UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN MATHÉMATIQUES ET INFORMATIQUE
APPLIQUÉES

PAR
ISAAC OGOUHOLA

PRÉDICTION DES BOGUES DANS LES APPLICATIONS MOBILES : UNE APPROCHE BASÉE SUR LES MÉTRIQUES LOGICIELLES ET L'APPRENTISSAGE AUTOMATIQUE

SEPTEMBRE 2021

Remerciements

En premier lieu, je remercie Dieu de m'avoir permis de mener ce travail de recherche à terme.

Je tiens à remercier mon directeur de recherche, Monsieur Fadel TOURE, qui a supervisé mon travail avec une attention particulière. Je le remercie pour son encadrement, sa disponibilité et la pertinence de ses remarques.

Merci également à tous les professeurs du département de Mathématiques et Informatique de l'UQTR avec qui j'ai suivi des cours tout au long de ma formation de maîtrise. Je m'en voudrais de ne pas remercier spécialement les différents responsables administratifs qui n'ont ménagé aucun effort pour la réussite de mon objectif principal à l'UQTR.

Merci également à mes parents qui m'ont apporté tout leur soutien afin que je puisse en arrivé là aujourd'hui, Dieu vous le rende au-delà de vos attentes.

Je ne saurais remercier assez ma femme et ma fille qui m'ont apporté leur soutien; surtout moral pour pouvoir avancer dans mes études. Merci à ma femme pour la patience dont elle a fait preuve jusque-là. A ma fille qui a grandi loin de son papa. A chaque fois que je pense à elles, je retrouve la force d'avancer et d'approcher l'objectif.

Ce mémoire n'aurait pas vu le jour sans la contribution de tout un chacun de vous citer précédemment. Ma plus profonde reconnaissance pour votre soutien indéfectible.

RESUME

Les applications mobiles sont de plus en plus présentes dans notre quotidien. Pendant que certaines sont de simples loisirs d'autres, en revanche plus larges, jouent des rôles plus critiques. Ainsi, la taille et le niveau d'exigences de qualité augmentent. Les développeurs mobiles sont donc appelés à faire preuve de vigilance et de professionnalisme dans les processus d'assurances qualité lors des phases de développement et durant le cycle de vie du logiciel, mis en place pour garantir une application mobile de qualité sur le marché des applications en ligne. Dans ce marché, les usagers sont friands des interfaces simples et intuitives. Se faisant, ils transfèrent la complexité sur le code engendrant un risque de faute élevé. Dans ce cadre, les métriques de code source logiciel peuvent jouer un rôle important dans la prédiction de bogues et la maintenance.

Le présent travail s'inscrit dans une démarche d'assurance qualité du logiciel plus précisément dans la prédiction de fautes dans les applications mobiles. L'approche consiste à utiliser des métriques logicielles existantes et à adapter certaines aux spécificités du mobile. Afin de mieux prédire et de classifier les bogues, nous utiliserons les algorithmes d'apprentissage automatique sur différentes métriques OO calculées à partir des codes sources des applications mobiles natives destinées à l'OS Android, écrite en Java. Nous utiliserons des méthodes de prédictions connues comme base de comparaisons puis nous évaluerons les performances de chacune d'elle à l'aide de la Matrice de Confusion, la Courbe ROC, la Précision et le Rappel.

ABSTRACT

Mobile applications are more and more present in our daily lives. While some are simple hobbies, others, on the other hand broader, play more critical roles. The size of a mobile application is often a determining factor in the various bogues that it can generate when published. Mobile developers are therefore called upon to exercise vigilance and professionalism in the bogue detection processes implemented to guarantee a competitive mobile application in smartphone markets. Mobile user used to interact with simple and easy app interfaces. Doing so, they transfer the complexity to the source code, increasing the risk of fault. In such a context, source code metrics can play an important role in bogues prediction and software maintenance phases.

The current work uses existing software metrics and adapting some to mobile specificities. In order to better predict and classify bogues, we use machine learning algorithms after descriptive statistics of the various metrics calculated from the source codes of Android mobile applications. We used known prediction methods as comparison basis and then, we evaluate each one using the Confusion Matrix, ROC Curve, Accuracy and Recall.

Table des matières

CHAPITRE 1 : INTRODUCTION 13

1.1 Introduction 13

1.3 Problématique 15

1.4 Objectifs 16

1.5 Organisation du mémoire 16

CHAPITRE 2 : ETAT DE L'ART 18

2.1 Introduction 18

2.2 Revue de littérature 18

2.2.1 Prédiction de fautes dans un système logiciel 18

2.2.1 Prédiction de fautes dans une application mobile Android 23

CHAPITRE 3 : METRIQUES LOGICIELLES 25

3.1 Contexte 25

3.2 Définition 25

3.3 Métriques de taille 26

3.4 Métriques de couplage 26

3.5 Métriques de complexité 26

3.6 Métriques d'héritage 27

CHAPITRE 4: APPRENTISSAGE AUTOMATIQUE 29

4.1 Contexte 29

4.2 Définition 29

4.3 Apprentissage non supervisé 29

4.4 Apprentissage supervisé 31

4.5 Algorithmes d'apprentissage automatique 31

CHAPITRE 5 : COLLECTE DE DONNÉES ET METHODES D'ANAYSE ..36

5.1 Outils de collecte 36

5.2 Systèmes analysés 37

5.2.1 Description des systèmes 37

5.2.2 Procédure de collecte 39

5.2.3 Méthodes d'analyse 40

5.2.4 Statistiques descriptives 41

5.5 Évaluation de la performance des modèles d'apprentissage automatique 43

5.5.1 Matrice de confusion 43

5.5.3 Le rappel 44

5.5.4 La précision 44

5.5.2 Courbe ROC et AUC 44

CHAPITRE 6 : RESULTATS ET DISCUSSION 46

6.1 Présentation des résultats 46

6.1.1 Modèle de prédiction M_1 46

6.1.2 Modèle de prédiction M2 50

6.1.3 Modèle de prédiction M3 55

6.1.4 Modèle de prédiction M_4 60

6.1.5 Modèle de prédiction M_5 65

6.1.6 Modèle de prédiction M6 70

6.1.7 Modèle de prédiction M7 75

6.1.8 Modèle de prédiction M_8 80

6.1.9 Modèle de prédiction M_9 85

6.2 Synthèse des résultats 90

6.3 Menace pour la validité 92

6.3.1 Validité interne 92

6.3.2 Validité externe 93

6.3.3 Validité de construction 93

CONCLUSION 94

REFERENCES BIBLIOGRAPHIQUES 96

Liste des figures

Figure 1: Courbe ROC modèle de prédiction modèle M1 46

Figure 2: Courbe ROC modèle de prédiction modèle M2 51

Figure 3:Courbes ROC modèle de prédiction M3 56

Figure 4: Courbe ROC modèle de prédiction M4 61

Figure 5:Courbe ROC modèle de prédiction M5 66

Figure 6:Courbes ROC modèle de prédiction M6 71

Figure 7:Courbes ROC du modèle de prédiction M7 76

Figure 8: Courbes ROC modèle de prédiction M8 81

Figure 9: Courbes ROC modèles de prédiction M9 86

Liste des tableaux

Tableau 1 : Statistiques descriptives système Android PDF Viewer 41

Tableau 2 : Statistiques descriptives système Quran 42

Tableau 3 : Statistiques descriptives système Password Store Master 42

Tableau 4: Modèle de matrice de confusion 44

Tableau 5: Résultats du modèle de prédiction M_1 46

Tableau 6: Matrice de confusion des réseaux de neurones pour le modèle M1 47

Tableau 7: Matrice de confusion Random Forest pour le modèle M1 48

Tableau 8: Matrice de confusion KNN pour le modèle M1 48

Tableau 9: Matrice de confusion Naïf Bayésien pour le modèle M1 49

Tableau 10:Matrice de confusion SVM pour le modèle M1 49

Tableau 11: Matrice de confusion Régression Logistique pour le modèle M1 .. 50

Tableau 12:Résultats du modèle de prédiction M2 51

Tableau 13: Matrice de confusion Réseaux de Neurones pour le modèle M2 ... 52

Tableau 14:Matrice de confusion Random Forest pour le modèle M2 52

Tableau 15:Matrice de confusion KNN pour le modèle M2 53

Tableau 16: Matrice de confusion Naïf Bayésien pour le modèle M2 54

Tableau 17: Matrice de confusion SVM pour le modèle M2 54

Tableau 18: Matrice de confusion Régression Logistique pour le modèle M2 .. 55

Tableau 19:Résultats du modèle de prédiction M3 56

Tableau 20:Matrice de confusion Réseaux de Neurones pour le modèle M3 57

Tableau 21:Matrice de confusion Random Forest pour le modèle M3 57

Tableau 22:Matrice de confusion KNN pour le modèle M3 58

Tableau 23:matrice de confusion Naïf Bayésien pour le modèle M3 59

Tableau 24: Matrice de confusion SVM pour le modèle M3 59

Tableau 25: Matrice de confusion Régression logistique pour le modèle M3 ... 60

Tableau 26: Résultats du modèle de prédiction M4 ... 61

Tableau 27: Matrice de confusion réseaux de neurones pour le modèle M4 ... 62

Tableau 28: Matrice de confusion Random Forest pour le modèle de prédiction M4 ... 62

Tableau 29: Matrice de confusion KNN pour le modèle de prédiction M4 ... 63

Tableau 30:Matrice de confusion Naïf Bayésien pour le modèle de prédiction M4 ... 64

Tableau 31: Matrice de confusion SVM pour le modèle de prédiction M4 ... 64

Tableau 32: Matrice de confusion régression logistique pour le modèle de prédiction M4 ... 65

Tableau 33:Résultats du modèle de prédiction M5 ... 66

Tableau 34:Matrice de confusion Réseaux de Neurones pour le modèle de prédiction M5 ... 67

Tableau 35: Matrice de confusion Random Forest pour le modèle de prédiction M5 ... 67

Tableau 36:Matrice de confusion KNN pour le modèle de prédiction M5 ... 68

Tableau 37: Matrice de confusion Naïf Bayésien pour le modèle de prédiction M5 ... 69

Tableau 38:Matrice de confusion SVM pour le modèle de prédiction M5 ... 69

Tableau 39:Matrice de confusion Régression Logistique pour le modèle de prédiction M5 ... 70

Tableau 40:Résultats du modèle de prédiction M6 ... 71

Tableau 41:Matrice de confusion Réseaux de Neurones pour le modèle de prédiction M6 ... 72

Tableau 42:Matrice de confusion Random Forest pour le modèle de prédiction M6 ... 72

Tableau 43:Matrice de confusion KNN pour le modèle de prédiction M6 ... 73

Tableau 44:Matrice de confusion Naïf Bayésien pour le modèle de prédiction M6 74

Tableau 45:Matrice de confusion SVM pour le modèle de prédiction M6 74

Tableau 46: Matrice de confusion Régression logistique pour le modèle de prédiction M6 75

Tableau 47: Résultats du modèle de prédiction M7 76

Tableau 48: Matrice de confusion Réseaux de Neurones pour le modèle de prédiction M7 77

Tableau 49:Matrice de confusion Random Forest pour le modèle de prédiction M7 77

Tableau 50:Matrice de confusion KNN pour le modèle de prédiction M7 78

Tableau 51: Matrice de confusion Naïf Bayésien pour le modèle de prédiction M7 79

Tableau 52:Matrice de confusion SVM pour le modèle de prédiction M7 79

Tableau 53:Matrice de confusion Régression logistique pour le modèle de prédiction M7 80

Tableau 54:Résultats modèle de prédiction M8 81

Tableau 55:Matrice de confusion Réseaux de Neurones pour le modèle de prédiction M8 82

Tableau 56:Matrice de confusion Random Forest pour le modèle de prédiction M8 82

Tableau 57:Matrice de confusion KNN pour le modèle de prédiction M8 83

Tableau 58: Matrice de confusion Naïf Bayésien pour le modèle de prédiction M8 84

Tableau 59: Matrice de confusion SVM pour le modèle de prédiction M8 84

Tableau 60:Matrice de confusion Régression logistique pour le modèle de prédiction M8 85

Tableau 61: Résultats modèle de prédiction M9 86

Tableau 62:Matrice de confusion Réseaux de Neurones pour le modèle de prédiction M9 87

Tableau 63:Matrice de confusion Random Forest pour le modèle de prédiction M9 87

Tableau 64: Matrice de confusion KNN pour le modèle de prédiction M9 88

Tableau 65: Matrice de confusion Naïf Bayésien pour le modèle de prédiction M9 88

Tableau 66:Matrice de confusion SVM pour le modèle de prédiction M9 89

Tableau 67: Matrice de confusion Régression logistique pour le modèle de prédiction M9 90

Tableau 68: Synthèse des résultats des modèles de prédiction 91

CHAPITRE 1 : INTRODUCTION

1.1 Introduction

Au cours des dernières années, les appareils mobiles, tels que les smartphones et les tablettes, sont devenus très populaires [1]. L'une des raisons de leur popularité est le nombre toujours croissant d'applications mobiles disponibles, ce qui rend les entreprises et leurs produits plus accessibles aux utilisateurs finaux. Les développeurs d'applications mobiles peuvent recourir à plusieurs outils, cadres logiciels et services pour développer et garantir la qualité de leurs applications [2]. Cependant, il est toujours un fait que les erreurs se glissent dans les applications déployées, ce qui est le propre de tout logiciel et peut considérablement réduire la qualité des applications, la réputation des développeurs et des entreprises.

Les tests constituent une phase importante du cycle de vie du logiciel car ils aident à identifier les bogues dans les systèmes avant qu'ils ne soient livrés aux utilisateurs finaux. Il existe de nombreux types de tests dans le cycle de vie du logiciel. Nous pouvons les regrouper en trois catégories à savoir : le test unitaire, le test d'intégration et le test de validation. Le test unitaire est une façon de tester chaque module du logiciel séparément. Le test unitaire est également appelé test de composant. Dans la programmation orientée objets, l'élément sous test peut être alors une méthode spécifique, une classe ou un groupe de classes formant un composant. Il permet de s'assurer que la logique du programme est respectée et que le composant a un comportement conforme à sa spécification [3]. Les tests unitaires sont des tests dits en isolation car les composants sont testés individuellement les uns des autres, et ce dans n'importe quel ordre. Un test d'intégration est un test dans lequel les composants logiciels, les composants matériels ou les deux sont combinés pour tester et évaluer leurs interactions [4]. Un test d'intégration quant à lui considère des composants déjà

testés unitairement. Alors que le test unitaire se focalise sur le comportement d'un composant isolé, le test d'intégration s'attache à vérifier le bon comportement de l'assemblage de plusieurs composants. Ensuite, vient le test de validation qui consiste à s'assurer que le système entièrement intégré respecte les exigences spécifiées [4]. Ce test permet de déterminer si un système satisfait ou non à ses critères d'acceptation et permet au client d'accepter ou non le système [4]. C'est donc la dernière étape avant la mise en œuvre opérationnelle d'un système. Il est d'abord effectué par l'équipe de développement, puis par un panel d'utilisateurs finaux à l'aide de scénarios réels. Par conséquent, l'apparition d'applications mobiles s'est accompagnée d'un nouvel écosystème où les outils de test traditionnels ne sont pas toujours appliqués [5]: les interactions complexes des utilisateurs (par exemple, balayer, pincer, etc) doivent être soutenus par des tests d'interface spécifique. Les applications doivent tenir compte des appareils aux ressources limitées (source d'alimentation limitée, capacité de traitement inférieure). Les développeurs doivent prendre en compte un nombre toujours croissant d'appareils ainsi que des versions de système d'exploitation. Les applications suivent généralement une stratégie de déploiement (publication) continue [6] basée sur le temps hebdomadaire/bihebdomadaire qui crée des contraintes de temps critiques dans les tâches de test. De plus, les tests manuels ne sont pas une approche idéale pour garantir la qualité des logiciels et devraient être remplacés par des techniques automatisées [7].

De telles applications Android enrichissent notre vie, mais présentent également de nouveaux défis liés par exemple à la consommation d'énergie, à la sécurité des données personnelles. Bien qu'Android n'existe que depuis quelques années, de nombreux chercheurs se sont concentrés sur divers problèmes liés aux applications tournant sous cet environnement. Leurs différents objectifs, tels que l'amélioration des performances des applications, la

protection des données confidentielles dans les téléphones intelligents contre les applications malveillantes et la détection des bogues potentiels et leur sévérité, les poussent à proposer différentes techniques et à mettre en œuvre diverses approches d'assurance qualité. C'est dans ce dernier volet que s'inscrit notre travail.

Dans notre travail, nous analysons les métriques de codes calculées à partir de trois différentes applications mobiles Android pour répondre au besoin de la prédiction des bogues dans chacune d'elle. En effet, la prédiction permet d'orienter l'effort de test afin de découvrir les bogues très tôt dans les phases de développement, ce qui coûte moins cher à fixer [8].

1.3 Problématique

Les applications mobiles Android sont utilisées partout dans le monde et dans presque tous les domaines. Ces dernières années, Android est devenu le système d'exploitation le plus populaire pour les appareils mobiles (smartphone, tablettes) [9]. Dans le même temps, les marchés des applications pour Android se développent très rapidement en raison du développement continu des applications Android. D'après les recherches effectuées par Statista Research Department [10] en mai 2019 on estime que plus de 2,6 millions d'applications Android sont disponibles sur Google Play [11]. La satisfaction des utilisateurs provient en partie de la qualité de l'application. Les développeurs se doivent donc de mettre sur le marché des applications mobiles qui répondent aux besoins de la clientèle, faciles d'utilisation et sans bogues.

