UQAC
Université du Québec
à Chicoutimi

COMMENT L'APPRENTISSAGE AUTOMATIQUE PEUT DÉMYSTIFIER LES VULNÉRABILITÉS DE SÉCURITÉ DES LOGICIELS : UNE ÉTUDE EMPIRIQUE

PAR MOHAMED DIOUF

MÉMOIRE PRÉSENTÉ À L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI EN VUE DE L'OBTENTION DU GRADE DE MAÎTRE ÈS SCIENCES (M. SC.) EN INFORMATIQUE

QUÉBEC, CANADA

RÉSUMÉ

L'augmentation des incidents de violation de données au cours de la dernière décennie met en lumière les failles de sécurité présentes dans divers systèmes logiciels. Il est essentiel de noter que les défauts de sécurité constituent l'une des principales sources de vulnérabilités logicielles, car ils peuvent être exploités pour accéder de manière non autorisée à un système d'information. Dans ce mémoire, nous examinons les mesures de qualité des codes sources afin de déterminer leur efficacité à signaler l'existence de défauts. Dans cette perspective, nous avons réalisé une étude empirique sur sept écosystèmes open source, démontrant la possibilité de développer un modèle de prédiction capable de détecter les défauts de sécurité avec un niveau de précision élevé. De plus, nos observations ont révélé que les valeurs médianes de certaines métriques de qualité dans les fichiers contenant des défauts étaient en moyenne trois fois supérieures à celles des fichiers exempts de défauts. Ainsi, nous avons pu conclure que l'association entre l'analyse des mesures de qualité des codes sources et l'application d'algorithmes d'apprentissage automatique peut être utilisée pour établir un système de prévision, facilitant ainsi la détection précoce des vulnérabilités de sécurité par les développeurs.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ABRÉVIATIONS

AUC	Area Under the Curve
GBM	Gradient Boosting Method
MDP	Metrics Data Program
CVE	Common Vulnerabilities and Exposures
JSE	JavaScript Engine de Mozilla
INP	Input Validation Error
MEM	Memory Errore
CON	Configuration Error
CWE	Common Weakness Enumeration
OSV	Open Source Vulnerability
SMOTE	Synthetic Minority Over-sampling Technique
AVNNet	Advanced Vulnerability Network Neural Network
GBM	Gradient Boosting Machine
MDP	NASA Metrics Data Program
DPCNN	Deep Pyramid Convolutional Neural Network
DNN	Deep Neural Network

DÉDICACE

À Dieu tout-puissant, sans qui rien n'aurait été possible, je rends grâce pour sa miséricorde et sa protection, même dans les moments les plus sombres.

À mes parents bien-aimés, ***Ndeye Saynabou Diouf*** *et* ***Niokhobaye Diouf****, vos sacrifices incommensurables, votre amour infini et votre dévouement ont porté chacun de mes pas. Vous avez tout donné pour que je puisse rêver et réussir. Je vous dois tout, et ce travail est une infime partie de la dette que je ne pourrai jamais entièrement rembourser.*

À mes oncles, qui ont été bien plus que des figures familiales :

Ousmane Biteye Diouf*, chez qui j'ai vécu cinq ans de formation intense. Il m'a non seulement encadré, mais il a façonné l'homme et l'étudiant que je suis aujourd'hui. Ses conseils m'ont guidé à chaque étape, et son soutien sans faille a illuminé même les jours les plus sombres.* ***Elhadj Alioune Badara Diou****f, dont la générosité financière et morale a été mon pilier tout au long de ce parcours éprouvant. Grâce à lui, j'ai pu poursuivre mes études sans jamais perdre espoir, malgré les difficultés.* ***Oumar Diogoye Diouf****, dont la sagesse et les encouragements constants m'ont réconforté dans les moments de doute. Chacun de vous a été une lumière dans mon cheminement, m'a donné la force quand tout semblait s'effondrer. Vous avez cru en moi même lorsque j'avais du mal à y croire. Ce mémoire est le fruit de votre amour, de vos sacrifices, et de votre foi en mon avenir.*

REMERCIEMENTS

Mon directeur de recherche, Monsieur Fehmi Jaafar, mérite ma plus sincère reconnaissance pour son soutien constant et précieux tout au long de la réalisation de ce mémoire. Les conseils avisés, son expertise et sa disponibilité ont joué un rôle crucial dans la réussite de ce projet. Son aptitude à me diriger de manière rigoureuse et bienveillante a renforcé mon travail et m'a donné l'opportunité de progresser au-delà de mes attentes. Je suis extrêmement reconnaissant pour sa confiance et son dévouement, qui ont été des sources de motivation inestimables tout au long de cette période académique.

Je tiens aussi à exprimer ma gratitude envers tout le personnel du Département de Mathématiques et Informatique (DIM) de l'UQAC pour leur cadre chaleureux et stimulant. Grâce à chaque discussion avec mes enseignants et collègues, j'ai pu approfondir mes connaissances et acquérir des compétences précieuses, tant pour ce travail que pour mon avenir professionnel.

Enfin, je souhaite exprimer ma gratitude envers mes proches, dont le soutien inébranlable, l'amour et la confiance en moi ont été des sources inestimables de force et de motivation. Ce parcours a été rendu possible grâce à leur présence à mes côtés, que ce soit lors des moments de doute ou de réussite. Ce travail leur est consacré, en signe de ma reconnaissance et de mon amour.

AVANT-PROPOS

Je présente ce mémoire avec une grande gratitude et une grande satisfaction, étant le fruit d'un parcours aussi exigeant qu'inspirant dans le domaine de la sécurité logicielle et de l'apprentissage automatique. Ce travail dépasse largement une perspective purement académique et reflète mon engagement à approfondir ma compréhension et à résoudre les problèmes de sécurité dans les logiciels, une question essentielle dans notre monde de plus en plus connectée.

J'ai été intéressé par l'idée de mettre les capacités de l'intelligence artificielle au service de la sécurité informatique dès nos premiers échanges avec mon directeur de recherche. Ce projet est issu de cet objectif commun : utiliser l'intelligence artificielle pour repérer et prévenir les vulnérabilités de sécurité, souvent difficiles à prévoir et pourtant si importantes. Non seulement l'objectif était d'explorer les possibilités techniques, mais également de favoriser la confiance envers les systèmes logiciels sur lesquels nous nous appuyons au quotidien.

Le processus de réalisation de cette tâche a été marqué par des défis et des interrogations constantes, tout en offrant des enseignements et des découvertes enrichissantes. Ma conviction s'est renforcée à chaque étape de recherche, d'analyse et de validation des résultats, car les progrès en apprentissage automatique offrent des perspectives réelles pour anticiper et réduire les risques de sécurité.

Ce travail est dédié à tous ceux qui, de près ou de loin, m'ont soutenu et inspiré, et j'espère qu'il contribuera, même modestement, à l'évolution des méthodes de sécurité logicielle dans le domaine du développement logiciel. Je vous convie à explorer cette étude pratique, résultant de mon intérêt pour la technologie et de mon désir de contribuer à un avenir numérique plus sécurisé.

INTRODUCTION

0.1 CONTEXTE

Les bugs de sécurité sont les erreurs de programmation les plus critiques qui peuvent engendrer des vulnérabilités non négligeables dans les logiciels. En réalité, de telles erreurs donnent à un individu malveillant la possibilité d'accéder au logiciel, de voler des informations ou d'empêcher le bon fonctionnement du logiciel Clemente *et al.* (2018). À l'heure actuelle, on assiste à une augmentation significative des attaques cybernétiques pour différentes raisons, comme le profit financier, la collecte de données personnelles à des fins de profilage, l'espionnage, etc. Par exemple, en 2022, les attaques menées par les pirates informatiques causèrent des pertes pouvant être estimées à 7000 milliards de dollars selon Cybersecurity Ventures [1]. Ce montant est extrêmement élevé, au point qu'il est supérieur au budget de l'ensemble des pays du monde, sauf les États-Unis et la Chine. Selon plusieurs recherches, les bugs de sécurité des logiciels sont l'une des principales raisons de l'apparition de vulnérabilités de sécurité dans les logiciels et ont un impact sur les éléments essentiels du logiciel Zaman *et al.* (2011). Par conséquent, avec l'augmentation rapide du nombre de bugs de sécurité, il est primordial de les analyser et de mettre en lumière leur lien avec la qualité et la fiabilité des systèmes logiciels Tan *et al.* (2014).

0.2 PROBLÉMATIQUE

Il est essentiel de prendre en compte l'aspect de la sécurité dès les premières étapes du processus de mise en place d'un système logiciel dans le but de réduire au minimum les risques de bugs de sécurité Neuhaus *et al.* (2007); Scandariato *et al.* (2014). Toutefois, l'analyse dynamique, bien que largement utilisée par les développeurs, produit un grand

1. https ://cybersecurityventures.com/boardroom-cybersecurity-report/

nombre de faux positifs, ce qui rend leur exploitation complexe. Un faux positif correspond à une alerte signalant une vulnérabilité alors qu'aucune faille réelle n'est présente. Par exemple, un outil d'analyse de code peut identifier une fonction comme vulnérable simplement parce qu'elle utilise une bibliothèque jugée risquée, alors qu'en réalité, toutes les bonnes pratiques de sécurité ont été suivies. Cette surabondance d'alertes erronées entraîne une surcharge de travail pour les équipes de développement, qui doivent examiner chaque signalement afin de distinguer les véritables failles des erreurs de détection Aggarwal & Jalote (2006). En outre, la sophistication grandissante des systèmes contemporains rend ardue l'identification manuelle des vulnérabilités de sécurité. Il est donc essentiel de mettre au point des méthodes automatiques et performantes afin de prédire et de corriger ces bugs avant qu'ils ne soient exploités par des attaquants.

0.3 CONTRIBUTION

Dans ce mémoire, nous appliquons des modèles d'apprentissage automatique pour analyser et prédire les bugs de sécurité dans sept systèmes logiciels open source. Cette recherche repose sur un ensemble de **25 métriques de qualité**, comprenant la complexité cyclomatique, le nombre de classes et de méthodes, ainsi que le niveau d'imbrication maximal. En complément, le jeu de données inclut des variables contextuelles pour chaque fichier, telles que la **date du commit**, **l'identifiant SHA du commit**, **le chemin du fichier dans le dépôt**, **l'écosystème** (par exemple, npm, Maven), et l'information de **source** (pré-commit ou post-commit).

Notre jeu de données couvre environ **340 000 fichiers**, parmi lesquels **34 294 fichiers sont identifiés comme buggés** et répartis dans plusieurs langages de programmation (Java, JavaScript, Python, PHP, C, etc.). Ces données proviennent de divers écosystèmes, notamment **Maven**, **npm**, **NuGet**, **Packagist**, **PyPI**, **GHSA**, et **OSS-Fuzz**.

Une étude empirique a été menée pour comprendre la corrélation entre les indicateurs de qualité et les problèmes de sécurité mais également pour prédire les bugs de sécurité. Tous les outils employés dans cette recherche sont Open Source et librement accessibles. En outre, nous proposons une analyse approfondie des résultats obtenus avec divers modèles de machine learning, en soulignant les métriques ayant la plus grande influence dans la prédiction des bugs de sécurité. Nos résultats fournissent des indications utiles pour guider les développeurs dans l'amélioration de la sécurité des logiciels.

0.4 MÉTHODOLOGIE

Tout d'abord, nous collectons les informations relatives aux vulnérabilités de sécurité en nous appuyant sur la base de données **OSV (Open Source Vulnerabilities)** [2] et sur la plateforme de dépôt de logiciels **Github** [3]. Par la suite, nous avons repéré les fichiers touchés par des problèmes de sécurité en utilisant des rapports de bugs et des journaux de mises à jour (commit). Nous nous sommes servi de l'outil **Understand tools** [4] afin d'extraire toutes les mesures de qualité des fichiers, qu'ils soient directement impactés ou qu'ils appartiennent à un paquage contenant au moins un fichier corrompu. Par la suite, dix modèles d'apprentissage automatique ont été utilisés pour étudier la corrélation entre les indicateurs de qualité des logiciels et les problèmes de sécurité, ainsi que pour étudier la prédiction des bugs de sécurité.

2. https ://osv.dev/
3. https ://github.com/
4. https ://scitools.com/

0.5 ORGANISATION DU DOCUMENT

Nous avons structuré ce mémoire en cinq parties et notre but est de réaliser une étude empirique sur les failles des logiciels engendrées par les bugs de sécurité :

Chapitre I : Revue de la littérature - Dans ce chapitre, nous allons examiner une série d'études antérieures qui ont analysé les problèmes logiciels en général, ainsi que les vulnérabilités et les problèmes de sécurité des logiciels spécifiquement. Grâce à ces travaux, il sera possible d'approfondir notre compréhension des approches actuelles et des méthodologies employées pour repérer et résoudre ces bugs de sécurité.

Chapitre II : Processus de collecte des données - Au cours de ce chapitre, nous allons examiner en détail le processus de collecte des données, en expliquant les différentes étapes d'extraction, la raison pour laquelle nous avons choisi des systèmes logiciels ainsi que la méthode de collecte des différentes mesures de qualité.

Chapitre III : Méthodologie de l'étude empirique - Ce chapitre présente la méthodologie adoptée pour mener l'étude empirique. Nous y détaillerons les algorithmes d'apprentissage automatique, les métriques utilisées pour évaluer leur performance, ainsi que les étapes du processus expérimental. Nous expliquerons également comment les modèles ont été entraînés, validés et testés pour détecter les vulnérabilités.

Chapitre IV : Résultats de l'étude empirique - Dans ce chapitre, nous présenterons les résultats obtenus en appliquant les méthodes et modèles décrits précédemment. Ces résultats incluront des analyses quantitatives et qualitatives des performances des algorithmes, ainsi que des comparaisons des différentes approches testées. Les résultats seront interprétés à la lumière des objectifs initiaux et des travaux précédents.

Chapitre V : Discussion et limites - Ce dernier chapitre propose une réflexion critique sur le travail réalisé et des orientations pour des recherches futures.

D’abord, nous identifions les principales limites de l’étude, telles que les défis liés à la collecte des données (incomplétude, biais) et au choix des modèles, qui peuvent restreindre la portée des résultats. La généralisation des conclusions à d’autres contextes ou types de logiciels demeure également un enjeu.

Ensuite, nous analysons la validité de l’étude en examinant la robustesse des résultats et les biais potentiels, qu’ils soient liés aux données ou aux méthodes. Nous discutons également de la reproductibilité et de l’applicabilité des résultats. Enfin, nous présentons des pistes de recherche future.

Conclusion - Dans ce chapitre, nous faisons un tour des points saillants traités dans cette étude, ainsi que les principales contributions apportées par notre travail. Nous présentons un résumé des résultats obtenus et abordons leur impact sur la détection des bugs de sécurité dans les logiciels analysés. Le présent résumé souligne l’efficacité des diverses méthodes utilisées et leur aptitude à améliorer la prédiction des bugs de sécurité.

CHAPITRE I
REVUE DE LA LITTÉRATURE

Dans ce chapitre, nous étudions les recherches antérieures concernant les problèmes logiciels en général, en mettant l'accent sur les vulnérabilités de sécurité. Nous examinons les méthodes employées pour repérer et prévenir ces vulnérabilités, en mettant en évidence les méthodes d'analyse les plus fréquemment utilisées. Finalement, nous repérons les principales faiblesses de la littérature existante, ce qui permettra de mieux situer l'apport et l'originalité de notre étude.

1.1 CONCEPTS CLÉS ET CONTEXTE THÉORIQUE

Dans le domaine de la sécurité des logiciels, il est crucial de prévenir et de détecter les vulnérabilités afin d'assurer la fiabilité des systèmes et de préserver les données confidentielles. Analyser et anticiper les risques nécessite une compréhension des concepts fondamentaux liés aux vulnérabilités de sécurité, aux métriques de qualité logicielle et à l'utilisation de l'apprentissage automatique. Dans cette partie, nous mettons en relief les concepts essentiels à cette analyse, en examinant des formes courantes de vulnérabilités de sécurité, leurs conséquences et l'importance de leur prédiction proactive. De plus, nous soulignons le rôle des indicateurs de qualité logicielle dans l'évaluation et l'amélioration du code source. Nous étudions aussi comment l'utilisation de l'apprentissage automatique et des bases de données spécialisées permet d'automatiser et de renforcer la détection des vulnérabilités, ce qui ouvre des perspectives prometteuses pour une sécurité logicielle améliorée.

1.1.1 LES VULNÉRABILITÉS DE SÉCURITÉ DES LOGICIELS

Les vulnérabilités de sécurité représentent des failles ou des erreurs dans un logiciel, permettant aux attaquants d'exploiter le système pour compromettre la confidentialité, l'intégrité ou la disponibilité des données. Ces failles peuvent survenir à cause de multiples facteurs, notamment des erreurs humaines (comme des fautes dans le code), de mauvaises pratiques de développement (comme un manque de tests de sécurité) ou des complexités inhérentes à des projets logiciels de grande envergure.

EXEMPLES COURANTS DE VULNÉRABILITÉS

— **Injections SQL** : Ces attaques exploitent des failles dans les bases de données en injectant des commandes SQL malveillantes via des entrées utilisateur non sécurisées.

— **Débordements de mémoire tampon (buffer overflow)** : Une mauvaise gestion de la mémoire au niveau du code source peut permettre à un attaquant de corrompre les données ou d'exécuter du code malveillant.

— **Erreurs d'authentification** : Une mauvaise implémentation des mécanismes d'authentification ou de gestion des sessions peut permettre à un attaquant de contourner les contrôles d'accès.

IMPACT DES VULNÉRABILITÉS

— **Pertes financières** : Par exemple, une fuite de données sensibles peut entraîner des amendes réglementaires ou des pertes de revenus.

— **Atteintes à la réputation** : Les entreprises victimes de cyberattaques subissent souvent une perte de confiance de leurs clients.

— **Violations de données personnelles** : Ces incidents peuvent exposer des informations confidentielles, affectant les individus.

LA PRÉDICTION DES VULNÉRABILITÉS

L'objectif est de détecter de façon proactive les zones du code les plus susceptibles de contenir des failles avant que celles-ci ne soient exploitées. Cela permet aux développeurs de corriger les problèmes avant qu'ils ne causent des dommages.

1.1.2 LES MÉTRIQUES DE QUALITÉ LOGICIELLE

Les métriques de qualité sont des outils permettant d'évaluer de manière objective divers aspects du code source, notamment sa maintenabilité, sa complexité et sa modularité. Ces métriques sont essentielles pour la détection des vulnérabilités, car elles mettent en lumière les pratiques de codage susceptibles d'introduire des erreurs.

EXEMPLES DE MÉTRIQUES DE QUALITÉ

— **Complexité cyclomatique** : Elle mesure la complexité d'un programme en analysant son graphe de contrôle. Un code très complexe est plus difficile à comprendre, à tester et à sécuriser.

— **Densité des commentaires** : Une documentation insuffisante rend le code plus difficile à maintenir et peut indiquer des zones de risque accru.

— **Nombre de lignes de code (LOC)** : Un code volumineux est plus susceptible de contenir des erreurs, car il est plus difficile à gérer.

— **Couplage et cohésion** : Un couplage élevé (interdépendance excessive entre modules) ou une faible cohésion (manque de focus des modules) peut rendre le logiciel plus vulnérable aux erreurs et aux attaques.

Ces métriques fournissent des indications cruciales pour cibler les zones nécessitant une amélioration.

1.1.3 L'APPRENTISSAGE AUTOMATIQUE DANS LE CONTEXTE DES VULNÉRABILITÉS

L'apprentissage automatique (ML) est devenu un outil incontournable pour prédire les vulnérabilités, grâce à sa capacité à analyser des données complexes et à identifier des motifs subtils dans les programmes.

TYPES DE PROBLÈMES ADRESSÉS PAR L'APPRENTISSAGE AUTOMATIQUE

— **Classification** : Déterminer si un fichier ou un module est vulnérable ou non (approche binaire).
— **Régression** : prédire la probabilité ou la gravité d'une vulnérabilité.

