide de Git. Il offre toutes les fonctionnalités de contrôle de version distribuée et de gestion du code source de Git, ainsi que l'ajout de ses propres fonctionnalités. Il fournit un contrôle d'accès et plusieurs fonctionnalités de collaboration telles que le suivi des bogues, les demandes de fonctionnalités et la gestion des tâches pour chaque projet. Les comptes gratuits GitHub sont couramment utilisés pour héberger des projets open source.

JUnit [49] est un Framework permettant l'écriture et l'exécution de tests unitaires automatisés pour des classes logicielles écrites en Java. Les cas de tests unitaires de JUnit sont écrits en Java par les testeurs. JUnit assiste les testeurs afin qu'ils puissent écrire les cas de test avec plus de commodité. Une utilisation typique de JUnit consiste à tester chaque classe logicielle du système à l'aide d'une classe test dédiée. Le test effectif de la classe logicielle est fait par sa classe test dédiée à travers l'appel de l'initialiseur de test JUnit. Après exécution, JUnit rapporte les tests qui ont réussi et les tests qui ont échoué.

Code MR [50] est un outil d'analyse (plugin intégré à l'environnement IntelliJ [51]) de la qualité logicielle et du code pour les projets Java, Kotlin et Scala. Il aide les développeurs à surveiller les indicateurs de qualité du logiciel en cours de développement par le calcul des différentes métriques de code source.

IntelliJ IDEA [51] est un environnement de développement intégré écrit en Java pour le développement de logiciels développé par JetBrains. Il est disponible sous forme d'édition communautaire sous licence Apache 2 et dans une édition commerciale propriétaire. Les deux peuvent être utilisés pour le développement commercial.

Cover [52] est un outil d'extraction de la couverture des tests (plugin intégré à l'environnement IntelliJ [51]). Il permet de capturer, pour chaque classe logicielle, sa couverture de test. Les niveaux de couvertures obtenues avec le plugin Cover sont de granularité méthodes et de granularité ligne de codes. Pour obtenir le rapport de couverture, il faudrait configurer l'option « code coverage [52] » dans l'environnement de développement IntelliJ IDEA [51]. Une fois la configuration faite, nous obtenons un rapport des différentes couvertures de test de chaque classe du système après exécution des tests JUNIT. Différents types de couvertures existent selon le niveau de granularité considéré.

3.3.1 Granularité méthodes

La couverture correspond au pourcentage de méthodes testées sur le nombre total de méthodes à tester. Dans ce cas, un taux de 100% signifie que toutes les méthodes de la classe ont été testées. Autrement dit, chaque méthode de la classe logicielle a été invoquée au moins une fois par une méthode de la classe test dédiée. Cependant, ce taux ne garantit pas que l'ensemble des chemins d'exécution possible à l'intérieur des méthodes ont été couverts durant le test.

3.3.2 Granularité lignes de codes

Elle correspond au pourcentage de lignes de code traversées par le flux de contrôle induit par les tests unitaires sur le nombre total de lignes

de code. Cette granularité est très utilisée notamment pour les tests de régression. Dans ce cas, un pourcentage de 100 % correspond à une couverture complète de toutes les lignes de code de la classe.
L'une des couvertures existantes dans la littérature, mais qui n'a pas été utilisée durant nos expérimentations, est la couverture selon le profil opérationnel.

3.3.3 Couverture selon le profil opérationnel

La couverture basée sur le flux calcule la proportion des chemins logiques couverts par le test. C'est la mesure la plus compliquée à appréhender. Elle conduit à une vision arborescente (graphe) du code, chaque structure de contrôle menant à un ou plusieurs embranchements. Une couverture de 100 % du profil opérationnel correspond à tous les scénarios possibles des flux de contrôle.

Pour notre travail, nous nous intéresserons aux granularités méthodes et ligne de codes. Pour déterminer les classes testées, nous avons considéré plusieurs seuils basés sur la valeur des couvertures de test.

3.4 Méthode de collecte des données

Nous avons considéré les codes sources de cinq versions différentes des systèmes POI [53] et ANT [54] sur les répertoires publics de GitHub [49]. Pour le système POI, nous avons considéré les versions 13, 14, 15, 16 et 17 et les versions 1.3, 1.4, 1.5 , 1.6 et 1.7 pour le système ANT. Par la suite, nous avons effectué les étapes suivantes pour chaque version des différents systèmes :

Étape 1 : Calcul des métriques de classes logicielles à l'aide de Code Mr pour toutes les versions considérées.

Etape 2 : Exécution des suites de tests unitaires (JUnit) et calcul du taux de couverture à l'aide de Cover.

Étape 3 : Préparation des données collectées. Le calcul des métriques de classes logicielles pour chaque composant met en évidence certaines observations aberrantes (outliers) pouvant biaiser nos résultats. Il s'agit des composants ou classes ayant comme métrique de complexité 0 (WMPC = 0) que nous avons exclues de nos données. Ces composants sont en général des artefacts OO comme les interfaces, les énumérations sans méthode.

Étape 4 : À chaque métrique, nous avons associé un rang. Ce choix est motivé par le fait que le classificateur sera entraîné sur plusieurs systèmes ayant différentes tailles. En effet, pour permettre au classificateur de prendre en compte cette différence de taille entre systèmes, nous avons jugé bon d'associer à chaque métrique son rang.

Étape 5 : Binarisation de la couverture de test des données. La binarisation consiste à libeller chaque classe par la valeur binaire 0 ou 1 selon que son taux de couverture considéré est supérieur ou non à un certain seuil.

L'objectif de cette binarisation est de modéliser notre problème en un problème de classification binaire. Pour ce faire, nous avons binarisé les taux de couverture obtenus. Nous considérons une classe comme testée si sa couverture de test est supérieure ou égale à un seuil fixé. Nous avons pris en compte 3 seuils pour chaque granularité considérée :

- Seuil1 : Les classes ayant une couverture supérieure ou égale à 50 % sont considérées comme testées. Les classes restantes sont considérées comme non testées ;
- Seuil2 : Les classes ayant une couverture supérieure ou égale à 30 % sont considérées comme testées. Les classes restantes sont considérées comme non testées ;

- Seuil3 : Les classes ayant une couverture supérieure à 0 sont considérées comme testées. Les classes restantes sont considérées comme non testées.

Pour chaque observation, les attributs formés par les métriques de code source, les rangs associés ainsi que le taux de couverture binarisé, forment une observation libellée. Notre objectif sera de construire un classificateur basé sur l'apprentissage profond, entraîné sur l'ensemble des données ainsi obtenues.

3.5 Statistiques descriptives

3.5.1 Système POI

Le tableau (1) un décrit les statistiques descriptives des 5 versions du système POI. Ainsi, nous remarquons une tendance croissante de la taille en termes de nombre de classes. On passe de 2259 pour la version 13 à 2436 classes pour la version 17. Notons une diminution significative entre la version 14 et la version 15. Nous pouvons observé cela par la figure une (Figure 1). Cette diminution significative s'explique par le fait que de la version 14 à la version 15, il y a eu au moins 200 classes qui ont été remaniées et un ajout approximatif de 30 classes. La complexité et la taille moyenne varient peu d'une version à l'autre (autour de 18 pour WMC et 130 pour LOC). Cette faible variation s'observe également pour les autres moyennes des métriques. La grande variabilité des métriques au sein d'une même version (les écarts types (σ) supérieurs ou égaux à la moyenne pour la plupart des métriques) traduit une grande disparité des caractéristiques des classes logicielles.

Versions	Obs	Stats	CBO	DIT	LCOM	LOC	NOC	RFC	WMC
V13	2259	Min	1	1	0	5	0	1	1
		Max	327	7	82	5156	151	538	546
		Mean	13.82	1.89	2.32	138.14	0.53	25.75	19.45
		σ	23.04	1.16	4.66	260.48	3.94	37.28	35.82
V14	2337	Min	1	0	0	5	0	1	1
		Max	328	4	82	3650	151	416	579
		Mean	13.86	0.74	2.28	130.19	0.52	19.35	18.56
		σ	23.05	1.04	4.16	226.71	3.8	30.05	32.41
V15	2208	Min	1	0	0	8	0	1	2
		Max	331	4	82	3763	151	424	591
		Mean	14.47	0.75	2.44	139.53	0.54	20.46	20
		σ	23.89	1.05	4.36	236.17	3.98	31.10	33.73
V16	2441	Min	1	1	0	5	0	1	1
		Max	344	7	82	3970	151	508	612
		Mean	14.06	1.90	2.26	130.65	0.51	25.58	18.76
		σ	23.82	1.19	4.20	230.49	3.80	35.68	33.16
V17	2436	Min	1	1	0	4	0	1	1
		Max	349	7	82	3943	151	510	615
		Mean	14.23	1.90	2.28	131.66	0.50	25.82	19.17
		σ	24.09	1.19	4.21	230.86	3.73	35.94	34.56

Tableau 1 : Statistiques descriptives des métriques de POI.

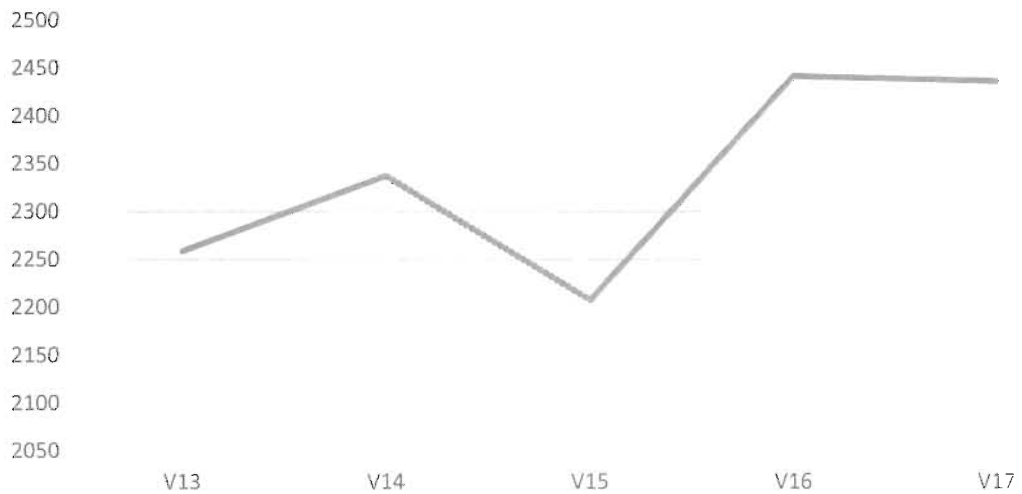

Figure 1 : Évolution du nombre de classes POI.