La présence des bogues dans un système logiciel présente d'énormes conséquences autant pour l'utilisateur que pour le concepteur; sachant que dans un processus logiciel, les tests occupent plus de la moitié des ressources [12], et ne garantissent pas pour autant la fiabilité totale du système.

Dans notre travail, nous avons extrait les métriques de code source de logiciels de trois applications mobiles Android avec lesquelles nous avons effectuées la prédiction des bogues en utilisant des algorithmes d'apprentissage automatique.

1.4 Objectifs

L'objectif général de notre étude est la prédiction des bogues dans les applications mobiles à partir des métriques de codes sources. Pour atteindre cet objectif, nous sommes passés par 3 étapes clés:

- Extraction les métriques des codes sources des applications mobiles
- Construction des modèles de prédiction de bogues à l'aide des algorithmes d'apprentissage automatique,
- Évaluation des performances des algorithmes d'apprentissage automatique entrainés sur des données de certaines métriques de code source afin de prédire les classes logicielles susceptibles de contenir des bogues.

1.5 Organisation du mémoire

Notre travail est divisé en 6 chapitres qui présentent chacun un axe spécifique du mémoire en plus de la conclusion.

Le premier est une introduction de la problématique portant sur la prédiction des bogues relative au développement des applications mobiles et la solution envisagée que nous proposons dans le cadre de ce projet.

Le deuxième fait référence à des récents articles qui traitent des sujets de la prédiction des bogues dans un système logiciel en général et mobile en particulier.

Le troisième chapitre présente de façon beaucoup plus détaillée les différentes métriques logicielles utilisées dans le cadre de ce travail.

Le quatrième chapitre quant à lui, aborde les techniques d'apprentissage automatique en général avec un accent particulier sur les différents algorithmes d'apprentissage automatique utilisés pour créer des modèles de prédiction que nous avons utilisés.

Ensuite, dans le cinquième chapitre nous parlons des données collectées pour atteindre notre objectif ainsi que la méthodologie utilisée.

Nous présentons par la suite dans le sixième chapitre, les différents résultats de nos expérimentations ainsi que le processus de comparaison de la performance des modèles, application par application et puis nous pointons les menaces pour la validité.

Finalement, nous conclurons ce mémoire avec une ouverture sur des travaux futurs.

CHAPITRE 2 : ETAT DE L'ART

2.1 Introduction

Le présent mémoire fait appel à plusieurs domaines de recherche en génie logiciel. Nous décrivons, dans cette section l'état des recherches et quelques publications en lien avec notre travail. Nous présentons sommairement l'état des recherches et les principales publications connexes à notre sujet tout en mettant en évidence les différentes approches utilisées et leur adaptabilité au mobile.

2.2 Revue de littérature

Pour mesurer les caractéristiques d'un logiciel, plusieurs recherches ont été effectuées. Les chercheurs ont réalisé des études pour trouver des manières de mesurer ces caractéristiques de bas niveau. Les travaux de Chidamber et Kemerer [13], Brito et Carapuça [14], Martin [15] sont quelques exemples de ces recherches. Ainsi, le domaine de la métrologie logicielle est très vaste. Un grand nombre de métriques ont été décrites et de nouvelles sont fréquemment proposées.

2.2.1 Prédiction de fautes dans un système logiciel

Une faute est une étape, un traitement, ou une définition de donnée incorrecte dans un programme [16]. La défaillance représente l'incapacité d'un logiciel ou d'un composant d'un système logiciel à réaliser ses spécifications avec les performances requises. La défaillance peut être due au matériel ou au logiciel [17].

Hall et al. [18] ont analysé 208 études publiées pendant onze ans sur des modèles de prédiction de bogues sur la base du code source; ils ont noté que la performance des modèles dépendait des données sélectionnées, des variables ou métriques indépendantes et des techniques de modélisation. Zimmermann et al.

[19] ont révélé que la prédiction de bogues entre projets est très importante pour les logiciels ayant des données évolutives insuffisantes ou peu nombreuses. Par conséquent, pour de tels projets, ils ont proposé de construire un modèle en utilisant les données évolutives issues d'autres projets similaires.

En investiguant des moyens de prédiction toujours plus précoces, des chercheurs se sont intéressés aux liens entre les métriques de spécifications et les fautes. Dans ce cadre, Kaur et al. [20] ont combiné un groupe de métriques de spécifications (métriques issues des spécifications logicielles et déterminées beaucoup plus tôt dans le cycle de développement) à un groupe de métriques de code source collectées plus tard dans le cycle. Ils ont montré qu'il était possible de prédire assez tôt, grâce aux métriques issues des spécifications, les classes sujettes aux fautes. En plus, en combinant ces métriques avec celles obtenues plus tard à partir du code source, l'étude montre que l'on peut nettement améliorer les performances des modèles de prédiction. Les auteurs recourent à la technique du clustering (classification) sur des données issues des dépôts du MDP (Metrics Data Program) [21] et qui concernent des systèmes écrits en C++. Ils produisent un modèle de prédiction plus performant que ceux basés sur l'un ou l'autre des deux groupes de métriques pris séparément.

Briand et al. [22] ont fait une étude empirique de la suite de métriques de conception orientée objet (OO) introduites dans les travaux de CK [13]. Leur objectif était d'évaluer ces métriques en tant que prédicteurs des classes sujettes aux fautes et par conséquent, de déterminer si elles peuvent être utilisées comme indicatrices de qualité précoces. Pour effectuer la validation avec précision, ils ont collecté des données sur le développement de huit systèmes de gestion de l'information de taille moyenne basés sur des exigences identiques. Les huit projets ont été développés en utilisant un modèle de cycle de vie séquentiel, une méthode d'analyse/conception OO bien connue et le langage de programmation C++. Sur la base d'une analyse empirique et quantitative, les avantages et les

inconvénients de ces mesures OO sont discutés. Plusieurs des métriques OO de Chidamber et Kemerer semblent être utiles pour prédire la propension aux fautes des classes au cours des premières phases du cycle de vie.

Aggarwal et al. [23] étendent l'étude de Briand et al. [22] à 12 systèmes écrits en JAVA, dans un environnement contrôlé. Après une analyse par composantes principales qui leur a permis de déceler le recoupement des informations capturées par les métriques, ils construisent des modèles logistiques univariés sur les données et parviennent à une exactitude de prédiction de l'ordre de 90 %. Ils notent, aussi, que les métriques de couplage sont particulièrement corrélées à la probabilité de détection de fautes dans les classes logicielles.

Rathore et al. [24] investiguent de leur côté la capacité de prédiction des modèles construits à partir des métriques orientées objet (associées aux caractéristiques du paradigme orienté objet - POO) telles que le couplage, la cohésion, la complexité, l'héritage et la taille afin de prédire les classes susceptibles de contenir des fautes. Les auteurs utilisent la régression logistique et l'analyse de la courbe ROC (Receiver Operating Characteristic) afin de sélectionner les métriques les plus performantes. Ces dernières servent ensuite de base d'apprentissage pour les algorithmes d'intelligence artificielle. Après avoir testé cette approche sur des données de PROMISE (Predictor Models In Software Engineering) [25], les auteurs concluent que les métriques orientées objet liées au couplage et à la complexité produisent des modèles significativement plus précis (pour la prédiction de fautes) que les autres métriques.

D'autres études font appel aux méthodes d'apprentissage automatique pour construire des prédicteurs de fautes à partir de (l'information sur) l'historique des fautes et/ou des métriques logicielles. C'est dans ce cadre que s'inscrit

l'objectif de notre étude. Ainsi, Gyimòthy et al. [26] se sont penchés sur le cas des logiciels open source. Ils bâtissent des modèles par apprentissage automatique directement sur les métriques CK [13]. L'analyse empirique est effectuée sur 7 versions de la suite logicielle Mozilla (le navigateur et le mailer). Elle montre, d'une part, qu'il est possible de prédire les classes les plus prédisposées aux fautes à partir de l'historique des attributs orientés objet et, d'autre part, que le couplage et la complexité présentent des résultats particulièrement intéressants. L'analyse de l'évolution de la suite de logiciels produite par la suite montre que les corrélations entre les fautes et les métriques CK [13] persistent sur plusieurs versions des logiciels open source.

En 2005 MMT. Thwin et T. Quah [27] ont fait une étude sur l'application des réseaux de neurones à la prédiction de la qualité des logiciels à l'aide de métriques orientées objet. Dans cette étude, deux types d'investigations ont été effectuées. La première portait sur la prédiction du nombre de défauts dans une classe et la seconde portait sur la prédiction du nombre de lignes modifiées par classe. Deux modèles de réseaux de neurones sont utilisés, ce sont le réseau de neurones de Ward et le réseau de neurones de régression générale (GRNN). Les métriques de conception orientées objet liées à l'héritage, la complexité, la cohésion, le couplage et l'allocation de mémoire sont utilisées comme variables indépendantes. Le modèle de réseau GRNN s'avère plus précis que le modèle de réseau Ward.

G.Sharma, S. Sharma, S. Gujral [28] ont élaboré en 2015 un dictionnaire de termes critiques pour la prédiction de la sévérité des bogues et ont utilisé plusieurs modèles comme les K-plus proches voisins et la classification multi-classes de Bayes. Cette approche de création d'un dictionnaire de termes critiques spécifiant la gravité à l'aide de deux méthodes de sélection de caractéristiques différentes, à savoir: le gain d'informations et le chi-carré. La classification des rapports de bogue est effectuée à l'aide des algorithmes Naïve

Bayes Multinomial (NBM) et K-plus proche voisin (KNN). Dans cette recherche, des dictionnaires spécifiques aux composants sont créés à partir de quatre composants d'Éclipse. Ces dictionnaires utilisent les 125 principaux termes contributifs construit à l'aide de deux méthodes de sélection de caractéristiques à savoir: Info-gain et Chi-carré. L'ensemble des termes du dictionnaire alimente deux algorithmes d'apprentissage automatique largement utilisés (Naïve Bayésien et KNN) pour la tâche de classification. Les performances sont analysées en termes de précision et d'exactitude. Il a été conclu que dans les conditions expérimentales utilisées, KNN performe mieux pour la classification de la gravité des bogues. L'algorithme KNN utilisait la distance euclidienne.

S. Bouktif propose une approche pour l'amélioration de la prédiction de la qualité du logiciel par combinaison et adaptation de modèles (CAMP) [29]. C'est une approche inspirée de la technique de mélange d'experts qui, à partir d'un ensemble de modèles existants et d'un échantillon reflétant les spécificités d'un environnement particulier (un contexte logiciel dans une organisation particulière), dérive le modèle le plus adapté possible à ce contexte. L'approche CAMP est basée sur une idée reposant sur trois principes se chargeant respectivement de la promotion des trois propriétés étudiées. Ces principes peuvent être résumés de la façon suivante : décomposer les modèles en expertises pour faciliter leur interprétation et leur manipulation, combiner les expertises pour enrichir et généraliser les modèles, adapter les expertises pour mieux convenir à un contexte spécifique. Dans le but d'évaluer l'approche CAMP, ils ont construit un environnement « semi-réel » dans lequel le contexte d'organisation est une évolution d'un vrai système logiciel (API Java), mais les modèles sont « simulés (c'est-à-dire, obtenus par apprentissage sur des données réelles). La construction de cet environnement est à double apport. En effet, elle définit d'une part, un facteur de qualité crucial pour la maintenance de logiciel

qui est la stabilité de classes dans un logiciel orienté objets. D'autre part, elle construit des modèles de stabilité de type arbres de décision et de type classificateurs bayésiens. Les résultats obtenus dans le cadre des différentes expériences montrent que les objectifs fixés pour améliorer la prédiction sont bien atteints, à savoir, la bonne capacité prédictive (exactitude), la bonne capacité de généralisation et la facilité d'interprétation du modèle dérivé par l'approche CAMP.

2.2.1 Prédiction de fautes dans une application mobile Android

Les techniques de prédiction de fautes dans une application mobile Android sont encore à leurs débuts. Linares-Vâsquez et al. ont utilisé la méthode DECOR [30] pour détecter 18 défauts de code dans des applications mobiles basées sur le langage Java Mobile Edition. Cette étude à large échelle sur 1343 applications montre que la présence des défauts de code affecte négativement les attributs de qualité du logiciel, en particulier les attributs qui sont réputés être liés à la présence de bogues, comme la complexité excessive. Ils ont aussi découvert que certains défauts sont plus courants dans certaines catégories d'applications système Java mobile.

Notons par ailleurs qu'il existe des études qui, bien que n'utilisant pas directement les fautes, fournissent des informations intéressantes sur les métriques et les structures de code pertinentes pour la détection des défauts de code. Ce volet constitue un des points fondamentaux de l'approche que nous utilisons dans notre travail pour la prédiction des bogues dans une application mobile Android.

Ainsi l'outil SAMOA utilisé dans [31] permet aux développeurs d'analyser leurs applications Android à partir du code source. Cet outil collecte des métriques comme le nombre de paquetages, de lignes de code ou encore la complexité cyclomatique. Ils ont effectué une étude sur 20 applications mobiles

et ont découvert que ces applications étaient significativement différentes des applications de bureau. En effet, les applications Android tendent à contenir moins de classes, mais utilisent intensivement des librairies externes. Le code ainsi produit serait donc plus complexe à comprendre durant les phases de maintenance et d'évolution [32].

L'utilisation de l'apprentissage automatique pour la prédiction de fautes dans les applications mobiles Android devient de plus en plus grande. Nombreuses sont les approches qui formulent le problème de prédiction de fautes en un problème de classification. En effet, les approches basées sur l'apprentissage automatique, utilisent des informations extraites soit statiquement comme dans le cadre de notre travail soit dynamiquement pour entraîner des classificateurs dans le but de détecter les classes susceptibles de contenir des fautes.

CHAPITRE 3 : METRIQUES LOGICIELLES

3.1 Contexte

Les métriques logicielles orientées objet mesurent les propriétés de bas niveaux des classes logicielles. La plupart de ces métriques sont issues de la suite de métriques OO de Chidamber et Kemerer (CK) [13]. Par ailleurs, certaines métriques présentées ci-dessous, définies initialement dans le cadre du paradigme procédural, ont été adaptées au contexte orienté objet par certains auteurs.

Dans ce travail, nous avons recouru aux métriques logicielles de code source capturant la complexité, la taille, le couplage et l'héritage afin de mesurer les propriétés des applications OO. Les sections suivantes présentent leur description.

3.2 Définition

Une métrique logicielle est une compilation de mesures issues des propriétés techniques ou fonctionnelles d'un logiciel. Il est possible de classer les métriques logicielles en trois catégories [33] :

- Maintenance applicative (appelée aussi TMA Tierce Maintenance Applicative)
- Qualité applicative
- Respect des processus de développement.

Nous les avons regroupés selon les artéfacts orientés objets qu'elles capturent.

3.3 Métriques de taille

- LOC: Cette métrique compte le nombre de lignes d'instructions dans la classe mesurée. Elle évalue la taille d'une classe. La taille est fortement liée à la complexité. Une classe de grande taille est souvent synonyme de responsabilités disparates et de forte probabilité de présence de fautes, ce qui suggère que ces classes doivent être restructurées.

3.4 Métriques de couplage

- CBO: Cette métrique appartient à la suite CK [13] et détermine, pour une classe logicielle donnée, le nombre de classes auxquelles elle est couplée et vice versa. Il s'agit d'une sommation des métriques FANIN et FANOUT. Elle évalue théoriquement l'interdépendance entre les classes du système. Briand et al. [22] ont montré qu'un couplage excessif entre les classes nuit à la modularité et constitue un obstacle à la réutilisation. Plus une classe est indépendante, plus il est facile de la réutiliser dans une autre application. Afin d'améliorer la modularité et d'améliorer l'encapsulation, le couplage des classes doit être le plus limité possible. Plus le couplage est élevé, plus les différentes parties du système sont sensibles aux modifications et plus la maintenance sera difficile. Par ailleurs, la mesure du couplage s'avère utile pour prévoir le niveau de complexité des tests des différentes parties du système.

3.5 Métriques de complexité

- **WMC**: issue de la suite CK [13]. Elle somme les complexités cyclomatiques de toutes les méthodes de la classe mesurée. La complexité cyclomatique d'une méthode est donnée par la formule de McCabe [34] comme étant le nombre de chemins linéairement indépendants qu'elle contient. WMC reflète donc la complexité globale d'une classe. Elle est liée à l'effort nécessaire au développement et à la maintenance d'une classe logicielle. Une valeur élevée

est synonyme d'un risque élevé de présence de bogues dans la classe, mais aussi d'une faible compréhensibilité du code. En tenant compte du nombre de méthodes, la complexité cyclomatique est aussi liée à une autre perspective de la taille des classes logicielles.

- **RFC**: Issue de la suite de CK [13], cette métrique compte, pour une classe donnée, le nombre de ses méthodes ainsi que le nombre de méthodes (des autres classes) appelées par la classe en réponse à un message. La métrique RFC reflète le niveau de communication potentiel entre une classe et les autres dont elle utilise les services. Cette métrique est théoriquement liée à la complexité et à la facilité de test d'une classe logicielle dans la mesure où plus une classe invoque des méthodes de diverses origines, plus ses fonctionnalités seront compliquées à vérifier. Une classe qui appelle un plus grand nombre de méthodes est considérée comme plus complexe et nécessitant plus d'effort de test.