EXEMPLES D'ALGORITHMES COURAMMENT UTILISÉS

— **Forêts aléatoires (Random Forests)** : Performants pour gérer les données avec de nombreuses caractéristiques.
— **Machines à vecteurs de support (SVM)** : Utilisées pour détecter les vulnérabilités dans des espaces de données complexes.

— **Réseaux neuronaux profonds (Deep Learning)** : particulièrement adaptés à l'analyse de grandes quantités de données non structurées.

IMPORTANCE DES DONNÉES

Les modèles d'apprentissage automatique sont performants lorsqu'ils utilisent des données de qualité, de quantité et de pertinence pour leur entraînement. Les données jouent un rôle essentiel dans la détection de vulnérabilités dans le code source afin d'assurer des prédictions fiables et exploitables. Ces informations sont issues de différentes sources, chacune offrant une perspective singulière qui enrichit l'apprentissage des modèles.Ces données proviennent de sources variées, telles que :

— Les journaux de commits.

— Les bases de données de bugs, comme OSV (Open Source Vulnerabilities).

— Les métriques de qualité logicielle extraites automatiquement.

1.1.4 LES BASES DE DONNÉES ET OUTILS POUR L'ANALYSE DES VULNÉRABILITÉS

Les chercheurs utilisent des bases de données publiques et des outils spécialisés pour analyser les vulnérabilités et repérer des tendances. Grâce à ces ressources, il est possible de recueillir, structurer et analyser des informations essentielles concernant les vulnérabilités, ce qui facilite leur repérage et leur correction.

EXEMPLE DE BASES DE DONNÉES PUBLIQUES

— **CVE (Common Vulnerabilities and Exposures)** : Une référence centralisée pour répertorier et décrire les failles de sécurité.

— **NVD (National Vulnerability Database)** : Une extension de la base CVE, enrichie avec des détails supplémentaires tels que les scores CVSS (Common Vulnerability Scoring System), les impacts techniques et les correctifs disponibles.

— **OSV (Open Source Vulnerabilities)** : Un dépôt spécifiquement dédié aux projets open source.

OUTILS D'ANALYSE DE CODE

Les outils d'analyse de code sont essentiels pour repérer et prévenir les éventuels problèmes présents dans le code source. Les développeurs et les analystes ont la possibilité d'évaluer la qualité, la structure et la sécurité du code, en fournissant une vision approfondie des éventuelles vulnérabilités ou inefficacités. La prédiction et la gestion des vulnérabilités logicielles sont particulièrement bénéfiques grâce à ces outils, qui automatisent la détection des erreurs et fournissent des rapports exploitables pour améliorer le développement logiciel. Voici quelques exemples d'outils fréquemment employés :

— **SciTools Understand** : Fournit des métriques détaillées pour évaluer la qualité et la structure du code.

— **SonarQube et Coverity** : Outils d'analyse statique qui détectent les problèmes potentiels dans le code avant son exécution.

— **Checkmarx** : Conçu pour renforcer la sécurité des applications, cet outil met l'accent sur l'identification des vulnérabilités critiques dans le code source, tout en s'intégrant facilement aux environnements de développement existants.

— **Pylint** : Destiné aux projets Python, cet outil détecte les erreurs de programmation, vérifie les conventions de style et analyse les performances potentielles du code.

— **FindBugs (et son successeur SpotBugs)** : Détecte les bugs courants dans les projets Java, en mettant en évidence des problèmes tels que des accès concurrentiels ou des erreurs de manipulation de mémoire.

— **Bandit** : Un outil spécifiquement conçu pour analyser les projets Python à la recherche de vulnérabilités de sécurité, en se concentrant sur les erreurs courantes dans les bibliothèques ou les fonctions critiques.

Ces ressources permettent une analyse approfondie pour identifier les vulnérabilités et améliorer les pratiques de développement.

1.2 TRAVAUX ANTÉRIEURS ET MÉTHODOLOGIES EXISTANTES

La sécurité des logiciels revêt une importance capitale à l'ère du numérique, car les failles peuvent mettre en péril des systèmes essentiels et divulguer des informations confidentielles. De manière traditionnelle, ils étaient détectés par des inspections manuelles et des outils d'analyse statique, mais ces méthodes sont limitées par la complexité grandissante des logiciels et des attaques. Le développement de l'apprentissage automatique (ML) a bouleversé ce secteur en proposant des outils qui permettent de repérer des motifs subtils et des corrélations qui étaient difficilement repérables manuellement à partir de données volumineuses et diversifiées. Progressivement, ces modèles ont été utilisés pour prédire et détecter les vulnérabilités logicielles, même s'ils restent confrontés à des défis techniques et méthodologiques. Dans cette partie, nous passerons en revue les travaux antérieurs et les méthodologies existantes, en

examinant les approches clés, les limitations actuelles et les perspectives qu'elles offrent pour des recherches futures.

1.2.1 PRÉDICTION DES BUGS DANS LES LOGICIELS

La capacité de repérage des bugs logiciels reste un enjeu extrêmement important dans le cycle de développement et de maintenance des logiciels, car elle impacte significativement le bon fonctionnement global du système logiciel. En effet, détecter les bugs dès les premières étapes du processus de développement est crucial, car cela améliore la qualité et la fiabilité des systèmes logiciels tout en réduisant les coûts, le temps et les efforts liés au développement Hammouri *et al.* (2018). C'est dans cette mouvance que différentes techniques de prédiction des bugs ont été proposées et utilisées, parmi lesquelles l'utilisation de métriques logicielles Couto *et al.* (2012); Punitha & Chitra (2013), la spécification des attributs de qualité Sharma *et al.* (2013); Schugerl *et al.* (2008), ainsi que l'exploitation des modèles d'apprentissage automatique et de l'apprentissage profond Clemente *et al.* (2018); Hammouri *et al.* (2018).

UTILISATION DE MÉTRIQUES LOGICIELLES

Les métriques logicielles jouent un rôle central dans l'évaluation de la qualité des systèmes logiciels et la prédiction des défauts, notamment dans le contexte des projets orientés objet et open source. Plusieurs études ont exploré la corrélation entre ces métriques et les bugs pour concevoir des modèles prédictifs fiables.

Misra & Bhavsar (2003) ont examiné 30 systèmes open source écrits en C++ et ont utilisé la densité de bugs comme métrique, calculée par le nombre de bugs divisé par le nombre de lignes de code. Ils ont démontré que les 15 métriques considérées dans l'étude ont une forte corrélation avec la densité de bug. Ces métriques incluent : la taille moyenne

des classes, la taille moyenne des méthodes, la profondeur moyenne des chemins, la densité de contrôle, la profondeur de l'arbre d'héritage, le facteur de masquage des méthodes, le vocabulaire du programme, la longueur du programme, le nombre de classes, le nombre de méthodes, le facteur de polymorphisme, le pourcentage des membres publics/protégés, la réponse des classes, le nombre de lignes de code source, et le poids des méthodes dans les classes.

De leur côté, El Emam *et al.* (2001) ont mis en relief l'importance de la détection précoce des composants défectueux dans les logiciels. Dans ce contexte, ils se sont concentrés sur les systèmes orientés objet où les modèles de prédiction utilisant des métriques de conception peuvent être employés pour identifier les classes défectueuses dès les premières étapes du cycle de développement. Ils ont proposé la conception de modèles de prédiction basés sur des métriques de conception orientées objet. Leur modèle proposé permet de prédire quelles classes, dans une future version d'une application Java commerciale, seront susceptibles d'être défectueuses. Leur modèle est également capable de fournir une estimation globale de la qualité, c'est-à-dire combien de classes de la future version risquent d'être défectueuses. Ils ont utilisé les données collectées à partir d'une version d'une application Java commerciale pour construire un modèle de prédiction, qu'ils ont ensuite validé sur une version ultérieure de la même application. Leurs résultats montrent une augmentation significative de la précision du modèle de prédiction. Ils ont également démontré que les métriques d'héritage et de couplage externe avaient la plus forte corrélation avec la propension aux défauts. En outre, ils ont prouvé que le modèle de prédiction construit avec ces deux métriques offrait une meilleure précision et que la méthode employée pour prédire la qualité d'un système futur à l'aide de métriques de conception présentait également une bonne précision.

OPTIMISATION ET ÉQUILIBRAGE

L'optimisation des paramètres et l'équilibrage des classes sont deux techniques clés pour améliorer les performances et la fiabilité des modèles de prédiction de bugs.

Tantithamthavorn *et al.* (2018b) ont examiné des aspects tels que l'amélioration des performances, la stabilité des performances, l'interprétation des modèles, la *transférabilité* des paramètres, le coût computationnel et le classement des techniques de classification des modèles de prédiction de bugs pour les techniques d'optimisation automatique des paramètres. Ils cherchaient avant tout à saisir l'effet de l'optimisation automatique des paramètres sur les modèles de prédiction de bugs. Les méthodes d'amélioration automatisée des paramètres ont examiné les différentes configurations envisageables et suggéré les réglages optimisés afin d'obtenir les meilleures performances. 26 méthodes de classification couramment employées ont été évaluées en utilisant 12 mesures de performance, dont 3 sont indépendantes du seuil (comme l'AUC) et 9 sont dépendantes du seuil (comme les mesures de précision et de rappel).À l'aide d'une étude de cas sur 18 ensembles de données, ils ont constaté que l'optimisation automatique des paramètres améliore la performance AUC jusqu'à 40%, fournit des classificateurs aussi stables que ceux formés avec des paramètres par défaut, modifie de manière significative le classement de l'importance des variables, propose des réglages optimisés pour 17 des 20 paramètres les plus sensibles transférables entre différents ensembles de données sans baisse significative des performances, et n'ajoute que 30 minutes de calcul supplémentaires pour 12 des 26 techniques de classification étudiées.

Dans une perspective complémentaire, Tantithamthavorn *et al.* (2018a) ont étudié l'impact de cinq techniques de rééquilibrage de classes largement utilisées, à savoir le suréchantillonnage, le sous-échantillonnage, le SMOTE par défaut, le SMOTE optimisé et la technique

ROSE, sur la performance et l'interprétation des modèles de prédiction de bugs. Ils ont entraîné leurs modèles de prédiction de défauts en utilisant sept techniques de classification couramment employées : random forest (RF), régression logistique (LR), naive Bayes (NB), réseau de neurones (AVNNet), Boosting (C5.0), gradient boosting extrême (xGBTree) et méthode de gradient boosting (GBM).Ils ont examiné les situations expérimentales où les méthodes de rééquilibrage de classes sont bénéfiques et susceptibles d'être utilisées dans les modèles de bugs. La performance des modèles a été évaluée à l'aide de 10 mesures de performance couramment utilisées, dont 3 indépendantes du seuil (telles que l'AUC) et 7 dépendantes du seuil (telles que la précision, le rappel et la F-mesure). Pour analyser plus en profondeur l'impact des techniques de rééquilibrage de classes sur les modèles de prédiction de défauts, les auteurs ont construit des modèles statistiques afin d'étudier la relation entre les facteurs expérimentaux (tels que les ratios de bugs, les techniques de classification) et la performance ainsi que l'interprétation des modèles. Une étude de cas basée sur 101 ensembles de données, issus de systèmes propriétaires et open source collectés à partir de cinq corpus différents, a été menée. Les résultats ont montré que l'impact des techniques de rééquilibrage dépend de la mesure de performance utilisée et de la technique de classification employée.Les auteurs ont constaté que la méthode SMOTE optimisée et la méthode de sous-échantillonnage présentent des avantages lorsque les équipes d'assurance qualité souhaitent accroître l'AUC et le rappel, respectivement. Toutefois, il convient de ne pas utiliser ces méthodes lorsqu'il s'agit d'extraire des connaissances et de comprendre les modèles de bugs.

Les métriques logicielles utilisées dans la prédiction des vulnérabilités sont généralement utilisées dans des contextes spécifiques, tels que certains langages de programmation ou types de projets, ce qui restreint leur portée générale.

Jiang *et al.* (2008) ont comparé la performance des modèles de prédiction utilisant des métriques au niveau de la conception avec ceux utilisant des métriques au niveau du code, ainsi que ceux qui combinent les deux types de métriques. Ils ont analysé 13 bases de données du programme NASA Metrics Data Program (MDP), chacun contenant trois groupes de métriques : conception, code et l'ensemble complet. À l'aide de diverses techniques de modélisation et de tests statistiques, ils ont confirmé que les modèles construits à partir de métriques de code surpassent généralement ceux basés sur des métriques de conception. À cet effet, ils ont utilisé cinq algorithmes d'apprentissage automatique (i.e. learner, random forest, bagging, régression logistique, boosting et naive Bayes) issus de l'outil *Weka*, pour modéliser la propension aux défauts. Cependant, les deux types de modèles s'avèrent utiles, car ils peuvent être construits à différentes phases du projet. Pour examiner si la performance des trois groupes de métriques sur chaque ensemble de données montrait des différences statistiquement significatives, ils ont appliqué des tests statistiques non paramétriques. Les modèles basés sur le code peuvent être utilisés pour améliorer la performance des modèles basés sur la conception, augmentant ainsi l'efficacité des activités de vérification et de validation tard dans le cycle de développement. Ils ont également constaté que les modèles combinant les métriques de conception et de code surpassent ceux utilisant seulement un des deux types de métriques. Un autre constat intéressant est que la performance des modèles de prédiction varie davantage en fonction des groupes de métriques utilisés qu'en fonction des différents algorithmes de modélisation (apprentissage automatique).

De leur côté, Rahman & Devanbu (2013) ont analysé l'applicabilité et l'efficacité des métriques logicielles (de processus et de code) selon plusieurs perspectives différentes.

Sur 85 versions de 12 grands projets open-source, ils ont développé différents modèles de prédiction afin d'évaluer divers paramètres du logiciel, tels que la performance, la portabilité, la stabilité et la stagnation de divers ensembles de métriques. Les modèles élaborés à partir de diverses métriques ont été comparés en utilisant à la fois des évaluations sensibles aux coûts et basées sur l'AUC pour différents objectifs de performance, de stabilité et de portabilité. L'objectif était de déterminer à quel moment un ensemble de métriques peut être pertinent pour une entreprise.Selon leurs résultats, il a été démontré que les métriques de code, bien que fréquemment employées dans la littérature sur la prédiction de bugs, sont généralement moins stables et moins flexibles que les métriques de processus pour la prédiction. Ils ont aussi observé une forte stagnation des métriques de code, c'est-à-dire qu'elles ne fluctuent pas beaucoup d'une version à l'autre. Cela conduit à un arrêt des modèles de prédiction, où les mêmes fichiers sont constamment signalés comme défectueux. Selon les auteurs, la cause de la résistance des métriques de code aux changements réside dans le fait que les métriques de code traditionnelles sont moins sensibles aux activités de développement.

UTILISATION DE L'APPRENTISSAGE AUTOMATIQUE ET DE L'APPRENTISSAGE PROFOND

Les modèles d'apprentissage supervisé ont prouvé leur efficacité pour prédire l'apparition de bugs logiciels. Par exemple, Immaculate *et al.* (2019) ont utilisé des modèles d'apprentissage supervisé (à savoir la régression logistique, le Naïve Bayes, la forêt aléatoire et l'arbre de décision) pour prédire l'apparition des bugs en se basant sur des données historiques. Leurs résultats ont démontré que la forêt aléatoire (Random Forest) a atteint une précision de 97%, surpassant ainsi tous les autres modèles d'apprentissage automatique.

D'autres chercheurs ont exploré des approches combinées, intégrant des techniques d'apprentissage automatique et profond. Jayanthi & Florence (2019), par exemple, ont présenté une approche combinée pour la prédiction des bugs logiciels en utilisant le principe des techniques de sélection de caractéristiques et un modèle de réseau de neurones. Leurs résultats ont montré que la méthode proposée a atteint une précision de 97,20 %, et ils en ont déduit qu'en utilisant des techniques de sélection de caractéristiques, la complexité en temps et en espace pour la prédiction des défauts est réduite sans affecter la précision de la prédiction.

Dans une perspective similaire, Li *et al.* (2017) ont développé un réseau de neurones convolutifs en apprentissage profond (DPCNN) afin de prédire les bugs logiciels. Les données sémantiques correspondent aux informations propres au domaine d'un système logiciel, tandis que les informations structurelles correspondent à la structure syntaxique réelle du programme, ainsi qu'au flux de contrôle et de données qu'il représente (i.e., l'organisation des fichiers du système logiciel) Maletic & Marcus (2001). Le modèle est composé de vingt métriques logicielles utilisées par les auteurs. Ils ont prouvé que le DPCNN pourrait être employé en collaboration avec d'autres modèles d'apprentissage automatique afin d'améliorer les prédictions.

Les approches hybrides continuent de montrer leur potentiel. Manjula & Florence (2019) ont proposé une méthode hybride qui combine des algorithmes génétiques (GA) pour améliorer les caractéristiques avec un réseau de neurones profonds (DNN) pour la classification. Les algorithmes génétiques étaient améliorés dans leur technique, avec une nouvelle méthode de conception de chromosomes et de calcul de la fonction de fitness. Selon eux, leur méthode a obtenu des résultats supérieurs à ceux d'autres méthodes, avec un taux de précision de 97,82 %.

En parallèle, Ferenc *et al.* (2020) ont appliqué un réseau de neurones profonds à un large ensemble de données de bugs (contenant 8780 classes Java avec bugs et 38,838 sans bugs) et ont comparé ses performances à plusieurs algorithmes « traditionnels ». Ils ont montré que l'apprentissage profond avec des métriques de qualité peut effectivement améliorer la précision de prédiction avec une F-mesure de 53,59 %.

Les techniques hybrides ont également été explorées par Rhmann *et al.* (2020) Les chercheurs ont utilisé des modèles hybrides et de pointe en apprentissage automatique pour la prédiction des bugs en analysant les métriques de changement. Ils ont constaté que les techniques hybrides permettent de prédire les bugs logiciels de manière bien plus efficace que l'utilisation de chaque technique séparément.

Pour résoudre des défis spécifiques comme le déséquilibre des classes et le surapprentissage, Pandey *et al.* (2020) ont proposé une méthode de prédiction des bugs logiciels, appelée BPDET. Cette méthode repose sur deux étapes : l'apprentissage profond (auto-encodeur) et l'apprentissage par ensemble. Ils visent à résoudre deux problèmes : (1) le problème de déséquilibre des classes, c'est-à-dire que le nombre de modules défectueux dans le système est inférieur à celui des modules non défectueux, et (2) le problème de surapprentissage, c'est-à-dire qu'un modèle obtient de meilleurs résultats en formation qu'en test. Les écrivains ont examiné leur méthode à partir de douze ensembles de données publics de la NASA et ont remarqué qu'elle réussit à éviter ces problèmes.

De nombreux chercheurs se sont intéressés à la prédiction des bugs logiciels, proposant diverses solutions pour aider les développeurs à améliorer la qualité de leur code et à réduire au minimum l'apparition de bugs. Dans ce mémoire actuel, dans le but de soutenir les

développeurs, notre attention est portée sur l'analyse des vulnérabilités de sécurité et sur la façon de les anticiper. Les failles de sécurité peuvent entraîner des répercussions bien plus graves que les failles traditionnelles, car elles peuvent être utilisées par des attaquants pour mettre en péril la sécurité des systèmes. Il est donc primordial de créer des modèles qui permettent de les repérer dès les premières étapes du développement logiciel.
De manière concrète, en se référant à des études précédentes Baca *et al.* (2009) qui ont défini les différentes catégories de failles de sécurité dans les systèmes logiciels, nous nous focalisons sur cinq catégories de failles : les failles liées aux erreurs de mémoire, les failles liées aux exceptions de pointeur nul, les failles liées aux vérifications d'initialisation, les failles liées aux conditions de concurrence et les failles liées aux problèmes de contrôle d'accès. Il est essentiel de détecter rapidement ces types de failles pour renforcer la sécurité globale des applications logicielles, car elles constituent des vulnérabilités critiques dans de nombreux systèmes.