3.5.2 Système ANT

Les statistiques descriptives des cinq versions du système ANT observées au tableau (2) deux montrent que la taille en termes de nombre de classes à une tendance croissante. On passe de 1118 pour la version 13 à 1165

classes pour la version 17 (Confer figure deux (2)). La complexité et la taille moyenne varient peu d'une version à l'autre (autour de 19 pour WMC et 156 pour LOC). Cette faible variation s'observe également pour les autres moyennes des métriques. La grande variabilité des métriques au sein d'une même version (les écarts types (σ) supérieurs ou égaux à la moyenne pour la plupart des métriques) traduit une disparité des caractéristiques des classes logicielles.

Versions	obs.	Stats	CBO	DIT	LCOM	LOC	NOC	RFC	WMC
V13	1118	obs.	1118	1118	1118	1118	1118	1118	1118
		Min	0	1	0	4	0	1	1
		Max	672	7	30	1899	143	326	236
		Mean	10.97	2.4	1.85	156.64	0.67	26.06	19.74
		σ	1111.16	1.82	5.36	49022.6	27.09	1130	864.26
V14	1124	obs.	1124	1124	1124	1124	1124	1124	1124
		Min	0	1	0	6	0	1	1
		Max	674	7	30	2204	143	321	235
		Mean	10.84	2.39	1.84	155.65	0.67	25.78	19.51
		σ	1104.3	1.82	5.35	50129.71	26.97	1106.36	841.08
V15	1124	obs.	1124	1124	1124	1124	1124	1124	1124
		Min	0	1	0	6	0	1	1
		Max	674	7	30	2204	143	321	235
		Mean	10.84	2.39	1.84	155.73	0.67	25.79	19.52
		σ	1104.29	1.82	5.35	50304.1	26.97	1108.06	843.18
V16	1164	obs.	1164	1164	1164	1164	1164	1164	1164
		Min	0	1	0	6	0	1	1
		Max	694	7	30	2214	145	321	252
		Mean	10.8	2.38	1.82	155.01	0.66	25.63	19.44
		σ	1122.97	1.82	5.18	50743.87	27.72	1096.15	840.85
V17	1165	obs.	1165	1165	1165	1165	1165	1165	1165
		Min	0	1	0	4	0	1	1
		Max	694	7	30	2214	145	321	255
		Mean	10.81	2.39	1.83	157.33	0.66	25.65	19.45
		σ	1125.11	1.83	5.25	53739.35	27.71	1100.13	843.81

Tableau 2 : Statistiques descriptives des métriques de ANT.

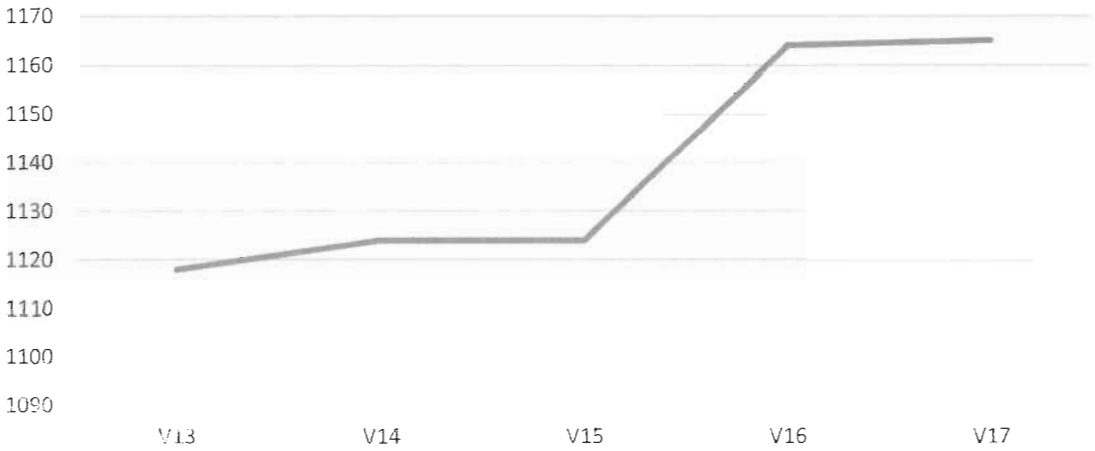

Figure 2 : Évolution du nombre de classes du système ANT.

Dans le prochain chapitre, nous discuterons de la méthodologie expérimentale utilisée pour résoudre notre problématique.

Chapitre 4 MÉTHODE EXPÉRIMENTALE

4.1 Le modèle d'apprentissage profond

L'apprentissage profond [55, 57] fait partie d'une famille plus large de méthodes d'apprentissage machine (machine learning) basées sur l'apprentissage de représentations de données. Il s'agit d'un ensemble de techniques permettant à un système de découvrir automatiquement les représentations nécessaires soit pour la détection de caractéristiques ou bien pour la classification à partir de données brutes, par opposition à des algorithmes spécifiques à une tâche. L'apprentissage peut être supervisé, non supervisé ou par renforcement.

4.1.1 Apprentissage supervisé

Durant l'apprentissage automatique avec supervision [58], les opérateurs présentent à l'ordinateur des exemples d'entrées et de sorties souhaitées. Grâce à un algorithme de classification ou d'apprentissage machine, l'ordinateur recherche des solutions pour bien classifier toutes les entrées en faisant correspondre les sorties calculées aux sorties des données. L'objectif à atteindre est la détermination de la règle générale qui fait correspondre les entrées et les sorties.

L'apprentissage automatique [58-59] avec supervision peut être utilisé pour faire des prédictions sur des données futures (on parle de « modélisation prédictive »). L'algorithme essaie de développer une fonction qui prédit avec précision la sortie à partir des variables d'entrée. Par exemple, prédire la valeur d'un bien immobilier (sortie) à partir d'entrées telles que le nombre de pièces, l'année de construction, la surface du terrain, l'emplacement, etc. L'apprentissage automatique avec supervision peut se subdiviser en deux types :

- Classification : La variable de sortie est une catégorie ;
- Régression : La variable de sortie est une valeur numérique.

Les principaux algorithmes d'apprentissage automatique avec supervision sont [60, 61] les forêts aléatoires, les arbres décisionnels, la méthode du k plus proche voisin (k-NN), la régression linéaire, la classification bayésienne naïve, la machine à vecteurs de support (SVM), la régression logistique et les réseaux de neurones [62-64].

4.1.2 L'apprentissage non supervisé

Durant l'apprentissage non supervisé [59, 60], l'algorithme est laissé à lui-même pour déterminer la structure de l'entrée (aucun label n'est communiqué à l'algorithme). Cette approche peut être un but en soi (qui permet de découvrir des structures enfouies dans les données) ou un moyen d'atteindre un autre but. Cette approche est également appelée « apprentissage des caractéristiques » (Features Learning) [65-66].

Un exemple d'apprentissage automatique sans supervision est l'algorithme de reconnaissance faciale prédictive de Facebook [67], qui identifie les personnes sur les photos publiées par les utilisateurs. Il existe deux types d'apprentissages automatiques sans supervision :

- Clustering : L'objectif consiste à trouver des regroupements dans les données [65-66].
- Association : L'objectif consiste à identifier les règles qui permettront de définir de grands groupes de données [66-67].

Les principaux algorithmes d'apprentissage automatique sans supervision sont le K-Means, le clustering/regroupement hiérarchique et la réduction de la dimensionnalité.

4.1.3 L'apprentissage automatique par renforcement

Durant l'apprentissage automatique par renforcement [59,60], un programme informatique interagit avec un environnement dynamique dans lequel il doit atteindre un certain but, par exemple conduire un véhicule ou affronter un adversaire dans un jeu. Le programme-apprenti reçoit une rétroaction sous forme de « récompenses » et de « punitions » pendant qu'il navigue dans l'espace du problème et qu'il apprend à identifier le comportement le plus efficace dans le contexte considéré [60, 61, 68]. En 2013, c'est un algorithme d'apprentissage automatique par renforcement (Q-learning) qui s'est rendu célèbre en apprenant à gagner six jeux vidéo Atari sans aucune intervention du programmeur. Il existe deux types d'apprentissages automatiques par renforcement [68] :

- Monte-Carlo – Le programme reçoit ses récompenses à la fin de l'état « terminal » ;
- L'apprentissage automatique par différence temporelle (TD) – Les récompenses sont évaluées et accordées à chaque étape.

Les principaux algorithmes d'apprentissage machine par renforcement [68] sont les suivants : Q-learning, Deep Q Network (DQN) et SARSA (State-Action-Reward-State-Action).

4.2 Le Réseau de Neurones artificiels

Les réseaux de neurones artificiels [69] sont des systèmes inspirés du fonctionnement des neurones biologiques (Confer figure trois (3)). Le plus célèbre d'entre eux est le perceptron multicouche (Confer figure quatre (4)), un système artificiel capable d'apprendre par l'expérience.

Introduit en 1957 par Franck Rosenblatt [70], il n'est véritablement utilisé que depuis 1982 après son perfectionnement. Grâce à la puissance

de calcul des années 2000, le perceptron s'est largement démocratisé et est de plus en plus utilisé.

4.2.1 Présentation générale

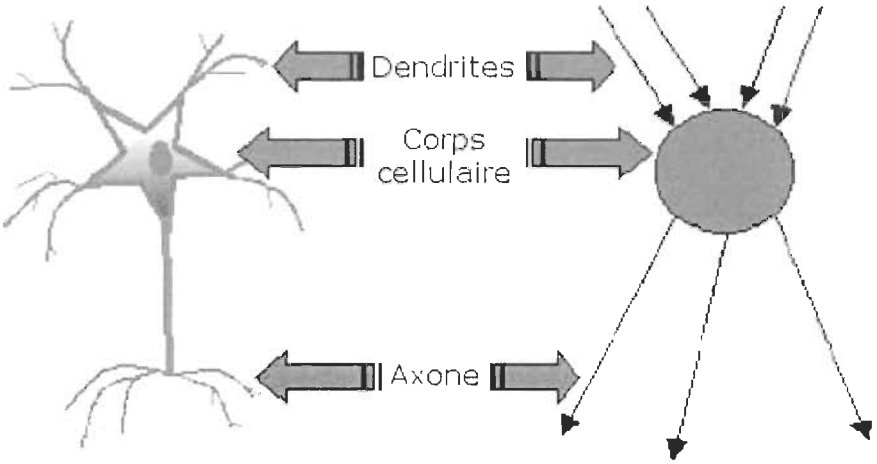

Figure 3 : Neurone biologique Vs Neurone artificiel [71]

À l'image de la biologie (Confer figure trois (3)), le perceptron est un ensemble de neurones organisés en couches. D'une couche à l'autre se propage le signal d'entrée jusqu'à la sortie, en activant les neurones ou non au fur et à mesure les neurones.