3.6 Métriques d'héritage

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

- **NOC**: Cette métrique appartient à la suite CK [13]. Elle compte le nombre de classes immédiatement dérivées de la classe mesurée. NOC reflète (théoriquement) l'impact potentiel d'une classe sur ses descendants. Elle évalue

aussi le niveau de réutilisabilité de la classe mesurée dans la mesure où une classe ayant de nombreux enfants (classes dérivées) est très générique. Cependant, un très grand nombre de dérivations directes est signe de mauvaise conception et d'une abstraction impropre. Théoriquement, le nombre de dérivations directes impacte fortement la hiérarchie des classes et peut nécessiter un effort de test particulier pour la classe mesurée [35].

CHAPITRE 4: APPRENTISSAGE AUTOMATIQUE

4.1 Contexte

La plupart des approches de détection de bogues pour les applications orientées objet reposent sur des techniques d'apprentissage machine encore appelées « Machine Learning » [35]. Dans cette partie, nous présentons une vue générale sur l'apprentissage machine et ses deux déclinaisons : l'apprentissage supervisé et l'apprentissage non supervisé.

4.2 Définition

Le Machine Learning ou apprentissage automatique est une méthode d'analyse des données qui automatise la création de modèles analytiques. C'est une branche de l'intelligence artificielle qui repose sur l'idée que les systèmes peuvent apprendre des données, identifier des tendances et prendre des décisions avec un minimum d'intervention humaine.

On distingue trois types d'apprentissage machines: l'apprentissage non supervisé, apprentissage supervisé et l'apprentissage semi-supervisé. Nous nous intéresserons aux deux premiers.

4.3 Apprentissage non supervisé

L'apprentissage non supervisé [35] est celui où l'algorithme doit opérer à partir d'exemples non annotés. Il doit faire émerger automatiquement des catégories à associer aux données qu'on lui soumet pour reconnaître par exemple un chat, une voiture (par exemple) comme sont capables de le faire les animaux et les humains. Le problème d'apprentissage non supervisé le plus fréquent est la segmentation (ou Clustering) où l'on essaie de séparer les données en groupes (catégorie, classe, cluster...) : regrouper des images de voitures, de chats, etc. Beaucoup de recherches s'appuient sur cette technique

pour la détection d'anomalies (bogues, comportement inhabituel ou inattendu) pour la maintenance prédictive, la cyber sécurité, mais aussi le dépistage précoce de maladies, etc.

De manière générale, l'algorithme cherche à maximiser d'une part l'homogénéité des données au sein des groupes de données et à former des groupes aussi distincts que possible : selon le contexte, on choisit d'utiliser un algorithme donné pour classer les données par exemple selon leur densité ou leur gradient de densité. Dans le cas de la détection de bogues dans un programme informatique, c'est plutôt le caractère extrême ou atypique des valeurs ou d'un pattern dans les données qui est recherché. La métrique sous-jacente joue un rôle clé pour déterminer ce qui est la norme et ce qui s'en éloigne [36].

Il s'agit, par exemple, de données qui ne suivent pas le même schéma ou qui sont atypiques pour la distribution de probabilité observée. La difficulté du problème provient du fait qu'on ne connait pas au préalable la distribution sous-jacente de l'ensemble des données. C'est à l'algorithme de déterminer une métrique appropriée pour détecter les différentes classes susceptibles de contenir des bogues. Parmi les exemples d'applications dans d'autres domaines, citons les transactions bancaires (où une anomalie sera vue comme une fraude potentielle), la surveillance des données physiologiques d'un malade (l'anomalie est un problème de santé possible), ou encore la détection de défauts dans des chaines de production. La détection d'anomalie est souvent un problème d'apprentissage de type non supervisé. Les algorithmes typiques de détection d'anomalie sont les one-class SVM, les méthodes d'apprentissage de distribution bayésienne et les Random Forest [37].

4.4 Apprentissage supervisé

Ce cadre d'apprentissage automatique part du fait que les données historiques (ou exemples) sont annotées. Dans le cadre de notre travail, considérons les données résultantes des métriques extraites des applications mobiles : un problème supervisé correspond au cas où le label « 1 » est bien associé, à la base, à des classes qui sont susceptibles de contenir véritablement des bogues et le label « 0 » est bien associé à des classes qui à la base ne contiennent aucun bogue. Après avoir classifié correctement les exemples, il peut ensuite généraliser ce classement à de nouvelles données autrement dit, classifier correctement des classes qui n'ont pas servi à l'apprentissage selon les labels « 1 » et « 0 » qui indique qu'elles contiennent ou non des bogues. C'est la capacité de généralisation. Dans le cadre business, on parle souvent d'analyse prédictive. Parmi les exemples d'applications, citons par exemple la classification de courriels en pourriels ou non selon le contenu du message, son expéditeur, son sujet..., et le diagnostic médical selon les symptômes, etc [36].

La technique de prédiction de bogues dans une application mobile peut être supervisée ou non selon le cas. Dans notre cas, il s'agit d'un problème de classification supervisé. Dans la classification supervisée, les observations contenant les métriques des classes logicielles ainsi que leurs labels précédemment classés forment un ensemble d'entraînement pour décider si la classe présente un bogue ou non.

4.5 Algorithmes d'apprentissage automatique

Il existe plusieurs algorithmes et techniques utilisés pour la classification supervisée et non supervisée des bogues dans un système logiciel en général.

Dans ce travail nous nous intéresserons aux algorithmes supervisés suivants :

- **Classificateur Bayésien Naïf (NB)**: Il s'agit d'une méthode de classification statistique qui se base principalement sur le théorème de Bayes avec une forte hypothèse sur les descripteurs (conditionnellement indépendants) d'où le qualificatif naïf. Elle est utilisée dans plusieurs applications telles que la détection de courriels spam, pour séparer les bons courriels des mauvais (c'est-à-dire les pourriels). Un avantage de cette méthode est la simplicité de programmation, la facilité d'estimation des paramètres et sa rapidité même sur de très grandes quantités de données. Sur un ensemble d'apprentissage, l'approche consiste à : (1) déterminer les probabilités à priori de chaque classe (analyse de fréquence), (2) à appliquer la règle de Bayes pour déterminer les probabilités à posteriori des classes, et (3) à choisir la classe la plus probable.

- **Machine à vecteurs de support (SVM)** : Il s'agit d'un ensemble de techniques destinées à résoudre des problèmes de discrimination (prédiction d'appartenance à des groupes prédéfinis) et de régression (analyse de la relation d'une variable par rapport à d'autres). Les SVM sont une famille d'algorithmes d'apprentissage automatique qui permettent de résoudre des problèmes tant de classification que de régression ou de détection d'anomalie (bogues, fraude, etc.). Ils sont connus pour leurs solides bases théoriques, leur grande flexibilité ainsi que leur simplicité d'utilisation même sans grande connaissance du Data Mining. Ils ont été développés dans les années 1990. Ils ont pour but de séparer les données en classes à l'aide d'une frontière aussi « simple » que possible, de telle façon que la distance entre les différents groupes de données et la frontière qui les sépare soit maximale. Cette distance est aussi appelée « marge » et les SVM sont ainsi qualifiés de « séparateurs à vaste marge », les « vecteurs de support » étant les données les plus proches de la frontière.

Les SVM sont utilisés dans une variété d'applications (bio-informatique, recherche d'informations, vision par ordinateur, finance, etc.) notamment parce qu'à la différence des réseaux de neurones, on peut les utiliser sans comprendre leur fonctionnement : il existe des jeux d'hyper paramètres par défaut – pour la classification, la régression ou la détection d'anomalie – qui fonctionnent dans l'immense majorité des cas. C'est un de leurs principaux avantages. Ces hyper paramètres sont, par ailleurs, en nombre très réduit : ils se limitent au choix de la technique de régularisation (de type lasso ou encore régularisation RKHS*, une méthode spécifique aux SVM) et au choix du noyau (noyaux polynomiaux, Sobolev, RBF**...). Concernant les algorithmes SVM, citons le Kernel Ridge Régression pour la régression ou le ONE CLASS SVM pour la détection d'anomalie.

Enfin, selon les données, la performance des SVM est en général de même ordre voire supérieure à celle d'un réseau de neurones ou d'un modèle de mélanges gaussiens, à l'exception de certains cas notables comme la classification d'images.

- **Les réseaux de neurones (ANN)** : À l'inverse des algorithmes de déduction, ces derniers sont des algorithmes de type induction, c'est-à-dire que par le biais d'observations limitées, ils essayent de tirer des généralisations plausibles. C'est un système basé sur l'expérience qui se constitue une mémoire lors de sa phase d'apprentissage (qui peut être aussi non supervisée), appelée entrainement.

Ce sont des fonctions mathématiques à plusieurs paramètres, ajustables. Comme dans les neurones du cerveau où des connexions se créent, disparaissent ou se renforcent en fonction de différents stimuli et produisent une action. Les réseaux de neurones artificiels (ou formels) ajustent des paramètres (appelés poids synaptiques en référence au fonctionnement

biologique du cerveau) en fonction de données d'entrée afin de fournir la meilleure réponse possible.

Un neurone construit une combinaison linéaire des entrées qu'il reçoit, auxquelles il ajoute une valeur appelée biais. Une fonction non linéaire, dite d'activation, (comme la tangente hyperbolique) est alors appliquée à la valeur de sortie. Cette valeur est ensuite transmise à la couche des neurones suivantes. Chaque neurone effectue ainsi un calcul très rudimentaire, et c'est la succession des couches de neurones qui permet d'obtenir des réseaux complexes. Durant la phase dite « d'entraînement », le réseau va ajuster automatiquement les paramètres de chaque neurone, c'est-à-dire les valeurs des poids et du biais afin de minimiser l'erreur moyenne calculée sur l'ensemble des exemples entre la sortie attendue et celle observée. L'hypothèse est qu'après cette phase d'entraînement, le réseau sera capable de traiter de manière satisfaisante de nouveaux exemples, dont la sortie est inconnue, en fonction de ce qu'il a « appris ». Cette phase d'apprentissage peut se faire sur des caractères manuscrits, des objets dans une image, des sons, etc.

Dans un réseau de neurones à deux couches, la première couche est constituée d'un ensemble de neurones connectés en parallèle et fournissant un ensemble de sorties, elles-mêmes combinées pour devenir les entrées d'un nouvel ensemble de neurones formant une seconde couche.

- **Forêt aléatoire (RF)** : Il s'agit d'une application de graphe en arbres de décision permettant ainsi la modélisation de chaque résultat sur une branche en fonction des choix précédents. Le résultat final sera le vote de la majorité des arbres [38].

- **Les K plus proches voisins (KNN):** l'algorithme des K plus proches voisins est une méthode de classification et de régression non paramétrée

permettant de reconnaitre des patrons dans les observations. Il consiste à estimer la classe d'une nouvelle entrée E en prenant en compte, et avec le même poids, les k échantillons d'entrainement les plus « proches » de la nouvelle entrée E.

La proximité est déterminée par une distance. Dans nos expérimentations, nous utiliserons la distance HEOM (Heterogeneous Euclidean-Overlap Metric).

- **La Régression logistique (RL) :** La régression logistique est un modèle mathématique qui combine un ensemble de variables prédictives (X) avec une variable aléatoire binomiale (Y). Elle est couramment utilisée dans le domaine de l'intelligence artificielle (IA) et de l'apprentissage machine. Elle est considérée comme l'un des modèles d'analyse multivariée les plus simples à déchiffrer et analyser. La régression est une technique de modélisation linéaire, qui étudie le rapport entre une variable principale et des variables explicatives. Elle est dite logistique, quand il existe un lien de fonction logistique entre la variable d'intérêt et les autres variables.

CHAPITRE 5 : COLLECTE DE DONNÉES ET METHODES D'ANAYSE

5.1 Outils de collecte

- **GitHub** [39] : est un service web d'hébergement de codes sources de logiciels, utilisant le logiciel de gestion de versions Git. Ce site est développé en Ruby on Rails et Erlang par **Chris Wanstrath**, **PJ Hyett** et **Tom Preston-Werner.** GitHub propose des comptes professionnels payants, ainsi que des comptes gratuits pour les projets de logiciels libres. Le site assure également un contrôle d'accès et des fonctionnalités destinées à la collaboration comme le suivi des bogues, les demandes de fonctionnalités, la gestion de tâches et un wiki pour chaque projet.

Cette plateforme nous a permis d'avoir à la fois accès au code source des applications mobiles et de relever les différents bogues qu'elles présentent.

- **IntelliJ IDEA** [40] : est un environnement de développement IDE destiné au développement de logiciels informatiques. Il supporte les langages de programmation Java, Kotlin, Groovy, Scala.... Plusieurs plugins sont intégrés dans son environnement de travail pour le traitement de données. Dans le cadre de notre travail, nous explorons CodeMR. **CodeMR** [41] est un outil d'analyse de la qualité logicielle et de code statique pour les projets Java, Kotlin et Scala. Il aide les éditeurs de logiciels à développer un meilleur code et des produits de meilleure qualité. Nous utilisons CodeMR pour calculer les métriques de code et les attributs de qualité de bas niveau (couplage, complexité, cohésion et taille).

- **Orange** [42] : c'est un logiciel libre de forage de données. Il propose des fonctionnalités de préparation, de modélisation des données et de validité des modèles à travers une interface visuelle, une grande variété de modalités de

visualisation et des affichages variés et dynamiques. Développé en Python, il existe des versions pour les systèmes Windows, Mac et Linux.

Dans le cadre de notre travail, nous utilisons ce logiciel pour créer des modèles de prédiction des bogues en exploitant différents algorithmes d'apprentissages automatiques supervisés que le système propose. Les résultats des modèles sont évalués à l'aide de la matrice de confusion, du composant Test & Score (composant proposé par Orange) et de la courbe ROC. Ces outils de validation sont fournis par Orange.

- **Xlstat** : il est à la fois simple d'utilisation et très puissant. Il permet à ses utilisateurs d'analyser, de visualiser et de modéliser leurs données tout en produisant des rapports sous Microsoft Excel, exportables vers d'autres formats. Avec plus de 220 fonctionnalités statistiques allant des statistiques descriptives au Machine Learning, Xlstat est un outil statistique très utilisé dans des entreprises, des universités et de plus de 100 000 utilisateurs situés dans plus de 120 pays [43].

Dans le cadre de notre travail il nous permet de faire une statistique descriptive des métriques logicielles calculées à partir des codes sources des applications mobiles Android.

5.2 Systèmes analysés

5.2.1 Description des systèmes

Nous avons analysé 3 applications mobiles Android à savoir : **PDF Viewer**, **Quran** et **Password Store Master**.

- **Android PDF Viewer** : Cette application permet de lire des fichiers au format PDF. Créé avec l'aide de Android-pdfview par Joan Zapata [44], elle est entièrement développée en Java. Elle inclut des fonctionnalités comme des animations, des gestuelles, du zoom et le double tap. Elle est basée sur Pdfium

Android pour le décodage des fichiers PDF [45]. Elle fonctionne sur API 11 (Android 3.0) et supérieur et est sous licence Apache License 2.0. En date du présent mémoire, l'application a connu plus de 14 contribiteurs pour son amélioration continue. Elle comporte 99 classes dont 33 contenant des bogues d'après les données recueillies dans les forums de discussion sur la plateforme Github.

- **Quran** : C'est une application de Coran simple (basée sur Madani) pour Android. Elle est développée en Java et Kotlin par un groupe de tunisien avec aujourd'hui plus de 46 contributeurs en vue de son amélioration continue. Les données de traduction, de Tafsir (explication) et d'arabe proviennent du Quranenc et de l' Université King Saud . Un petit nombre de traductions proviennent également de Tanzil [46]. La version 2.9.3 de l'application comporte 236 classes dont 29 qui contiennent des bogues d'après les donnés recueillies dans les forums de discussion sur la plateforme Github. Les bogues mineurs étaient constatés sur le rendu de l'application (police illisible, mise en page non adaptée) avec certains terminaux. Les bogues majeurs sont beaucoup plus liés à certaines fonctionnalités de l'application (plantage, résultat de verset coranique non conforme à la requête de la recherche).

- **Password Store Master** : Cette application peut remplir les champs de mot de passe dans les applications natives, les navigateurs et les sites web. Il existe deux variantes de livraison dans l'application. Une implémentation soutenue par un service d'accessibilité qui est désormais héritée et obsolète, et une implémentation remaniée, résistante au hameçonnage et riche en fonctionnalités pour Android 8.0 Oreo et supérieur, soutenue par la nouvelle fonctionnalité de remplissage automatique d'Android [47]. Elle est développée en Kotlin et Script Shell. Elle compte actuellement plus de 70 contributeurs en vue de son amélioration continue. La version 1.3.2 de l'application comporte 44 classes dont 11 qui contiennent des bogues entachant le bon fonctionnement de

l'application d'après les données recueillies dans les forums de discussion sur la plateforme Github.

5.2.2 Procédure de collecte

Pour atteindre nos objectifs, nous explorons plusieurs codes sources des applications mobiles à l'aide de GitHub. Nous relevons minutieusement tous les bogues découverts durant les phases de test et d'utilisation de ces différentes applications Android. Rappelons que sur la plateforme Github, les différents problèmes rencontrés en phase de test ou d'utilisation d'un système logiciel sont étiquetés de plusieurs façons (mineur, majeure, amélioration) à travers plusieurs couleurs (violet, rouge, orange, bleu). Les différents problèmes relevés dans les fils de discussion à propos de chaque application mobile nous ont amené à mieux comprendre les différents types de bogue signalés. Ensuite, en suivant chaque intitulé de problème relevé, nous avons pu identifier les différentes classes concernées.