1.2.2 PRÉDICTION DES VULNÉRABILITÉS DANS LES LOGICIELS

La prédiction des vulnérabilités dans les logiciels est un domaine de recherche qui vise à identifier de manière proactive les failles potentielles dans le code, permettant ainsi de renforcer la sécurité dès les premières étapes de développement.Ce domaine utilise différentes méthodes, comme l'analyse de métriques logicielles, les modèles d'apprentissage automatique et les techniques statistiques, pour repérer les éléments qui pourraient entraîner des vulnérabilités de sécurité. Les recherches antérieures dans ce domaine ont examiné divers aspects de la problématique, soulignant les difficultés et les possibilités que présente l'anticipation des vulnérabilités de sécurité.

Camilo *et al.* (2015) ont examiné la manière dont les bugs de sécurité sont classés par rapport aux bugs classiques. Leur recherche s'est focalisée sur l'analyse des liens entre les erreurs de sécurité et les failles de sécurité. Le système Chromium et cinq versions de Google

Chrome, à savoir les versions 5, 11, 19, 27 et 35, ont été examinés de manière empirique. Grâce au système de suivi des erreurs de Google Chrome, ils ont identifié 374 686 erreurs qui ont été détectées au cours de six années de développement. Les chercheurs ont établi que les vulnérabilités et les bugs forment deux catégories distinctes, car ils ont constaté une faible corrélation entre les fichiers qui ont été identifiés comme contenant des bugs par le passé et les bugs postérieurs à la sortie. Dans notre recherche actuelle, nous étudions de manière empirique la corrélation entre les métriques logicielles et les failles de sécurité, et nous étudions si les modèles d'apprentissage automatique et les tests statistiques peuvent expliquer cette corrélation.

Par ailleurs, selon Zaman *et al.* (2011), les problèmes de sécurité sont considérés comme les plus importants dans un système logiciel et sont détectés et résolus beaucoup plus rapidement que tout autre problème. De plus, ils ont observé que les vulnérabilités de sécurité touchent plus de développeurs et ont un impact sur un plus grand nombre de fichiers dans un système logiciel. Sur cette base, dans ce mémoire, nous nous concentrons sur la prédiction des bugs de sécurité à l'aide de métriques de qualité, où nous mettons en évidence les valeurs-seuil critiques que les développeurs doivent prendre en compte.

En échantillonnant 2 060 bugs réels dans trois systèmes majeurs, à savoir le noyau Linux, Mozilla et Apache, Tan *et al.* (2014) ont examiné les caractéristiques des bugs logiciels. Selon leurs résultats, le système d'exploitation du noyau Linux présente davantage de vulnérabilités sémantiques que les autres systèmes analysés. Sur cette base, les auteurs ont suggéré de fournir davantage de soutien pour aider les développeurs à diagnostiquer et corriger les bugs de sécurité, en particulier les bugs de sécurité sémantiques, c'est-à-dire les bugs qui ne provoquent pas l'arrêt de l'exécution du programme, mais qui affectent le comportement du

programme par rapport aux attentes du programmeur.

De leur côté, Alves *et al.* (2016) ont examiné la relation entre les métriques logicielles et les bugs de sécurité. Leur ensemble de données provenait de cinq systèmes logiciels open-source : Mozilla, httpd, glibc, le noyau Linux et Xen HV. Ils ont extrait des rapports de sécurité provenant des avis de sécurité de la Fondation Mozilla (MFSA), des avis de sécurité Xen et des expositions communes aux vulnérabilités (CVE) pour constituer une partie des données sur les vulnérabilités. Leurs résultats ont montré que les métriques logicielles peuvent discriminer les fonctions défectueuses, mais qu'il est difficile de trouver une forte corrélation entre ces métriques et le nombre de bugs de sécurité existant dans les fonctions analysées.

Dans notre recherche, nous faisons appel à des modèles d'apprentissage automatique de pointe et à des tests statistiques afin de revoir cette relation et de repérer les indicateurs de qualité qui pourraient avoir le plus d'impact sur la hausse des bugs de sécurité. Comme les systèmes logiciels sont très vulnérables aux failles de sécurité, nous cherchons non seulement à les repérer aux dernières étapes du cycle de vie d'un logiciel, mais aussi à les prédire le plus tôt possible pour les éliminer dès les premières étapes de développement.

Clemente *et al.* (2018), à leurs tours, ont étudié plusieurs tailles et métriques de complexité pour trois applications logicielles open-source développées par Mozilla, à savoir SeaMonkey, Firefox et Thunderbird. Ces trois applications ont été employées par les auteurs afin de prédire les failles. Selon leurs résultats, ils ont confirmé que les métriques logicielles peuvent servir d'indicateurs de bugs de sécurité. De plus, ils ont conclu que l'apprentissage profond peut entraîner des prédictions plus précises par rapport aux algorithmes d'apprentis-

sage automatique traditionnels, tels que les arbres de décision, les forêts aléatoires, les SVM et Naïve Bayes.

Dans ce mémoire, nous approfondissons cette étude précédente pour anticiper les vulnérabilités de sécurité sur une gamme plus étendue de systèmes logiciels en utilisant plus de modèles d'apprentissage automatique. En outre, nous repérons toutes les mesures logicielles les plus liées aux vulnérabilités de sécurité et nous analysons leur impact sur l'émergence de ces vulnérabilités dans le but d'aider les développeurs à les prévenir.
Ainsi, notre étude actuelle se distingue de l'étude précédente par l'analyse d'un plus grand nombre de métriques de qualité, l'utilisation de techniques supplémentaires d'apprentissage automatique et la présentation de nouveaux résultats concernant l'impact de certaines métriques de qualité sur la prédiction des bugs de sécurité.

Une perspective complémentaire a été apportée par Ouedraogo *et al.* (2013) qui ont proposé une nouvelle taxonomie des métriques de qualité nécessaires pour obtenir une assurance dans un processus de vérification de la sécurité. Dans le même esprit que le modèle de maturité des capacités utilisé en ingénierie des systèmes, ils ont ajouté cinq niveaux de qualité standard pour un processus de vérification qui se concentre sur l'évaluation de la correction des mécanismes de sécurité en direct. En outre, ils ont examiné la corrélation entre les niveaux de qualité et les divers niveaux de capacité des familles de mesures de vérification suivantes : couverture, rigueur, profondeur et indépendance de la vérification. Leur classification de qualité est incluse dans un cadre visant à garantir la sécurité des systèmes logiciels. Les métriques proposées peuvent également être utilisées pour obtenir une assurance dans d'autres domaines tels que la conformité légale et la sécurité. Par ailleurs, en identifiant des exigences de sécurité de qualité appropriées, la taxonomie des métriques proposée pourrait aider les

fabricants informatiques dans le développement de leurs produits ou systèmes.

Dans le contexte de l'analyse des métriques de complexité, Shin & Williams (2008) ont réalisé une analyse statistique sur neuf métriques de complexité du moteur JavaScript. L'objectif était de repérer les disparités entre les métriques de code vulnérables et non vulnérables et de prédire les vulnérabilités. Pour leur étude de cas, ils ont opté pour le moteur JavaScript de Mozilla (JSE), car le code source, les bugs et les informations sur les vulnérabilités sont accessibles à tous, et le nombre de défauts et de vulnérabilités signalés était adéquat pour une analyse initiale. D'après leurs premiers résultats, il a été démontré que les mesures de complexité pouvaient anticiper les vulnérabilités avec un taux de faux positifs faible, mais un taux de faux négatifs élevé. Leurs résultats ont également indiqué que la complexité de l'imbrication pourrait être un facteur de différenciation entre les fonctions vulnérables et les fonctions défectueuses dans le JSE. Par conséquent, accorder plus d'attention aux fonctions fortement imbriquées que les autres lors des inspections de sécurité pourrait être une stratégie efficace.

Ganesh *et al.* (2022) ont mené une étude dont l'objectif est de repérer les failles de sécurité dans le code source en utilisant des modèles d'apprentissage automatique. Elle élabore divers algorithmes de classification en utilisant des métriques statiques provenant de code source Java et d'un ensemble de données provenant de projets open-source tels qu'Apache Struts, tels que Naive Bayes, la régression logistique, les arbres de décision et XGBoost. Le modèle XGBoost a démontré sa performance en matière de prédiction des vulnérabilités de sécurité, dépassant ainsi les autres modèles testés. La méthode utilisée englobe la collecte de données, la purification et l'échantillonnage des classes, la sélection des caractéristiques pertinentes en utilisant des techniques telles que l'élimination de caractéristiques récursives,

ainsi que l'entraînement des modèles. L'étude examine aussi les difficultés liées à la prédiction entre versions et projets, mettant en évidence l'importance de standardiser les données afin d'améliorer les performances prédictives. Bien que le modèle XGBoost ait été efficace, les méthodes d'apprentissage ensemblistes n'ont pas dépassé les modèles individuels, et l'étude met en évidence l'importance d'améliorer ces approches pour de futures études.

Hemmati (2024) traite de l'anticipation des bugs logiciels en mettant l'accent sur la différenciation entre les diverses gravités des bugs, un élément essentiel pour donner la priorité et traiter efficacement les bugs dans le domaine de la maintenance technologique. En utilisant les fichiers Defects4J et Bugs *.jar*, ainsi que dix métriques de code source, l'étude évalue à la fois la présence et la gravité des erreurs. Malgré la capacité de certaines métriques à prédire la présence de code défectueux, elles ne peuvent pas évaluer avec précision la gravité des imperfections. La détection des bugs et l'évaluation de leur gravité sont également insuffisantes pour les outils d'analyse statique tels que SpotBugs et Infer, avec des taux de précision et de rappel faibles. Selon ces études, il est constaté que les outils actuels classent souvent ou ignorent les bugs graves, ce qui entraîne des problèmes majeurs dans la maintenance des logiciels. L'article met en évidence l'importance de recherches ultérieures afin d'améliorer les méthodes de prédiction des erreurs, en particulier en intégrant des aspects liés au type d'erreur pour mieux comprendre la complexité des erreurs logicielles. Il suggère l'utilisation d'approches plus précises afin de saisir la corrélation entre les caractéristiques du code et la gravité des erreurs, dans le but d'améliorer la détection et la gestion des erreurs.

Dans leur étude sur les bugs de sécurité, Wei *et al.* (2021) ont élaboré des critères de classification pour la catégorie des bugs de sécurité, tels que la cause initiale, les conséquences et l'emplacement. En outre, ils ont choisi 1076 rapports de bugs pour l'enquête provenant

de cinq projets (Apache Tomcat, Apache HTTP Server, Mozilla Firefox, Linux Kernel et Eclipse). Les résultats de classification ont été analysés par les auteurs et ont démontré que les opérations sur la mémoire sont le type de bugs de sécurité le plus fréquent. De plus, ils ont rapporté que les principales causes racines des bugs de sécurité sont CON (Erreur de configuration), INP (Erreur de validation des entrées) et MEM (Erreur de mémoire). Par ailleurs, les bugs de sécurité causés par LOG (erreur de logique) entraînent généralement des attaques par déni de service (DoS).

En se basant sur des connaissances dans le domaine de la sécurité logicielle, Zheng *et al.* (2022) ont suggéré d'améliorer l'efficacité de la prédiction des rapports de bugs de sécurité (SBR) en utilisant l'apprentissage machine supervisé. Ils ont employé un ensemble de règles pour reconnaître les entités et établir des liens entre elles, créant ainsi des graphes de connaissances. Leur corpus a été enrichi grâce aux données du CWE (Common Weakness Enumeration), et des mots et des passes liés à la sécurité ont été ajoutés. Enfin, ils ont démontré comment prédire les SBR à partir du projet cible en calculant la similarité cosinus entre le corpus intégré et les rapports de bugs cibles. Les auteurs ont présenté une évaluation expérimentale sur cinq ensembles de données SBR open-source, montrant que l'approche guidée par les connaissances du domaine pouvait améliorer l'efficacité des SBR.

Fehrer *et al.* (2024) ont mené une étude visant à automatiser l'analyse des dépôts de code en utilisant des analyseurs statiques et des techniques d'apprentissage automatique pour prédire si un commit corrige une vulnérabilité. Dans un premier temps, ils ont développé un ensemble de données de 1 821 commits provenant de projets open source, en mettant l'accent sur des commits ayant une pertinence industrielle. Une fois les données prétraitées avec soin, ils ont extrait des caractéristiques des fichiers modifiés par les commits en utilisant

différents outils tels que PMD, Checkstyle, CK et Progex. Des métriques logicielles, des listes de bugs et des graphes de flux de contrôle ont été générés grâce à ces outils, ce qui a enrichi les représentations des fichiers et des commits. Par la suite, l'équipe a instauré un processus d'apprentissage automatique, incluant l'utilisation de sept algorithmes d'apprentissage et l'emploi de techniques globales afin d'améliorer les résultats. Les performances de la combinaison de modèles par empilement et vote ont été plus élevées, avec une précision moyenne de 77,5%. Cependant, ils ont mis en évidence des contraintes, telles que le manque de clarté dans la classification des commits et le risque de biais dans leur ensemble de données.

En résumé, les apports des études antérieures constituent une fondation solide pour notre exploration. Notre objectif est d'atteindre un niveau supérieur en incorporant des méthodes avancées et en étendant le domaine d'analyse, afin de proposer des solutions solides pour prédire de manière proactive les vulnérabilités logicielles.

1.3 LIMITES DES RECHERCHES EXISTANTES

Les avancées réalisées dans le domaine de la prédiction des bugs de sécurité en utilisant des modèles d'apprentissage automatique ont été importantes, mais elles comportent certaines limites qui permettent d'apporter des contributions supplémentaires. En premier lieu, de nombreuses études se focalisent exclusivement sur l'évaluation des performances des modèles sans analyser en profondeur les métriques de qualité du code qui pourraient avoir un impact sur l'apparition de ces bugs. De plus, la plupart de ces études sont spécifiques à un langage de programmation, ce qui limite leur applicabilité aux projets utilisant d'autres langages et réduit la portée de leurs conclusions. Enfin, il existe peu de recherches qui abordent de manière approfondie les tendances et les corrélations qui peuvent orienter les développeurs vers l'adoption de meilleures pratiques de développement sécurisé.

1.4 POSITIONNEMENT DE NOTRE ÉTUDE

Dans ce mémoire, nous présentons une évaluation empirique des performances des modèles d'apprentissage automatique pour la prédiction des bugs de sécurité. De plus, nous explorons l'existence de seuils critiques de métriques de qualité qui pourraient influencer l'introduction de bugs de sécurité. Notre étude empirique est menée sur sept systèmes open-source et repose sur une analyse approfondie des données afin d'aider les développeurs à mieux comprendre les caractéristiques des bugs de sécurité.

Les principales contributions de cette étude sont les suivantes :

— **Identification des métriques de qualité les plus corrélées aux bugs de sécurité** grâce à des techniques d'apprentissage automatique. Cette analyse permet de mieux comprendre les facteurs qui influencent l'apparition de vulnérabilités dans les logiciels.
— **Détermination de seuils critiques pour certaines métriques de qualité**, pouvant servir d'indicateurs d'alerte aux développeurs afin d'anticiper les risques liés aux bugs de sécurité.
— **Démonstration empirique de l'efficacité des modèles d'apprentissage automatique** pour la prédiction des bugs de sécurité, en mettant en avant l'importance du choix d'une stratégie de validation adaptée.

Enfin, notre étude couvre plusieurs langages de programmation, ce qui permet d'avoir une vision plus généralisable des problèmes liés aux bugs de sécurité. Cette approche contribue à l'amélioration des pratiques de développement sécurisé en proposant des recommandations basées sur des analyses empiriques.

CHAPITRE II

PROCESSUS DE COLLECTE DES DONNÉES

2.1 PRÉSENTATION DES OUTILS

2.1.1 PLATEFORME OSV (OPEN SOURCE VULNERABILITY)

QU'EST-CE QU'OSV ?

OSV est une base de données publique de vulnérabilités qui se concentre exclusivement sur les projets open source. Son objectif est de fournir des renseignements précis et facilement accessibles concernant les vulnérabilités qui affectent les bibliothèques, les frameworks et les outils utilisés par les développeurs. Il vise à relier les vulnérabilités à des versions spécifiques du code tout en simplifiant la correction.

ÉCOSYSTÈMES PRIS EN CHARGE

OSV couvre une large gamme d'écosystèmes populaires du développement logiciel. Parmi les principaux écosystèmes couverts, on retrouve **npm**, **Pipy**, **Maven**, **Packagist** etc. Les développeurs qui utilisent différents langages de programmation peuvent bénéficier de ce support multi-écosystèmes pour obtenir des informations à jour sur les vulnérabilités qui impactent leurs logiciels.

PRINCIPALES FONCTIONNALITÉS DE LA PLATEFORME OSV

— **Accès direct aux informations sur les vulnérabilités :** OSV propose une interface conviviale et facile à utiliser qui permet aux utilisateurs de trouver des informations précises sur des vulnérabilités déjà identifiées. Des informations telles que la description de la vulnérabilité, les versions impactées et les versions corrigées des packages sont incluses dans les résultats.

— **Commits de correction associés :** La particularité d'OSV par rapport aux autres bases de données de vulnérabilités réside dans la faculté d'accéder directement aux commits de correction liés à une vulnérabilité spécifique. Cela donne aux développeurs la possibilité de saisir précisément les modifications effectuées pour résoudre le problème.

— **Accès aux versions corrigées du code :** Les versions spécifiques du code source ou des packages où la vulnérabilité a été corrigée sont facilement accessibles avec OSV. Cela diminue la durée requise pour repérer et mettre en œuvre les corrections dans les projets en cours.

COMMENT FONCTIONNE OSV ?

OSV recueille ses informations à partir de diverses sources :

— Rapports de sécurité des mainteneurs de projets open-source.

— Bases de données de sécurité d'autres écosystèmes.

— Commit logs publics qui incluent des informations sur les correctifs de sécurité.

Après l'agrégation des informations, OSV les actualise en fonction des commits et des nouvelles versions publiées, assurant ainsi l'accès des développeurs aux données les plus récentes concernant les vulnérabilités.

CONTRIBUTIONS DE LA COMMUNAUTÉ

Les contributeurs ont la possibilité de soumettre des vulnérabilités ou des corrections à OSV, qui est un projet ouvert. Grâce à cette approche communautaire, OSV peut maintenir sa dynamique et sa mise à jour, en soumettant directement des rapports aux mainteneurs de projets open-source et aux chercheurs en sécurité.

IMPORTANCE POUR LA SÉCURITÉ OPEN-SOURCE

OSV joue un rôle crucial dans l'écosystème open-source en permettant :

— Une transparence sur les vulnérabilités : En rendant accessibles à tous des informations publiques, OSV contribue à renforcer la sécurité des projets open-source.

— Des mises à jour plus rapides et efficaces : Les équipes de développement ont la possibilité d'appliquer des mises à jour de manière plus précise grâce à l'accès direct aux correctifs.

— Un suivi automatisé des vulnérabilités dans des projets complexes avec de nombreuses dépendances, ce qui diminue le risque de laisser une vulnérabilité non corrigée.

Dans le cadre de ce mémoire, nous avons choisi la plateforme OSV, car elle permet d'accéder directement aux informations sur les vulnérabilités, aux commits de correction et aux versions corrigées. OSV simplifie non seulement la gestion proactive des vulnérabilités, mais renforce également la sécurité des projets open-source à travers différents écosystèmes.