Le principe est de mesurer la différence entre la sortie calculée et celle attendue et de mettre à jour les liaisons entre les neurones en conséquence (les renforcer ou les inhiber) pour réduire cette différence.

4.2.2 Les couches du perceptron

Le perceptron (Confer figure quatre (4)) est organisé en trois parties [72] :

La couche d'entrée : C'est un ensemble de neurones qui portent le signal d'entrée. Par exemple, si notre réseau essaie d'apprendre à réaliser un XOR entre 2 bits, on aura en entrée bit1 et bit2 (donc 2 neurones, un pour chaque information) [72]. Si nous voulons apprendre au réseau à estimer le prix d'un appartement, nous aurons autant de neurones dans cette

couche que de variables. Tous les neurones de cette couche sont ensuite reliés à ceux de la couche suivante.

Les couches cachées : Il s'agit du cœur du perceptron dans lequel, les relations entre les variables vont être calculées.

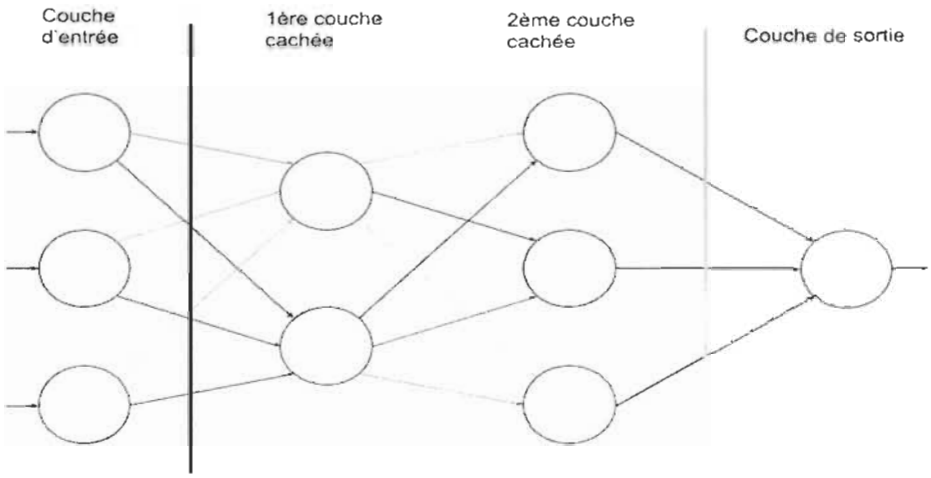

Figure 4 : Réseau de neurones artificiel ou perceptron multi couche [73]

La couche de sortie : cette couche classifie et fournit la classe à associer à chaque entrée ; la prédiction. Un perceptron est constitué de 3 couches de neurones : une couche d'entrée, une couche cachée et une couche de sortie.

4.2.3 *Le neurone*

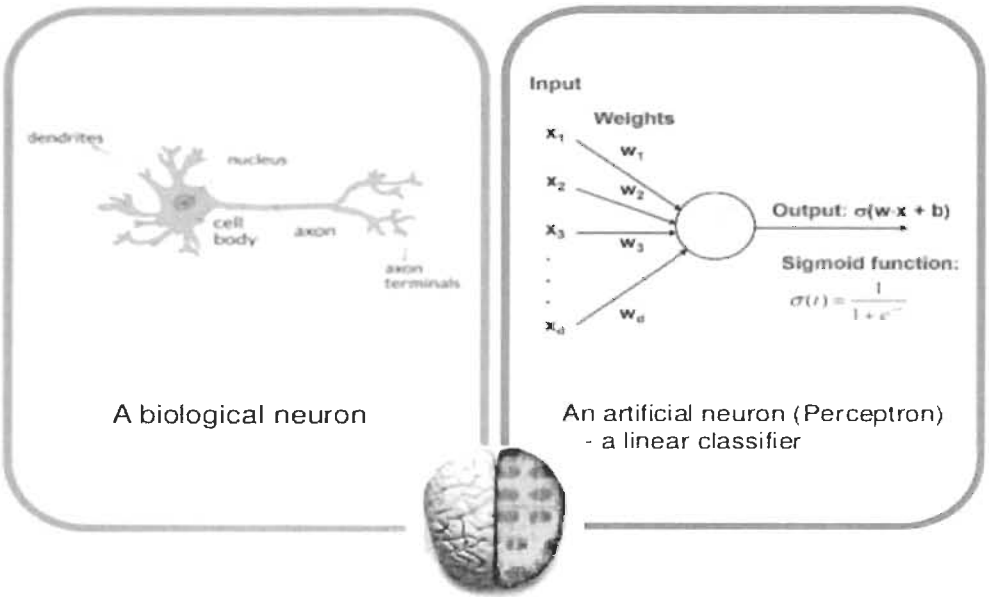

Figure 5 : Neurone biologique Vs Neurone artificiel [71]

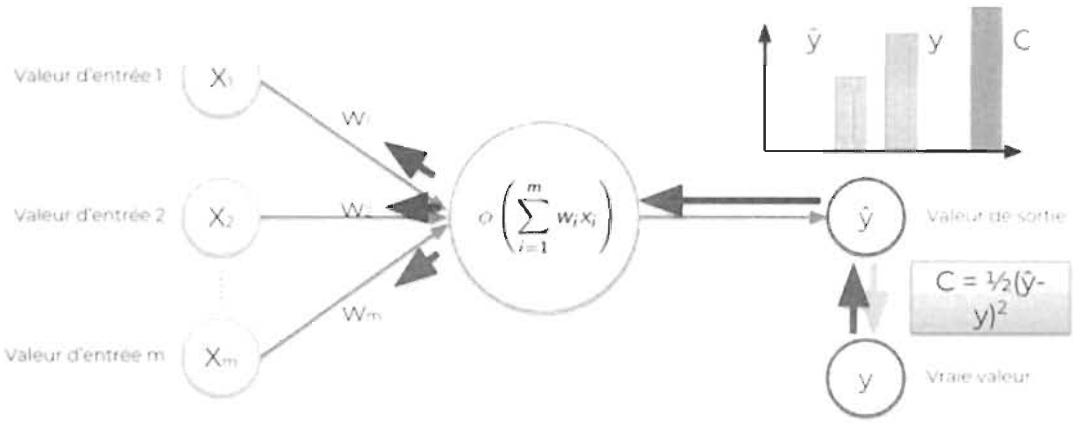

Figure 6:Architecture d'un neurone artificiel [74]

Un neurone (Confer figure cinq (5) et six (6)) est composé d'une somme pondérée de signaux et d'une fonction d'activation. Les réseaux de neurones sont généralement organisés en couches interconnectées au niveau des neurones.

Les données entrent dans le réseau via la couche d'entrée, qui communique avec une ou plusieurs autres couches cachées consécutives où le traitement proprement dit s'effectue via un système de "liens" pondérés. La dernière

couche cachée est liée à la couche de sortie qui détermine la réponse (la classe) de l'entrée fournie, comme indiqué dans le graphique ci-dessous.

Comme on le voit dans la figure 6, des signaux x_1, x_2... x_m arrivent à notre neurone. Chaque lien qui amène le signal est pondéré, respectivement par w^i_1, w^i_2...w^i_m pour chaque couche i. C'est ce poids qui va être adapté tout au long de l'apprentissage pour permettre au réseau de prédire efficacement. Ensuite, la somme pondérée de tous ces signaux est calculée. Un biais b (considéré comme un neurone externe supplémentaire) est ajouté et permet d'envoyer systématiquement le signal 1 au neurone. Grâce à ce biais, la fonction d'activation va être décalée et le réseau aura donc de plus grandes opportunités d'apprentissage.

Une fois cette somme calculée, on applique une fonction d'activation pour obtenir un signal de sortie. Cette fonction d'activation aura un seuil à partir duquel le neurone va émettre un signal (il a été suffisamment stimulé), il s'agit du potentiel d'action du neurone biologique. La formule de sortie d'un neurone caché sera donc toujours de la forme : $Y = factivation\ (b + \sum_i w_i \cdot x_i\ )$. Celle d'un neurone d'entrée sera : $Y = x$. Celle d'un neurone de sortie est : $Y = \sum_i w_i \cdot x_i$.

Dans la pratique, les poids sont initialisés au hasard lorsqu'on crée le réseau de neurones. Il en va de même pour le biais. Les fonctions d'activations sont seulement appliquées au niveau des couches cachées et de la couche de sortie.

4.2.4 La règle delta

La plupart des réseaux de neurones artificiels (RNA) contiennent des « règles d'apprentissage » qui modifient les poids des connexions en fonction des données [75]. Autrement dit, la plupart des RNA apprennent

par l'exemple, tout comme leurs homologues biologiques. Bien qu'il existe de nombreuses règles d'apprentissage utilisées par les réseaux de neurones, l'exemple présenté ci-dessous ne concerne que la règle du delta. Elle est souvent utilisée par la classe la plus courante de RNA appelée "réseaux de neurones à propagation arrière" [76]. Avec la règle delta, l'apprentissage est un processus supervisé continu (rétropropagation) s'effectuant pendant un certain nombre de cycles ou "époques". Une rétropropagation (Confer figure sept (7)) est une propagation d'erreur en arrière faite pour ajuster les poids des liaisons. La rétropropagation [77] effectue une descente de gradient dans l'espace vectoriel de la solution. La descente de gradient permet d'évaluer l'impact d'un poids sur l'erreur et donc de l'améliorer. Plus simplement, lorsque par exemple, une image est initialement présentée à un réseau de neurones, il « devine » de manière aléatoire ce qu'elle pourrait être. Par un calcul de fonction de coût, il, détermine à quel point sa réponse est éloignée de la réponse réelle et apporte des ajustements appropriés à ses poids de connexion.