Dans le cadre de notre travail, nous nous sommes basés sur des bogues majeurs relevés (bogues majeurs). Dans le code de chaque classe logicielle, les bogues majeures sont étiquetés en rouge avec à l'appui une jauge de niveau qui permet de savoir précisément le nombre de modifications (majeures ou mineures) effectuées.

A chaque fois que nous remarquons que des bogues majeurs ont été relevés dans une classe logicielle, nous attribuons la modalité **1** à cette classe dans notre tableau de données. Si plusieurs bogues majeurs proviennent de la même classe, nous comptons le nombre total de bogues et nous les reportons dans notre tableau de données. Dans notre tableau récapitulatif des données, nous avons nommé cette colonne **Bogues Effectifs (BE)**. À toutes les classes qui ne présentent pas de bogue, nous attribuons la modalité **0**.

5.2.3 Méthodes d'analyse

Le plugin **CodeMR** nous a permis de calculer pour chaque application mobile, les **métriques de classe** décrites plus haut (CBO, LOC, RFC, DIT, LCOM, NOC, WMC). Toutes les données de ces métriques sont consignées dans un fichier Excel.

Nous avons relevé les différents bogues présents dans les systèmes logiciels en explorant les différents fils de discussions sur GitHub. À cet effet afin d'effectuer une statistique descriptive sur chacune des métriques logicielles calculées, nous avons créé une nouvelle colonne nommée **Bogues Effectifs** « **BE** » qui veut dire bogues décelés à l'usage. Dans cette dernière, toutes les classes dans lesquelles des bogues ont été relevés sont remplacées par 1 puis en absence de bogue elles sont remplacées par 0.

Une statistique descriptive est faite sur les différentes données collectées dans le cadre de notre travail avec l'outil statistique Xlstat intégré à Excel. Ainsi différents modèles d'apprentissage sont créés en séparant les données d'entrainement des données de test. Les données d'une application sont entièrement utilisées dans certains modèles comme données d'entraînement et dans un autre cas comme données de test. Chacune des applications sert une fois au moins de données de test et de données d'entraînement. Dans d'autres modèles les données de deux applications sont combinées pour servir de données d'entrainement et le reste constitue des données de test et vice-versa. Cette méthode s'appelle **leave-one-out cross-validation** [48].

5.2.4 Statistiques descriptives

Nous avons choisi 3 applications mobiles pour Android qui ont été développées en Java. Les statistiques descriptives des métriques de ces applications mobiles se présentent comme suit :

- Android PDF Viewer

Métriques	Obs	Mini	Max	Moy.	Écart-type
WMC	66	0,00	200,00	15,24	34,76
CBO	66	0,00	31,00	4,78	5,12
LOC	66	1,00	931,00	82,25	182,71
RFC	66	0,00	96,00	10,84	20,16
DIT	66	0,00	2,00	0,95	0,53
NOC	66	0,00	17,00	0,50	2,14
BE	66	0,00	5,00	0,86	1,11

Tableau 1 : Statistiques descriptives système Android PDF Viewer

Le tableau 1 décrit les statistiques descriptives du système Android PDF Viewer. Ce dernier comporte 66 classes logicielles avec en moyenne 82 lignes de code. Quatre (4) classes en moyennes sont couplées entre elles (CBO s'élève à 4,78). Le système possède en moyenne 10 méthodes (RFC moy) et un maximum de 96 méthodes (RFC max) pour les 66 classes. Le nombre de méthodes est assez considérable et justifie les 931 lignes d'instructions (LOC) constatées pour toutes les classes. De même le niveau d'utilisation de l'héritage est assez considérable (DIT+NOC donnent 19).

- Quran

Métriques	Obs	Mini	Max	Moy.	Écart-type
WMC	207	0,00	313,00	17,47	914,49
CBO	207	1,00	83,00	8,51	92,25
LOC	207	3,00	1639,00	99,38	27246,08
RFC	207	0,00	239,00	16,92	549,65
DIT	207	0,00	4,00	1,12	0,21
NOC	207	0,00	9,00	0,19	0,73
BE	207	0,00	13,00	0,35	1,76

Tableau 2 : Statistiques descriptives système Quran

Le tableau 2 décrit les statistiques descriptives du système Quran. Ce dernier contient 207 classes avec en moyenne 99 lignes de code. Le couplage (CBO) moyen entre les classes est de 8. Le système possède en moyenne 17 méthodes (RFC moy) et au maximum 239 méthodes (RFC max) pour les 207 classes logicielles. Le nombre de méthodes est assez considérable et justifie les 1639 lignes d'instructions (LOC) que compte le système. On constate également que le niveau d'utilisation de l'héritage est assez considérable (DIT+NOC s'élèvent à 13).

- **Password Store Master**

Métriques	Obs	Mini	Max	Moy.	Écart-type
WMC	23	1,00	104,00	18,91	26,78
CBO	23	1,00	21,00	6,91	5,05
LOC	23	6,00	655,00	120,21	165,32
RFC	23	2,00	91,00	19,43	21,38
DIT	23	0,00	2,00	1,08	0,41
NOC	23	0,00	2,00	0,08	0,41
BE	23	0,00	5,00	1,13	1,60

Tableau 3 : Statistiques descriptives système Password Store Master

Le tableau 3 décrit les statistiques descriptives du système Password Store Master. Ce dernier comporte 23 classes contenant en moyenne 120 lignes de code. Le couplage (CBO) moyen entre les classes est de 7. Le système possède en moyenne 19 méthodes (RFC moy) et 91 méthodes au total (RFC max) pour les 23 classes. Le nombre de méthodes est assez considérable et justifie les 655 lignes d'instructions constatées pour ce système. Nous constatons également que ce système utilise peu d'héritage puisque DIT+NOC s'élève à 3.

5.5 Évaluation de la performance des modèles d'apprentissage automatique

Nous avons créé plusieurs modèles de prédiction afin d'atteindre notre objectif principal. Après, les performances des différents modèles de prédiction sont évaluées à l'aide de la matrice de confusion, l'AUC, le rappel et la précision.

5.5.1 Matrice de confusion

Encore appelé tableau de contingence; la matrice de confusion est un outil permettant de mesurer les performances d'un modèle d'apprentissage automatique en vérifiant notamment le niveau d'exactitude de ces prédictions par rapport à la réalité, dans des problèmes de classification.

Cette matrice permet de comprendre de quelle façon le modèle de classification n'est pas clair lorsqu'il effectue des prédictions. Ceci permet non seulement de savoir quelles sont les erreurs commises, mais surtout le type d'erreurs commises. On peut analyser les types d'erreurs commises pour déterminer comment les erreurs sont commises.

		Vraies Classes	
		Positive	**Négative**
Classes prédites	**Positive**	VP	FP
	Négative	FN	VN

Tableau 4: Modèle de matrice de confusion

VP : vraies classes positives, **VN** : vraies classes négatives, **FP** : fausses classes positives, **FN** : fausses classes négatives.

5.5.3 Le rappel

Le rappel permet de connaître la proportion de résultats positifs réels qui ont été correctement identifiée. Il peut être défini comme suit : $\mathbf{Rappel} = \frac{\mathbf{VP}}{\mathbf{VP+FN}}$

5.5.4 La précision

La précision est la proportion d'observation classifiée positive par le classificateur sur le nombre de total classifications positives. Elle est définie comme suit : $\mathbf{Précision} = \frac{\mathbf{VP}}{\mathbf{VP+FP}}$

5.5.2 Courbe ROC et AUC

- Une **courbe ROC** (Receiver Operating Characteristic) est un graphique représentant les performances d'un modèle de classification pour tous les seuils de classification. Cette courbe trace le taux de vrais positifs en fonction du taux de faux positifs.

- L'**AUC** fournit une mesure agrégée des performances pour tous les seuils de classification possibles. On peut interpréter l'AUC comme une mesure de la probabilité pour que le modèle classe de manière positive un exemple aléatoire au-dessus d'un exemple négatif aléatoire.

Les valeurs d'AUC sont comprises dans une plage allant de 0 à 1. Un modèle dont 50 % des prédictions sont erronées à un AUC de 0,5 et est jugé aléatoire. Si toutes ses prédictions sont correctes, son AUC est de 1,0. Un AUC >= 70% est considéré comme une bonne classification [49].

CHAPITRE 6 : RESULTATS ET DISCUSSION

6.1 Présentation des résultats

6.1.1 Modèle de prédiction M_1

Dans le modèle M_1, le jeu de données de l'application **Password Store Master** a servi à entrainer le modèle. Par la suite, le classificateur obtenu a été testé sur les jeux de données de l'application **Android PDF Viewer**.

6.1.1.1 Test et Score - Courbe ROC

MODELS	AUC	PRECISION	RAPPEL
KNN	0,72	0,72	0,76
SVM	0,81	0,88	0,80
RF	0,78	0,82	0,77
ANN	0,85	0,90	0,85
NB	0,77	0,73	0,77
RL	0,75	0,73	0,77

Tableau 5: Résultats du modèle de prédiction M_1

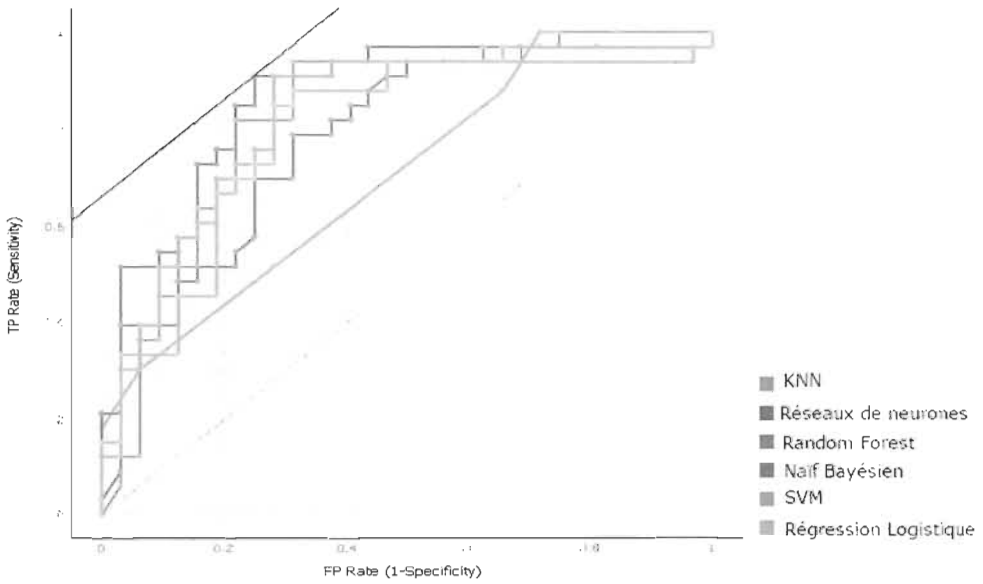

Figure 1: Courbe ROC modèle de prédiction modèle M1

On constate que pour ce modèle, tous les classificateurs sont précis dans la prédiction des classes qui contiennent des bogues. On peut constater que les valeurs de l'aire sous la courbe (AUC) varient entre 0,72 et 0,85 et de 72% à 90% pour la proportion des classes susceptibles de contenir des bogues. De même, on peut remarquer que les proportions des classes contenant effectivement des bogues sont également significatives. Elles varient de 76 % à 85%. Les Réseaux de Neurones sont meilleurs pour ce modèle de prédiction.

6.1.1.2 Matrices de confusion

- **Réseaux de Neurones**

	0	**1**	**Total**
0	0	3	3
1	2	18	20
Total	**2**	**21**	**23**

Tableau 6: Matrice de confusion des réseaux de neurones pour le modèle M1

Le tableau 6 nous indique horizontalement que les 3 classes logicielles qui initialement sont prédites comme ne contenant pas de bogues, contiennent en réalité des bogues. Sur les 20 classes logicielles qui initialement contiennent des bogues, 18 classes ont été correctement classées comme telles contre 2 classes n'ayant aucun bogue.

On remarque verticalement que toutes les 2 classes logicielles prédites comme classes ne contenant aucun bogue contiennent en fait des bogues. Tandis que sur les 21 classes logicielles prédites comme des classes logicielles comportant des bogues, 3 n'en contiennent pas en réalité.

- **Random Forest**

	0	1	Total
0	2	4	6
1	3	14	17
Total	**5**	**18**	**23**

Tableau 7: Matrice de confusion Random Forest pour le modèle M1

Le tableau 7 indique horizontalement que sur les 6 classes logicielles ne contenant aucun bogue initialement, 2 sont prédites comme telles. Quant aux 17 classes qui contiennent initialement des bogues, 14 sont effectivement classées comme telles.

On remarque verticalement que sur les 5 classes logicielles prédites comme n'ayant pas de bogues, 2 classes logicielles sont correctement prédites. Tandis que sur les 18 classes logicielles prédites comme des classes ayant de bogues, 14 classes sont effectivement classées comme telles.

- **KNN**

	0	1	Total
0	1	4	5
1	5	13	18
Total	**6**	**17**	**23**

Tableau 8: Matrice de confusion KNN pour le modèle M1

Le tableau 8 nous indique horizontalement que sur les 5 classes logicielles ne contenant aucun bogue initialement, une seule est prédite comme telle. Quant aux 18 classes qui contiennent initialement des bogues, 13 sont effectivement classées comme telles.

On remarque verticalement que sur les 6 classes logicielles prédites comme n'ayant pas de bogues, une seule est correctement classée. Tandis que sur les 17

classes logicielles prédites comme des classes ayant de bogues, 13 classes sont effectivement classées comme telles.

- **Naïf Bayésien**

	0	1	Total
0	0	4	4
1	5	14	19
Total	**5**	**18**	**23**

Tableau 9: Matrice de confusion Naïf Bayésien pour le modèle M1

Le tableau 9 nous indique horizontalement qu'aucune des 4 classes logicielles prédites comme ne contenant pas de bogue initialement, n'est classée correctement. Quant aux 19 classes qui contiennent initialement des bogues, 14 sont effectivement classées comme telles.

On remarque verticalement que sur les 5 classes logicielles prédites comme n'ayant pas de bogues, aucune n'est correctement classée. Tandis que sur les 18 classes logicielles prédites comme des classes ayant des bogues, 14 classes sont effectivement classées comme telles.

- **SVM**

	0	1	Total
0	1	4	5
1	2	16	18
Total	**3**	**20**	**23**

Tableau 10:Matrice de confusion SVM pour le modèle M1

Le tableau 10 nous indique horizontalement que sur les 5 classes logicielles ne contenant aucun bogue initialement, une seule est prédite comme telle. Quant aux 18 classes qui contiennent initialement des bogues, 16 sont effectivement classées comme telles.

On remarque aussi verticalement que sur les 3 classes logicielles prédites comme n'ayant pas de bogues, une seule classe logicielle est correctement prédite. Tandis que sur les 20 classes logicielles prédites comme des classes ayant des bogues, 16 classes sont effectivement classées comme telles.

- **Régression Logistique**

	0	**1**	**Total**
0	0	4	4
1	5	14	19
Total	**5**	**18**	**23**

Tableau 11: Matrice de confusion Régression Logistique pour le modèle M1

Le tableau 11 nous indique horizontalement que sur les 4 classes logicielles ne contenant aucun bogue initialement, aucune n'est prédite comme telle. Quant aux 19 classes logicielles qui contiennent initialement des bogues, 14 sont effectivement classées comme telles.

On remarque verticalement que sur les 5 classes logicielles prédites comme n'ayant pas de bogues, aucune n'est correctement classée. Tandis que sur les 18 classes logicielles prédites comme des classes ayant des bogues, 14 classes sont effectivement classées comme telles.

6.1.2 Modèle de prédiction M2

Dans le modèle M_2, les jeux de données des applications **Password Store Master** et **Quran** ont servi à former les classificateurs qui ont ensuite été testés par la suite sur de nouveaux jeux de données de l'application mobile **Android PDF Viewer**.

6.1.2.1 Test et Score - Courbes ROC

MODELS	AUC	PRECISION	RAPPEL
KNN	0,68	0,71	0,77
SVM	0,63	0,63	0,69
RF	0,68	0,68	0,76
ANN	0,77	0,74	0,87
NB	0,73	0,73	0,79
RL	0,75	0,71	0,83

Tableau 12:Résultats du modèle de prédiction M2

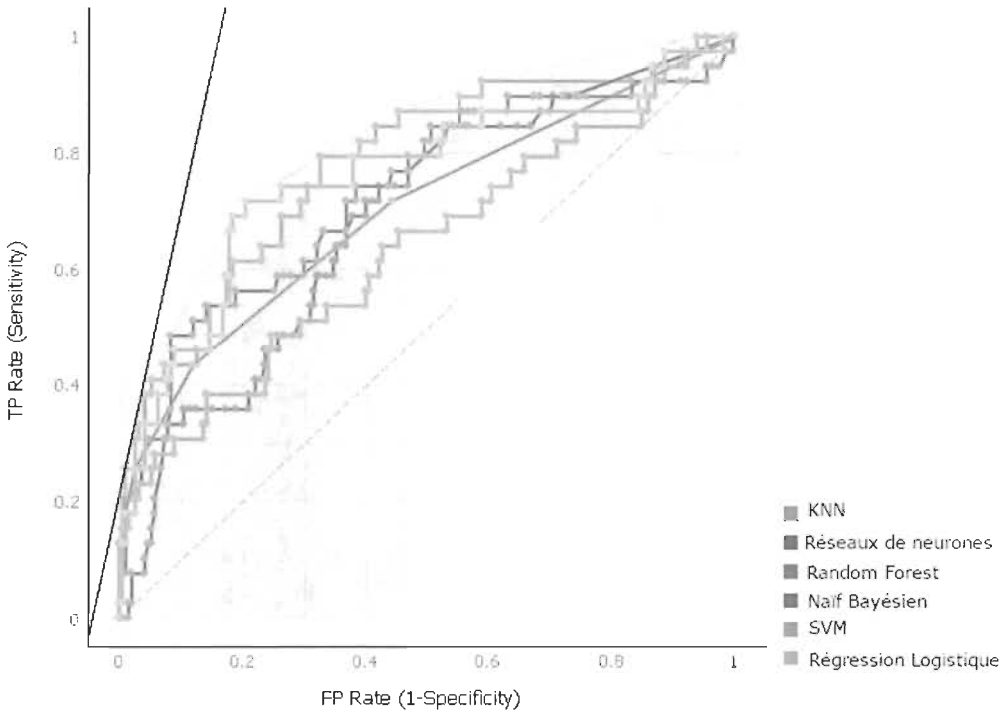

Figure 2: Courbe ROC modèle de prédiction modèle M2

On constate que pour ce modèle d'apprentissage (figure 2) les Réseaux de Neurones, le Naïf Bayésien et les KNN sont plus précis dans la prédiction des classes qui contiennent des bogues à l'usage. Les performances des Réseaux de Neurones sont plus significatives avec une AUC=0,77. Ensuite la Régression Logistique avec une AUC=0,75 et enfin le Naïf Bayésien avec une AUC=0,73. Ces 3 classificateurs sont non seulement significatifs dans la proportion des

classes susceptibles de contenir des bogues (entre 71% et 74%) mais aussi dans la proportion des classes contenant effectivement des bogues; soit entre 79% et 87%. Les Réseaux de Neurones sont également meilleurs pour ce modèle de prédiction.