2.1.2 GITHUB

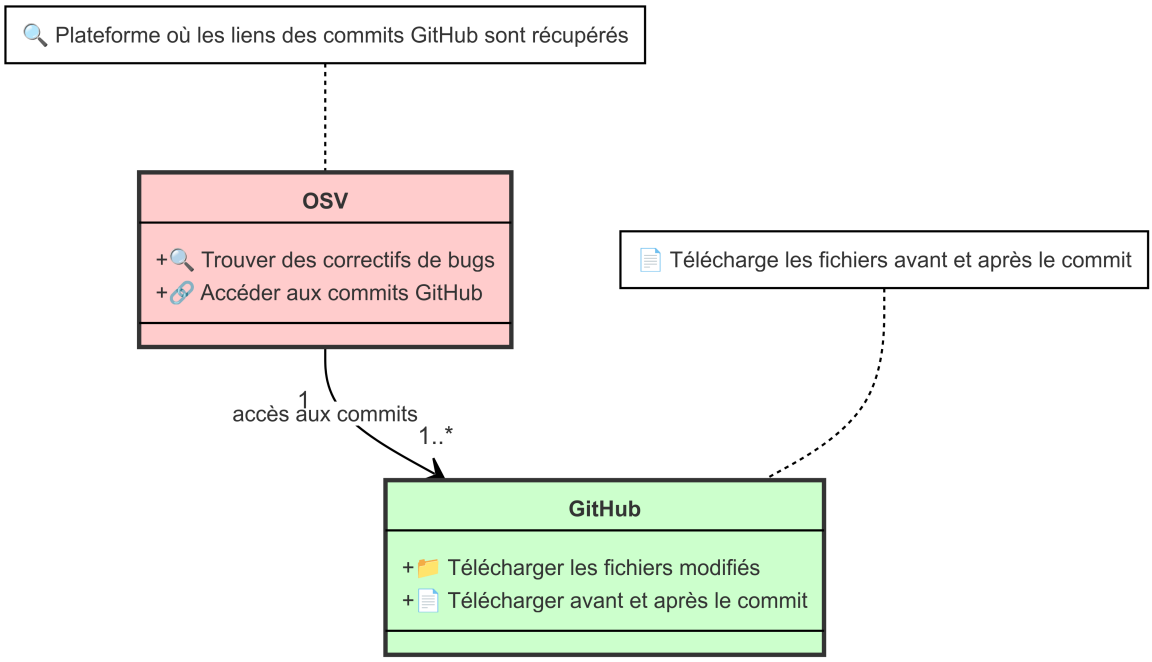

FIGURE 2.1 : Téléchargements de fichiers avant/après correction

Les fichiers sources modifiés lors des commits de correction de vulnérabilité peuvent être consultés sur GitHub. En utilisant les données de vulnérabilité fournies par OSV, il est possible d'utiliser GitHub afin de repérer les fichiers touchés et de télécharger les versions préalables et postérieures à la correction.

2.1.3 OUTILS ET BIBLIOTHÈQUES SUPPLÉMENTAIRES

Pour automatiser la collecte des données, nous avons utilisé divers outils afin d'interagir avec les dépôts GitHub et d'analyser les changements effectués aux fichiers entre les versions précédentes et postérieures à un commit. À titre d'exemple, `GitPython` a donné la possibilité

de collecter des données détaillées sur les commits, telles que les métadonnées (dates, auteurs, messages de commit) et les fichiers mis à jour. Par la suite, on a utilisé l'outil `unidiff` afin de comparer les variations entre les versions de fichiers avant et après les modifications effectuées par chaque commit.

2.2 PRÉSENTATION DES ÉCOSYSTÈMES

Les commits collectés pour cette étude proviennent de différents environnements logiciels spécifiques, où des problèmes de sécurité ont été signalés et réglés. Dans ce contexte, un écosystème fait référence à un environnement technologique spécifique où des bibliothèques, des packages ou des logiciels sont développés, gérés et diffusés. Des langages de programmation, des gestionnaires de packages ou des frameworks spécifiques peuvent être inclus dans ces écosystèmes afin de construire et de maintenir des applications. Notre étude se concentre sur les écosystèmes suivants, chacun d'eux étant sélectionné pour la fréquence ou la gravité des vulnérabilités détectées, ainsi que pour la disponibilité des données de sécurité associées. Ces environnements permettent d'analyser des commits liés à des correctifs de sécurité, en offrant une vue sur la manière dont les développeurs corrigent des failles critiques.

2.2.1 GHSA (GITHUB SECURITY ADVISORY)

GHSA est une plateforme gérée par GitHub, fournissant des conseils de sécurité pour les projets hébergés sur GitHub.Les avis de sécurité comprennent des renseignements concernant les vulnérabilités, les mesures correctives, ainsi que des recommandations pour les réduire.

2.2.2 PYPI (PYTHON PACKAGE INDEX)

PyPI est l'écosystème officiel des packages Python. Il fournit une large base de données de bibliothèques Python open-source. Les vulnérabilités dans PyPI concernent généralement les bibliothèques utilisées dans des projets Python et les commits de correction permettent de sécuriser ces bibliothèques.

2.2.3 OSS-FUZZ

OSS-Fuzz est une initiative de Google qui utilise des techniques de fuzzing pour détecter automatiquement des vulnérabilités dans les logiciels open-source. Les projets soutenus par OSS-Fuzz sont fréquemment cruciaux, et les correctifs de vulnérabilités dans cet écosystème sont essentiels pour la sécurité du code source.

2.2.4 NPM (NODE PACKAGE MANAGER)

L'écosystème le plus apprécié pour les paquets JavaScript est npm. Ces vulnérabilités sont principalement liées aux projets JavaScript et Node.js dans cet écosystème. Il est primordial de prendre des mesures de correction de vulnérabilités dans cet écosystème afin de garantir la sécurité des applications JavaScript.

2.2.5 PACKAGIST

Packagist est le principal dépôt des packages pour l'écosystème PHP. Les faiblesses présentes dans cet écosystème sont liées aux bibliothèques PHP employées dans des applications web et d'autres programmes. Les commits de correction touchent souvent des composants critiques dans des applications PHP.

2.2.6 MAVEN

Maven est un écosystème de packages pour les projets Java. Les bibliothèques Java utilisées dans des projets complexes tels que les systèmes d'entreprise ou les applications distribuées sont principalement touchées par ces vulnérabilités.Les correctifs apportés dans cet écosystème visent à améliorer la sécurité des bibliothèques Java.

2.2.7 NUGET

NuGet est l'écosystème des packages pour les projets .NET. Les vulnérabilités dans cet écosystème affectent des bibliothèques utilisées dans des applications basées sur .NET, que ce soit pour le développement web, des applications de bureau ou d'autres systèmes.

2.3 ÉTAPES DE LA COLLECTE DES DONNÉES

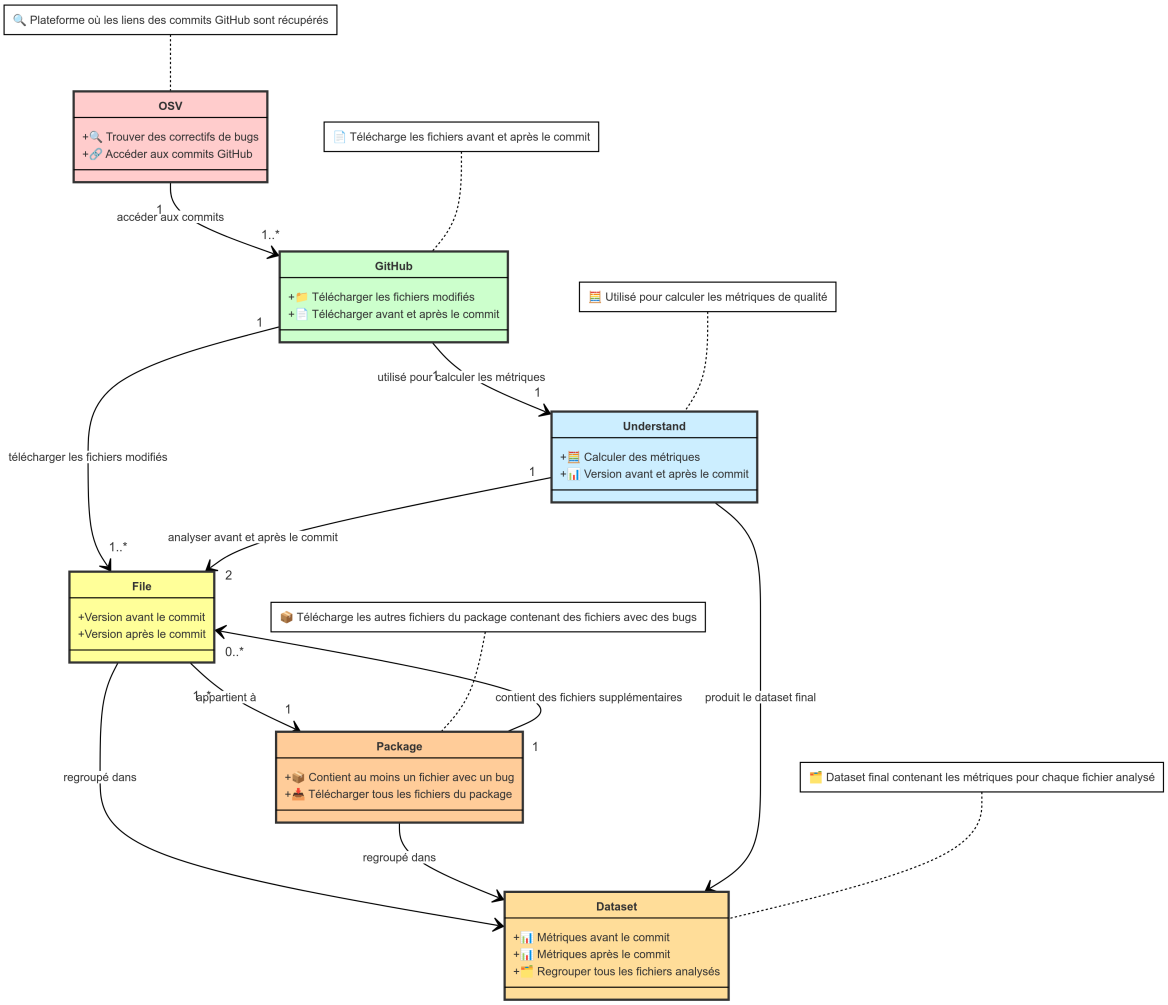

FIGURE 2.2 : Processus de collecte des données

2.3.1 IDENTIFICATION DES VULNÉRABILITÉS

Les vulnérabilités sont identifiées via la plateforme OSV en fonction de critères comme le type de vulnérabilité, la date de correction et l'écosystème concerné. Chaque vulnérabilité recensée est associée à un ou plusieurs commits de correction dans un dépôt GitHub. Afin

d'automatiser ce processus, nous utilisons un script Python qui communique avec l'API d'OSV pour détecter les vulnérabilités pertinentes pour divers écosystèmes. Grâce à ce script, nous pouvons trier les vulnérabilités selon différents paramètres, tels que la gravité, le package affecté ou la version restaurée. Par la suite, on extrait les commits de correction correspondants afin d'effectuer une analyse plus approfondie des correctifs appliqués au code source.

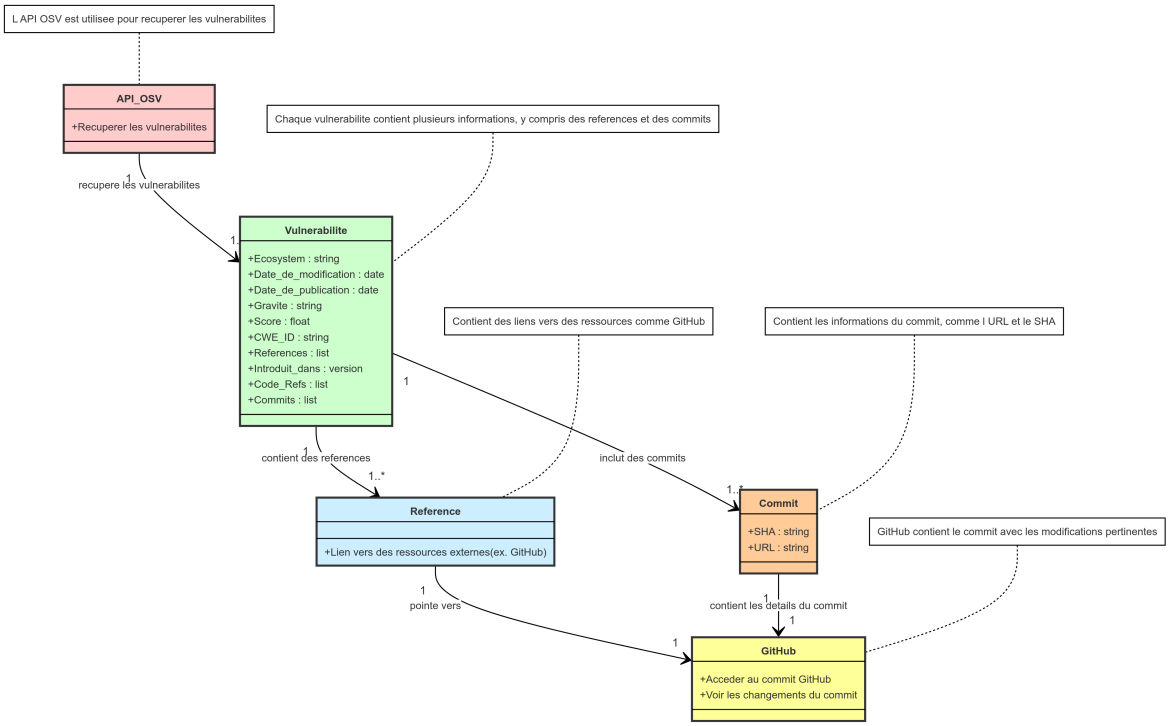

FIGURE 2.3 : Extraction des vulnérabilités via la plateforme OSV

2.3.2 EXTRACTION DES COMMITS DE CORRECTION

Après avoir repéré une vulnérabilité, le commit correspondant à sa correction est extrait de GitHub en utilisant son URL ou son identifiant (SHA). Chaque modification renferme

des données essentielles telles que les fichiers modifiés, le message de modification et les métadonnées liées (auteur, date).

2.3.3 ACCÈS AUX FICHIERS MODIFIÉS LORS DU COMMIT

À partir du commit de correction, les fichiers modifiés sont identifiés et téléchargés. Ce processus est automatisé à l'aide de la bibliothèque pygithub [5] qui nous donne la possibilité d'interagir avec l'API GitHub et d'extraire les chemins et contenus des fichiers avant et après modification.

2.3.4 ACCÈS AUX FICHIERS DES PACKAGES CONTENANT DES FICHIERS VULNÉRABLES

Outre le téléchargement des fichiers modifiés directement lors du commit, les fichiers dans les mêmes packages que les fichiers vulnérables sont également récupérés. Cela donne la possibilité de prendre en compte l'état global des packages touchés par la vulnérabilité, car les dépendances internes au sein du package peuvent influencer de manière indirecte la vulnérabilité corrigée.

2.4 TÉLÉCHARGEMENT DES VERSIONS AVANT ET APRÈS LE COMMIT

2.4.1 VERSION AVANT COMMIT

Chacun des fichiers modifiés et chaque fichier du package est récupéré avec la version avant l'application du correctif. Cela offre la possibilité d'examiner le code dans son état défectueux avant la correction.

5. https ://github.com/PyGithub

2.4.2 VERSION APRÈS COMMIT

Après application du commit de correction, la version corrigée des fichiers et des packages est téléchargée. Les fichiers incluent le patch appliqué pour corriger la vulnérabilité ainsi que les autres fichiers non modifiés, mais présents dans le même package.

2.4.3 GESTION DES DIFFÉRENCES ENTRE LES VERSIONS

En utilisant des bibliothèques telles que Unidiff, on peut comparer les versions avant et après commit, ce qui permet d'analyser les modifications apportées au niveau du code source et de comprendre la nature du correctif.

2.5 CALCUL DES MÉTRIQUES DE CODE AVEC L'OUTIL UNDERSTAND

2.5.1 PRÉSENTATION DE L'OUTIL UNDERSTAND

Understand [6] est un outil d'analyse statique de code conçu pour permettre aux développeurs et aux chercheurs de modifier, de refaçonner le code, de visualiser des graphes de dépendances et de collecter des métriques de qualité. Cet outil prend en charge de nombreux langages de programmation et offre une interface intuitive, ainsi qu'une gamme d'options avancées pour analyser la structure et la qualité du code. Understand est souvent utilisé pour évaluer la maintenabilité et la complexité du code, ce qui en fait un atout pour les équipes cherchant à améliorer la qualité de leurs projets logiciels.

6. https://support.scitools.com/support/solutions/articles/70000582223-what-metrics-does-understand-have-

2.5.2 PROCESSUS DE CALCUL DES MÉTRIQUES

Nous extrayons d'abord le code source de deux versions de chaque fichier défectueux avant et après la correction (commit), ainsi que deux versions de chaque fichier non défectueux (environ 340 000 fichiers). Ensuite, nous avons utilisé l'outil Understand pour générer une liste de métriques de qualité pour chacun des fichiers (c'est-à-dire, impliquant des bugs de sécurité ou non) de chaque écosystème étudié.

L'outil est largement utilisé dans la littérature pour extraire des métriques de qualité du code Barrak *et al.* (2018); Gupta *et al.* (2021). *Understand* propose différentes méthodes pour collecter des métriques de qualité. Nous avons utilisé l'outil à partir de la ligne de commande, ce qui permet de l'intégrer facilement dans des flux de travail automatisés. Dans notre cas, nous avons intégré *Understand* dans un script Python, configuré pour analyser automatiquement le code source et calculer les métriques de qualité pour chaque fichier. Le script parcourt les fichiers du projet, en extrait les informations pertinentes, puis génère un fichier de sortie consolidant les données issues de plusieurs fichiers CSV. Ce fichier inclut toutes les métriques fournies par *Understand*, offrant une vue d'ensemble complète des performances et de la qualité du code analysé.

2.6 MÉTRIQUES CALCULÉES

2.6.1 COMPLEXITÉ CYCLOMATIQUE

Ces métriques sont utilisées pour calculer la complexité cyclomatique du code source, en identifiant les différents chemins possibles (comme les boucles `if/else`).

— **SumCyclomaticModified** : Complexité cyclomatique totale des chemins modifiés.

— **SumCyclomatic** : Complexité cyclomatique totale.

— **SumCyclomaticStrict** : Complexité cyclomatique stricte.

— **MaxCyclomatic** : Complexité cyclomatique maximale.

— **AvgCyclomaticStrict** : Moyenne de la complexité cyclomatique stricte.

— **AvgCyclomaticModified** : Moyenne de la complexité cyclomatique pour les chemins modifiés.

— **AvgCyclomatic** : Moyenne générale de la complexité cyclomatique.

— **MaxCyclomaticModified** : Complexité cyclomatique maximale pour les chemins modifiés.

— **MaxEssential** : Complexité essentielle maximale.

2.6.2 VARIABLES, MÉTHODES ET CLASSES

Ces métriques évaluent le nombre d'unités de code (par exemple, méthodes, variables, classes, etc.) dans le projet.

— **CountDeclClass** : Nombre de déclarations de classes.

— **CountStmtDecl** : Nombre de déclarations d'instructions.

— **CountDeclFunction** : Nombre de déclarations de fonctions.

— **CountDeclMethod** : Nombre de déclarations de méthodes.

— **CountStmt** : Nombre total d'instructions.

— **MaxNesting** : Niveau maximal d'imbrication des blocs de code.

2.6.3 MÉTADONNÉES ADDITIONNELLES

Le jeu de données contient également les colonnes suivantes pour fournir des informations contextuelles sur les fichiers analysés :

— **Date de commit** : La date du commit associé au fichier.

— **SHA du commit** : L'identifiant SHA du commit.

— **Chemin du fichier** : Le chemin du fichier dans le dépôt.

— **Écosystème** : L'écosystème auquel le fichier appartient (par exemple, npm, maven, etc.).

— **Source** : Indique si le fichier provient de la version avant le commit (pré-commit) ou après le commit (post-commit).

2.7 STRUCTURATION ET STOCKAGE DES DONNÉES COLLECTÉES

2.7.1 ORGANISATION DES FICHIERS COLLECTÉS

Les fichiers collectés, ainsi que les fichiers des packages associés, sont structurés en fonction de l'écosystème, de la vulnérabilité et du commit. Chaque fichier est associé à des métadonnées comprenant la date du commit, le nom du fichier, le type de modification et l'auteur du commit.En outre, nous avons collecté des mesures de code pour chaque fichier en utilisant l'outil *Understand*, ce qui offre des indicateurs quantitatifs sur la qualité du code, tels que la complexité, la profondeur des tests et le nombre de lignes de code. En incorporant ces mesures, nous disposons d'une vision globale de l'état et de l'évolution du code. Grâce à cette organisation des informations, complétée par les métriques de qualité, nous sommes en mesure de réaliser une analyse approfondie et éclairée des fichiers au sein des divers écosystèmes.