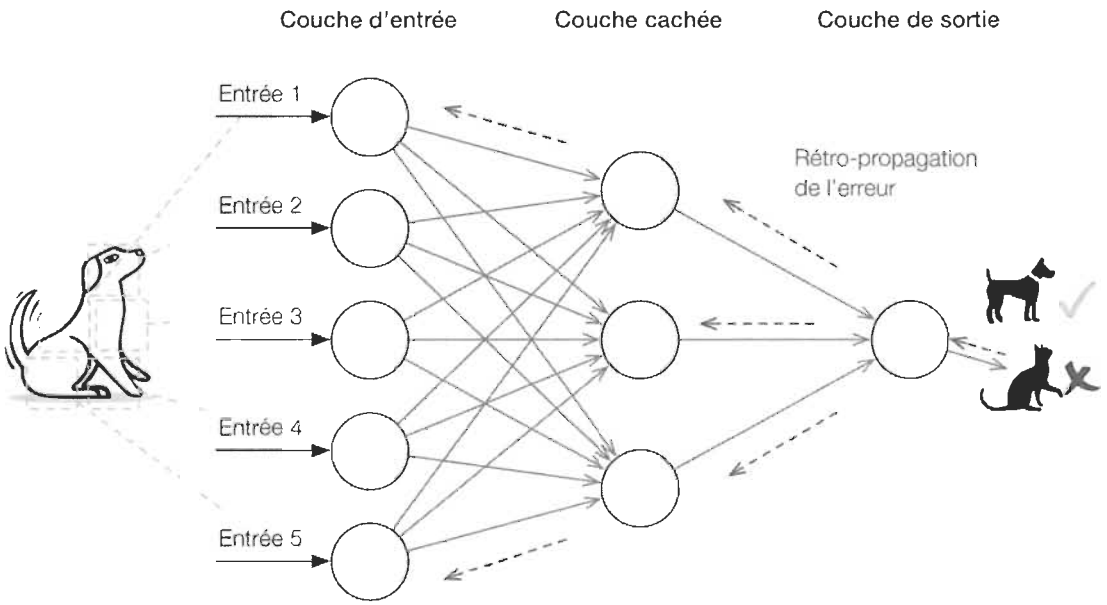

Figure 7 : Illustration de la méthode de rétro propagation [78]

Nous pouvons noter également que chaque nœud de couche cachée contient une fonction d'activation sigmoïdale qui polarise l'activité du réseau et l'aide à se stabiliser.

4.2.5 Le minimum global

La rétro propagation effectue une descente de gradient dans l'espace vectoriel de la solution vers un « minimum global » le long du vecteur de plus forte pente en sens opposée à la normale de la surface d'erreur. Le minimum global (Confer figure huit (8)) est cette solution théorique avec l'erreur la plus faible possible. La surface d'erreur elle-même est un hyper paraboloïde comme le montre la figure 8. En effet, dans la plupart des problèmes, l'espace de la solution est assez irrégulier et comporte de nombreux « creux » et « collines » qui peuvent amener l'algorithme du gradient à s'installer dans un « minimum local » qui n'est pas la solution globale.

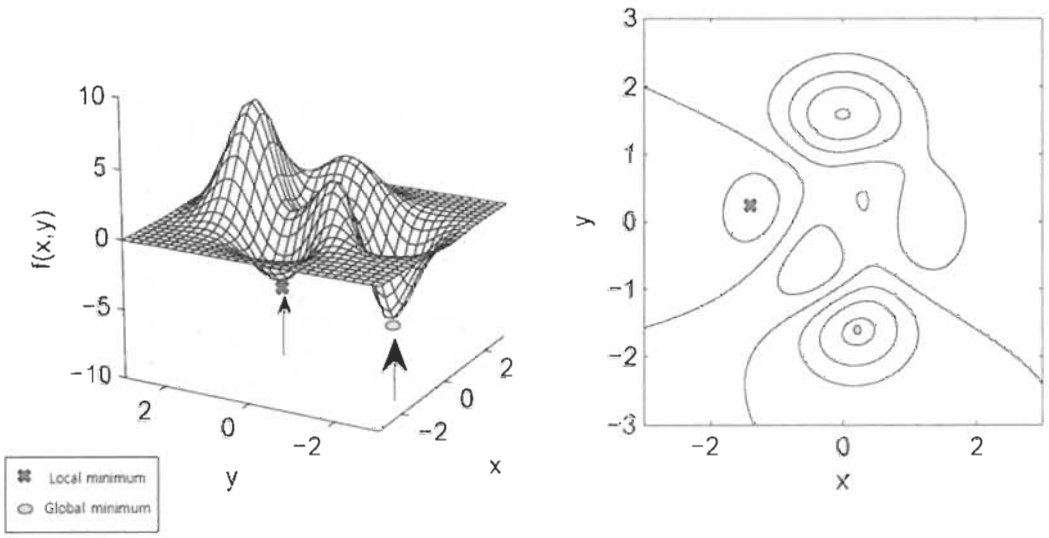

Figure 8 : Minimum local, minimum global et surface d'erreur [79]

Étant donné que la nature de l'espace d'erreur ne peut être connue a priori, la convergence de cette méthode nécessite un grand nombre d'ajustements. La plupart des règles d'apprentissage comportent un grand

nombre de paramètres contrôlables et ajustables permettant de superviser l'apprentissage (coefficient bêta -vitesse-, moment). La vitesse d'apprentissage est le taux de convergence entre la solution actuelle et le minimum global. Le moment évite au réseau de s'installer dans les minima locaux de la surface.

Une fois le modèle entraîné à un niveau satisfaisant, il peut être utilisé comme classificateur de données. Cependant, un entraînement soutenu sur un jeu de données conduit à un sur-apprentissage (convergence vers un minimum local) du réseau de neurones sur ces données, le rendant moins performant face à de nouvelles entrées.

4.3 Construction du Réseau de neurones

Nous avons utilisé les bibliothèques TensorFlow [80], Keras[81] et Pandas[82] et le langage de programmation Python pour la construction de notre classificateur. Nous avons procédé à plusieurs essais pour finalement aboutir à la configuration architecturale ci-dessous, nous donnant de bons résultats indépendamment des systèmes et de leurs versions.

Le réseau de neurones construit après plusieurs essais possède les caractéristiques suivantes :

- À la couche d'entrée, nous avons placé 14 neurones, car ayant 7 métriques de classes plus leurs rangs respectifs. Ce qui nous donne 14 entrées ;
- 13 couches cachées. À la première couche cachée, nous aurons 169 neurones ;
- Elle est suivie de plusieurs autres couches cachées successives décrémentant chacune 13 neurones de la couche précédente ;
- 2 neurones de la couche de sortie qui traduisent la sortie binaire de notre problème de classification (Classe testée ou non) ;

- Le nombre d'époques choisi varie entre 100 et 700.

Outre l'architecture du réseau de neurones, voici d'autres paramètres à considérer.

4.3.1 La fonction d'activation

Nous avons utilisé la fonction Relu [80, 81] comme fonction d'activation pour tous les neurones sauf ceux de la sortie qui utilisent les fonctions sigmoïdes.

4.3.2 L'optimiseur

L'optimiseur [83] contrôle le taux d'apprentissage. Nous avons utilisé l'optimiseur « Adam ». Le taux d'apprentissage détermine à quelle vitesse les poids du réseau de neurones sont calculés et ajustés. Un taux d'apprentissage plus faible peut conduire à des poids plus précis (jusqu'à un certain point), mais le temps nécessaire pour calculer les poids sera plus long et le risque de tomber sur un minimum local (sur-apprentissage) est plus grand.

4.3.3 La fonction perte

Nous avons utilisé « l'erreur quadratique moyenne [83] » comme fonction de perte. Elle est calculée en prenant la différence quadratique moyenne entre les valeurs prédites et réelles. Plus petite est l'erreur, plus proches seront les sorties prédites aux sorties des données d'entraînement.

4.4 Évaluation et Validation du classificateur

4.4.1 Évaluation

L'évaluation du modèle s'est faite par la matrice de confusion (Confer figure neuf (9)) et le calcul du taux de bonne classification [84].

La matrice de confusion est un outil standard d'évaluation des modèles de classification. Elle dénombre le nombre de classes prédites correctement ainsi que le nombre de classes non prédites correctement par le classificateur. Ainsi, pour une classification binaire (+ et -), nous obtenons une matrice 2x2 dans laquelle les lignes représentent le nombre d'observations prédites par le classificateur pour les classes + et -, tandis que les colonnes représentent les classes réelles des observations + et -. Les quatre cellules de la matrice correspondent au dénombrement des : faux positif, vrai positif, faux négatif et vrai négatif.

		Classe réelle	
		Négative (-)	Positive (+)
Classe prédite	Négative (-)	True Negatives (vrais négatifs)	False Negatives(Faux négatifs)
	Positive (+)	False Positives (Faux Positifs)	True Positives (Vrai positifs)

Figure 9 : Illustration de la matrice de confusion

TP (True positives = Vrais positifs) : les cas où la prédiction est positive, et que la valeur réelle est positive.

TN (True Négatives = Vrais négatifs) : les cas où la prédiction est négative, et que la valeur réelle est effectivement négative.

FP (False Positive = Faux positifs) : les cas où la prédiction est positive, mais que la valeur réelle est négative.

FN (False Négative = Faux négatifs) : les cas où la prédiction est négative, mais que la valeur réelle est positive.

4.4.2 Validation du classificateur

La validation de notre classificateur s'est faite de quatre façons.

- CVV (Cross Version validation). Durant le processus de CVV, le modèle de réseaux de neurones artificiels est entraîné sur les données d'une version v d'un système, et le classificateur obtenu est validé sur les données des versions v + k (postérieures) du même système ou k > 0.

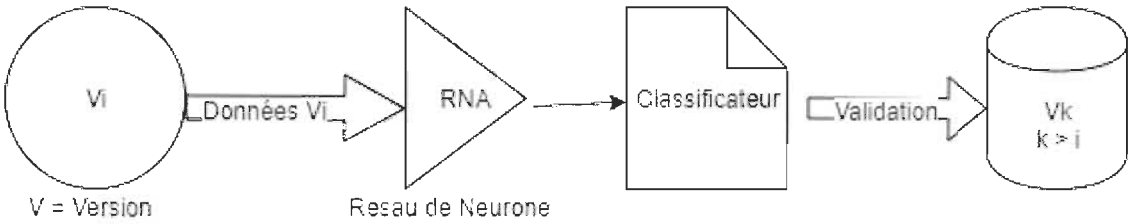

Figure 10 : Cross Version Validation : CVV

- CPVV (Combined Previous Version Validation). Pour une version V donnée, cette validation CPVV consiste à entraîner les réseaux de neurones artificiels sur la combinaison de toutes les versions antérieures à V (V est exclue) puis tester le classificateur obtenu sur les données de la version V. Le but de cette validation est de tester la possibilité de suggérer des classes à tester grâce à un apprentissage sur les données combinées de plusieurs versions antérieures.