6.1.2.2 Matrices de confusion

- **Réseaux de Neurones**

	0	1	Total
0	14	5	19
1	12	35	47
Total	**26**	**40**	**66**

Tableau 13: Matrice de confusion Réseaux de Neurones pour le modèle M2

Le tableau 13 nous indique horizontalement que sur les 19 classes logicielles qui initialement sont prédites comme ne contenant pas de bogues, 14 classes sont effectivement classées comme telles. Sur les 47 classes logicielles qui initialement contiennent des bogues, 35 classes sont effectivement classées comme telles.

On remarque verticalement que sur les 26 classes logicielles prédites comme classes ne contenant aucun bogue; 12 contiennent en fait des bogues. Tandis que sur les 40 classes logicielles prédites comme des classes logicielles comportant des bogues, 5 n'en contiennent pas en réalité.

- **Random Forest**

	0	1	Total
0	9	10	19
1	15	32	47
Total	**24**	**42**	**66**

Tableau 14:Matrice de confusion Random Forest pour le modèle M2

Le tableau 14 nous indique horizontalement que sur les 19 classes logicielles initialement prédites comme ne contenant pas de bogues, 9 classes sont effectivement classées comme telles. Sur les 47 classes logicielles qui initialement contiennent des bogues, 32 classes sont effectivement classées comme telles.

On remarque verticalement que sur les 24 classes logicielles prédites comme classes ne contenant aucun bogue; 15 contiennent en fait des bogues. Tandis que sur les 42 classes logicielles prédites comme des classes logicielles comportant des bogues, 10 n'en contiennent pas en réalité.

- **KNN**

	0	**1**	**Total**
0	7	10	17
1	14	35	49
Total	**21**	**45**	**66**

Tableau 15:Matrice de confusion KNN pour le modèle M2

Le tableau 15 nous indique horizontalement que sur les 17 classes logicielles initialement prédites comme ne contenant pas de bogues, 7 classes sont effectivement classées comme telles. Sur les 49 classes logicielles qui initialement contiennent des bogues, 35 classes sont correctement classées comme telles.

On remarque verticalement que sur les 21 classes logicielles prédites comme classes ne contenant aucun bogue; 14 contiennent en fait des bogues. Tandis que sur les 45 classes logicielles prédites comme des classes logicielles comportant des bogues, 10 n'en contiennent pas en réalité.

- **Naïf Bayésien**

	0	1	Total
0	11	9	20
1	12	34	46
Total	**23**	**43**	**66**

Tableau 16: Matrice de confusion Naïf Bayésien pour le modèle M2

Le tableau 16 nous indique horizontalement que sur les 20 classes logicielles qui initialement sont prédites comme ne contenant pas de bogues, 11 classes sont effectivement classées comme telles. Sur les 46 classes logicielles qui initialement contiennent des bogues, 34 classes sont effectivement classées comme telles.

On remarque verticalement que sur les 23 classes logicielles prédites comme classes ne contenant aucun bogue; 12 contiennent en fait des bogues. Tandis-que sur les 43 classes logicielles prédites comme des classes logicielles comportant des bogues, 9 n'en contiennent pas en réalité.

- **SVM**

	0	1	Total
0	6	13	19
1	17	30	47
Total	**23**	**43**	**66**

Tableau 17: Matrice de confusion SVM pour le modèle M2

Le tableau 17 nous indique horizontalement que sur les 19 classes logicielles initialement prédites comme ne contenant pas de bogues, 6 sont effectivement classées comme telles. Sur les 47 classes logicielles qui initialement contiennent des bogues, 30 sont effectivement classées comme telles.

On remarque verticalement que sur les 23 classes logicielles prédites comme classes ne contenant aucun bogue; 17 en contiennent. Tandis-que sur les 43 classes logicielles prédites comme des classes logicielles comportant des bogues, 13 n'en contiennent pas en réalité.

- **Regression Logistique**

	0	**1**	**Total**
0	10	7	17
1	14	35	49
Total	**24**	**42**	**66**

Tableau 18: Matrice de confusion Régression Logistique pour le modèle M2

Le tableau 18 nous indique horizontalement que sur les 17 classes logicielles initialement prédites comme ne contenant pas de bogues, 10 sont effectivement classées comme telles. Sur les 49 classes logicielles qui initialement contiennent des bogues, 35 sont effectivement classées comme telles.

On remarque verticalement que sur les 24 classes logicielles prédites comme classes ne contenant aucun bogue ; 14 en contiennent. Tandis que sur les 42 classes logicielles prédites comme des classes logicielles comportant des bogues, 7 n'en contiennent pas en réalité.

6.1.3 Modèle de prédiction M3

Dans le modèle M_3, le jeu de données de l'application **Quran** a servi à former le modèle et testé par la suite sur de nouveaux jeux de données de l'application mobile **Android PDF Viewer**.

6.1.3.1 Test et Score- Courbes ROC

MODELS	AUC	PRECISION	RAPPEL
KNN	0,76	0,73	0,80
SVM	0,70	0,68	0,73
RF	0,79	0,79	0,86
ANN	0,81	0,80	0,88
NB	0,79	0,76	0,81
RL	0,82	0,89	0,93

Tableau 19:Résultats du modèle de prédiction M3

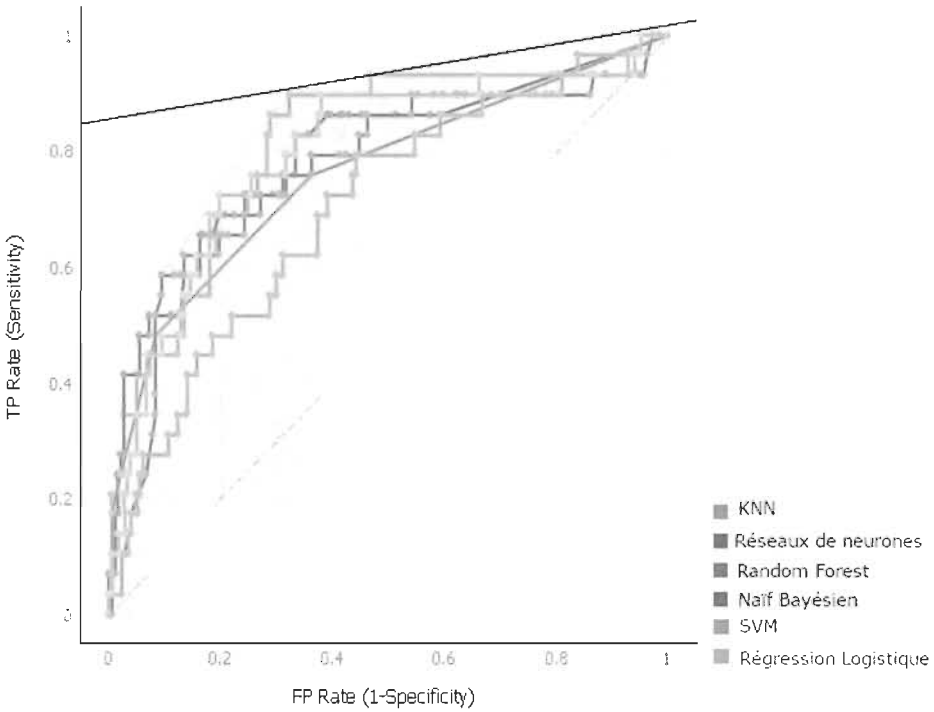

Figure 3:Courbes ROC modèle de prédiction M3

On constate que pour ce modèle, tous les classificateurs sont précis dans la prédiction des classes qui contiennent des bogues à l'usage. On peut constater que les valeurs de l'aire sous la courbe (AUC) varient de 0,70 à 0,82 et de 68% à 89% pour la proportion des classes susceptibles de contenir des bogues. On peut également, remarquer que les proportions des classes contenant effectivement des bogues sont également significatives. Elles varient de 73 % à 93%. La

Régression Logistique et les Réseaux de Neurones sont meilleurs pour ce modèle de prédiction.

6.1.3.2 Matrices de confusion

- Réseaux de Neurones

	0	1	Total
0	11	5	16
1	10	40	50
Total	**21**	**45**	**66**

Tableau 20:Matrice de confusion Réseaux de Neurones pour le modèle M3

Le tableau 20 nous indique horizontalement que sur les 16 classes logicielles initialement prédites comme ne contenant pas de bogues, 11 sont effectivement classées comme telles. Sur les 50 classes logicielles qui initialement contiennent des bogues, 40 sont effectivement classées comme telles.

On remarque verticalement que sur les 21 classes logicielles prédites comme classes ne contenant aucun bogue; 10 en contiennent. Tandis que sur les 45 classes logicielles prédites comme des classes logicielles comportant des bogues, 5 n'en contiennent pas en réalité.

- **Random Forest**

	0	1	Total
0	12	6	21
1	10	38	48
Total	**22**	**44**	**66**

Tableau 21:Matrice de confusion Random Forest pour le modèle M3

Le tableau 21 nous indique horizontalement que sur les 21 classes logicielles initialement prédites comme ne contenant pas de bogues, 12 sont

effectivement classées comme telles. Sur les 48 classes logicielles qui initialement contiennent des bogues, 38 sont effectivement classées comme telles.

On remarque verticalement que sur les 22 classes logicielles prédites comme classes ne contenant aucun bogue; 10 en contiennent. Tandis que sur les 44 classes logicielles prédites comme des classes logicielles comportant des bogues, 6 n'en contiennent pas en réalité.

- **KNN**

	0	**1**	**Total**
0	8	9	17
1	13	36	49
Total	**21**	**45**	**66**

Tableau 22:Matrice de confusion KNN pour le modèle M3

Le tableau 22 nous indique horizontalement que sur les 17 classes logicielles initialement sont prédites comme ne contenant pas de bogues, 8 sont effectivement classées comme telles. Sur les 49 classes logicielles qui initialement contiennent des bogues, 36 sont effectivement classées comme telles.

On remarque verticalement que sur les 21 classes logicielles prédites comme classes ne contenant aucun bogue; 13 en contiennent. Tandis que sur les 45 classes logicielles prédites comme des classes logicielles comportant des bogues, 9 n'en contiennent pas en réalité.

- **Naïf Bayésien**

	0	1	Total
0	11	8	19
1	11	36	47
Total	**22**	**44**	**66**

Tableau 23:matrice de confusion Naïf Bayésien pour le modèle M3

Le tableau 23 nous indique horizontalement que sur les 19 classes logicielles initialement prédites comme ne contenant pas de bogues, 11 sont effectivement classées comme telles. Sur les 47 classes logicielles qui initialement contiennent des bogues, 36 sont effectivement classées comme telles.

On remarque verticalement que sur les 22 classes logicielles prédites comme classes ne contenant aucun bogue; 11 en contiennent. Tandis que sur les 44 classes logicielles prédites comme des classes logicielles comportant des bogues, 8 n'en contiennent pas en réalité.

- **SVM**

	0	1	Total
0	6	12	18
1	15	33	48
Total	**21**	**45**	**66**

Tableau 24: Matrice de confusion SVM pour le modèle M3

Le tableau 24 nous indique horizontalement que sur les 18 classes logicielles initialement prédites comme ne contenant pas de bogues, 6 sont effectivement classées comme telles. Sur les 48 classes logicielles qui initialement contiennent des bogues, 33 sont effectivement classées comme telles.

On remarque verticalement que sur les 21 classes logicielles prédites comme classes ne contenant aucun bogue; 15 en contiennent. Tandis que sur les 45 classes logicielles prédites comme des classes logicielles comportant des bogues, 12 n'en contiennent pas en réalité.

- **Régression Logistique**

	0	1	Total
0	16	3	19
1	5	42	47
Total	**21**	**45**	**66**

Tableau 25: Matrice de confusion Régression logistique pour le modèle M3

Le tableau 25 nous indique horizontalement que sur les 19 classes logicielles initialement prédites comme ne contenant pas de bogues, 16 sont effectivement classées comme telles. Sur les 47 classes logicielles qui initialement contiennent des bogues, 42 sont effectivement classées comme telles.

On remarque verticalement que sur les 21 classes logicielles prédites comme classes ne contenant aucun bogue; 5 en contiennent. Tandis que sur les 45 classes logicielles prédites comme des classes logicielles comportant des bogues, n'en contiennent pas en réalité.

6.1.4 Modèle de prédiction M_4

Dans le modèle M_4, les jeux de données de l'application **Android PDF Viewer** ont servi à former le modèle et testés par la suite sur de nouveaux jeux de données de l'application mobile **Password Store Master**.

6.1.4.1 Test et Score- Courbes ROC

MODELS	AUC	PRECISION	RAPPEL
KNN	0,72	0,64	0,64
SVM	0,81	0,72	0,76
RF	0,81	0,72	0,76
ANN	0,84	0,87	0,77
NB	0,77	0,75	0,70
RL	0,75	0,75	0,66

Tableau 26: Résultats du modèle de prédiction M4

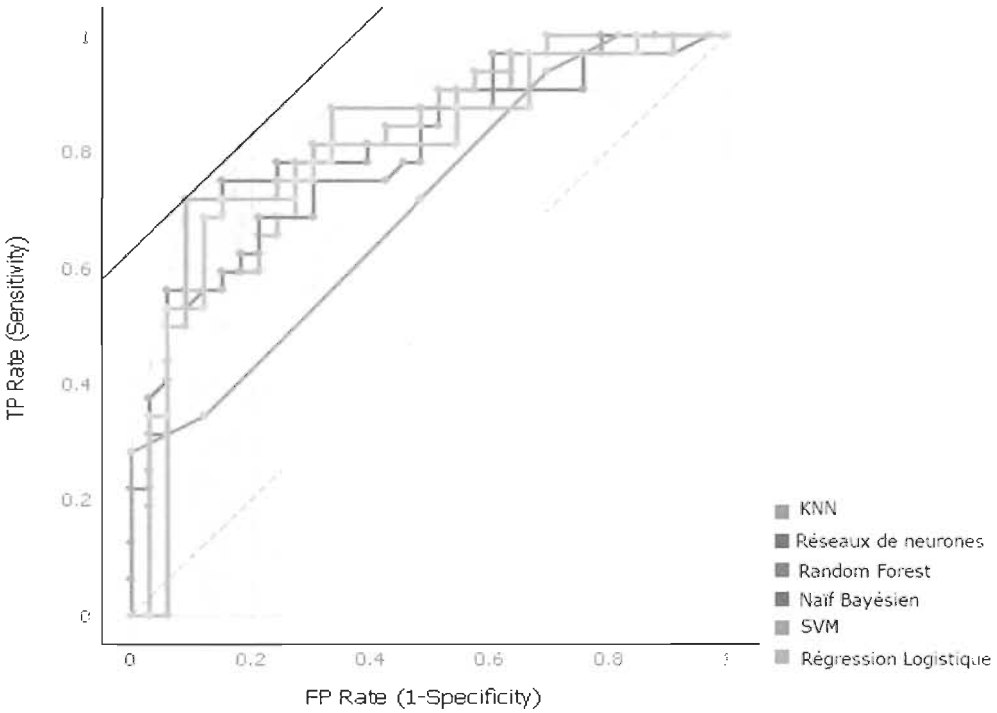

Figure 4: Courbe ROC modèle de prédiction M4

On constate que pour ce modèle, tous les classificateurs sont précis dans la prédiction des classes qui contiennent des bogues à l'usage. On peut constater que les valeurs de l'aire sous la courbe (AUC) varient de 0,72 à 0,84 et de 64% à 87% pour la proportion des classes susceptibles de contenir de bogues. De même on peut remarquer que les proportions des classes contenant

effectivement des bogues sont également significatives. Elles varient de 64 % à 77%. Les Réseaux de Neurones sont meilleurs pour ce modèle de prédiction.

6.1.4.2 Matrices de confusion

- **Réseaux de Neurones**

	0	1	Total
0	3	4	7
1	2	14	16
Total	**5**	**18**	**23**

Tableau 27: Matrice de confusion réseaux de neurones pour le modèle M4

Le tableau 27 nous indique horizontalement que sur les 7 classes logicielles initialement prédites comme ne contenant pas de bogues, 3 sont effectivement classées comme telles. Sur les 16 classes logicielles qui initialement contiennent des bogues, 14 sont effectivement classées comme telles.