2.7.2 PRÉPARATION DES DONNÉES POUR L'APPRENTISSAGE AUTOMATIQUE

Les fichiers sont ensuite formatés pour être utilisés comme données d'entrée pour les modèles d'apprentissage automatique. Des métriques de code, telles que la complexité cyclomatique, les lignes de code ou les fonctions modifiées, sont calculées pour chaque version avant et après commit. En outre, on intègre une variable buggy afin de déterminer si un fichier présente des problèmes de sécurité ou non, ce qui permet de mieux cibler l'apprentissage et d'évaluer la performance des modèles dans la détection des fichiers problématiques.

2.8 DÉFIS ET LIMITATIONS DE LA COLLECTE DES DONNÉES

2.8.1 LIMITATIONS DES DONNÉES EXTRAITES

Certaines données peuvent être incomplètes ou manquantes, soit du côté de la plateforme OSV, soit sur GitHub. De plus, il peut arriver que certains commits touchent plusieurs fichiers non directement liés à la vulnérabilité. L'ajout des fichiers de packages non directement modifiés peut également introduire des redondances ou des fichiers non pertinents.

2.8.2 CONTRAINTES TECHNIQUES

Les limitations de l'API GitHub, telles que le volume de données ou le taux de requêtes, nous ont également posé des défis lors de la collecte. De plus, la gestion des dépendances entre les packages peut rendre difficile l'accès aux versions spécifiques dans les écosystèmes comme `npm` ou `PyPI`.

CHAPITRE III
MÉTHODOLOGIE DE L'ÉTUDE EMPIRIQUE

3.1 INTRODUCTION

Il est essentiel de suivre un processus rigoureux pour mettre en place une approche de prédiction des bugs de sécurité, allant de la préparation des données à l'évaluation des modèles. Les étapes de la méthodologie employée pour rendre possible l'utilisation des algorithmes d'apprentissage automatique dans la prédiction des bugs de sécurité sont détaillées dans ce chapitre. Notre analyse est guidée par trois questions de recherche centrales :

— **Quelles sont les métriques de qualité avec la plus grande importance dans la prédiction des bugs de sécurité ?**

L'objectif de cette question est de repérer les traits particuliers du code qui sont les plus pertinents pour prédire la présence de bugs de sécurité. Effectivement, différentes mesures, peuvent avoir un impact plus ou moins significatif sur l'émergence de vulnérabilités. L'importance relative des différentes métriques utilisées dans notre modèle sera donc analysée.

— **Existe-t-il des différences significatives entre les valeurs des métriques de qualité dans les fichiers affectés par des bugs de sécurité confirmés et ceux qui ne le sont pas ?**

Cette interrogation offre la possibilité d'examiner si certaines mesures de qualité diffèrent de manière significative entre les fichiers impliqués dans des problèmes de sécurité et les autres fichiers du projet. En analysant les fichiers qui ont été touchés et ceux qui n'ont pas été touchés par des bugs, nous pourrons déterminer si certaines caractéristiques du code sont plus observables dans les fichiers vulnérables.

— **Quelle est l'efficacité des algorithmes de machine learning dans la prédiction des bugs de sécurité ?**

Il est crucial d'évaluer les performances des algorithmes afin de déterminer leur utilité dans la réalité. Différents algorithmes d'apprentissage automatique seront testés sur nos données pour évaluer leur précision, leur rappel et leur score F1, en analysant leurs performances à l'aide de techniques de validation croisée.

3.2 PRÉTRAITEMENT DES DONNÉES

Avant l'entraînement des modèles de machine learning, il est crucial de procéder au prétraitement des données. Il comprend l'intégration de la variable **buggy**, le choix des métriques communes à divers langages de programmation, l'organisation du dataset en couples de versions buggy et non-buggy, ainsi que le tri des données en fonction de leur ordre chronologique.

3.2.1 AJOUT DE LA VARIABLE `BUGGY`

La première étape du prétraitement a consisté à introduire une nouvelle variable appelée `buggy`, qui indique si un fichier contient ou non un bug de sécurité confirmé. Cette variable est définie comme suit :

— `1` si le fichier est buggy (présence d'un bug de sécurité),

— `0` Sinon.

Cette variable joue un rôle central dans l'entraînement des modèles de machine learning, en servant de cible pour la prédiction.

3.2.2 SÉLECTION DES MÉTRIQUES COMMUNES AUX LANGAGES DE PROGRAMMATION

Pour chacun des 340 000 fichiers analysés dans notre étude, nous avons calculé 40 métriques de qualité avec L'outil **undertand**. Cependant, pour garantir l'homogénéité de notre jeu de données, nous avons conservé les 25 métriques communes à tous les langages de programmation. Cette sélection permet d'assurer que les résultats de l'analyse sont applicables à une variété de langages de programmation.

3.2.3 ORGANISATION DU JEU DE DONNÉES PAR COUPLES (BUGGY VS NON-BUGGY)

Nos données ont été organisées sous forme de couples, où chaque fichier est représenté par deux versions :

— Une version *before* (avant la correction du bug),

— Une version *after* (après la correction du bug).

Cette organisation permet une comparaison directe entre les fichiers avant et après la correction des bugs de sécurité.

3.2.4 TRIAGE CHRONOLOGIQUE DES DONNÉES

Les versions les plus anciennes seront utilisées pour l'entraînement des modèles, tandis que les versions les plus récentes seront utilisées pour les tests. La validation rigoureuse de cette approche est assurée en veillant à ce que les modèles soient testés sur des données qu'ils n'ont jamais vues, tout en respectant la chronologie des événements.

3.3 IDENTIFICATION DES MÉTRIQUES CLÉS POUR LA PRÉDICTION DES BUGS DE SÉCURITÉ

Dans le cadre de la première question de recherche, notre objectif est de fournir une description détaillée des indicateurs de qualité logicielle examinés dans ce mémoire. En outre, notre objectif est de souligner leur influence sur l'émergence des bugs de sécurité en identifiant l'importance des caractéristiques, c'est-à-dire les métriques qui ont le plus d'impact sur l'émergence des bugs de sécurité dans les systèmes applicatifs. Il s'agit de repérer les indicateurs de qualité auxquels les professionnels doivent accorder une attention particulière, car ils peuvent augmenter le risque d'apparition de problèmes de sécurité.

Approche Pour répondre à cette première question, nous commençons par identifier, parmi les fichiers du code source du projet, ceux qui impliquent des bugs de sécurité. Dans le tableau 3.1, nous présentons le nombre de fichiers dans chaque système logiciel impliquant des bugs de sécurité. Une fois les fichiers contenant des bugs de sécurité identifiés, nous avons créé une nouvelle colonne dans les fichiers CSV appelée `buggy`, où nous avons attribué une valeur de 1 si le fichier contient des bugs de sécurité, et une valeur de 0 s'il n'en contient pas.

TABLEAU 3.1 : Informations sur les bugs de sécurité

Systems	Buggy files	Security bugs first report date	Security bugs last report date
Maven	533	2009-06-21	2022-04-07
npm	552	2011-02-28	2022-04-11
NuGet	750	2019-04-22	2022-03-27
Packagist	2274	2012-12-12	2022-04-27
Pypi	4024	2011-09-01	2022-04-12
GHSA	9052	2011-02-28	2022-04-29
OSS-Fuzz	16109	2016-12-10	2022-05-07

Nous avons employé cinq méthodes pour évaluer l'importance des caractéristiques, à savoir la régression linéaire, la régression logistique, le classificateur XGB Chen & Guestrin (2016), le F-Score et Boruta Kursa & Rudnicki (2010), afin de repérer les caractéristiques dont l'importance pourrait influencer la colonne cible (c'est-à-dire la présence de bugs de sécurité dans les fichiers analysés). Dans cette étape, notre but est d'analyser l'importance des caractéristiques en utilisant diverses méthodes couramment utilisées pour sélectionner les meilleures caractéristiques, afin de déterminer éventuellement les métriques qui revêtent le plus d'importance et qui sont les plus utiles pour prédire les bugs de sécurité. La prédiction des bugs de sécurité serait probablement le plus influencée par les métriques de qualité les plus importantes.

Résultats : Les résultats sont présentés dans les tableaux 3.2, 3.3, 3.4, 3.5 and 3.6, où les métriques ont un impact par ordre décroissant d'importance, avec chaque méthode de sélection de caractéristiques utilisant une mesure d'importance distincte. L'impact de la caractéristique spécifique sur le modèle utilisé pour prédire la présence ou non d'un bug de sécurité est plus élevé à mesure que le score augmente.

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n + \varepsilon$$

Dans cette formule :

y est la variable dépendante (cible),

β_0 est l'ordonnée à l'origine (ou intercept),

$\beta_1, \beta_2, \ldots, \beta_n$ sont les coefficients de régression associés aux variables indépendantes $x_1, x_2, \ldots, x_n$,

$x_1, x_2, \ldots, x_n$ représentent les variables indépendantes (caractéristiques),

ε est le terme d'erreur (ou bruit).

Le coefficient β_i mesure l'effet d'une variation d'une unité de la variable x_i sur la moyenne de y, tout en maintenant les autres variables indépendantes constantes. Un coefficient positif signifie une corrélation positive, tandis qu'un coefficient négatif indique une corrélation négative.

Le coefficient de régression est utilisé à la fois dans la régression linéaire et la régression logistique. Ce coefficient permet de déterminer si chaque variable indépendante présente une corrélation positive ou négative avec la variable cible dépendante. Le coefficient détermine la variation de la moyenne de la variable dépendante lorsque la variable indépendante varie d'une unité, tout en maintenant les autres variables du modèle constantes et isolées.

En ce qui concerne le classificateur XGB, l'importance des caractéristiques renvoyées pour un seul arbre de décision est calculée en fonction de la quantité avec laquelle chaque point de division d'attribut améliore la mesure de performance, pondérée par le nombre d'observations dont le nœud est responsable. La formule est donnée par :

$$\text{Importance}(f_i) = \sum_{j=1}^{T} \frac{\Delta\text{Perf}(f_i, j) \cdot N_j}{N_{\text{total}}}$$

où :

— $\text{Importance}(f_i)$ est l'importance de la caractéristique f_i,

— T est le nombre total de nœuds dans l'arbre de décision,

— $\Delta\text{Perf}(f_i, j)$ est l'amélioration de la mesure de performance due à la division sur la caractéristique f_i au nœud j,

— N_j est le nombre d'observations qui passent par le nœud j,

— N_{total} est le nombre total d'observations dans l'arbre.

Pour la méthode F-score, chaque caractéristique f_i est évaluée individuellement par la formule :

$$F(f_i) = \frac{2 \cdot (\text{Precision}(f_i) \cdot \text{Recall}(f_i))}{\text{Precision}(f_i) + \text{Recall}(f_i)}$$

Un score plus élevé indique une meilleure performance de la caractéristique sans prendre en considération qu'une caractéristique peut s'améliorer en association avec une autre.

Finalement, la méthode Boruta agit comme un algorithme de contournement autour de la forêt aléatoire. Elle génère des copies aléatoires des caractéristiques, appelées "bruit", et utilise la formule suivante pour tester l'importance d'une caractéristique f_i par rapport à ces copies :

$$Z(f_i) > Z(\text{ShadowMax})$$

où :

— $Z(f_i)$ est l'importance de la caractéristique f_i dans la forêt aléatoire,

— $Z(\text{ShadowMax})$ est l'importance maximale parmi les copies aléatoires de bruit générées (caractéristiques fictives).

Si l'importance $Z(f_i)$ de la caractéristique réelle dépasse l'importance maximale du bruit $Z(\text{ShadowMax})$, alors la caractéristique est considérée comme pertinente et mérite d'être conservée.

TABLEAU 3.2 : Impactes des métriques en utilisant les coefficients de la régression linéaire

	Metrics	**Linear Regression coefficient**
1	CountStmtExe	8.2
2	CountDeclClass	7.5
3	extension	6.3
4	AvgLineBlank	6.6
5	CountStmtDecl	3.2
6	CountDeclFunction	2.3
7	Ecosystem	1.5
8	AvgCyclomaticModified	1.8
9	MaxNesting	1.4
10	MaxCyclomaticModified	1.0
11	MaxEssential	1.4
12	AvgLineComment	1.3
13	AvgLine	1.2
14	CountLineCode	1.1
15	CountLineBlank	-2.2
16	CountStmt	-3.7
17	AvgEssential	-1.8
18	CountLineComment	-1.3
19	commitdate	-1.7
20	CountLine	-1.2
21	MaxCyclomatic	-5.5
		Continued on next page

Continued from previous page		
	Metrics	**Linear Regression coefficient**
22	AvgLineCode	-9.4
23	SumEssential	-1.2
24	AvgCyclomatic	-1.6
25	AvgCyclomaticStrict	-2.1
26	SumCyclomaticStrict	-2.4
27	SumCyclomaticModified	-8.3
28	RatioCommentToCode	-1.3
29	SumCyclomatic	-3.5

TABLEAU 3.3 : Impactes des métriques en utilisant les coefficients de la régression logistique

	Metrics	**Logistic Regression coefficient**
1	CountStmtExe	6.243662
2	SumCyclomatic	2.671614
3	CountStmtDecl	2.317193
4	SumCyclomaticModified	1.968960
5	AvgCyclomaticModified	1.686614
6	MaxNesting	0.480617
7	extension	0.442367
8	CountDeclClass	0.426263
9	AvgLineBlank	0.197796
10	MaxEssential	0.140259
		Continued on next page

Continued from previous page		
	Metrics	**Logistic Regression coefficient**
11	CountLineBlank	0.124659
12	MaxCyclomaticModified	0.099790
13	AvgLineComment	0.095030
14	AvgLineCode	0.076280
15	AvgCyclomatic	0.056402
16	RatioCommentToCode	0.000803
17	MaxCyclomatic	-0.011370
18	CountLineCode	-0.025963
19	AvgEssential	-0.037578
20	CountLine	-0.050032
21	CountLineComment	-0.056943
22	CountDeclFunction	-0.083573
23	AvgCyclomaticStrict	-0.294391
24	SumEssential	-0.551190
25	AvgLine	-0.641006
26	SumCyclomaticStrict	-1.818907
27	CountStmt	-10.093714

TABLEAU 3.4 : Impactes des métriques en utilisant les coefficients de XGB

	Metrics	**Classifier Feature Importance**
1	CountDeclClass	0.465517
Continued on next page		

Continued from previous page		
	Metrics	**Classifier Feature Importance**
2	CountStmtExe	0.125924
3	extension	0.076086
4	SumCyclomatic	0.040734
5	MaxNesting	0.030293
6	SumCyclomaticModified	0.016307
7	AvgCyclomaticModified	0.016200
8	AvgLine	0.013371
9	CountStmt	0.012795
10	AvgCyclomatic	0.012013
11	AvgEssential	0.011858
12	AvgCyclomaticStrict	0.009821
13	RatioCommentToCode	0.007681
14	CountDeclFunction	0.007589
15	MaxEssential	0.007159
16	MaxCyclomatic	0.005614
17	CountLineBlank	0.005606
18	CountLineComment	0.004248
19	Ecosystem	0.004073
20	CountStmtDecl	0.003718
21	CountLine	0.003071
22	AvgLineCode	0.002771
23	MaxCyclomaticModified	0.002624
Continued on next page		

Continued from previous page		
	Metrics	**Classifier Feature Importance**
24	SumEssential	0.002550
25	AvgLineComment	0.002318
26	AvgLineBlank	0.002274
27	commitDate	0.002242
28	CountLineCode	0.002017
29	SumCyclomaticStrict	0.001612

TABLEAU 3.5 : Impactes des métriques en utilisant les coefficients de F-Score

	Metrics	**F-score Feature Score**
1	CountStmtExe	47332.3
2	CountDeclClass	9278.3
3	SumCyclomatic	8828.3
4	SumCyclomaticModified	7257.5
5	MaxEssential	6788.4
6	AvgCyclomaticModified	3784.1
7	CountStmtDecl	3910.1
8	extension	4174.5
9	CountLine	3124.5
10	CountLineComment	2379.0
11	MaxNesting	2282.1
12	SumEssential	2102.5
Continued on next page		

Continued from previous page		
	Metrics	**F-score Feature Score**
13	CountStmt	2183.5
14	MaxCyclomatic	1981.4
15	AvgEssential	1658.1
16	Ecosystem	1591.6
17	CountLineBlank	2258.5
18	MaxCyclomaticModified	1770.3
19	CountDeclFunction	1367.8
20	RatioCommentToCode	544.7
21	AvgLineComment	461.8
22	AvgLineCode	295.9
23	AvgCyclomaticStrict	123.6
24	AvgCyclomatic	121.5
25	CountLineCode	120.8
26	AvgLine	166.3
27	AvgLineBlank	165.5

TABLEAU 3.6 : Impactes des métriques en utilisant les coefficients de Boruta

	Metrics	**Boruta Feature Ranking**
1	CountStmtExe	1
2	SumCyclomaticModified	2
3	SumCyclomatic	3
		Continued on next page

Continued from previous page		
	Metrics	**Boruta Feature Ranking**
4	AvgLineCode	4
5	CountDeclClass	5
6	AvgCyclomaticModified	6
7	extension	7
8	AvgCyclomaticStrict	8
9	CountLine	9
10	AvgCyclomatic	10
11	commitdate	11
12	SumEssential	12
13	MaxCyclomatic	13
14	AvgLineComment	14
15	RatioCommentToCode	15
16	AvgEssential	16
17	Ecosystem	17
18	CountLineComment	18
19	CountLineCode	19
20	MaxCyclomaticModified	20
21	CountLineBlank	21
22	CountStmtDecl	22
23	AvgLine	23
24	MaxEssential	24
25	CountDeclFunction	25
Continued on next page		

Continued from previous page		
	Metrics	**Boruta Feature Ranking**
26	MaxNesting	26
27	AvgLineBlank	27
28	SumCyclomaticStrict	28
29	CountStmt	29

TABLEAU 3.7 : Les Hyper-Paramètres de nos algorithmes

ML algorithm	**Hyper-parameters**
Nearest Neighbors	Number of neighbors = 5
Decision Tree	Max_depth = 5
Random Forest	max_depth=5, n_estimators=100, max_features=1
Multi-layer Perceptron	activation='relu', solver = 'adam', alpha=1, batch_size = 'auto'
AdaBoost	N_estimators = 50
XGBClassifier	N_estimators = 100
Naive Bayes	priors=None, var_smoothing=1e-09
QDA	priors=None, reg_param=0.0, store_covariance=False, tol=0.0001
SVM linear	kernel='linear'
SVM Gaussian	kernel='rbf'
LogisticRegression	random_state=42, solver='lbfgs', max_iter=1000.

CHAPITRE IV
RÉSULTATS DE L'ÉTUDE EMPIRIQUE

4.1 LES MÉTRIQUES DE QUALITÉ AVEC LA PLUS GRANDE IMPORTANCE DANS L'APPARITION DES BUGS DE SÉCURITÉ

La sélection de différentes approches de sélection de caractéristiques a été effectuée de manière réfléchie afin de maximiser la robustesse de notre analyse. En effet, différentes méthodes peuvent être employées pour sélectionner les caractéristiques, notamment des approches basées sur des tests statistiques, des techniques de réduction de dimension ou encore des méthodes intégrées aux modèles d'apprentissage automatique.

Notre choix s'est porté sur une combinaison de plusieurs méthodes afin de garantir qu'en présence d'un ensemble de caractéristiques hautement significatives, celles-ci puissent être identifiées indépendamment de la technique utilisée. En particulier, nous avons utilisé des approches basées sur les coefficients de régression, les scores F ainsi que des algorithmes tels que XGBoost et Boruta. Chacune de ces techniques offre une perspective différente : certaines privilégient la relation linéaire entre les variables et la cible, tandis que d'autres exploitent des interactions complexes et non linéaires.

En combinant ces méthodes, notre objectif était d'identifier de manière fiable les variables essentielles influençant la présence de vulnérabilités de sécurité dans les fichiers, tout en minimisant les biais liés au choix d'une seule technique.