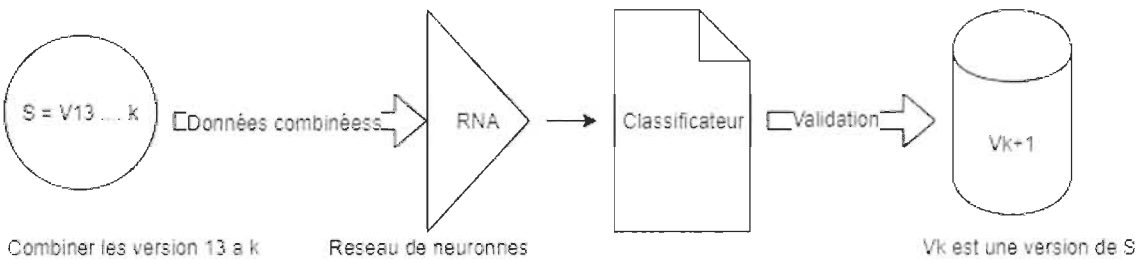

Figure 11 : Combined Previous Version Validation CPVV

- CSPVV (Cross Systèmes Validation). Durant cette validation, nous entraînerons les réseaux de neurones sur toutes les versions combinées d'un système S + les k premières versions de S'puis nous testerons le classificateur obtenu sur les k + i versions postérieures S'(i >0).

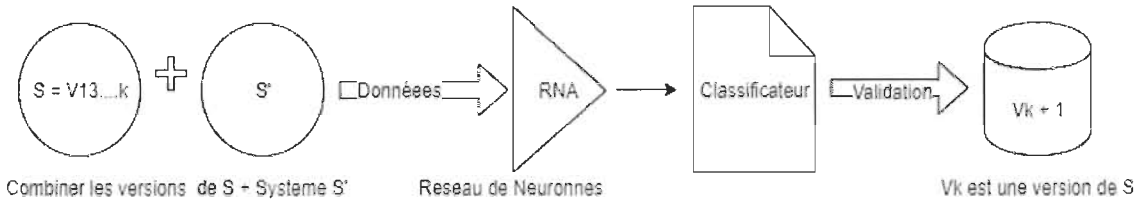

Figure 12 : Combined System and Previous Version Validation. CSPVV

- LOSOV (Leave One System Out Validation). Pour un système S donné, cette validation consiste à entraîner le classificateur obtenu grâce aux réseaux de neurones artificiels sur la combinaison de toutes les versions d'un même système ou de plusieurs systèmes dans le but de prédire les classes à tester d'un système différent S'. Le but de cette validation est de tester la possibilité de suggérer des classes à tester grâce à un apprentissage sur les données combinées d'un ou de plusieurs systèmes.

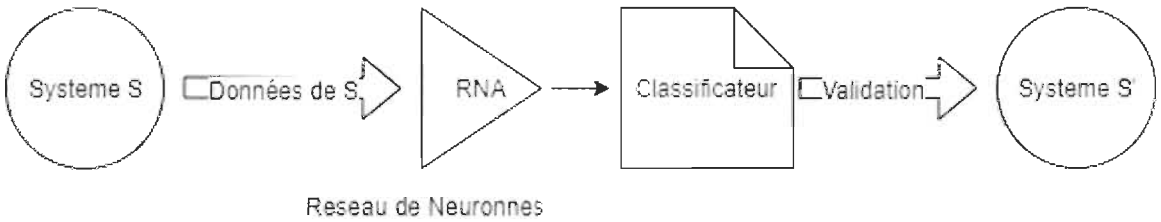

Figure 13 : Leave One System Out Validation.

Dans le prochain chapitre nous présenterons les résultats obtenus à la suite des expérimentations puis nous discuterons et ferons une interprétation de ces résultats.

Chapitre 5 RÉSULTATS ET INTERPRÉTATIONS

5.1 CVV Validation

La validation CVV consiste à prédire les classes à tester d'une version v d'un système S par un classificateur qui a été entraîné sur les données de la version précédente v-1 du même système S.

Les résultats suivants ont été obtenus par cette technique de validation appliquée sur les différentes versions de POI et de ANT

5.1.1 Validation sur le système POI

Le tableau 3 suivant nous donne les résultats obtenus après une validation CVV sur le système POI en considérant les deux niveaux de granularité et les trois types de seuils pour chaque niveau. Les résultats obtenus montrent des indices de surapprentissage durant l'entraînement du classificateur sur le premier jeu de données (version 13). En effet, nous obtenons des taux d'exactitude pratiquement de 99% (98.90%). Cela se remarque par le fait que ce classificateur (obtenu après entraînement sur la version 13) testé sur d'autres versions nous donne de très faible taux d'exactitude. En raison du surapprentissage du modèle, on a une mauvaise classification, durant la validation de la version 14. Cette mauvaise classification est déduite par l'obtention des taux d'exactitude relativement bas (variant entre 65% et 67%) par rapport aux autres taux obtenus durant la validation CVV. Pour résoudre ce problème, nous avons en premier lieu appliqué la fonction dropout sur les couches du réseau de neurone. Mais cette technique n'a pas été très efficace car nous n'avons pas assez de données d'entrainement et le réseau de neurones ne comporte pas un très grand nombre de couches cachées et de neurones. En second lieu, nous

avons légèrement diminué le nombre d'itérations durant l'entraînement mais aussi diminue le learning rate. Ces techniques nous ont donc permis d'obtenir un meilleur classificateur.

Versions	Granularité Méthodes			Granularité lignes de Codes		
	CM50	CM30	CM0	CLOC50	CLOC30	CLOC0
13->13	96.65%	97.75%	98.41%	94.48%	97.04%	98.90%
13->14	67.09%	66.12%	64.32%	67.67%	67.25%	65.11%
14->14	85.68%	86.48%	86.78%	79.43%	79.35%	76.20%
14->15	83.87%	85.09%	84.77%	79.03%	78.81%	75.51%
15->15	88.70%	89.06%	88.57%	82.15%	79.76%	79.94%
15->16	80.08%	80.88%	80.92%	78.39%	75.90%	76.99%
16->16	88.84%	90.00%	89.00%	81.29%	80.28%	77.55%
16->17	88.25%	89.70%	87.53%	81.57%	80.52%	78.07%
17->17	91.43%	92.84%	89.94%	82.62%	79.24%	80.64%

Tableau 3 : Validation CVV sur le système POI.

Mais, à partir de la version 15, nous assistons à une amélioration de la classification du modèle entraîné car nous obtenons des taux d'exactitudes variant entre 75% et 89%. En moyenne les taux d'exactitude sont supérieurs à 75%. Cela montre que pour un seuil du taux d'exactitude fixé à 70%, un classificateur entraîné sur une version antérieure V du système POI, est capable de prédire les classes à tester de la version suivante V+1 du même système.

En considérant les niveaux de couvertures de test, nous obtenons des taux d'exactitude élevés pour un seuil fixé à 30% (CM30) pour les méthodes et un seuil de 50% (CLOC50) pour les lignes de code.

5.1.2 Validations CVV sur les versions du Système ANT

Le tableau 4 suivant nous donne les résultats obtenus après une validation CVV sur le système ANT en considérant les deux niveaux de granularité et les trois types de seuils pour chaque niveau.

Versions	Granularité Méthodes			Granularité lignes de Codes		
	CM50	CM30	CM0	CLOC50	CLOC30	CLOC0
13->13	93.20%	91.77%	88.28%	94.63%	93.47%	88.37%
13->14	92.97%	91.19%	87.99%	94.48%	93.06%	87.28%
14->14	94.04%	90.84%	88.79%	95.37%	94.22%	89.86%
14->15	93.77%	90.57%	88.26%	95.02%	93.95%	89.32%
15->15	93.51%	92.62%	89.06%	94.48%	94.57%	90.12%
15->16	93.64%	92.35%	88.57%	94.67%	94.24%	89.69%
16->16	94.67%	94.07%	89.86%	95.79%	94.76%	88.57%
16->17	94.25%	93.56%	89.70%	95.62%	94.42%	88.58%
17->17	94.16%	94.33%	88.76%	95.45%	95.11%	91.07%

Tableau 4 : Validation CVV sur le système ANT.

Les résultats obtenus après la validation CVV, faite sur le système ANT, ne montrent aucun signe de surapprentissage du classificateur. En moyenne, nous obtenons des taux d'exactitude supérieurs à 87%. Ceci montre la capacité du classificateur à bien distinguer les classes à tester durant les validations sur les versions du système ANT.

Les taux d'exactitudes élevés supérieurs à la moyenne sont observés lorsque le niveau de granularité choisi sont les lignes de code avec un seuil de couverture fixé à 50% (CLOC50).

En général, nous remarquons que les taux d'exactitude obtenus durant la validation sur le système ANT sont plus élevés et moins volatiles que ceux obtenus dans le cas du système POI. Les statistiques descriptives

faites sur les deux systèmes ont montré que la taille des versions du système POI varient beaucoup entre elles. Ces statistiques sont beaucoup plus stables et moins variables dans le cas du système ANT. Cette variation de la taille des versions pour POI pourrait donc expliquer la tendance du classificateur à obtenir des taux d'exactitude moins bons durant les validations entre versions.

5.1.3 Conclusion partielle.

La validation CVV faite sur les deux systèmes POI et ANT nous permet de conclure qu'il est possible de prédire à partir d'une version antérieure, les classes à tester d'une version en cours de développement.

5.2 CPVV Validation

La validation de type CPVV consiste à prédire les classes à tester d'une version V d'un système S en cours de développement par un classificateur qui a été entraîné sur toutes les versions antérieures à la version V du même système S. La table 5 présente les résultats obtenus pour les différentes versions de POI puis de ANT.

5.2.1 Validations sur le système POI

Le tableau 5 suivant nous donne les résultats obtenus après une validation CPVV sur le système POI en considérant les deux niveaux de granularité et les trois types de seuils pour chaque niveau.

Versions combinées	Granularité Méthodes			Granularité lignes de Codes		
	CM50	CM30	CM0	CLOC50	CLOC30	CLOC0
[13,14] ->15	74.47%	71.13%	73.02%	72.03%	74.20%	76.73%
[13,14,15] ->16	70.48%	68.27%	66.95%	72.33%	68.11%	65.42%
[13,14,15,16] ->17	76.14%	74.81%	72.19%	76.66%	74.97%	81.61%

Tableau 5 : Validation CPVV sur le système POI

Les résultats de CPVV sur le système POI nous montrent une tendance croissante des taux d'exactitude d'une version à une autre. Pour un seuil S de couverture de test fixé à 50 % (CM50 et CLOC50), nous obtenons des taux d'exactitudes supérieurs à 70 %. Cependant, nous observons une légère diminution du taux d'exactitude au niveau de la version 16. Cette diminution du taux d'exactitude implique la présence de faible corrélation entre les données d'entraînement (métriques des versions combinées 13, 14 et 15) et des données de validation (métriques de la version 16).