On remarque verticalement que sur les 5 classes logicielles prédites comme classes ne contenant aucun bogue; 2 en contiennent. Tandis que sur les 18 classes logicielles prédites comme des classes logicielles comportant des bogues, 4 n'en contiennent pas en réalité.

- **Random Forest**

	0	1	Total
0	1	4	5
1	5	13	18
Total	**6**	**17**	**23**

Tableau 28: Matrice de confusion Random Forest pour le modèle de prédiction M4

Le tableau 28 nous indique horizontalement que sur les 5 classes logicielles initialement prédites comme ne contenant pas de bogues, une seule est

effectivement classée comme telle. Sur les 18 classes logicielles qui initialement contiennent des bogues, 13 sont effectivement classées comme telles.

On remarque verticalement que sur les 6 classes logicielles prédites comme classes ne contenant aucun bogue; une seule classe contient en fait des bogues. Tandis que sur les 17 classes logicielles prédites comme des classes logicielles comportant des bogues, 4 n'en contiennent pas en réalité.

- **KNN**

	0	**1**	**Total**
0	0	6	6
1	6	11	17
Total	**6**	**17**	**23**

Tableau 29: Matrice de confusion KNN pour le modèle de prédiction M4

Le tableau 29 nous indique horizontalement que sur les 6 classes logicielles initialement prédites comme ne contenant pas de bogues, aucune classe n'est effectivement classée comme telle. Sur les 17 classes logicielles qui initialement contiennent des bogues, 11 sont effectivement classées comme telles.

On remarque verticalement que sur les 6 classes logicielles prédites comme classes ne contenant aucun bogue; aucune classe n'est effectivement classée comme telle. Tandis que sur les 17 classes logicielles prédites comme des classes logicielles comportant des bogues, 6 n'en contiennent pas en réalité.

- **Naïf Bayésien**

	0	1	Total
0	2	5	7
1	4	12	16
Total	**6**	**17**	**23**

Tableau 30:Matrice de confusion Naïf Bayésien pour le modèle de prédiction M4

Le tableau 30 nous indique horizontalement que sur les 7 classes logicielles initialement prédites comme ne contenant pas de bogues, deux sont effectivement classées comme telles. Sur les 16 classes logicielles qui initialement contiennent des bogues, 12 sont effectivement classées comme telle.

On remarque verticalement que sur les 6 classes logicielles prédites comme classes ne contenant aucun bogue; 4 en contiennent. Tandis que sur les 17 classes logicielles prédites comme des classes logicielles comportant des bogues, 5 n'en contiennent pas en réalité.

- **SVM**

	0	1	Total
0	1	4	5
1	5	13	18
Total	**6**	**17**	**23**

Tableau 31: Matrice de confusion SVM pour le modèle de prédiction M4

Le tableau 31 nous indique horizontalement que sur les 5 classes logicielles initialement prédites comme ne contenant pas de bogues, une seule est effectivement classée comme telle. Sur les 18 classes logicielles qui initialement contiennent des bogues, 13 sont effectivement classées comme telles.

On remarque verticalement que sur les 6 classes logicielles prédites comme classes ne contenant aucun bogue; une seule classe contient en fait des bogues. Tandis que sur les 17 classes logicielles prédites comme des classes logicielles comportant des bogues, 4 n'en contiennent pas en réalité.

- **Regression Logistique**

	0	**1**	**Total**
0	0	6	6
1	5	12	17
Total	**5**	**18**	**23**

Tableau 32: Matrice de confusion régression logistique pour le modèle de prédiction M4

Le tableau 32 nous indique horizontalement que sur les 6 classes logicielles initialement prédites comme ne contenant pas de bogues, aucune n'est effectivement classée comme telle. Sur les 17 classes logicielles qui initialement contiennent des bogues, 12 sont effectivement classées comme telles.

On remarque verticalement que sur les 5 classes logicielles prédites comme classes ne contenant aucun bogue; 5 classes contiennent en fait des bogues. Tandis que sur les 18 classes logicielles prédites comme des classes logicielles comportant des bogues, 6 n'en contiennent pas en réalité.

6.1.5 Modèle de prédiction M_5

Dans le modèle de prédiction M_5, les jeux de données des applications **Android PDF Viewer** et **Quran** ont servi à former le modèle et testés par la suite sur de nouveaux jeux de données de l'application mobile **Password Store Master**.

6.1.5.1 Test et Score- Courbes ROC

MODELS	AUC	PRECISION	RAPPEL
KNN	0,67	0,70	0,66
SVM	0,61	0,62	0,66
RF	0,67	0,70	0,66
ANN	0,71	0,73	0,87
NB	0,71	0,82	0,82
RL	0,63	0,64	0,73

Tableau 33:Résultats du modèle de prédiction M5

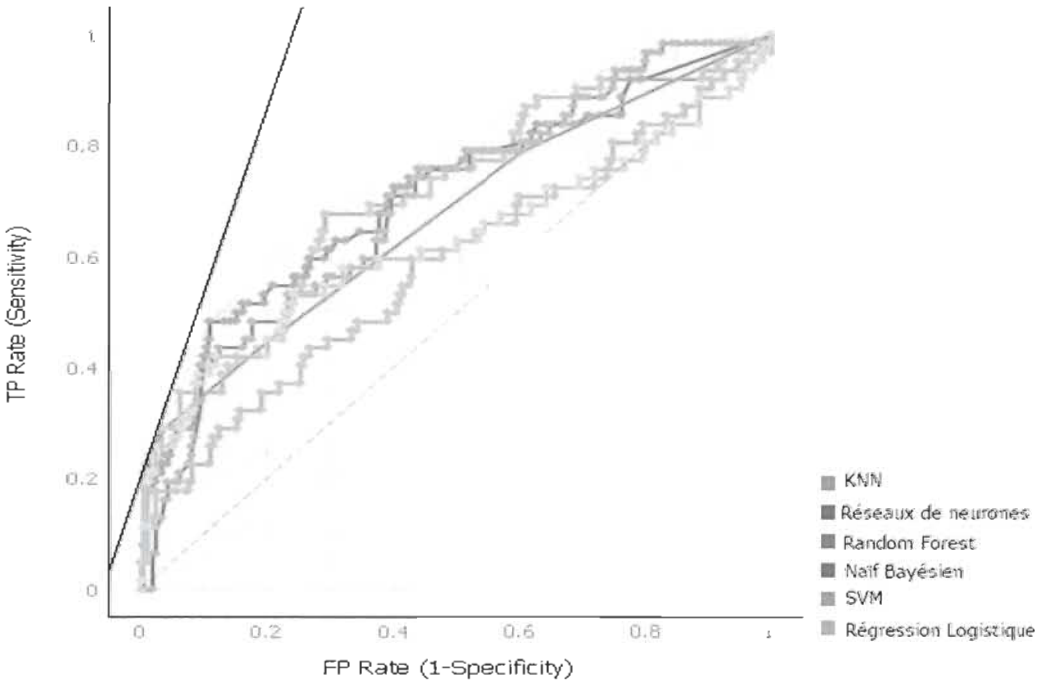

Figure 5:Courbe ROC modèle de prédiction M5

On constate que pour ce modèle d'apprentissage les Réseaux de Neurones et le Naïf Bayésien sont plus précis dans la prédiction des classes contenant des bogues. On remarque également, que ces deux classificateurs ont la même valeur de AUC; soit 0,71. Ils sont non seulement significatifs dans la proportion des classes susceptibles de contenir des bogues (respectivement 73% et 82%) mais aussi dans la proportion des classes contenant effectivement des bogues; soit respectivement 87% et 82%.

6.1.5.2 Matrices de confusion

- **Réseaux de Neurones**

	0	1	Total
0	2	2	4
1	5	14	19
Total	**7**	**16**	**23**

Tableau 34:Matrice de confusion Réseaux de Neurones pour le modèle de prédiction M5

Le tableau 34 indique horizontalement que sur les 4 classes logicielles initialement prédites comme ne contenant pas de bogues, 2 sont effectivement classées comme telles. Sur les 19 classes logicielles qui initialement contiennent des bogues, 14 sont effectivement classées comme telles.

On remarque verticalement que sur les 7 classes logicielles prédites comme classes ne contenant aucun bogue; 5 en contiennent. Tandis que sur les 16 classes logicielles prédites comme des classes logicielles comportant des bogues, 2 n'en contiennent pas.

- **Random Forest**

	0	1	Total
0	0	6	6
1	5	12	17
Total	**5**	**18**	**23**

Tableau 35: Matrice de confusion Random Forest pour le modèle de prédiction M5

Le tableau 35 nous indique horizontalement que sur les 6 classes logicielles initialement prédites comme ne contenant pas de bogues, aucune n'est effectivement classée comme telle. Sur les 17 classes logicielles qui initialement contiennent des bogues, 12 sont effectivement classées comme telles.

On remarque verticalement que sur les 5 classes logicielles prédites comme classes ne contenant aucun bogue aucune n'est effectivement classée comme telle. Tandis que sur les 18 classes logicielles prédites comme des classes logicielles comportant des bogues, 6 n'en contiennent pas.

- **KNN**

	0	**1**	**Total**
0	0	6	6
1	5	12	17
Total	**5**	**18**	**23**

Tableau 36:Matrice de confusion KNN pour le modèle de prédiction M5

Le tableau 36 nous indique horizontalement que sur les 6 classes logicielles initialement prédites comme ne contenant pas de bogues, aucune n'est effectivement classée comme telle. Sur les 17 classes logicielles qui initialement contiennent des bogues, 12 sont effectivement classées comme telles.

On remarque verticalement que sur les 5 classes logicielles prédites comme classes ne contenant aucun bogue aucune n'est effectivement classée comme telle. Tandis que sur les 18 classes logicielles prédites comme des classes logicielles comportant des bogues, 6 n'en contiennent pas.

- **Naïf Bayésien**

	0	1	Total
0	3	3	6
1	3	14	17
Total	**6**	**17**	**23**

Tableau 37: Matrice de confusion Naïf Bayésien pour le modèle de prédiction M5

Le tableau 37 nous indique horizontalement que sur les 6 classes logicielles initialement prédites comme ne contenant pas de bogues, 3 sont effectivement classées comme telles. Sur les 17 classes logicielles qui initialement contiennent des bogues, 14 sont effectivement classées comme telles.

On remarque verticalement que sur les 6 classes logicielles prédites comme classes ne contenant aucun bogue, 3 sont effectivement classées comme telles. Tandis que sur les 17 classes logicielles prédites comme des classes logicielles comportant des bogues, 3 n'en contiennent pas.

- **SVM**

	0	1	Total
0	2	5	7
1	6	10	16
Total	**8**	**15**	**23**

Tableau 38:Matrice de confusion SVM pour le modèle de prédiction M5

Le tableau 38 nous indique horizontalement que sur les 7 classes logicielles initialement prédites comme ne contenant pas de bogues, 2 sont effectivement classées comme telles. Sur les 16 classes logicielles qui initialement contiennent des bogues, 10 sont effectivement classées comme telles.

On remarque verticalement que sur les 8 classes logicielles prédites comme classes ne contenant aucun bogue 2 classes sont effectivement classées comme

telles. Tandis que sur les 15 classes logicielles prédites comme des classes logicielles comportant des bogues, 5 n'en contiennent pas.

- **Régression Logistique**

	0	1	Total
0	2	4	6
1	6	11	17
Total	**8**	**15**	**23**

Tableau 39:Matrice de confusion Régression Logistique pour le modèle de prédiction M5

Le tableau 39 nous indique horizontalement que sur les 6 classes logicielles initialement prédites comme ne contenant pas de bogues, 2 sont effectivement classées comme telles. Sur les 17 classes logicielles qui initialement contiennent des bogues, 11 sont effectivement classées comme telles.

On remarque verticalement que sur les 8 classes logicielles prédites comme classes ne contenant aucun bogue, 2 sont effectivement classées comme telles. Tandis que sur les 15 classes logicielles prédites comme des classes logicielles comportant des bogues, 4 n'en contiennent pas.

6.1.6 Modèle de prédiction M6

Dans le modèle de prédiction M_6, les jeux de données de l'application **Quran** ont servi à former le modèle et testés par la suite sur de nouveaux jeux de données de l'application mobile **Password Store Master**.

6.1.6.1 Test et Score- Courbes ROC

MODELS	AUC	PRECISION	RAPPEL
KNN	0,76	0,72	0,72
SVM	0,70	0,64	0,64
RF	0,72	0,70	0,70
ANN	0,82	0,84	0,88
NB	0,79	0,77	0,73
RL	0,82	0,84	0,88

Tableau 40:Résultats du modèle de prédiction M6

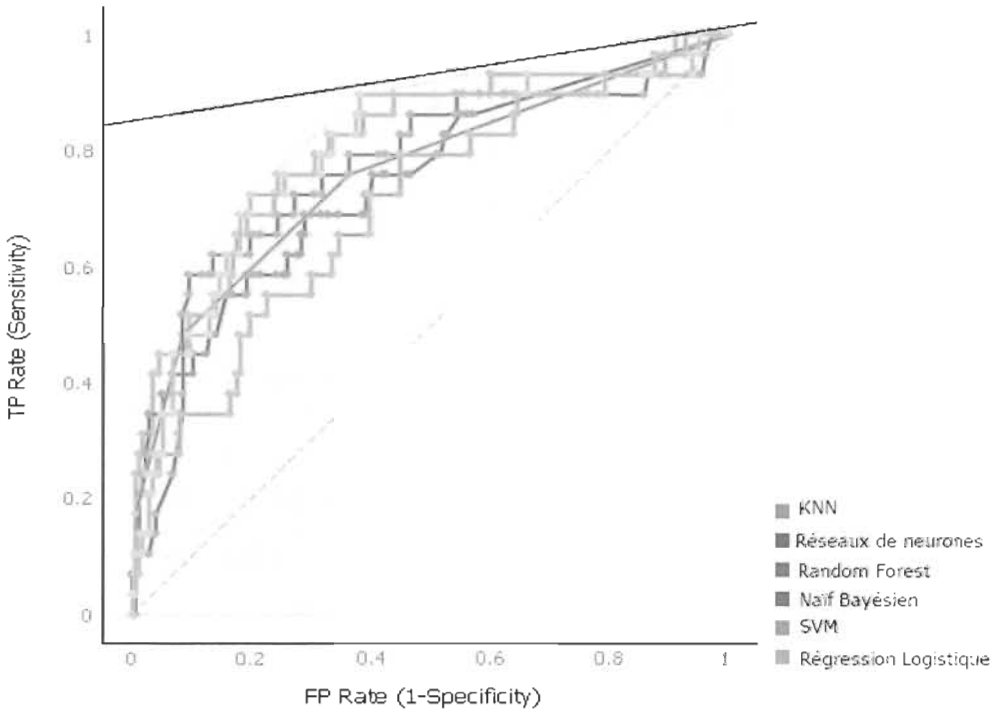

Figure 6:Courbes ROC modèle de prédiction M6

On constate que dans ce modèle, tous les classificateurs sont précis dans la prédiction des classes contenant des bogues. On peut constater que les valeurs de l'aire sous la courbe (AUC) varient de 0,70 à 0,82 et de 64% à 84% pour la proportion des classes susceptibles de contenir des bogues. De même on peut remarquer que les proportions des classes contenant effectivement des bogues sont également significatives. Elles varient de 64 % à 88%. Les Réseaux de

Neurones et la Régression Logistique sont meilleurs pour ce modèle de prédiction.

6.1.6.2 Matrices de confusion

- **Réseaux de Neurones**

	0	1	Total
0	2	2	4
1	3	16	19
Total	**5**	**18**	**23**

Tableau 41:Matrice de confusion Réseaux de Neurones pour le modèle de prédiction M6

Le tableau 41 nous indique horizontalement que sur les 4 classes logicielles initialement prédites comme ne contenant pas de bogues, 2 sont effectivement classées comme telle. Sur les 19 classes logicielles qui initialement contiennent des bogues, 16 sont effectivement classées comme telles.

On remarque verticalement que sur les 5 classes logicielles prédites comme classes ne contenant aucun bogue; 3 contiennent en fait des bogues. Tandis que sur les 18 classes logicielles prédites comme des classes logicielles comportant des bogues, 2 n'en contiennent pas.

- **Random Forest**

	0	1	Total
0	1	5	6
1	5	12	17
Total	**6**	**17**	**23**

Tableau 42:Matrice de confusion Random Forest pour le modèle de prédiction M6

Le tableau 42 nous indique horizontalement que sur les 6 classes logicielles initialement prédites comme ne contenant pas de bogues, une seule est

effectivement classée comme telle. Sur les 17 classes logicielles qui initialement contiennent des bogues, 12 sont effectivement classées comme telles.

On remarque verticalement que sur les 6 classes logicielles prédites comme classes ne contenant aucun bogue; 5 contiennent en fait des bogues. Tandis que sur les 17 classes logicielles prédites comme des classes logicielles comportant des bogues, 5 n'en contiennent pas en réalité.

- **KNN**

	0	**1**	**Total**
0	0	5	5
1	5	13	18
Total	**5**	**18**	**23**

Tableau 43:Matrice de confusion KNN pour le modèle de prédiction M6

Le tableau 43 nous indique horizontalement que sur les 5 classes logicielles initialement prédites comme ne contenant pas de bogues, aucune n'est effectivement classée comme telle. Sur les 18 classes logicielles contenant initialement des bogues, 13 sont effectivement classées comme telles.