Toutefois, les résultats des tableaux 3.2 à 3.6 montrent que chaque méthode fournit une liste exclusive des caractéristiques les plus significatives. En comparant les 10 variables les plus significatives provenant de chaque méthode, nous avons constaté que 6 variables

(CountStmtExe, SumCyclomaticModified, AvgCyclomaticModified, extension, CountDeclClass, SumCyclomatic) ont été identifiées par toutes les techniques. Cela laisse entendre que ces variables sont extrêmement importantes et nécessitent une attention particulière en ce qui concerne leur possible lien avec les problèmes de sécurité. Cependant, même si ces variables apparaissent de manière systématique, nous estimons que toutes les variables étudiées ont un impact sur la prédiction des bugs de sécurité. Il est donc important de continuer à les prendre en compte dans les analyses futures dans le but d'affiner nos modèles de prédiction.

4.2 ANALYSE COMPARATIVE DES FICHIERS (BUGGY VS NON-BUGGY)

En guise de réponse à la deuxième question de recherche, nous analysons les valeurs des métriques de qualité afin de déterminer s'il y a un bug de sécurité. En effet, notre objectif est de confirmer une différence significative entre les valeurs des métriques de qualité en ce qui concerne l'existence de bugs de sécurité. Ainsi, lorsque les praticiens observent de telles différences, ils peuvent être alertés et commencer à réorganiser ou à corriger le code comme mesures préventives. En outre, il est possible de repérer des liens possibles entre des valeurs spécifiques des métriaues de qualité et les vulnérabilités du code logiciel, ce qui peut améliorer notre capacité à repérer les vulnérabilités déjà présentes et leur impact potentiel sur les problèmes de sécurité, voire à prédire de futures faiblesses du code source.

4.2.1 APPROCHE POUR LA COMPARAISON

TABLEAU 4.1 : Les 10 meilleures Métriques par méthode de sélection

Méthodes	10 meilleures Métriques
LinearRegression	CountStmtExe

Suite à la page suivante

Tableau 4.1 – *Suite de la page précédente*

Méthodes	**10 meilleures Métriques**
	CountDeclClass
	extension
	AvgLineBlank
	SumCyclomaticModified
	MaxCyclomatic
	SumCyclomatic
	AvgCyclomaticModified
	MaxNesting
	CountStmt
Logistic Regression	CountStmt
	CountStmtExe
	SumCyclomatic
	CountStmtDecl
	SumCyclomaticStrict
	SumCyclomaticModified
	AvgCyclomaticModified
	MaxNesting
	extension
	CountDeclClass
XGBClassifier	CountDeclClass
	CountStmtExe
	extension

Suite à la page suivante

Tableau 4.1 – *Suite de la page précédente*

Méthodes	**10 meilleures Métriques**
	SumCyclomatic
	MaxNesting
	SumCyclomaticModified
	AvgCyclomaticModified
	AvgLine
	CountStmt
	AvgCyclomatic
F-score	CountStmtExe
	CountDeclClass
	SumCyclomatic
	SumCyclomaticModified
	MaxEssential
	AvgCyclomaticModified
	CountStmtDecl
	extension
	CountLine
	CountLineComment
Boruta	CountStmtExe
	SumCyclomaticModified
	SumCyclomatic
	AvgLineCode
	CountDeclClass

Suite à la page suivante

Tableau 4.1 – *Suite de la page précédente*

Méthodes	**10 meilleures Métriques**
	AvgCyclomaticModified
	extension
	AvgCyclomaticStrict
	CountLine
	AvgCyclomatic

Les dix meilleures caractéristiques extraites par les cinq techniques de sélection de caractéristiques citées dans la première question de recherche sont présentées dans le tableau 4.1. Il est observé que les 10 caractéristiques les plus performantes obtenues varient en fonction des méthodes employées. Parmi les 25 métriques initiales, nous avons décidé d'analyser les valeurs de cinq métriques dans le cadre d'une analyse préliminaire. Effectivement, CountStmtExe, SumCyclomaticModified, AvgCyclomaticModified, CountDeclClass et SumCyclomatic ont été choisis car ces cinq métriques se classent parmi les dix premières(les plus importantes) dans nos différentes méthodes de sélection de caractéristiques.

Comme nous pouvons l'observer dans le tableau 4.2, nous identifions les valeurs de la moyenne, de l'écart-type, du minimum, du maximum, de la médiane et des quartiles (25% et 75%) pour chaque métrique. Pour les deux cibles, c'est-à-dire Buggy = 0 et Buggy = 1, nous recueillons ces valeurs individuellement afin de pouvoir éventuellement observer une différence significative entre elles. À partir de cette base, nous affichons les graphes correspondants pour les fichiers contenant des bugs de sécurité et les fichiers sans bugs de sécurité, où nous illustrons visuellement la disparité entre la médiane, Q1 (25%) et Q3 (75%).

En plus de cela afin d'approfondir et d'obtenir une perspective globale, nous exposons dans cette partie l'analyse associée à nos cinq métriques de qualité. Finalement, nous effectuons un test statistique dans l'optique de vérifier la disparité observée entre les valeurs des métriques des fichiers avec et sans bugs de sécurité. Nous utilisons le test statistique non paramétrique de Mann-Whitney U Hollander *et al.* (2013) pour vérifier si les valeurs des métriques des fichiers propres (qui ne sont pas affectés par des bugs de sécurité) ou des fichiers affectés par des bugs de sécurité sont identiques. $\alpha = 0{,}05$. Nous optons pour ce test en raison de sa popularité et de son utilisation dans de nombreux travaux précédents sur l'analyse des bugs et des logiciels afin de vérifier si deux échantillons sont probablement issus de la même population (c'est-à-dire que les deux populations ont la même forme) Meléndez *et al.* (2020) Xiao *et al.* (2019).

Ce test est approprié pour notre ensemble de données, car il n'est pas paramétrique et ne requiert pas l'hypothèse d'une distribution normale des données. Ceci en fait un outil très pratique pour des données asymétriques ou avec des valeurs extrêmes, telles que celles que nous examinons. En outre, le test suppose que l'échantillon est aléatoire, ce qui assure que les observations choisies, qu'elles proviennent de fichiers impliqués dans des bugs de sécurité ou non, sont représentatives de toutes les données. Le test est également basé sur l'indépendance des échantillons, c'est-à-dire que les observations d'un groupe ne sont pas associées à celles de l'autre. Dans notre situation, chacun des fichiers fait partie du groupe des fichiers présentant des problèmes de sécurité ou bien du groupe des fichiers sans problèmes, mais jamais des deux. Grâce à cette indépendance, il est possible de faire une comparaison fiable entre les deux groupes sans entrave externe.

4.3 RÉSULTATS

Effectivement, il est possible d'observer une disparité entre les valeurs des métriques obtenues pour les fichiers sans problèmes de sécurité et celles pour les fichiers avec des

problèmes de sécurité dans le tableau 4.2. Prenons l'exemple de la médiane : pour toutes les métriques observées, la médiane des fichiers avec des bugs de sécurité est en moyenne trois fois supérieure à celle des fichiers sans bugs de sécurité, ce qui peut indiquer une corrélation entre la valeur élevée de la métrique et l'apparition d'un dysfonctionnement de sécurité.

Nous pouvons en déduire que plus la complexité cyclomatique du système logiciel (par exemple, SumCyclomatic) est élevée, plus le risque d'introduire des bugs de sécurité est élevé. En outre, des tendances similaires sont observées avec les métriques de qualité qui décrivent le nombre d'instructions exécutables (par exemple, CountStmtExe) et le nombre de déclarations de classes (par exemple, CountDeclClass). Le nombre de chemins indépendants à travers le code source d'un programme est mesuré par la complexité cyclomatique du code pour évaluer la complexité d'un programme. La complexité cyclomatique du code augmente avec la complexité du logiciel, et plus le code est complexe, moins la sécurité du système logiciel est élevée [7] Alenezi & Zarour (2020).

Au niveau des figures de 4.1 à 4.5,nous présentons les graphes des quartiles pour les cinq métriques discutées : CountStmtExe, SumCyclomatic, AvgCyclomaticModified, CountDeclClass et SumCyclomaticModified. Ces figures montrent clairement une différence significative concernant les mesures statistiques des métriques de qualité entre un fichier propre et un fichier contenant un bug de sécurité.

Afin de vérifier la disparité des valeurs des métriques de qualité, comme cela a été démontré précédemment, nous employons le test U de Mann-Whitney pour de démontrer que les deux groupes (Buggy=0, Buggy=1) présentent des différences significatives.

— Les données sont sous-échantillonnées Shelke *et al.* (2017) pour garantir un nombre équivalent d'instances entre les fichiers contenant des problèmes de sécurité et ceux où

7. http ://www.mccabe.com/pdf/MoreComplexEqualsLessSecure-McCabe.pdf

aucun problème de sécurité n'a été détecté. Cela vise à équilibrer les classes afin de les comparer.

— À partir des données d'entraînement, nous sélectionnons un échantillon afin de générer des résultats pour le test U de Mann-Whitney. Par conséquent, nous sélectionnons un échantillon aléatoire de 25 % des données d'entraînement équilibrées.

— Hypothèse nulle (H0) : Les distributions des deux groupes (fichiers avec bugs de sécurité et fichiers sans bugs) sont égales.

— Hypothèse alternative (HA) : les distributions des deux groupes ne sont pas égales.

— Le test U de Mann-Whitney révèle une P-value importante (< 0,05) pour toutes les mesures de qualité. Par exemple, les résultats du test pour des ensembles de métriques de qualité choisis sont présentés :

— P-value= 3,97e-05 pour la métrique CountStmtExe

— P-value= 2,58e-16 pour la métrique SumCyclomatic

— P-value= 2,222e-7 pour la métrique AvgCyclomaticModified

— P-value= 1,7e-02 pour la métrique CountDeclClass

— P-value= 1,53e-25 pour la métrique SumCyclomaticModified

Puisqu'il s'agit de valeurs p inférieures à alpha = 0,05 pour chaque instance, nous disposons de preuves solides contre l'hypothèse nulle. Ainsi, il est évident que les valeurs des métriques logicielles analysées dans les fichiers contenant des bugs de sécurité diffèrent considérablement de celles observées dans les fichiers sans bugs de sécurité. Les résultats corroborent notre constat visuel selon lequel les valeurs des métriques de qualité peuvent être utilisées comme indicateurs de bugs de sécurité potentiels. Dans l'ensemble, nous avons constaté que les valeurs moyennes des métriques de qualité dans les fichiers avec bugs de sécurité sont trois fois plus élevées que celles

des fichiers sans bugs de sécurité. Il est donc conseillé de notifier les développeurs dès que des tendances similaires sont repérées.

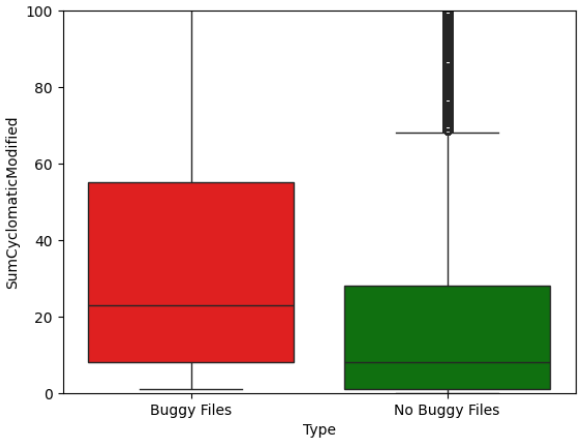

FIGURE 4.1 : Diagramme en boite (SumCyclomaticModified)

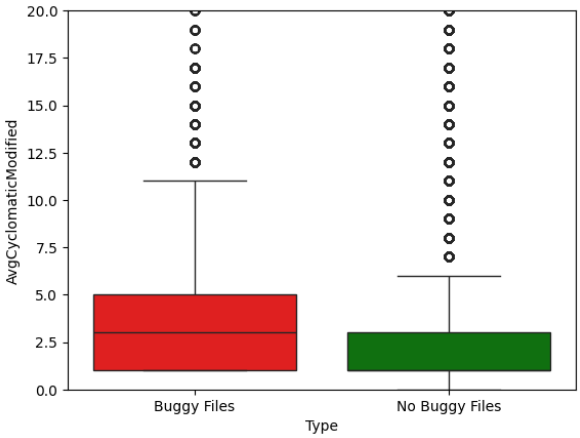

FIGURE 4.2 : Diagramme en boite (AvgCyclomaticModified)

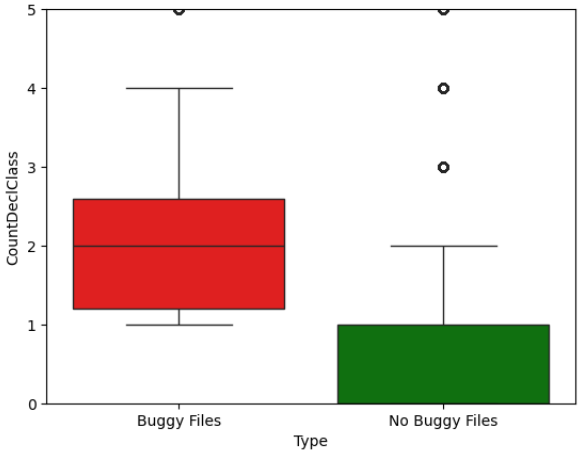

FIGURE 4.3 : Diagramme en boite (CountDeclClass)

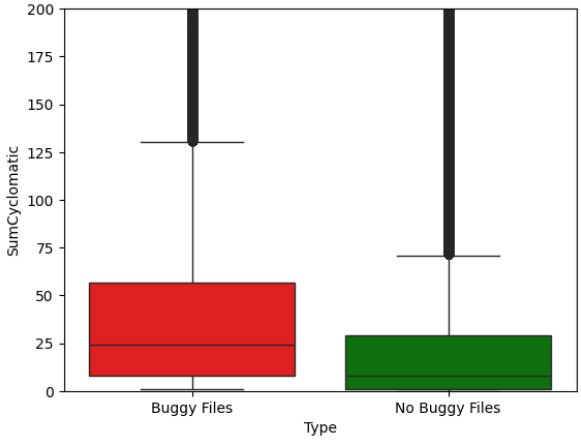

FIGURE 4.4 : Diagramme en boite (SumCyclomatic)

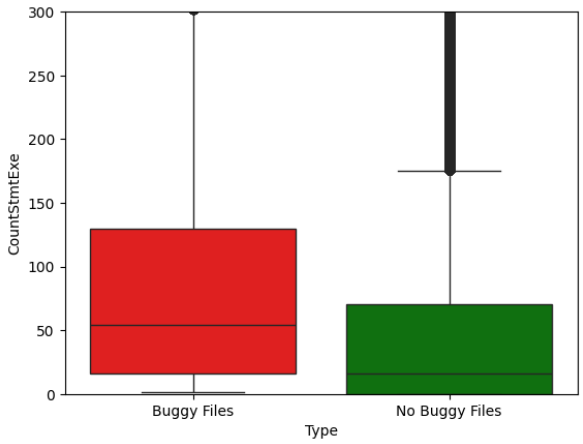

FIGURE 4.5 : Diagramme en boite (CountStmtExe)

4.4 PRÉDICTION DES BUGS DE SÉCURITÉ AVEC DES ALGORITHMES DE MACHINE LEARNING

Dans cette Partie, nous étudions comment les méthodes d'apprentissage automatique peuvent contribuer à la prédiction des vulnérabilités de sécurité des logiciels. De cette manière, nous examinons 11 méthodes de classification et repérons les plus performantes pour anticiper les problèmes de sécurité en utilisant 6 indicateurs d'évaluation. Les données utilisées ont été extraites et collectées afin de prédire les bugs de sécurité, comme indiqué dans la section **Étapes de la collecte des données**. Nous considérons les algorithmes suivants dans le cadre de cette étude :

— Nearest Neighbors

— Decision Trees

— Random Forests

— Multi-layer Perceptron

— XgBoost

— AdaBoost

— Naive Bayes

— Quadratic Discriminant Analysis

— SVM linear

— SVM Gaussian

— Logistic Regression

Les métriques d'évaluation considérées après la validation croisée (5-fold cross-validation) sont : la précision (Precision), le rappel (Recall), le F1-score, le coefficient de corrélation de Matthews (MCC) et la courbe ROC (AUC-ROC). Les métriques listées ont été calculées lors de l'analyse des données combinées de tous les projets. Notre approche pour cette question de recherche suit les étapes présentées dans Brownlee (2014).

— **Déterminer le but** : Il s'agit de mesurer la capacité des techniques d'apprentissage automatique à prédire l'émergence de bugs de sécurité en se référant aux valeurs des métriques de qualité. En outre, nous examinerons différentes méthodes d'apprentissage automatique couramment employées afin de déterminer la prédiction la plus précise de ces bugs.

— **Analyser les données** : Dans un premier temps, nous avons utilisé des scripts Python pour générer des statistiques descriptives sur les données utilisées (telles que le nombre de caractéristiques, les données dupliquées, les types de valeurs, etc.) ainsi que des graphiques de visualisation afin de mieux appréhender nos métriques.

— **Préparer les données** : nous avons effectué une vérification du type de données pour éviter toute transformation afin de mieux présenter la structure du problème de prédiction aux algorithmes de modélisation. Dans notre situation, les informations étaient conformes au format approprié, car toutes les mesures de qualité étaient exprimées en nombres.

— **Évaluer les techniques** : Cette étape consiste à évaluer les techniques en deux étapes : d'abord, nous entraînons les algorithmes afin de créer le modèle prédictif, puis nous évaluons le modèle en se basant sur les métriques. Dans cette optique, nous avons adopté une méthode de validation croisée en cinq étapes.

Afin d'obtenir une quantité accrue de données d'entraînement, nous avons également effectué un équilibrage (suréchantillonnage) des classes de données d'entraînement à chaque itération de la validation croisée en 5 volets en utilisant la méthode SMOTE Blagus & Lusa (2013).

Notre étude s'appuie sur l'utilisation de SMOTE (Synthetic Minority Over-sampling Technique) en raison du déséquilibre important observé entre les fichiers « buggy » (fichiers contenant des bugs de sécurité) et les fichiers « non-buggy » dans les différents écosystèmes de logiciels, comme le montre le tableau ci-dessous :

Ecosystem	Total Files	Buggy Files	Percentage Buggy (%)
GHSA	100783	9052	8.98
Maven	8752	533	6.09
NuGet	8985	750	8.35
OSS-Fuzz	129129	16109	12.48
Packagist	19960	2274	11.39
PyPI	51645	4024	7.79
npm	19189	552	2.88

TABLEAU 4.3 : Fichiers buggy par écosystème

Il est évident que dans la plupart des écosystèmes, le taux de fichiers buggy demeure faible par rapport au nombre total de fichiers. Par exemple :

— Au sein de l'écosystème **npm**, seulement 2,88% des fichiers présentent des problèmes.

— Dans d'autres écosystème tels que **Maven** et **PyPI**, le taux de fichiers incorrects est de respectivement 6,09 % et 7,79 %. Même si la proportion de fichiers buggy est légèrement

plus élevée dans les écosystèmes (comme OSS-Fuzz avec 12,48% de fichiers buggy), les fichiers non-buggy restent majoritaires.

Les algorithmes d'apprentissage automatique ont tendance à favoriser la classe majoritaire (les fichiers non-buggy) lorsque les données sont fortement déséquilibrées. Cela peut conduire à une détection erronée, dans laquelle le modèle est extrêmement efficace pour détecter les fichiers non-buggy (classe majoritaire), mais ne parvient pas à détecter correctement les fichiers buggy (classe minoritaire). Cela pose un problème majeur lorsqu'il s'agit de prédire les bugs de sécurité, où l'identification des bugs est la priorité.