En effet, les statistiques descriptives du système POI montrent une grande variation du nombre de classes entre les versions consécutives du système. De la version 13 à la version 14, nous notons une augmentation de près de 100 classes, de la version 14 à 15 une diminution de près de 150 classes et de la version 15 à 16 une forte augmentation de près de 250 classes. Ce qui peut traduire une grande refactorisation du système POI ou à des modifications dues aux changements de besoins. Cette variation du nombre de classes entraîné dans notre cas, une baisse du taux d'exactitude des modèles durant la phase d'apprentissage et une mauvaise prédiction des classes à tester sur la version 16.

En revanche, nous remarquons qu'une fois l'évolution du nombre de classes se stabilise (plus aucune refactorisation majeure ou aucune modification apportée), le classificateur devient plus performant puisqu'à partir de la version 17 nous notons une augmentation du taux d'exactitude.

De cette remarque, nous pouvons conclure qu'une étude préalable sur l'évolution du système est très importante. Car cela permettra de mieux apprécier la performance du classificateur durant les validations.

5.2.2 Validations CPVV sur les versions du Système ANT

Le tableau 6 suivant nous donne les résultats obtenus après une validation CPVV sur le système ANT en considérant les deux niveaux de granularité et les trois types de seuils pour chaque niveau.

Versions combinées	Granularité Méthodes			Granularité lignes de Codes		
	CM50	CM30	CM0	CLOC50	CLOC30	CLOC0
[13,14] ->15	94.04%	95.82%	90.57%	98.75%	96.26%	93.68%
[13,14,15] ->16	96.56%	96.65%	94.24%	98.63%	97.16%	93.73%
[13,14,15,16] ->17	97.08%	97.60%	97.94%	99.40%	97.08%	98.37%

Tableau 6 : Validation CPVV sur le système ANT

La validation CPVV faite sur les versions du système ANT nous montre des taux d'exactitude relativement élevés par rapport à ceux obtenus dans le cas du système POI.

Nous assistons à une augmentation graduelle du taux d'exactitude d'une validation à une autre. Cette augmentation du taux d'exactitude s'explique par l'augmentation des jeux de données à chaque fois qu'on ajoute une nouvelle version aux données d'entraînement. En effet, plus nous avons des jeux de données et mieux l'apprentissage par le classificateur se fait. Les taux d'exactitude élevés sont plus remarquables pour un seuil S de couverture de test fixé à 50% (CLOC50) comme dans le cas du système POI.

En moyenne, nous obtenons des taux d'exactitude supérieurs à 90%. Ce taux d'exactitude élevé démontre la présence de fortes corrélations entre les données (métriques) du système ANT, ce qui n'est pas le cas pour le système POI.

5.2.3 Conclusion partielle

La validation CPVV faite sur les versions des deux systèmes nous permet de dire qu'a un seuil du taux d'exactitude fixé à 70%, il est possible d'apprendre à partir des versions antérieures dans le but de prédire les classes à tester pour une version en cours de développement. Les expérimentations faites nous permettent de conclure que dans le cas d'une validation CPVV, le niveau de refactorisation due à l'évolution du système ou à des modifications de certains besoins du système est un facteur à prendre en compte pour mieux évaluer la performance de ce classificateur ainsi que les résultats obtenus. Une validation de type CPVV sur un système avec peu de modifications (exemple du système ANT) peut donner de meilleurs résultats que dans le cas d'un système moins stable (exemple du système POI).

Connaître la nature des modifications apportées au niveau de chaque version précédente ou système, permettrait de mieux sélectionner les versions antérieures sur lesquelles le classificateur pourrait être entraîné pour qu'il soit plus efficace. Mais, entraîner le classificateur sur un grand nombre de jeux est aussi nécessaire pour obtenir une diversité de données de bonnes classifications et tendre vers des extrema globaux.

5.3 Validation CSPVV Validation

La validation CSPVV consiste à prédire les classes à tester d'une version V d'un système S en entraînant le classificateur sur les versions antérieures à la version V et sur les données d'un autre système S', différent de S.

Dans notre cas, une validation CSPVV sur la version 15 du système ANT se fera en entraînant le classificateur sur les versions 13 et 14 ainsi que sur toutes les versions du système POI.

L'objectif de cette validation est d'investiguer la possibilité de prédire les classes à tester d'une version en se basant sur des connaissances acquises à la fois sur les versions antérieures et sur un autre système.

Les résultats du tableau 7 ont été obtenus par cette technique de validation appliquée sur les différentes versions de POI et de ANT.

5.3.1 Validation sur les versions du système POI

Le tableau 7 suivant nous donne les résultats obtenus après une validation CSPVV sur le système POI en considérant les deux niveaux de granularité et les trois types de seuils pour chaque niveau.

Versions combinées	Granularité Méthodes			Granularité lignes de codes		
	CM50	CM30	CM0	CLOC50	CLOC30	CLOC0
[ANT+ POI 13] ->13	79.66%	73.43%	75.64%	80.58%	78.11%	74.89%
[ANT+ POI 13] ->14	68.60%	66.58%	66.16%	69.02%	67.63%	65.11%
[ANT + POI 13,14] ->15	70.00%	66.20%	65.21%	70.49%	68.55%	65.43%
[ANT + POI 13,14,15] ->16	70.64%	67.43%	66.02%	74.26%	71.41%	66.39%
[ANT + POI 13,14,15,16] -	75.09%	72.03%	69.18%	75.29%	74.04%	71.99%

Tableau 7 : Validation CSPVV sur le système POI

Les résultats obtenus après validation CSPVV montrent une augmentation graduelle du taux d'exactitude d'une version V à une version V+1. Cela s'explique par une augmentation graduelle des jeux de données (données d'entraînement) d'une validation à une autre. Cette augmentation des données d'entraînement permet au classificateur de mieux apprendre et d'être plus performant. Néanmoins, au niveau de la version 14, nous assistons à une baisse du taux d'exactitude suivie d'une amélioration de ce taux à partir de la version 15. Cette baisse du taux pourrait être due à la présence de bruit dans les données d'entraînement. En effet, la combinaison des données de deux systèmes différents peut engendrer du bruit dans les données. Cependant, lorsque le classificateur est entraîné sur un grand nombre de jeux de données il devient performant. Ainsi, comme

nous le remarquons, à partir de la version 15, le taux d'exactitude croît considérablement. Après validation sur toutes les versions du système, nous obtenons en moyenne des taux d'exactitude supérieurs à 70 %. Les forts taux d'exactitude obtenus se remarquent plus au niveau de la granularité lignes de code avec un seuil de couverture de test S fixe à 50 % (CLOC50).

5.3.2 Validations sur les versions du Système ANT

Le tableau 8 suivant nous donne les résultats obtenus après une validation CSPVV sur le système ANT en considérant les deux niveaux de granularité et les trois types de seuils pour chaque niveau.

Versions combinées	Granularités Méthodes			Granularités lignes de Codes		
	CM50	CM30	CM0	CLOC50	CLOC 30	CLOC 0
[POI + ANT 13] ->13	68.25%	68.43%	68.16%	76.57%	74.60%	62.97%
[POI+ANT13] ->14	86.03%	80.78%	79.72%	91.28%	86.30%	78.29%
[POI+ANT 13,14] ->15	91.37%	89.86%	85.68%	93.51%	89.77%	84.61%
[POI+ANT 13,14,15] ->16	93.47%	92.44%	87.03%	94.42%	90.81%	87.63%
[POI+ANT 13,14,15,16] ->17	93.05%	93.48%	89.10%	94.42%	93.22%	88.76%

Tableau 8: Validation CSPVV sur le système ANT

Les résultats obtenus après validation de type CSPV sur le système ANT nous permettent de constater qu'a la première validation faite sur la version 13 du système ANT, le classificateur semble avoir de la peine à faire de bonnes classifications. Cependant, aux prochaines validations, nous remarquons une augmentation graduelle du taux d'exactitude. Cette baisse du taux d'exactitude remarquée à la première validation est due à la présence de bruits dans les données d'entraînement causée par la combinaison des données de deux systèmes différents. Après validation sur chaque version du système ANT, nous obtenons en moyenne des taux d'exactitude supérieurs à 78 %. Notons aussi que les taux d'exactitude

élevés sont obtenus pour un niveau de granularité ligne de code avec un seuil S de couverture de test fixé à 50 % (CLOC50).

5.3.3 Conclusion partielle

La validation CSPVV est similaire à la validation CPVV avec la seule différence que nous ajoutons un autre système (versions d'un autre système) au jeu de données d'entraînement. Les résultats obtenus après validation CSPVV sur ces deux systèmes nous permettent de conclure qu'il est possible de cibler les classes nécessitant un test en se basant sur un historique construit sur les versions antérieures ainsi qu'un autre système quelconque. La baisse des taux d'exactitude lors de la validation sur certaines versions (Version 14 pour le système POI et version 13 pour le système ANT) comme nous l'avons souligné est due à la présence de bruit dans les données d'entraînement. Ainsi, dans le cas d'un apprentissage sur plusieurs données venant de différents systèmes, il est fort possible qu'il y ait du bruit dans les données d'entraînement et cela rendrait le classificateur moins performant. Pour remédier à ce phénomène, l'idéal serait de : (1) faire une étude sur la qualité des systèmes utilisés, mais aussi de (2) faire un entraînement sur beaucoup d'autres systèmes et sur un grand nombre de données. Cela rendrait l'apprentissage du classificateur plus général et moins spécifique à une version ou à un système.

5.4 LOSOV Validation

La validation LOSOV consiste à prédire les classes à tester d'un système entier S (versions combinées du système) par un classificateur qui a été entrainé sur plusieurs autres systèmes différents de S.

Exemple : Soit un ensemble de systèmes S, S1, S2, S3. Une validation LOSOV sur le système S consisterait a entraîné avant un classificateur sur

les données des systèmes S1, S2 et S3, ensuite l'utiliser pour prédire les classes à tester sur le système S.

L'objectif de cette validation est d'investiguer la possibilité de prédire les classes à tester d'un système par un classificateur entraîné sur un ou plusieurs autres systèmes différents de S.

5.4.1 Résultats obtenus après validation sur le système ANT

Le tableau 9 suivant nous donne les résultats obtenus après une validation LOSOV sur le système ANT en considérant les deux niveaux de granularité et les trois types de seuils pour chaque niveau.

Dans ce premier cas, nous avons entraîné le classificateur sur toutes les versions du système POI et validé le classificateur en le testant sur le système ANT (toutes les versions combinées du système ANT).

Système	Granularité Méthodes			Granularité lignes de codes		
	CM50	CM30	CM0	CLOC50	CLOC30	CLOC0
POI	73.59 %	71.53 %	68.49 %	75.49 %	73.42 %	69.11 %

Tableau 9: Validation LOSOV sur le système ANT.