On remarque verticalement que sur les 5 classes logicielles prédites comme classes ne contenant aucun bogue; aucune n'est effectivement classée comme telle. Tandis que sur les 18 classes logicielles prédites comme des classes logicielles comportant des bogues, 5 n'en contiennent pas.

- **Naïf Bayésien**

	0	1	Total
0	0	5	5
1	4	14	18
Total	**4**	**19**	**23**

Tableau 44:Matrice de confusion Naïf Bayésien pour le modèle de prédiction M6

Le tableau 44 nous indique horizontalement que sur les 5 classes logicielles initialement prédites comme ne contenant pas de bogues, aucune n'est effectivement classée comme telle. Sur les 18 classes logicielles contenant initialement des bogues, 14 sont effectivement classées comme telles.

On remarque verticalement que sur les 4 classes logicielles prédites comme classes ne contenant aucun bogue; aucune n'est effectivement classée comme telle. Tandis que sur les 19 classes logicielles prédites comme des classes logicielles comportant des bogues, 5 n'en contiennent pas en réalité.

- **SVM**

	0	1	Total
0	0	6	6
1	6	11	17
Total	**6**	**17**	**23**

Tableau 45:Matrice de confusion SVM pour le modèle de prédiction M6

Le tableau 45 nous indique horizontalement que sur les 6 classes logicielles initialement prédites comme ne contenant pas de bogues, aucune n'est effectivement classée comme telles. Sur les 17 classes logicielles contenant initialement des bogues, 11 classes sont effectivement classées comme telles.

On remarque verticalement que sur les 6 classes logicielles prédites comme classes ne contenant aucun bogue; aucune n'est effectivement classée comme

telles. Tandis que sur les 17 classes logicielles prédites comme des classes logicielles comportant des bogues, 6 n'en contiennent pas en réalité.

- **Régression Logistique**

	0	1	Total
0	2	2	4
1	3	16	19
Total	**5**	**18**	**23**

Tableau 46: Matrice de confusion Régression logistique pour le modèle de prédiction M6

Le tableau 46 nous indique horizontalement que sur les 5 classes logicielles initialement prédites comme ne contenant pas de bogues, 2 classes sont effectivement classées comme telles. Sur les 18 classes logicielles qui initialement contiennent des bogues, 16 sont effectivement classées comme telles.

On remarque verticalement que sur les 5 classes logicielles prédites comme classes ne contenant aucun bogue; 2 sont effectivement classées comme telles. Tandis que sur les 18 classes logicielles prédites comme des classes logicielles comportant des bogues, 2 n'en contiennent pas en réalité.

6.1.7 Modèle de prédiction M7

Dans le modèle de prédiction M_7, les jeux de données de l'application **Android PDF Viewer** ont servi à former le modèle et testés par la suite sur de nouveaux jeux de données de l'application mobile **Quran**.

6.1.7.1 Test et Score- Courbes ROC

MODELS	AUC	PRECISION	RAPPEL
KNN	0,73	0,76	0,73
SVM	0,80	0,83	0,85
RF	0,74	0,78	0,75
ANN	0,80	0,86	0,77
NB	0,69	0,66	0,67
RL	0,78	0,82	0,81

Tableau 47: Résultats du modèle de prédiction M7

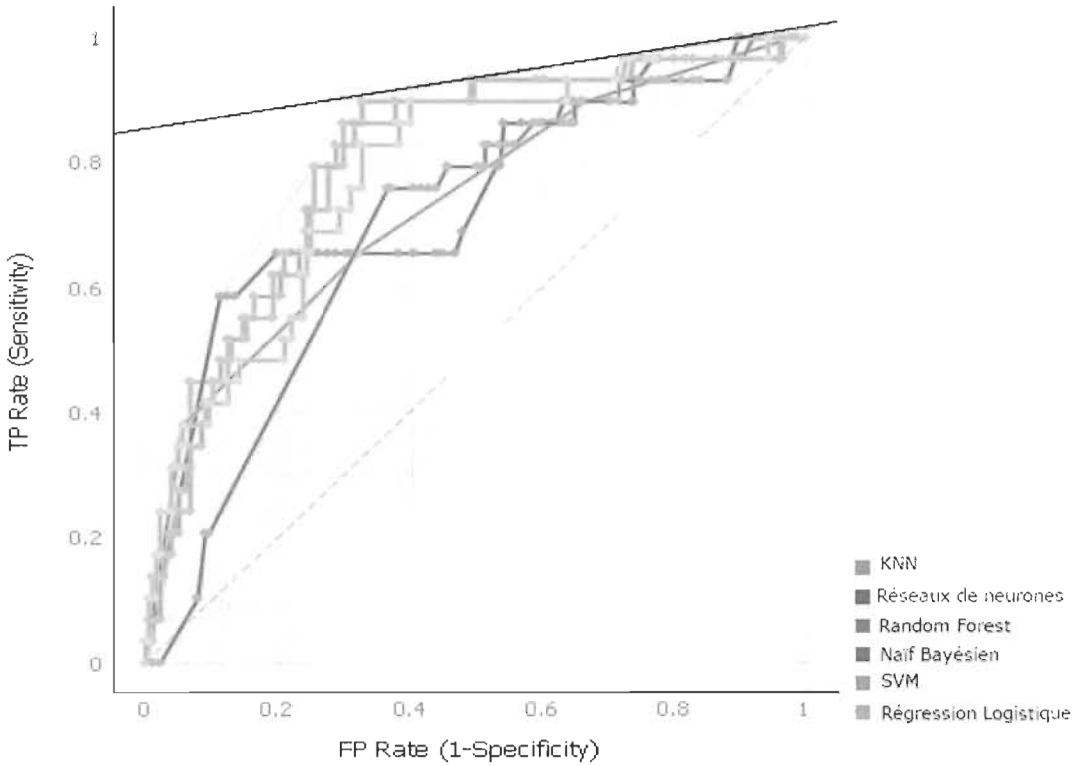

Figure 7:Courbes ROC du modèle de prédiction M7

On constate que pour ce modèle, tous les classificateurs sont précis dans la prédiction des classes qui contiennent des bogues sauf le Naïf Bayésien (AUC=0,69). On peut constater que les valeurs de l'aire sous la courbe (AUC) varient de 0,73 à 0,80 et de 76% à 86% pour la proportion des classes susceptibles de contenir des bogues. De même on peut remarquer que les

proportions des classes contenant effectivement des bogues sont également significatives. Elles varient de 73 % à 85%. Les Réseaux de Neurones et les SVM sont meilleurs pour ce modèle de prédiction.

6.1.7.2 Matrices de Confusion

- **Réseaux de Neurones**

	0	1	Total
0	63	29	92
1	15	100	115
Total	**78**	**129**	**207**

Tableau 48: Matrice de confusion Réseaux de Neurones pour le modèle de prédiction M7

Le tableau 48 nous indique horizontalement que sur les 92 classes logicielles initialement prédites comme ne contenant pas de bogues, 63 sont effectivement classées comme telles. Sur les 115 classes logicielles qui initialement contiennent des bogues, 100 sont effectivement classées comme telles.

On remarque verticalement que sur les 78 classes logicielles prédites comme classes ne contenant aucun bogue; 15 contiennent en fait des bogues. Tandis que sur les 129 classes logicielles prédites comme des classes logicielles comportant des bogues, 29 n'en contiennent pas en réalité.

- **Random Forest**

	0	1	Total
0	63	29	92
1	25	90	115
Total	**88**	**119**	**207**

Tableau 49:Matrice de confusion Random Forest pour le modèle de prédiction M7

Le tableau 49 nous indique horizontalement que sur les 92 classes logicielles initialement prédites comme ne contenant pas de bogues, 63 sont effectivement classées comme telles. Sur les 115 classes logicielles qui initialement contiennent des bogues, 90 sont effectivement classées comme telles.

On remarque verticalement que sur les 88 classes logicielles prédites comme classes ne contenant aucun bogue; 25 contiennent en fait des bogues. Tandis que sur les 119 classes logicielles prédites comme des classes logicielles comportant des bogues, 29 n'en contiennent pas en réalité.

- **KNN**

	0	**1**	**Total**
0	57	33	90
1	27	90	117
Total	**84**	**123**	**207**

Tableau 50:Matrice de confusion KNN pour le modèle de prédiction M7

Le tableau 50 nous indique horizontalement que sur les 90 classes logicielles initialement prédites comme ne contenant pas de bogues, 57 sont effectivement classées comme telles. Sur les 117 classes logicielles qui initialement contiennent des bogues, 90 sont effectivement classées comme telles.

On remarque verticalement que sur les 84 classes logicielles prédites comme classes ne contenant aucun bogue; 27 contiennent en fait des bogues. Tandis que sur les 123 classes logicielles prédites comme des classes logicielles comportant des bogues, 33 n'en contiennent pas en réalité.

- **Naïf Bayésien**

	0	1	Total
0	48	39	87
1	40	80	120
Total	**88**	**119**	**207**

Tableau 51: Matrice de confusion Naïf Bayésien pour le modèle de prédiction M7

Le tableau 51 nous indique horizontalement que sur les 87 classes logicielles initialement prédites comme ne contenant pas de bogues, 48 classes sont effectivement classées comme telle. Sur les 120 classes logicielles qui initialement contiennent des bogues, 80 sont effectivement classées comme telles.

On remarque verticalement que sur les 88 classes logicielles prédites comme classes ne contenant aucun bogue; 40 contiennent en fait des bogues. Tandis que sur les 119 classes logicielles prédites comme des classes logicielles comportant des bogues, 39 n'en contiennent pas en réalité.

- **SVM**

	0	1	Total
0	80	12	92
1	20	100	120
Total	**90**	**117**	**207**

Tableau 52:Matrice de confusion SVM pour le modèle de prédiction M7

Le tableau 52 nous indique horizontalement que sur les 92 classes logicielles initialement prédites comme ne contenant pas de bogues, 80 sont effectivement classées comme telles. Sur les 120 classes logicielles qui initialement contiennent des bogues, 100 classes sont effectivement classées comme telles.

On remarque verticalement que sur les 90 classes logicielles prédites comme classes ne contenant aucun bogue; 20 contiennent en fait des bogues. Tandis que sur les 100 classes logicielles prédites comme des classes logicielles comportant des bogues, 12 n'en contiennent pas en réalité.

- **Régression Logistique**

	0	1	Total
0	64	22	86
1	21	100	121
Total	**85**	**122**	**207**

Tableau 53:Matrice de confusion Régression logistique pour le modèle de prédiction M7

Le tableau 53 nous indique horizontalement que sur les 86 classes logicielles initialement prédites comme ne contenant pas de bogues, 64 sont effectivement classées comme telles. Sur les 121 classes logicielles qui initialement contiennent des bogues, 100 classes sont effectivement classées comme telles.

On remarque verticalement que sur les 85 classes logicielles prédites comme classes ne contenant aucun bogue; 21 contiennent en fait des bogues. Tandis que sur les 122 classes logicielles prédites comme des classes logicielles comportant des bogues, 22 n'en contiennent pas en réalité.

6.1.8 Modèle de prédiction M_8

Dans le modèle de prédiction M_8, les jeux de données de l'application **Password Store Master** ont servi à former le modèle et testés par la suite sur de nouveaux jeux de données de l'application mobile **Quran**.

6.1.8.1 Test et Score- Courbes ROC

MODELS	AUC	PRECISION	RAPPEL
KNN	0,55	0,50	0,50
SVM	0,45	0,50	0,45
RF	0,38	0,20	0,20
ANN	0,62	0,58	0,60
NB	0,61	0,52	0,57
RL	0,42	0,40	0,33

Tableau 54:Résultats modèle de prédiction M8

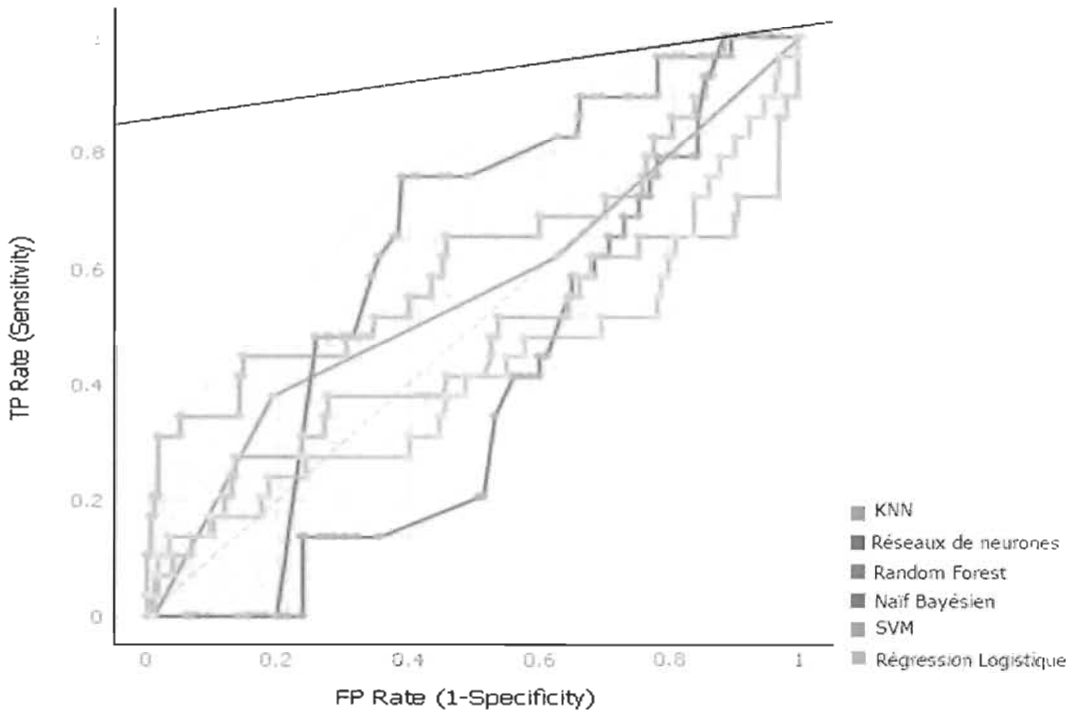

Figure 8: Courbes ROC modèle de prédiction M8

On remarque qu'aucun classificateur n'est précis dans la prédiction des classes contenant des bogues à l'usage. Les différentes valeurs de l'AUC indiquent des classifications presque aléatoires.

Ceci résulte du fait que les jeux d'entraînement ne sont pas suffisants pour entrainer le modèle. De plus nous disposons dans notre cas de plus de données test que des données ayant servi à entraîner le modèle.

6.1.8.2 Matrices de confusion

- **Réseaux de Neurones**

	0	1	Total
0	41	46	87
1	50	70	120
Total	**91**	**116**	**207**

Tableau 55:Matrice de confusion Réseaux de Neurones pour le modèle de prédiction M8

Le tableau 55 nous indique horizontalement que sur les 87 classes logicielles initialement prédites comme ne contenant pas de bogues, 41 sont effectivement classées comme telle. Sur les 120 classes logicielles qui initialement contiennent des bogues, 70 classes sont effectivement classées comme telles.

On remarque verticalement que sur les 91 classes logicielles prédites comme classes ne contenant aucun bogue; 50 contiennent en fait des bogues. Tandis que sur les 116 classes logicielles prédites comme des classes logicielles comportant des bogues, 46 n'en contiennent pas en réalité.

- **Random Forest**

	0	1	Total
0	27	80	107
1	80	20	100
Total	**107**	**100**	**207**

Tableau 56:Matrice de confusion Random Forest pour le modèle de prédiction M8

Le tableau 56 nous indique horizontalement que sur les 107 classes logicielles initialement prédites comme ne contenant pas de bogues, 27 sont effectivement classées comme telles. Sur les 100 classes logicielles qui

initialement contiennent des bogues, 20 sont effectivement classées comme telles.

On remarque verticalement que sur les 107 classes logicielles prédites comme classes ne contenant aucun bogue; 80 contiennent en fait des bogues.

Tandis que sur les 100 classes logicielles prédites comme des classes logicielles comportant des bogues, 80 n'en contiennent pas en réalité.

- **KNN**

	0	**1**	**Total**
0	27	60	87
1	60	60	120
Total	**87**	**120**	**207**

Tableau 57:Matrice de confusion KNN pour le modèle de prédiction M8

Le tableau 57 nous indique horizontalement que sur les 87 classes logicielles initialement prédites comme ne contenant pas de bogues, 27 sont effectivement classées comme telles. Sur les 120 classes logicielles qui initialement contiennent des bogues, 60 sont effectivement classées comme telles.

On remarque verticalement que sur les 87 classes logicielles prédites comme classes ne contenant aucun bogue; 60 contiennent en fait des bogues. Tandis que sur les 120 classes logicielles prédites comme des classes logicielles comportant des bogues, 60 n'en contiennent pas en réalité.

- **Naïf Bayésien**

	0	**1**	**Total**
0	31	49	80
1	60	67	127
Total	**91**	**116**	**207**

Tableau 58: Matrice de confusion Naïf Bayésien pour le modèle de prédiction M8

Le tableau 58 nous indique horizontalement que sur les 80 classes logicielles initialement prédites comme ne contenant pas de bogues, 31 sont effectivement classées comme telles. Sur les 127 classes logicielles qui initialement contiennent des bogues, 60 sont effectivement classées comme telles.

On remarque verticalement que sur les 91 classes logicielles prédites comme classes ne contenant aucun bogue; 60 contiennent en fait des bogues. Tandis que sur les 116 classes logicielles prédites comme des classes logicielles comportant des bogues, 49 n'en contiennent pas en réalité.