Extension	Total Files	Buggy Files	Percentage Buggy (%)
.c	62385	11001	17.63
.cc	20983	1535	7.32
.cpp	21794	2518	11.55
.h	52523	2859	5.44
.hh	3779	625	16.54
.hpp	3966	349	8.80
.java	21020	1249	5.94
.js	33935	3155	9.30
.php	39861	4222	10.59
.py	72615	5423	7.47
.ts	5582	358	6.41

TABLEAU 4.4 : Nombre de fichiers vulnérables par langage de programmation

4.4.1 AVANTAGES DE L'UTILISATION DE SMOTE DANS NOTRE ÉTUDE

Amélioration de la détection des bugs de sécurité - En augmentant le taux de fichiers buggy dans le jeu de données d'entraînement, SMOTE permettra aux algorithmes d'acquérir une meilleure capacité à reconnaître ces fichiers, ce qui diminue le risque d'ignorer les bugs de sécurité.

Meilleur rappel et F1-score - La classe majoritaire (fichiers non-buggy) ne sera plus favorisée par les algorithmes, ce qui entraînera une amélioration des mesures de performance telles que le rappel et le F1-score, qui sont essentiels pour évaluer la capacité du modèle à détecter correctement les fichiers contenant des bugs.

Utilisation appropriée pour des ensembles de données déséquilibrés - Dans les écosystèmes comme npm, où seulement 2.88% des fichiers sont buggy, le déséquilibre est extrême, ce qui rend SMOTE encore plus pertinent. Pour des écosystèmes comme OSS-Fuzz (12.48% de fichiers buggy), bien que l'écart soit moins important, l'utilisation de SMOTE reste bénéfique pour maximiser les chances de détection des bugs de sécurité.

4.4.2 MISE EN ŒUVRE DES MODÈLES

Dans cette partie, nous exposons la mise en œuvre des divers modèles d'apprentissage automatique utilisés pour prédire les problèmes de sécurité dans les projets logiciels. Les modèles ont été sélectionnés pour leur ancienneté et leur efficacité dans des opérations de classification. Chaque modèle a été entraîné à partir de notre jeu de données prétraité, en utilisant la technique de suréchantillonnage SMOTE dans le but de compenser le déséquilibre des classes.

K-NEAREST NEIGHBORS (KNN)

Le modèle K-Nearest Neighbors (KNN) classe un point x en fonction de ses k voisins les plus proches dans l'espace des caractéristiques, selon une distance telle que la distance euclidienne :

$$d(x,y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

où x et y sont des points dans un espace à n dimensions. Le modèle attribue la classe majoritaire parmi les k voisins les plus proches.

DECISION TREES

Le modèle arborescent des arbres de décision repose sur des règles de décision. Le choix des caractéristiques est basé sur la diminution de *l'impuretés*, généralement évaluée par l'entropie :

$$H(S) = -\sum_{i=1}^{c} p_i \log_2(p_i)$$

où p_i est la proportion des éléments de la classe i dans l'ensemble de données S.

RANDOM FORESTS

Le modèle Random Forests est une méthode d'ensemble qui construit plusieurs arbres de décision aléatoires. La prédiction finale est obtenue en moyennant les prédictions des arbres individuels :

$$\hat{y} = \frac{1}{T} \sum_{t=1}^{T} f_t(x)$$

où T est le nombre d'arbres, et $f_t(x)$ est la prédiction de l'arbre t.

MULTI-LAYER PERCEPTRON (MLP)

Le perceptron multicouche est un réseau de neurones artificiels qui transforme les entrées x en sorties y à travers plusieurs couches de neurones. Chaque neurone effectue une

combinaison linéaire de ses entrées suivie d'une fonction d'activation f, par exemple la fonction sigmoïde :

$$f(x) = \frac{1}{1+e^{-x}}$$

Le réseau est entraîné via la rétropropagation en minimisant une fonction de perte, telle que l'erreur quadratique moyenne (MSE) :

$$L = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

ADABOOST

AdaBoost combine plusieurs classificateurs faibles $h_t(x)$ et ajoute les poids des exemples à chaque tour de boucle. La dernière prédiction est obtenue en additionnant des classificateurs faibles :

$$H(x) = \text{sign}\left(\sum_{t=1}^{T}\alpha_t h_t(x)\right)$$

où α_t est le poids attribué au classificateur $h_t(x)$.

XGBOOST

XGBoost est une méthode de boosting qui construit un modèle en ajoutant successivement des arbres de décision (classificateurs faibles) $h_t(x)$. La prédiction finale est donnée par :

$$H(x) = \sum_{t=1}^{T} h_t(x)$$

où T est le nombre total d'arbres.

XGBoost minimise une fonction de perte L avec un terme de régularisation R :

$$L = \sum_{i=1}^{N} L(y_i, H(x_i)) + \sum_{t=1}^{T} R(h_t)$$

où y_i est la vraie étiquette et N le nombre d'exemples. La mise à jour du modèle à chaque itération s'écrit :

$$H_{new}(x) = H_{old}(x) + \eta h_t(x)$$

avec η le taux d'apprentissage.Les techniques d'optimisation utilisées par XGBoost permettent d'améliorer l'efficacité et les performances tout en gérant la complexité du modèle.

NAIVE BAYES

Le classificateur Naive Bayes utilise le théorème de Bayes pour calculer la probabilité d'appartenance d'un échantillon à une classe C_k donnée de ses caractéristiques x :

$$P(C_k|x) = \frac{P(x|C_k)P(C_k)}{P(x)}$$

Naive Bayes part du principe que les caractéristiques x_i sont indépendantes (à une certaine condition), ce qui simplifie le calcul de $P(x|C_k)$.

QUADRATIC DISCRIMINANT ANALYSIS (QDA)

L'analyse discriminante quadratique suppose que chaque classe suit une distribution normale multivariée. La fonction de décision pour une classe C_k est donnée par :

$$\delta_k(x) = -\frac{1}{2}\log|\Sigma_k| - \frac{1}{2}(x-\mu_k)^T\Sigma_k^{-1}(x-\mu_k) + \log P(C_k)$$

où μ_k et Σ_k sont respectivement la moyenne et la matrice de covariance de la classe k.

SVM (SUPPORT VECTOR MACHINES)

Les SVM trouvent une hyperplane qui sépare les classes de manière optimale. Pour un SVM linéaire, l'hyperplane est défini par :

$$w^T x + b = 0$$

où w est le vecteur normal à l'hyperplane, et b est le biais. La marge entre les classes est maximisée en résolvant le problème suivant :

$$\min_{w,b} \frac{1}{2}||w||^2 \quad \text{sous la contrainte} \quad y_i(w^T x_i + b) \geq 1$$

Pour les SVM à noyau gaussien (RBF), une fonction noyau est utilisée pour projeter les données dans un espace de dimension supérieure :

$$K(x,x') = \exp\left(-\gamma||x-x'||^2\right)$$

RÉGRESSION LOGISTIQUE

La régression logistique prédit la probabilité qu'un échantillon appartienne à une classe $y = 1$ en fonction d'une combinaison linéaire de ses caractéristiques, transformée par une fonction sigmoïde :

$$P(y = 1|x) = \frac{1}{1 + e^{-(w^T x + b)}}$$

Le modèle est entraîné en maximisant la vraisemblance, c'est-à-dire en minimisant la fonction de coût logistique :

$$L = -\frac{1}{n} \sum_{i=1}^{n} (y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i))$$

4.4.3 ÉVALUATION DES MODÈLES

Les données ont été réparties en ensembles d'apprentissage (70%) et de test (30%). Dans un premier temps, les données ont été classées chronologiquement, des plus anciennes aux plus récentes, pour garantir l'intégrité temporelle des versions étudiées. Nous avons par la suite encodé les variables catégorielles en utilisant le *One-Hot Encoding* et avons normalisé les variables numériques. Afin de remédier à l'inégalité des classes, nous avons employé la méthode de suréchantillonnage *SMOTE*.

L'évaluation de la performance de nos différent modèles repose sur une validation croisée en 5 plis (5-fold cross-validation). Ce qui nous a permis de réduire le risque de surapprentissage en offrant une évaluation plus fiable des performances des différents modèles.

Les performances globales des modèles sont évaluées avec les métriques suivantes : Accuracy, MCC et ROC AUC, Precision, Recall et F1 Score.

4.4.4 COMPARAISON DE LA PERFORMANCE DES MODÈLES

Algorithme	Accuracy	Precision	Recall	F1 Score	MCC	ROC-AUC
KNN	0.94	0.77	0.48	0.59	0.58	0.73
Decision Trees	0.96	0.78	0.83	0.80	0.78	0.90
Random Forests	0.97	0.89	0.82	0.85	0.84	0.90
MLP	0.97	0.90	0.77	0.83	0.82	0.88
AdaBoost	0.97	0.90	0.79	0.84	0.83	0.89
XGBoost	0.98	0.95	0.82	0.88	0.87	0.91
Naive Bayes	0.84	0.16	0.15	0.15	0.07	0.53
QDA	0.62	0.15	0.61	0.24	0.15	0.62
SVM (linear)	0.86	0.35	0.30	0.52	0.25	0.71
SVM (Gaussian)	0.87	0.29	0.36	0.47	0.22	0.69
Logistic Regression	0.90	0.31	0.03	0.05	0.06	0.51

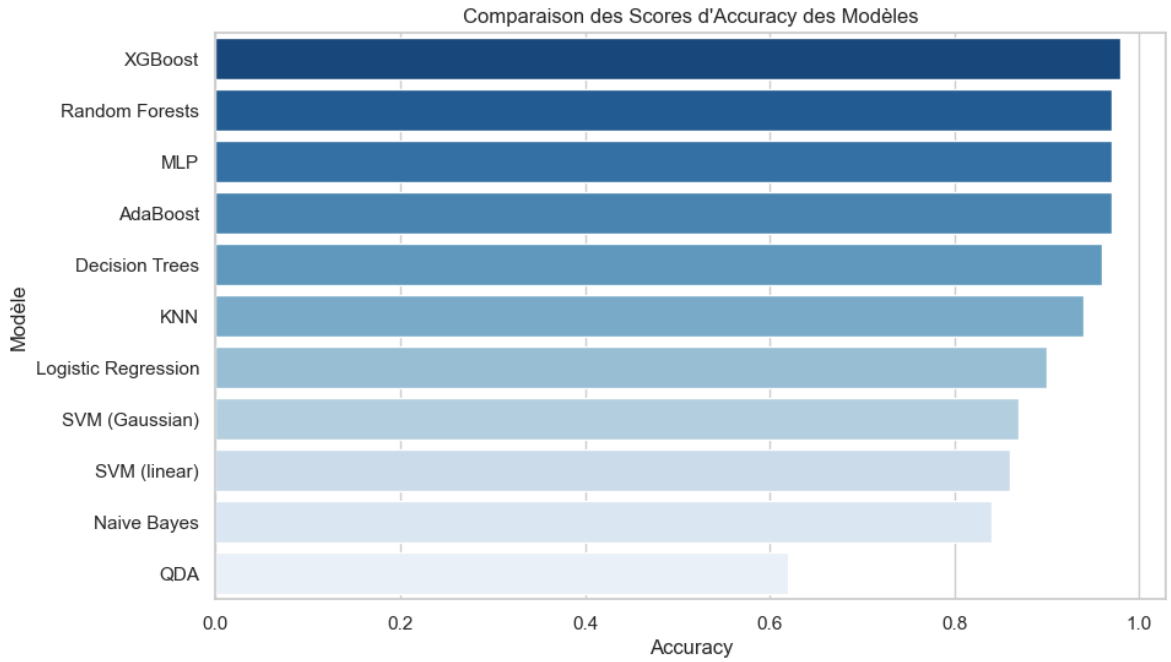

FIGURE 4.6 : scores de performance

Les modèles **XGBoost**, **Random Forests**, **MLP** et **AdaBoost** sont les plus efficaces, avec des scores de performance proches de 0.97 ou supérieurs, comme le montre le graphique

des scores. Par contre, les scores de performance les plus bas sont obtenus par **QDA** et **Naive Bayes**, ce qui suggère qu'il est difficile pour ces derniers de généraliser les prédictions sur des données de test. Les performances des modèles SVM (linear et Gaussian) ainsi que de la regression logistique sont modérées.

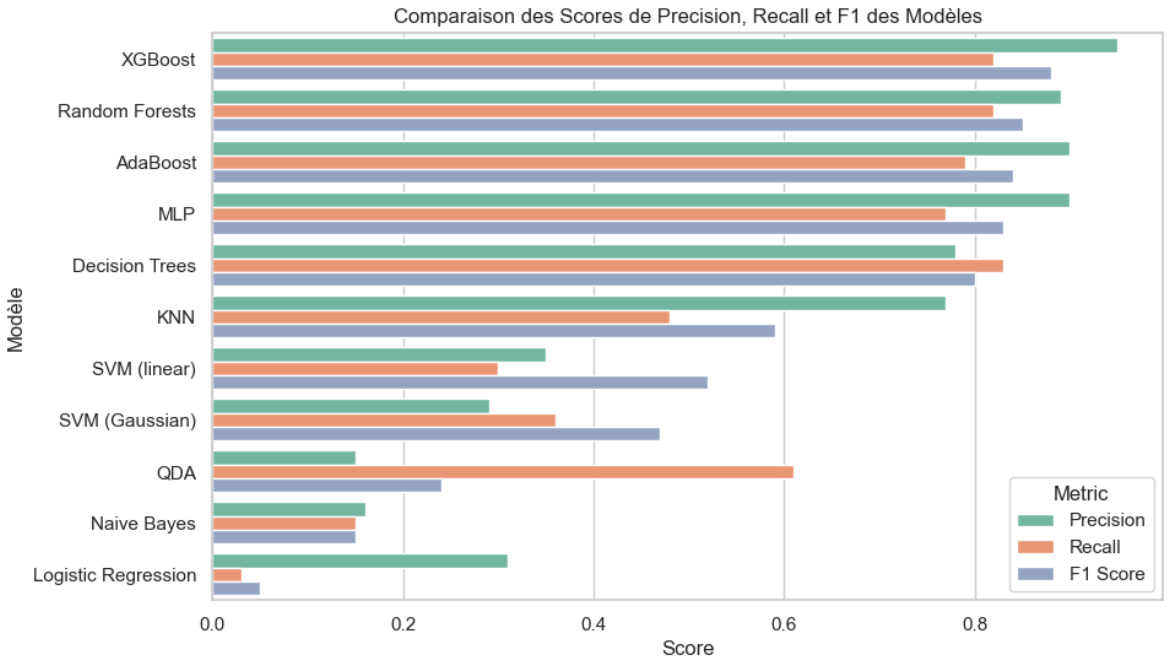

FIGURE 4.7 : Scores de précision, Rappel et F1 Score

Lorsqu'on analyse les scores de précision, de rappel et de F1, plusieurs éléments sont mis en évidence :

— Les meilleurs scores de précision sont obtenus par les modèles **XGBoost**, **AdaBoost**, **MLP** et **Random Forests**, ce qui indique qu'ils génèrent peu de faux positifs et sont fiables pour éviter les alertes non pertinentes. Les scores de précision les plus bas sont obtenus par les modèles **Naive Baye**s et **QDA**, ce qui démontre qu'ils produisent un grand nombre de fausses alertes.

— Malgré l'importance de la précision, le rappel met en évidence la capacité des modèles à détecter tous les bugs de sécurité. La distinction entre **Decision Trees** et **Random Forests** réside dans leurs scores de rappel élevés, ce qui témoigne de leur efficacité à repérer la plupart des bugs, même si cela peut parfois nécessiter des compromis en termes de précision. Les résultats bas des modèles de régression logistique, de **SVM (linear)** et de **KNN** révèlent qu'ils peuvent présenter certains problèmes, ce qui restreint leur efficacité dans un contexte de sécurité.

— Le F1-score équilibre la précision et le rappel. Une fois de plus, les modèles **XGBoost**, **Random Forests** et **AdaBoost** sont en tête, mettant en évidence leur capacité équilibrée à repérer les bugs sans générer de faux positifs ni en manquer. Le classement des modèles **Naive Bayes** et **QDA** est à nouveau en bas, car ils rencontrent des difficultés à concilier les deux aspects.

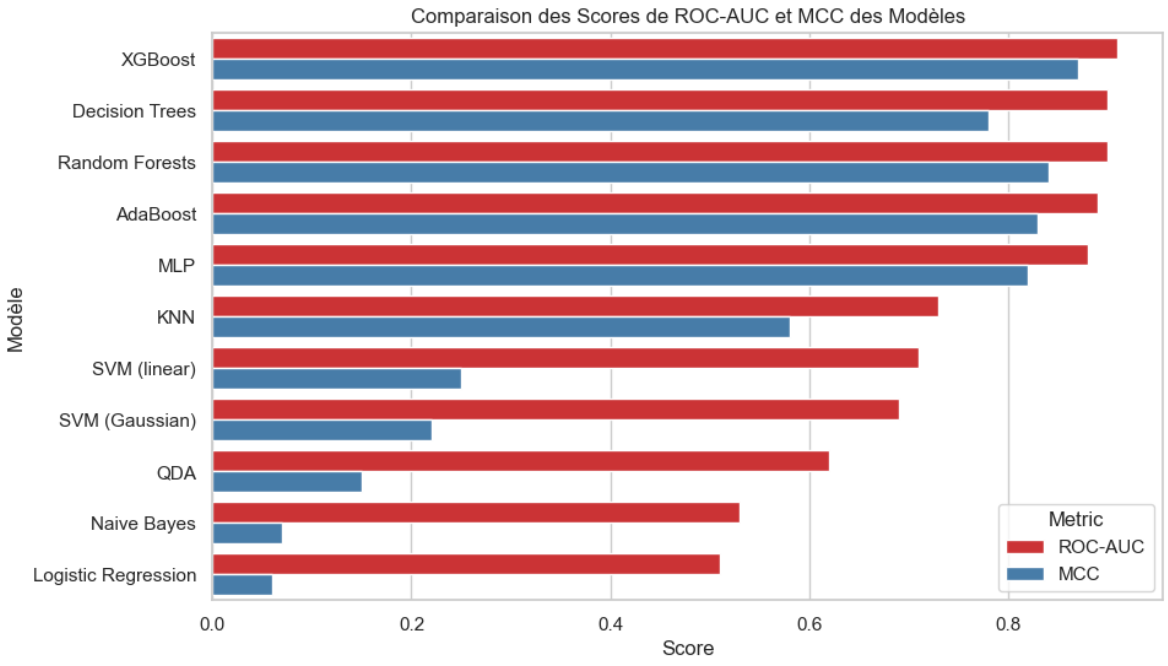

FIGURE 4.8 : Scores de ROC-AUC et MCC

— ROC-AUC : Les scores les plus élevés sont obtenus par les modèles **XGBoost**, **Random Forests** et **Decision Trees**, ce qui indique qu'ils sont parfaitement adaptés pour distinguer les fichiers vulnérables des fichiers non vulnérables. Les faibles valeurs de ROC-AUC de **Naive Baye**s et **QDA** indiquent qu'ils rencontrent des difficultés à séparer de manière adéquate les deux classes.

— MCC : cette métrique reflète l'équilibre global du modèle en tenant compte des prédictions vraies et fausses(positives et négatives). Les meilleurs scores MCC sont obtenus par **XGBoost**, **Random Forests** et **AdaBoost**, mettant en évidence leur aptitude à produire des prédictions fiables et équilibrées. En revanche, les scores MCC les plus bas sont observés chez **Naive Bayes** et **Logistic Regression**, ce qui suggère un accord global faible avec les données réelles.

4.4.5 CONCLUSION DE L'ANALYSE COMPARATIVE

— **XGBoost** (Précision : 0.95, Rappel : 0.82, F1 Score : 0.88, MCC : 0.87, ROC-AUC : 0.91), **Random Forests** (Précision : 0.89, Rappel : 0.82, F1 Score : 0.85, MCC : 0.84, ROC-AUC : 0.90), et **AdaBoost** (Précision : 0.90, Rappel : 0.79, F1 Score : 0.84, MCC : 0.83, ROC-AUC : 0.89) sont les modèles les plus efficaces pour prédire les bugs de sécurité car ils ont une grande capacité pour limiter les faux positifs (Précision élevée) et détecter efficacement les bugs (Rappel élevé).

— **MLP** (Précision : 0.90, Rappel : 0.77, F1 Score : 0.83, MCC : 0.82, ROC-AUC : 0.88) montre également des performances satisfaisantes bien que le rappel soit légèrement en dessous des algorithmes cités ci dessus.