En entrainant un classificateur sur toutes les versions du système POI pour prédire les classes à tester du système ANT (toutes les versions du système ANT), le taux d'exactitude le plus élevé (75.59 %) est obtenu pour un niveau de granularité méthode avec un seuil S de niveau de couverture de test fixé à 50 % (CLOC50). Le reste des taux d'exactitude obtenus avec différents seuils de niveau de couverture de test varient peu, mais gravitent autour de 70 %. Ainsi, grâce aux résultats obtenus et pour un seuil de taux d'exactitude fixé à 70 %, nous sommes capables de prédire les classes à tester d'un système S grâce à un classificateur entraîné sur un système S'.

5.4.2 Résultats obtenus après validation sur le système POI

Le tableau 10 suivant nous donne les résultats obtenus après une validation LOSOV sur le système POI en considérant les deux niveaux de granularité et les trois types de seuils pour chaque niveau.

Dans ce premier cas, nous avons entrainé le classificateur sur toutes les versions du système ANT et validé le classificateur en le testant sur le système POI (en combinant toutes les versions du system POI).

Système	Granularité Méthode			Granularité lignes de codes		
	CM50	CM30	CM0	CLOC50	CLOC30	CLOC0
ANT	75.72 %	70.83 %	70.01 %	81.63 %	76.14 %	69.45 %

Tableau 10: Validation LOSOV sur le système POI.

En entraînant un classificateur sur les données du système ANT puis en le validant sur le système POI, nous obtenons le taux d'exactitude le plus élevé (81.63 %) pour un niveau de granularité « lignes de code » avec un seuil S de niveau de couverture de test fixé à 50 % (CLOC50). Nous remarquons que le reste des taux d'exactitude obtenus pour différents niveaux de granularité et différents seuils de couverture de test sont supérieurs ou presque égaux à 70%. Nous pouvons conclure qu'a un seuil de taux d'exactitude à 70%, nous sommes capables à partir d'un apprentissage effectué sur un autre système, de prédire les classes à tester des versions d'un système en cours de développement.

Nous remarquons aussi que le classificateur entraîné sur les versions du système ANT et validé sur le système POI nous donne de meilleurs résultats que celui entraîné sur le système POI pour prédire le système ANT. Cela soutient notre opinion concernant l'existence de fortes corrélations entre les données (métriques) des versions du système ANT. Ce qui n'est pas le cas des données du Système POI.

5.4.3 Conclusion partielle

Les résultats montrent qu'avec un seuil de 70 %, il est possible qu'à partir d'un classificateur entraîné sur un système donné S, de prédire les classes à tester en priorité pour un système diffèrent S'.

Les résultats obtenus après la validation LOSOV nous montrent que les taux d'exactitude élevés sont obtenus lorsque nous considérons un niveau de granularité lignes de codes. En effet, pour le système POI nous obtenons en moyenne un taux d'exactitude de 73 % pour le niveau granularité méthode et 75 % pour le niveau granularité lignes de code. Au niveau du système ANT, nous obtenons en moyenne un taux d'exactitude de 75 % pour le niveau granularité méthode et 80 % pour le niveau granularité lignes de code. La grande différence observée encourage donc le choix du niveau de granularité lignes de code.

5.5 Limitations

Ces résultats sont certes assez significatifs au regard du nombre et de la taille des logiciels analysés, mais doivent néanmoins être considérés comme exploratoires. Des facteurs cachés peuvent, en effet, concourir à expliquer certains résultats (et limites observées). Dans ce cadre, nous pouvons remarquer que les systèmes open source utilisés présentent certaines spécificités dans leurs processus. Par exemple, le choix du nombre et des classes à tester explicitement est dans la plupart du temps laissé au bon vouloir des développeurs. Cela peut entraîner des classes partiellement testées et des systèmes ayant peu de classes testées, ce qui en retour, peut influencer les résultats obtenus dans nos différentes expérimentations.

La généralisation de nos résultats nécessite des investigations complémentaires notamment des tests de qualité des systèmes logiciels,

l'analyse de systèmes propriétaires (non-open source) car les processus de développement et de suivi de ces systèmes (souvent) industriels se font en général selon des normes de qualité plus rigoureuses comparées aux normes de qualité des systèmes open source. Les équipes de développement peuvent aussi être plus restreintes. Par ailleurs, les domaines d'application des logiciels analysés peuvent aussi impacter les résultats et restreindre leurs portées. La diversification des sources de données d'analyse avec notamment le choix de logiciels propriétaires et (du facteur) du domaine d'application des logiciels analysés dans nos investigations faciliterait la généralisation des résultats observés. Dans nos analyses, nous nous sommes limités au langage Java. Même si ce dernier est aujourd'hui une référence en matière de langage de développement, mature, populaire et intégrant tous les artéfacts de la technologie orientée objet, il reste néanmoins que nos résultats peuvent être biaisés par la considération de systèmes uniquement développés dans ce langage, ce qui contribue à restreindre leur portée.

Dans le prochain chapitre nous ferons une conclusion globale et présenterons nos futures perspectives.

Chapitre 6 CONCLUSION GÉNÉRALE

Rappel de la Problématique

Dans ce mémoire, nous avons investigué la problématique de la priorisation des tests unitaires dans les systèmes orientés objet. L'objectif était de proposer une approche basée sur l'apprentissage profond avec pour but de suggérer des classes à tester en priorité en se basant sur l'historique du système logiciel, les métriques de code source, les suites de test JUnit ainsi que les couvertures de tests JUnit. Pour cela, nous avons construit des classificateurs entraînés sur les données logicielles des systèmes logiciels sur lesquels nous avons travaillé (POI et ANT). Une fois l'apprentissage terminé, il fallait valider ces classificateurs en faisant des séries de tests.

Contribution

Dans ce contexte, nous avons exploré différentes facettes des liens entre les métriques logicielles utilisées (OO) et les tests unitaires écrits. Les études menées sur deux systèmes logiciels OO open source, ont montré la possibilité de prédire les classes à tester. Cela nous a donc permis de proposer un modèle de priorisation des classes à tester à partir des données logicielles.

Cette recherche nous a permis de mettre en évidence la possibilité : (1) d'entraîner un réseau de neurones profond sur des données d'une version antérieure et de construire un classificateur pouvant prédire les classes à tester pour une version en cours de développement, (2) de combiner les versions antérieures d'un même système logiciel afin de faire des prédictions sur la version postérieure, et (3) de bâtir des classificateurs sur les données combinées de différents systèmes (dans notre cas deux) capables de faire des prédictions sur un autre système.

Recherches futures

Ces résultats entrouvrent les possibilités d'utiliser les métriques logicielles et l'expérience des développeurs dans l'orientation de l'effort global des tests unitaires. En effet, se basant sur ces résultats, il serait intéressant de développer des outils basés sur le Cloud et les techniques d'analyse du Big data actuelles (impliquant des algorithmes d'intelligence artificielle) pour bâtir une nouvelle génération d'outils de priorisation et d'orientation des tests qui seront intégrés aux environnements de développement et seront basés sur la collaboration communautaire des développeurs logiciels. Cette piste sera notre futur axe de recherche.

Chapitre 7 RÉFÉRENCES BIBLIOGRAPHIQUES

[1] P. Gajalakshmi, "Software Development Lifecycle Model (SDLC) Incorporated With Release Management", International Research Journal of Engineering and Technology (IRJET) 2016.

[2] L. Craig, B. Victor, "Iterative and Incremental Development: A Brief History", IEEE Computer, pp. 1- 10, June 2003.

[3] W. Royce, "Managing the Development of Large Software Systems", Proceedings of IEEE WESCON 26, pp.1-9, 1970.

[4] B. Boehm. "Software Engineering Economics", Prentice Hall, Englewood Cliffs, NJ, 1981.

[5] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Prioritizing test cases for regression testing". Proceding of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pages 102-112. Portland, August 2000.

[6] G. McGraw, "Software Security Testing". IEEE Security and Privacy 2,2, (80-83), Sept/Oct 2004.

[7] S. R. Pressman, "Software Engineering: A Practitioner's Approach" 7th Edition, Mc Gray Hill, 2004.

[8] S. Elbaum, A. G. Malishevsky and G. Rothermel, "Test Case Prioritization: A Family of Empirical Studies," IEEE Transactions Software Engineering, Vol. 28, No. 2, pp.159-182, 2002.

[9] Y. Huang, K. Peng, "A history-based cost-cognizant test case prioritization technique in regression testing", Published in Journal of Systems and Software, on Mar 1, 2012.

[10] T. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection techniques". In

Proceedings of the 20th International Conference on Software Engineering, pages 188–197. IEEE Computer Society Press, April 1998.

[11] F. Toure, M. Badri & L. Lamontagne, "Investigating the Prioritization of Unit Testing Effort using Software Metrics", Published in ENASE, 2017.

[12] F. Toure, M. Badri & L. Lamontagne, "Predicting different levels of the unit testing effort of classes using source code metrics: a multiple case study on open-source software", Publication: Innovations in Systems and Software Engineering March 2018.

[13] S. R. Chidamber , C. F. Kemmerer, "A Metrics Suite for Object Oriented Design", IEEE Transactions on Software Engineering, vol. 20, no. 6, pp. 476–493, 1994.

[14] V. Gupta, K. K Aggarwal & Y. Singh, "A Fuzzy Approach for Integrated Measure of Object-Oriented Software Testability", Journal of Computer Science, Vol. 1, No. 2, pp. 276-282, 2005.

[15] M. Bruntink, A. V. Deursen, "Predicting Class Testability using Object-Oriented Metrics", 4th Int. Workshop on Source Code Analysis and Manipulation (SCAM), IEEE, 2004.

[16] M. Bruntink, A. Van Deursen, "An Empirical Study into Class Testability", Journal of Systems and Software, Vol. 79, No. 9, pp. 1219-1232, 2006.

[17] P. Stone, R. Brooks, E. Brynjolfsson, R. Calo, O. Etzioni, G. Hager, J. Hirschberg, S. Kalyanakrishnan, E. Kamar, S. Kraus, K. Leyton-Brown, D. Parkes, W. Press, A. Saxenian, J. Shah, M. Tambe, and A. Teller, "Artificial Intelligence and Life in 2030", One Hundred Year Study on Artificial Intelligence: Report of the 2015-2016 Study Panel, Stanford University, Stanford, CA, Doc: http://ai100.stanford.edu/2016-report. Accessed: September 6, 2016.

[18] L. Yann, Y. Bengio & H. Geoffrey, "Deep learning", Macmillan Publishers Limited, Received 25 February; accepted 1 May 2015.