— **Decision Trees** (Précision : 0.78, Rappel : 0.83, F1 Score : 0.80, MCC : 0.78, ROC-AUC : 0.90), bien qu'il soit efficace pour détecter les fichiers vulnérable (Rappel élevé)

génère un nombre non négligeable de faux positifs comme nous pouvons l'observer à travers la valeur de la précision.

— **Naive Bayes** (Précision : 0.16, Rappel : 0.15, F1 Score : 0.15, MCC : 0.07, ROC-AUC : 0.53), **QDA** (Précision : 0.15, Rappel : 0.61, F1 Score : 0.24, MCC : 0.15, ROC-AUC : 0.62), et **Logistic Regression** (Précision : 0.31, Rappel : 0.03, F1 Score : 0.05, MCC : 0.06, ROC-AUC : 0.51) sont moins efficaces si nous prenons en compte l'ensemble des métriques. Cela pourrait être expliqué par le fait qu'ils ne soient pas adaptés pour prédire les bugs de sécurité, en particulier dans des situations qui exigent à la fois une précision élevée et une grande robustesse.

TABLEAU 4.2 : Mesures statistiques

Métriques	Mesures	Fichiers sans bug(s)	Fichiers avec bug(s)
CountStmtExe	count	305149	33294
	Mean	96.02	192.41
	Std	646.47	1045.52
	Min	0.0	1.0
	25%	0.0	16.0
	Median	16.0	54.0
	75%	70.0	130.0
	Max	88818.0	72885.0
SumCyclomatic	Count	305149	33294
	Mean	41.98	91.78
	Std	523.73	603.12
	Min	0.0	1.0
	25%	1.0	8.0
	Median	8.0	24.0
	75%	29.0	56.9
	Max	121890.0	22705.0
AvgCyclomaticModified	Count	305149	33294
	Mean	4.23	15.25
	Std	140.97	425.27
	Min	0.0	1.0
	25%	1.0	1.0
	Median	1.0	3.0
	75%	3.0	5.0
	Max	21181.0	21161.0
CountDeclClass	Count	305149	33294
	Mean	1.14	2.78
	Std	4.60	6.09
	Min	0.0	1.0
	25%	0.0	1.2
	Median	0.0	2.0
	75%	1.0	2.6
	Max	335.0	333.0
SumCyclomaticModified	Count	305149	33294
	Mean	40.49	88.34
	Std	481.85	587.07
	Min	0.0	1.0
	25%	1.0	8.0
	Median	8.0	23.0
	75%	28.0	55.0
	Max	110753.0	21483.0

CHAPITRE V
DISCUSSION ET LIMITES

Ce chapitre est consacré à l'analyse critique des résultats obtenus et aux enseignements tirés de cette étude. Nous abordons d'abord les limites inhérentes à notre méthodologie et aux données utilisées, afin de mieux situer les conclusions dans leur contexte. Ensuite, nous explorons la validité de l'étude en examinant les dimensions de la validité interne et externe, afin d'évaluer la robustesse et la généralisation des résultats. Enfin, nous identifions des perspectives d'application et proposons des pistes de recherche future susceptibles de renforcer et d'approfondir les travaux sur la prédiction des vulnérabilités de sécurité à partir des métriques de code.

5.1 LIMITES DE L'ÉTUDE

Malgré des résultats prometteurs, notre recherche comporte plusieurs contraintes. Tout d'abord, l'emploi de mesures de code statiques, calculées à l'aide de l'outil **SciTools Understand**, peut avoir limité notre aptitude à détecter certains aspects dynamiques ou contextuels des problèmes de sécurité. Les vulnérabilités de sécurité peuvent être causées par des relations complexes lors de l'exécution qui ne sont pas apparentes dans une analyse statique. En outre, notre étude s'est focalisée sur un groupe limité d'écosystèmes logiciels, ce qui pourrait restreindre la généralisation des résultats à d'autres types de logiciels ou langages utilisés. En outre, **SciTools Understand** supporte un nombre limité de langages de programmation. Il est donc possible que notre approche ne soit pas applicable à des systèmes utilisant des langages non supportés, ce qui limite la portée de l'étude.

5.2 VALIDITÉ DE L'ÉTUDE

5.2.1 VALIDITÉ INTERNE

Les risques en ce qui concerne l'intégrité de notre étude portent sur les écosystèmes choisis, les algorithmes employés, les modèles d'apprentissage automatique sélectionnés et les outils employés.

Afin de réduire ces risques, nous avons tout d'abord consulté la littérature afin de repérer des systèmes qui ont déjà fait l'objet de nombreuses études précédentes. Par la suite, nous avons créé divers scripts Python que nous avons exécuté sur l'API OSV afin de collecter des données sur les problèmes de sécurité présents dans ces fichiers. Dans le même cadre, nous nous sommes assurés que la majorité des indicateurs de qualité utilisés dans notre étude soient couramment employés pour prédire les défauts en génie logiciel, même si notre objectif principal est de détecter les vulnérabilités de sécurité. Enfin, en ce qui concerne l'outil **Understand** utilisé, il est vrai que d'autres outils existants pourraient donner des résultats différents de ceux de l'outil utilisé dans cette étude, mais l'outil **Understand** est l'un des meilleurs outils qui offre de nombreuses métriques de qualité différentes utilisées dans différentes études de recherche antérieures Grichi *et al.* (2019); Grichi *et al.* (2020). En outre, il offre la possibilité de calculer des métriques dans plusieurs langages de programmation et propose la possibilité d'être exécuté en ligne de commande dans des scripts.

5.2.2 VALIDITÉ EXTERNE

Les risques de validité externe portent sur les éléments susceptibles d'influencer la généralisation de nos résultats. Il est possible que nos résultats ne puissent pas être appliqués à tous les systèmes logiciels existants, car nous avons seulement examiné un échantillon de 7 écosystèmes open source et analysé uniquement leurs fichiers. En effet les propriétés d'un

système logiciel peuvent différer selon divers critères et éléments. Toutefois, afin de minimiser cette menace, nous avons choisi des écosystèmes plus considérables qui possèdent une histoire longue (plusieurs versions) et un nombre de lignes de code variant de plusieurs centaines de milliers à des millions de lignes.

5.3 PERSPECTIVES

Les résultats obtenus nous ont montré que les méthodes d'apprentissage automatique présentent un potentiel prometteur pour améliorer la prédiction des vulnérabilités de sécurité dans les systèmes logiciels. Il est possible que cela incite les chercheurs et les professionnels à adopter ces modèles dans le but d'améliorer constamment la qualité des logiciels. Cependant, il faudra également varier les sources de données au-delà des métriques statiques et intégrer des analyses dynamiques ou contextuelles afin d'avoir une vision plus globale des vulnérabilités potentielles.

Ces résultats pourraient être exploités dans le monde professionnel dans l'optique de donner la priorité aux fichiers à analyser en profondeur ou de guider les efforts de changement du code en repérant les fichiers les plus susceptibles d'être vulnérables. Cependant, il serait nécessaire de faire des modifications pour adopter ces modèles, en particulier en ce qui concerne la personnalisation des métriques de code en fonction des types de projets spécifiques ou des environnements de développement.

5.4 PISTES DE RECHERCHE FUTURE

L'intégration de méthodes d'analyse dynamique ou de métriques provenant de l'exécution pourrait être étudiée dans les recherches à venir afin de compléter les données statiques utilisées dans cette étude. En outre, l'emploi de méthodes d'apprentissage semi-supervisé ou

non supervisé pourrait favoriser une meilleure exploitation des données non étiquetées, qui sont fréquentes dans les environnements de développement logiciel. À l'avenir, nous avons l'intention de :

1. Élargir notre étude à un éventail plus vaste de systèmes logiciels pour améliorer leur utilisation généralisée ;
2. Analyser les résultats de l'étude en prenant en considération la classification des fichiers en fonction des divers types de vulnérabilités de sécurité ainsi que leur gravité.
3. intégrer des écosystèmes écrits dans des langages de programmation qui ne sont pas encore compatibles à **understand**.
4. Mener d'autres études qualitatives pour évaluer le niveau de gravité des vulnérabilités de sécurité.

CONCLUSION

Tous les acteurs de l'industrie du logiciel sont très préoccupés par la nécessité de proposer un système logiciel sécurisé. La présence de vulnérabilités de sécurité dans les logiciels est l'un des principaux défis qui a entraîné une baisse de la sécurité des logiciels. Dans ce mémoire, nous avons examiné comment prédire ces problèmes de sécurité en utilisant des méthodes d'apprentissage automatique, en nous basant sur des métriques de qualité. Le but étant d'apporter une assistance aux développeurs et de les prévenir sur les valeurs essentielles des indicateurs de qualité qui pourraient suggérer l'apparition éventuelle d'un problème de sécurité à venir dans le logiciel. Ainsi, nous avons réalisé une étude empirique sur sept écosystèmes de logiciels principalement basée sur deux étapes. La première étape concerne la collecte des fichiers de bugs de sécurité et le calcul des métriques de qualité correspondantes, tandis que la deuxième étape est relative à la prédiction de ces bugs de sécurité logicielle à l'aide de onze modèles d'apprentissage automatique. Nos principaux résultats montrent que :

CountStmtExe, SumCyclomaticModified, AvgCyclomaticModified, CountDeclClass et SumCyclomatic sont les cinq principales métriques qui sont le plus liées à la présence de vulnérabilités de sécurité dans les systèmes logiciels.

La médiane des fichiers contenant des bugs de sécurité est, en moyenne, trois fois plus élevée que celle des fichiers sans bugs de sécurité.

La différence entre les valeurs des fichiers contenant des bugs de sécurité (buggy=1) et les fichiers sans bugs de sécurité (buggy=0) a été confirmée par le test U de Mann-Whitney, qui a donné des p-valeurs significatives.

Parmi les modèles évalués, XGBoost s'est démarqué avec les meilleures performances, suivi de près par Random Forest, AdaBoost et le perceptron multicouche (MLP), qui ont tous montré des résultats comparables. Les arbres de décision se sont également bien comportés.

En revanche, des modèles comme Naive Bayes et Quadratic Discriminant Analysis (QDA) ont présenté des performances nettement inférieures, révélant des limites significatives pour ces méthodes dans la prédiction des bugs de sécurité.

BIBLIOGRAPHIE

Aggarwal, A. & Jalote, P. (2006). Integrating static and dynamic analysis for detecting vulnerabilities. Dans *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, Vol. 1, pp. 343–350. IEEE.

Alenezi, M. & Zarour, M. (2020). On the relationship between software complexity and security. *arXiv preprint arXiv :2002.07135*.

Alves, H., Fonseca, B. & Antunes, N. (2016). Software metrics and security vulnerabilities : dataset and exploratory study. Dans *2016 12th European Dependable Computing Conference (EDCC)*, pp. 37–44. IEEE.

Baca, D., Petersen, K., Carlsson, B. & Lundberg, L. (2009). Static Code Analysis to Detect Software Security Vulnerabilities. Dans *Conference on Availability, Reliability and Security*.

Barrak, A., Laverdière, M.-A., Khomh, F., An, L. & Merlo, E. (2018). Just-in-Time Detection of Protection-Impacting Changes on WordPress and MediaWiki. Dans *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, CASCON '18, p. 178–188., USA. IBM Corp.

Blagus, R. & Lusa, L. (2013). SMOTE for High-Dimensional Class-Imbalanced Data. *BMC bioinformatics*, p. 106. doi: 10.1186/1471-2105-14-106

Brownlee, J. (2014). Machine learning mastery. *URL : http ://machinelearningmastery.com/discover-feature-engineering-howtoengineer-features-and-how-to-getgood-at-it*.

Camilo, F., Meneely, A. & Nagappan, M. (2015). Do bugs foreshadow vulnerabilities ? a study of the chromium project. Dans *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 269–279. IEEE.

Chen, T. & Guestrin, C. (2016). XGBoost : A Scalable Tree Boosting System. Dans *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pp. 785–794., New York, NY, USA. ACM. doi: 10.1145/2939672.2939785

Clemente, C. J., Jaafar, F. & Malik, Y. (2018). Is Predicting Software Security Bugs Using Deep Learning Better Than the Traditional Machine Learning Algorithms ? Dans *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 95–102.

Couto, C., Silva, C., Valente, M. T., Bigonha, R. & Anquetil, N. (2012). Uncovering causal relationships between software metrics and bugs. Dans *2012 16th European Conference on Software Maintenance and Reengineering*, pp. 223–232. IEEE.

El Emam, K., Melo, W. & Machado, J. C. (2001). The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software*, *56*(1), 63–75.

Fehrer, T., Lozoya, R. C., Sabetta, A., Nucci, D. D. & Tamburri, D. A. (2024). Detecting Security Fixes in Open-Source Repositories using Static Code Analyzers. Dans *28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024)*, pp. June 18–21. ACM. doi: 10.11453661167.3661217

Ferenc, R., Bán, D., Grósz, T. & Gyimóthy, T. (2020). Deep learning in static, metric-based bug prediction. *Array*, p. 100021.

Ganesh, S., Palma, F. & Olsson, T. (2022). Are Source Code Metrics "Good Enough" in Predicting Security Vulnerabilities ? *Data*, *7*, 127. Publisher's Note : MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations, doi: 10.3390/data7090127

Grichi, M., Abidi, M., Guéhéneuc, Y.-G. & Khomh, F. (2019). State of Practices of Java Native Interface. Dans *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, CASCON '19, p. 274–283., USA. IBM Corp.

Grichi, M., Abidi, M., Jaafar, F., Eghan, E. E. & Adams, B. (2020). On the Impact of Interlanguage Dependencies in Multilanguage Systems Empirical Case Study on Java Native Interface Applications (JNI). *IEEE Transactions on Reliability*, pp. 1–13. doi: 10.1109/TR.2020.3024873

Gupta, A., Suri, B., Kumar, V. & Jain, P. (2021). Extracting rules for vulnerabilities detection with static metrics using machine learning. *International Journal of System Assurance Engineering and Management*, *12*(1), 65–76.

Hammouri, A., Hammad, M., Alnabhan, M. & Alsarayrah, F. (2018). Software bug prediction using machine learning approach. *International Journal of Advanced Computer Science and Applications*, *9*(2), 78–83.

Hemmati, H. (2024). Predicting Security Vulnerabilities in Java-Based Systems Using Static Source Code Metrics and Machine Learning. *The Journal of Systems and Software*, *217*, 112179. Available online 6 August 2024, Corresponding author at York University, Toronto, Ontario, Canada. E-mail : hemmati@yorku.ca, doi: 10.1016/j.jss.2024.112179

Hollander, M., Wolfe, D. A. & Chicken, E. (2013). *Nonparametric statistical methods*, Vol. 751. John Wiley & Sons.

Immaculate, S. D., Begam, M. F. & Floramary, M. (2019). Software bug prediction using supervised machine learning algorithms. Dans *2019 International Conference on Data Science and Communication (IconDSC)*, pp. 1–7. IEEE.

Jayanthi, R. & Florence, L. (2019). Software defect prediction techniques using metrics based on neural network classifier. *Cluster Computing*, *22*(1), 77–88.

Jiang, Y., Cuki, B., Menzies, T. & Bartlow, N. (2008). Comparing Design and Code Metrics for Software Quality Prediction. Dans *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering*, PROMISE '08, p. 11–18., New York, NY, USA. Association for Computing Machinery. doi: 10.1145/1370788.1370793

Kursa, M. B. & Rudnicki, W. R. (2010). Feature Selection with the Boruta Package. *Journal of Statistical Software*, *36*(11), 1–13. doi: 10.18637/jss.v036.i11

Li, J., He, P., Zhu, J. & Lyu, M. R. (2017). Software defect prediction via convolutional neural network. Dans *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 318–328. IEEE.

Maletic, J. I. & Marcus, A. (2001). Supporting program comprehension using semantic and structural information. Dans *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pp. 103–112.

Manjula, C. & Florence, L. (2019). Deep neural network based hybrid approach for

software defect prediction using software metrics. *Cluster Computing*, *22*(4), 9847–9863.

Meléndez, R., Giraldo, R. & Leiva, V. (2020). Sign, Wilcoxon and Mann-Whitney tests for functional data : An approach based on random projections. *Mathematics*, *9*(1), 44.

Misra, S. C. & Bhavsar, V. C. (2003). Relationships between selected software measures and latent bug-density : Guidelines for improving quality. Dans *International Conference on Computational Science and Its Applications*, pp. 724–732. Springer.

Neuhaus, S., Zimmermann, T., Holler, C. & Zeller, A. (2007). Predicting Vulnerable Software Components. Dans *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, p. 529–540., New York, NY, USA. Association for Computing Machinery.

Ouedraogo, M., Savola, R. M., Mouratidis, H., Preston, D., Khadraoui, D. & Dubois, E. (2013). Taxonomy of quality metrics for assessing assurance of security correctness. *Software Quality Journal*, *21*(1), 67–97.

Pandey, S. K., Mishra, R. B. & Tripathi, A. K. (2020). BPDET : An effective software bug prediction model using deep representation and ensemble learning techniques. *Expert Systems with Applications*, *144*, 113085.

Punitha, K. & Chitra, S. (2013). Software defect prediction using software metrics-A survey. Dans *2013 International Conference on Information Communication and Embedded Systems (ICICES)*, pp. 555–558. IEEE.

Rahman, F. & Devanbu, P. (2013, 05). How, and why, process metrics are better. pp. 432–441. doi: 10.1109/ICSE.2013.6606589

Rhmann, W., Pandey, B., Ansari, G. & Pandey, D. (2020). Software fault prediction based on change metrics using hybrid algorithms : An empirical study. *Journal of King Saud University-Computer and Information Sciences*, *32*(4), 419–424.

Scandariato, R., Walden, J., Hovsepyan, A. & Joosen, W. (2014). Predicting Vulnerable Software Components via Text Mining. *IEEE Transactions on Software Engineering*, *40*(10), 993–1006.

Schugerl, P., Rilling, J. & Charland, P. (2008). Mining bug repositories–a quality assessment. Dans *2008 International Conference on Computational Intelligence for Modelling Control & Automation*, pp. 1105–1110. IEEE.

Sharma, M., Kumari, M. & Singh, V. (2013). Understanding the meaning of bug attributes and prediction models. Dans *Proceedings of the 5th IBM Collaborative Academia Research Exchange Workshop*, pp. 1–4.

Shelke, M. S., Deshmukh, P. R. & Shandilya, V. K. (2017). A review on imbalanced data handling using undersampling and oversampling technique. *Int J Recent Trends in Eng & Res*, *3*, 444–449.

Shin, Y. & Williams, L. (2008). An empirical model to predict security vulnerabilities using code complexity metrics. Dans *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pp. 315–317.

Tan, L., Liu, C., Li, Z., Wang, X., Zhou, Y. & Zhai, C. (2014, 6). Bug Characteristics in Open Source Software. *Empirical Softw. Engg.*, p. 1665–1705.

Tantithamthavorn, C., Hassan, A. E. & Matsumoto, K. (2018). The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *IEEE Transactions on Software Engineering*, *46*(11), 1200–1219.

Tantithamthavorn, C., McIntosh, S., Hassan, A. E. & Matsumoto, K. (2018). The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering*, *45*(7), 683–711.

Wei, Y., Sun, X., Bo, L., Cao, S., Xia, X. & Li, B. (2021). A comprehensive study on security bug characteristics. *Journal of Software : Evolution and Process*, *33*(10), e2376.

Xiao, Y., Keung, J., Bennin, K. E. & Mi, Q. (2019). Improving bug localization with word embedding and enhanced convolutional neural networks. *Information and Software Technology*, *105*, 17–29.

Zaman, S., Adams, B. & Hassan, A. E. (2011). Security versus Performance Bugs : A Case Study on Firefox. Dans *Proceedings of the 8th Working Conference on Mining Software*

Repositories, MSR '11, p. 93–102., New York, NY, USA. Association for Computing Machinery.

Zheng, W., Cheng, J., Wu, X., Sun, R., Wang, X. & Sun, X. (2022). Domain knowledge-based security bug reports prediction. *Knowledge-Based Systems*, *241*, 108293.