[19] A. M. Turing, "On computable numbers with computable numbers with an application to the ENTSCHEIDUNGS problem", Proceedings of the London Mathematical Society, Volume s2-42, Pages 230–265 November 1936.

[20] P. Maguire, P. Moser and M. Rebecca, "A clarification on Turing's test and its implications for machine intelligence", Journal of Cognitive Science 17-1: 63-94, 2016.

[21] B. Yoshua, "Learning Deep Architectures for AI, Foundation and Trends in Machine Learning", vol 2, no 1, pp 1–127, 2009.

[22] L. Yann, Y. Bengio & H. Geoffrey, "Deep learning", Macmillan Publishers Limited, Received 25 February; accepted 1 May 2015.

[23] M. Agarwal, A. Barham, "TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorfiow.org", Preliminary White Paper, November 9, 2015.

[24] P. Baldi and P. Sadowski. "The dropout learning algorithm. Artificial Intelligence", Information Science and Applications (ICISA) 2016.

[25] P. Baldi, P. J. Sadowski "Understanding dropout", Advances in Neural Information Processing Systems 26, NIPS 2013.

[26] "Deep Mind", https://deepmind.com/research/case-studies/alphago-the-story-so-far, Visité en Mars 2020.

[27] I. Goodfellow, Y. Bengio and A. Courville, "Deep learning, From Adaptive Computation and Machine Learning series", First edition, MIT Press, 2016.

[28] G. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from

overfitting", The Journal of Machine Learning Research (JMLR) 1929-1958. January 2014.

[29] D. P. Kingma, J. Ba, Adam: "A method for stochastic optimization". Published as a conference paper at The International Conference on Learning Representations (ICLR) 2015.

[30] "THE MNIST Database, of handwritten digits", http://yann.lecun.com/exdb/mnist/, Visité en Mars 2020.

[31] R. Pascanu, T. Mikolov, and Y. Bengio, "Understanding the exploding gradient problem", ArXiv abs/1211.5063: n. pag, 2012.

[32] Y. Yua, M. F. Laub, "Fault-based test suite prioritization for specification-based testing", Information and Software Technology Volume 54, Issue 2, Pages 179–202, February 2012.

[33] Lazić, L.: Software Testing Optimization by Advanced Quantitative Defect Management. Computer Science and Information Systems, Vol. 7, No. 3, 459-487, https://doi.org/10.2298/CSIS090923008L , (2010).

[34] G. Rothermel, M. J. Harrold, J. Ronne and C. Hong, "Empirical Studies of Test-Suite Reduction", In Journal of Software Testing, Verification, and Reliability, Vol. 12, No.4, 2002.

[35] S. Mirarab, A. Hassouna, & L. Tahvildar. "Using Bayesian belief networks to predict change propagation in software systems" in Proceedings of the 15th IEEE International Conference on Program Comprehension, pages 177-188, 2007.

[36] W. Wong, J. Horgan, S. London, and H. Agrawal, "A study of effective regression in practice", Proceedings of the 8th International Symposium on Software Reliability Engineering, November, p.230–238, 1997.

[37] R. Bryce, A. M. Memon, "Test Suite Prioritization by Interaction Coverage", Proceedings of the Workshop on Domain Specific Approaches to Software Test Automation (DOSTA 2007). ACM: New York, 2007.

[38] J. Kim, and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments", In Proceedings of International Conference on Software Engineering, May 2002.

[39] C. T Lin, C.D Chen, and C.S Tsai and G. M. Kapfhammer, "History-based Test Case Prioritization with Software Version Awareness", International Conference on Engineering of Complex Computer Systems, 2013.

[40] R. Carlson, A. Denton, "A clustering approach to improving test case prioritization, An industrial case study", 27th IEEE International Conference on Software Maintenance (ICSM), 2011.

[41] S. Elbaum, G. Rothermel, S. Kanduri and A. G Malishevsky, "Selecting a cost-effective test case prioritization technique", Software Quality Control, 12(3):185–210, 2004.

[42] B. Qu, C. Nie, B. Xu and X. Zhang, "Test case prioritization for black box testing", 31st Annual International Computer Software and Application conference, COMPSAC, 2007.

[43] H. Spieker, G. Arnaud, M. Dusica and M. Morten "Reinforcement learning for automatic test case prioritization and selection in continuous integration", Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis Pages 12–22https://doi.org/10.1145/3092703.30927092017, July 2017.

[44] "Apache Software Fondation", https://www.apache.org/ Visité en Mars 2020.

[45] L. Briand, J. Wust, J. Daly, and D. Porter, "Exploring the Relationship between Design Measures and Software Quality in Object Oriented Systems", J. Systems and Software, vol. 51, no. 3, pp. 245-273, 2000.

[46] M. Hitz, B. Montazeri, Chidamber and Kemerer's Metrics Suite, "A Measurement Theory Perspective", IEEE Transactions on Software Engineering, Vol. 22, No. 4, pp. 266-270, 1996.

[47] T. J McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering, Volume: SE-2 , 308–320, Dec 1976.

[48] "Github", https://github.com/, Visité en Mars 2020.

[49] "Junit 5", https://junit.org/junit5/, Visité en Mars 2020.

[50] "Code Mr", https://plugins.jetbrains.com/plugin/10811-codemr/, Visité en Mars 2020.

[51] "IntelliJ idea", https://www.jetbrains.com/idea/, Visité en Mars 2020.

[52] "Code-Coverage", https://www.jetbrains.com/help/idea/code-coverage.html, Visité en Mars 2020.

[53] "POI releases", https://github.com/apache/poi/releases, Visité en Mars 2020.

[54] "ANT releases", https://github.com/apache/ant/releases, Visité en Mars 2020.

[55] "Introduction to Neurol networks", http://pages.cs.wisc.edu/~bolo/shipyard/neural/local.html, Visité en Mars 2020.

[56] C. Maureen, & B. Charles, "Understanding Neural Networks", Bradford Books, Trends in Neurosciences, 1992.

[57] A. Smola, S.V.N. Vishwanathan, "Introduction to Machine Learning", published by the press syndicate of the university of Cambridge, Library of Congress Cataloguing in Publication data available, ISBN 0 521 82583 0 hardback, First published 2008.

[58] W. Richert, L. P. Coelho, "Building Machine Learning Systems with Python", Packt Publishing Ltd., ISBN 978-1-78216-140-0, First published July 2013.

[59] M. Welling, D. Bren "A First Encounter with Machine Learning", Donald Bren School of Information and Computer Science University of California Irvine, Published 2010.

[60] M. Bowles, "Machine Learning in Python: Essential Techniques for Predictive Analytics", John Wiley & Sons Inc, ISBN: 978-1-118- 96174-2, First Edition.

[61] S.B. Kotsiantis, "Supervised Machine Learning: A Review of Classification Techniques", Emerging Artificial Intelligence Applications in Computer Engineering, Artificial Intelligence Review, November 2006.

[62] L. Rokach, O. Maimon, "Top – Down Induction of Decision Trees Classifiers – A Survey", IEEE Transactions on Systems, IEEE transactions on systems, man and cybernetics: Part C, Vol. 1, No. 11, November 2002.

[63] D. Lowd, P. Domingos, "Naïve Bayes Models for Probability Estimation", Machine Learning, Proceedings of the Twenty-Second International Conference, Bonn, Germany, August 7-11, 2005.

[64] J. G. Dy and C. E. Brodley, "Feature Selection for Unsupervised Learning", Journal of Machine Learning Research 5, 845–889, 2004.

[65] L. K. Saul and S. T. Roweis. "Think Globally, Fit Locally: Unsupervised Learning of Low Dimensional Manifolds", Journal of Machine Learning Research 4, 119-155, 2003.

[66] O. Bousquet, G. Raetsch, and U. von Luxburg, "Unsupervised Learning", Advanced Lectures on Machine Learning, ML Summer Schools, Canberra, Australia, February 2-14, 2003.

[67] "Face recognition", https://towardsdatascience.com/an-intro-to-deep-learning-for-face-recognition-aa8dfbbc51fb, Visité en Mars 2020.

[68] V. François-Lavet, P. Henderson, R., G. Marc, "An Introduction to Deep Reinforcement Learning", Foundations and Trends in Machine Learning: Vol. 11, No. 3-4. DOI : 10.1561/2200000071, 2018.

[69] "Support de cours, Toulouse, Réseaux de neurones artificiels", https://www.math.univ-toulouse.fr/~besse/Wikistat/pdf/st-m-app-rn.pdf.

[70] F. Rosenblatt, "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain", Psychological Review, 65(6), 386–408. https://doi.org/10.1037/h0042519, 1958.

[71] "Neurone biologique vs neurone artificiel", http://penseeartificielle.fr/focus-reseau-neurones-artificiels-perceptron-multicouche/analogie-entre-neurone-biologique-et-artificiel/, Visité en Mars 2020.

[72] M. Marvin and P. Seymour, "Perceptrons: An Introduction to Computational Geometry", The M.I.T. Press Cambridge Mass, Volume 17, Issue 5, 1969.

[73] "Représentation Neurone biologique vs neurone artificiel", https://patducjacquet.wordpress.com/2017/06/16/perceptron/, Visité en Mars 2020.

[74] "Architecture d'un neurone Artificiel", https://ludo-louis.fr/types-reseaux-neurones-reseaux-neurones-artificiels/, Visité en Mars 2020.

[75] P. Barnabas, "Introduction to Machine Learning (Lecture Notes) and Perceptron", http://alex.smola.org/teaching/cmu2013-10-701/index.html, Visité en Mars 2020.

[76] S.Haykin, "Neural Networks and Learning Machines", Pearson, Prentice Hall, Third Edition, 2008.

[77] E. R. David, H. E Geoffrey, and J. Ronald, "Learning representations by back-propagating errors", published in D. E. Rumelhart & i. L. McClelland (Eds.), Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1: Foundations. Cambridge, MA: Bradford Books/MIT Press, September 1985.

[78] "Rétropropagation", https://www.cairn.info/revue-reseaux-2018-5-page-173.html, Visité en Mars 2020.

[79] "Minimum local et minimum global", https://www.researchgate.net/figure/the-difference-between-a-local-minimum-and-a-global-minimum-the-local-minimum-represents_fig44_30577305, Visité en Mars 2020.

[80] "Tensorflow", https://www.tensorflow.org/, Visité en Mars 2020.

[81] "Keras", https://keras.io/, Visité en Mars 2020.

[82] "Pandas", https://pandas.pydata.org/, Visité en Mars 2020.

[83] H. E. Geoffrey, S. Osindero, and T. Yee-Whye, "A fast learning algorithm for deep belief nets", Publication : Neural Computation, https://doi.org/10.1162/neco.2006.18.7.1527, Neural computation 1527-1554, 2006.