- **SVM**

	0	**1**	**Total**
0	47	60	107
1	50	50	100
Total	**97**	**110**	**207**

Tableau 59: Matrice de confusion SVM pour le modèle de prédiction M8

Le tableau 59 nous indique horizontalement que sur les 107 classes logicielles qui initialement sont prédites comme ne contenant pas de bogues, 47 classes sont effectivement classées comme telles. Sur les 100 classes logicielles qui initialement contiennent des bogues, 50 classes sont effectivement classées comme telles.

On remarque verticalement que sur les 97 classes logicielles prédites comme classes ne contenant aucun bogue; 50 contiennent en fait des bogues. Tandis que sur les 110 classes logicielles prédites comme des classes logicielles comportant des bogues, 60 n'en contiennent pas en réalité.

- **Régression Logistique**

	0	1	Total
0	27	80	107
1	60	40	100
Total	**87**	**120**	**207**

Tableau 60:Matrice de confusion Régression logistique pour le modèle de prédiction M8

Le tableau 60 nous indique horizontalement que sur les 107 classes logicielles qui initialement sont prédites comme ne contenant pas de bogues, 27 classes sont effectivement classées comme telles. Sur les 100 classes logicielles qui initialement contiennent des bogues, 40 classes sont effectivement classées comme telles.

On remarque verticalement que sur les 87 classes logicielles prédites comme classes ne contenant aucun bogue; 60 contiennent en fait des bogues. Tandis que sur les 120 classes logicielles prédites comme des classes logicielles comportant des bogues, 80 n'en contiennent pas en réalité.

6.1.9 Modèle de prédiction M_9

Dans le modèle M_9, les jeux de données des applications **Password Store Master** et **Android PDF Viewer** ont servi à former les classificateurs qui ont ensuite été validés et testé par la suite sur de nouveaux jeux de données de l'application mobile **Quran**.

6.1.9.1 Test et Score- Courbes ROC

MODELS	AUC	PRECISION	RAPPEL
KNN	0,68	0,83	0,70
SVM	0,79	0,79	0,79
RF	0,70	0,88	0,78
ANN	0,82	0,94	0,88
NB	0,76	0,87	0,75
RL	0,78	0,86	0,76

Tableau 61: Résultats modèle de prédiction M9

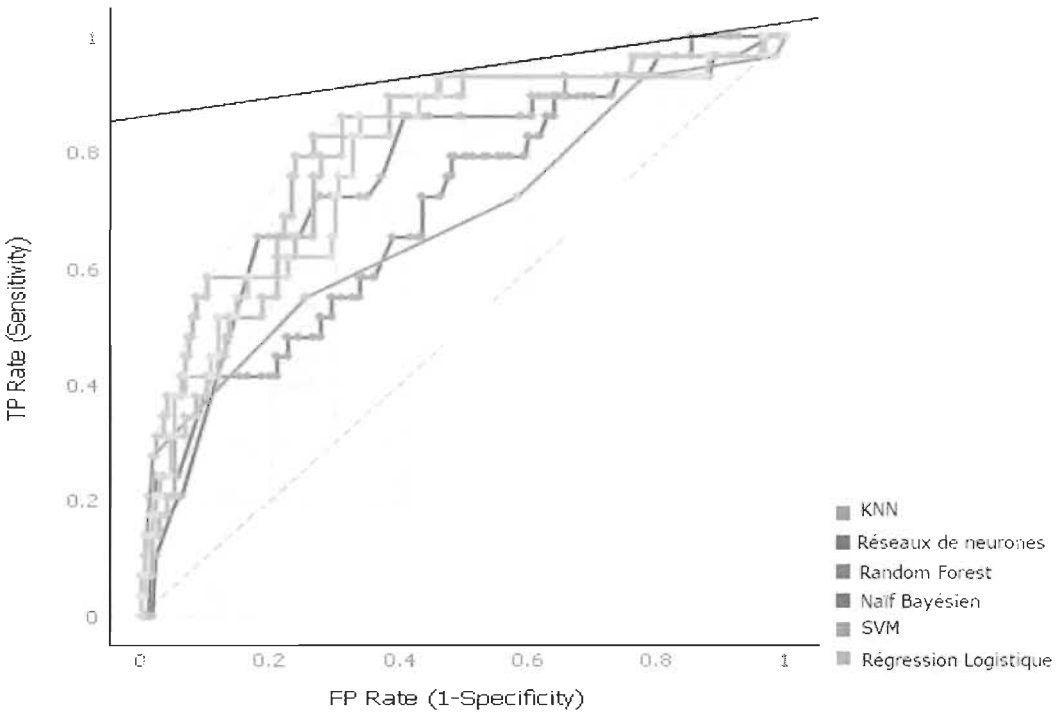

Figure 9: Courbes ROC modèles de prédiction M9

On constate que pour ce modèle, tous les classificateurs sont précis dans la prédiction des classes qui contiennent des bogues à l'usage sauf les KNN (AUC=0,68). On peut constater que les valeurs de l'aire sous la courbe (AUC) varient de 0,70 à 0,82 et de 79% à 94% pour la proportion des classes susceptibles de contenir de bogues. De même on peut remarquer que les proportions des classes contenant effectivement des bogues sont également

significatives. Elles varient de 75 % à 88%. Les Réseaux de Neurones sont meilleurs pour ce modèle de prédiction.

6.1.9.2 Matrices de Confusion

- **Réseaux de Neurones**

	0	1	Total
0	56	16	72
1	8	127	135
Total	**64**	**143**	**207**

Tableau 62:Matrice de confusion Réseaux de Neurones pour le modèle de prédiction M9

Le tableau 62 nous indique horizontalement que sur les 72 classes logicielles qui initialement sont prédites comme ne contenant pas de bogues, 56 classes sont effectivement classées comme telle. Sur les 135 classes logicielles qui initialement contiennent des bogues, 127 classes sont effectivement classées comme telles.

On remarque aussi verticalement que sur les 64 classes logicielles prédites comme classes ne contenant aucun bogue; 8 contiennent en fait des bogues. Tandis que sur les 143 classes logicielles prédites comme des classes logicielles comportant des bogues, 16 n'en contiennent pas en réalité.

- **Random Forest**

	0	1	Total
0	42	33	75
1	15	117	132
Total	**57**	**150**	**207**

Tableau 63:Matrice de confusion Random Forest pour le modèle de prédiction M9

Le tableau 63 nous indique horizontalement que sur les 75 classes logicielles qui initialement sont prédites comme ne contenant pas de bogues, 42

classes sont effectivement classées comme telle. Sur les 132 classes logicielles qui initialement contiennent des bogues, 117 classes sont effectivement classées comme telles.

On remarque verticalement que sur les 57 classes logicielles prédites comme classes ne contenant aucun bogue; 15 contiennent en fait des bogues. Tandis que sur les 150 classes logicielles prédites comme des classes logicielles comportant des bogues, 33 n'en contiennent pas en réalité.

- **KNN**

	0	**1**	**Total**
0	31	45	76
1	21	110	131
Total	**52**	**155**	**207**

Tableau 64: Matrice de confusion KNN pour le modèle de prédiction M9

Le tableau 64 nous indique horizontalement que sur les 76 classes logicielles qui initialement sont prédites comme ne contenant pas de bogues, 31 classes sont effectivement classées comme telles. Sur les 131 classes logicielles qui initialement contiennent des bogues, 110 classes sont effectivement classées comme telles. On remarque verticalement que sur les 52 classes logicielles prédites comme classes ne contenant aucun bogue; 21 contiennent en fait des bogues. Tandis que sur les 155 classes logicielles prédites comme des classes logicielles comportant des bogues, 45 n'en contiennent pas en réalité.

- **Naïf Bayésien**

	0	**1**	**Total**
0	38	37	75
1	16	116	132
Total	**54**	**153**	**207**

Tableau 65: Matrice de confusion Naïf Bayésien pour le modèle de prédiction M9

Le tableau 65 nous indique horizontalement que sur les 75 classes logicielles qui initialement sont prédites comme ne contenant pas de bogues, 38 classes sont effectivement classées comme telle. Sur les 132 classes logicielles qui initialement contiennent des bogues, 116 classes sont effectivement classées comme telles.

On remarque verticalement que sur les 54 classes logicielles prédites comme classes ne contenant aucun bogue; 16 contiennent en fait des bogues. Tandis que sur les 153 classes logicielles prédites comme des classes logicielles comportant des bogues, 37 n'en contiennent pas en réalité.

- **SVM**

	0	**1**	**Total**
0	39	31	70
1	16	121	137
Total	**55**	**152**	**207**

Tableau 66:Matrice de confusion SVM pour le modèle de prédiction M9

Le tableau 66 nous indique horizontalement que sur les 70 classes logicielles qui initialement sont prédites comme ne contenant pas de bogues, 39 classes sont effectivement classées comme telle. Sur les 137 classes logicielles qui initialement contiennent des bogues, 121 classes sont effectivement classées comme telles.

On remarque verticalement que sur les 55 classes logicielles prédites comme classes ne contenant aucun bogue; 16 contiennent en fait des bogues. Tandis que sur les 152 classes logicielles prédites comme des classes logicielles comportant des bogues, 31 n'en contiennent pas en réalité.

- **Régression Logistique**

	0	1	Total
0	34	36	70
1	18	119	137
Total	**52**	**155**	**207**

Tableau 67: Matrice de confusion Régression logistique pour le modèle de prédiction M9

Le tableau 67 nous indique horizontalement que sur les 70 classes logicielles qui initialement sont prédites comme ne contenant pas de bogues, 34 classes sont effectivement classées comme telle. Sur les 137 classes logicielles qui initialement contiennent des bogues, 119 classes sont effectivement classées comme telles.

On remarque aussi verticalement que sur les 52 classes logicielles prédites comme classes ne contenant aucun bogue; 18 contiennent en fait des bogues. Tandis que sur les 155 classes logicielles prédites comme des classes logicielles comportant des bogues, 36 n'en contiennent pas en réalité.

6.2 Synthèse des résultats

Le tableau 68 résume de classement des différents modèles de prédiction que nous avons construits avec l'approche utilisée dans notre travail. Dans ce tableau nous avons inscrit pour chaque modèle de prédiction les résultats d'expérimentation d'apprentissage automatique des deux meilleurs classificateurs les plus significatifs. Le modèle de prédiction M8 fait un classement aléatoire avec une faible liaison entre les bogues et les différentes classes logicielles concernées. Il est à noter que tous les classificateurs des modèles M1, M3, M4, M6 ont effectué une classification assez significative dans la prédiction des classes susceptibles de contenir des bogues; mais nous avons ressorti dans le tableau 68 ci-dessous les meilleurs parmi les meilleurs.

Rang	Modèles	Meilleurs algorithmes	AUC	Précision
1	M1	Réseaux de Neurones	0.85	88%
		SVM	0.81	80%
2	M3	Réseaux de Neurones	0.81	80%
		Régression Logistique	0.82	89%
3	M4	Réseaux de Neurones	0.84	87%
		Random Forest/SVM	0.81	72%
4	M6	Réseaux de Neurones	0.82	84%
		Régression Logistique	0.82	84%
5	M7	Réseaux de Neurones	0.86	100%
		SVM	0.80	83%
6	M9	Réseaux de Neurones	0.82	94%
		Régression Logistique	0.86	76%
7	M2	Réseaux de Neurones	0.77	74%
		Régression Logistique	0.75	71%
8	M5	Réseaux de Neurones	0.71	73%
		Naïf Bayésien	0.71	82%

Tableau 68: Synthèse des résultats des modèles de prédiction

Rappelons qu'un classificateur parfait a une AUC=1, un modèle sans aucun pouvoir de classification a une AUC autour de 0.5 et un modèle de prédiction aléatoire a une AUC= 0.5. Rappelons aussi que plus le score de précision est proche de 100%, plus le modèle de prédiction est efficace dans la classification des vraies positives. Considérant ces rappels et en observant les résultats expérimentaux d'apprentissage automatique; on constate que tous les modèles de prédiction obtiennent pour chacun des deux meilleurs classificateurs une AUC > 0.70 et une Précision ≥ 71%. Nous pouvons en déduire que les classificateurs concernés dans ces différents modèles de prédiction ont été très bons dans la prédiction des classes susceptibles de contenir des bogues. Les **Réseaux de Neurone** et la **Régression Logistique** se démarquent nettement des autres classificateurs du fait de leur présence répétitive dans le rang des meilleurs classificateurs.

Il est à noter que les systèmes qui ont permis de construire les meilleurs modèles de prédiction sont majoritairement développés en Java et contiennent 2570 lignes d'instructions toutes classes confondues. On remarque aussi que les deux systèmes comportent 230 classes logicielles pour un total de 32 superclasses dans une hiérarchie d'héritage (DIT+NOC). Ces dernières ont un impact immédiat sur toutes les classes dérivées.

6.3 Menace pour la validité

Les résultats obtenus à travers cette étude sont assez concluants. Cependant, il y a un certain nombre de considérations qu'il faudra prendre en compte en ce qui concerne ses limitations et sa validité.

6.3.1 Validité interne

La collecte de données sur les fautes logicielles avec Github est faite de façon empirique. Malgré les améliorations apportées, l'utilisation des commentaires des « commits » ne garantit pas à coup sûr la détection de patch pour un bogue donné. En plus, l'assignation d'un bogue à une classe est faite sans prendre en considération le fait qu'il puisse s'agir d'une répercussion de la correction d'un bogue. En effet, aucune distinction n'est faite entre les classes effectivement fautives et celles impactées par la correction d'un bogue. De ce fait, il peut arriver que certaines classes, probablement celles ayant un fort couplage et une complexité importante, puissent être atteintes par des fautes sévères, alors qu'elles ne sont pas en réalité à l'origine de ces fautes.

Par ailleurs, le choix de fusionner les données de différents systèmes pourrait être source de questionnements car certains systèmes ne sont pas entièrement développés en (Java). Mais cette approche, utilisée dans d'autres études, a fait ses preuves dans la qualité des résultats obtenus. Elle permet en même temps de réduire le biais de nos résultats pour les systèmes étudiés.

6.3.2 Validité externe

Dans ce travail, nous avons mené des expérimentations empiriques sur trois systèmes Android développées en majorité en JAVA. Par conséquent, toutes les conclusions qui découlent de nos résultats restent limitées. Nous ne pourrons pas les généraliser pour le reste des langages orientés objet, bien que le langage JAVA soit un bon représentant de ces langages. Pour cela, il faudrait conduire d'autres études basées sur d'autres langages orientés objets et/ou d'autres systèmes dans divers domaines.

6.3.3 Validité de construction

Pour construire les différents modèles de prédiction, nous avons calculé les métriques logicielles de chaque système à l'aide de l'outil « **CodeMR** ». Ce dernier est implémenté dans le logiciel Android Studio pour calculer les différentes métriques orientées objets utilisées dans notre étude. Plusieurs autres métriques logicielles n'ont pas été prises en compte dans notre étude. Par ailleurs, nous avons sélectionné seulement quelques algorithmes de classification pour l'apprentissage automatique de nos modèles de prédiction basés sur les métriques logicielles suivantes : la complexité, le couplage, l'héritage et la taille.

CONCLUSION

On peut souvent constater des défaillances à l'utilisation d'une application mobile Android. La présence de bogues est la cause fondamentale de ces défaillances. Au cours du processus de développement, une bonne partie du développement est consacrée au test logiciel afin d'assurer la qualité de l'application. Ainsi l'approche basée sur les métriques logicielles et l'apprentissage automatique pour la prédiction des bogues peut être un outil pour prédire les classes susceptibles de contenir des bogues et y focaliser les efforts de test.

Au cours de notre approche, nous avons mené des études empiriques sur environ 3225 lignes de code reparties dans environs 296 classes à travers 03 applications mobiles Android Open Source écrites en Java, Kotlin et Script Shell. Après avoir relevé le nombre de bogues (bogues majeurs affectant le fonctionnement normal du système) dans chaque classe logicielle, nous avons calculé les métriques logicielles (DIT, NOC, WMC, RFC, CBO, LOC) en analysant leurs codes sources respectifs. Afin d'atteindre le dernier objectif spécifique de notre étude, nous avons utilisé des classificateurs issus de différentes approches d'apprentissages supervisés suivants : les Réseaux de Neurones, le Naïf Bayésien, les machines à vecteur de support (SVM), la méthode des k plus proche voisins (KNN), les forêts d'arbres décisionnels (Random Forest) et la Régression Logistique. Ces modèles ont permis de construire 09 classificateurs en appliquant la méthode leave-one-out cross-validation. Les performances des classificateurs obtenus ont été évaluées grâce aux matrices de confusion et à l'AUC. Les différents résultats d'expérimentation de l'apprentissage automatique indiquent que parmi les 03 systèmes étudiés, seul le système Password Store Master développé en Kotlin (conçu pour interagir pleinement avec Java) et Script shell (conçu pour interpréter des commandes sous Unix), a été difficile à prédire. Il se retrouve dans les modèles de prédiction M2, M5 et

M8 qui ont le moins performé d'après nos expérimentations. Outre la complexité de ce système, nous avons remarqué un problème de compatibilité effective avec les autres systèmes dans la fusion des données pour l'entraînement des modèles d'une part et pour les données de test d'autre part. Quant aux autres modèles de prédiction, les résultats obtenus sont assez concluants avec des aires sous la courbe (AUC) significatives et des précisions pour la plupart très significatives. Il en résulte donc que dans un modèle de prédiction, la fusion des données peut influencer négativement les résultats obtenus si tous les systèmes étudiés ne sont pas entièrement développés dans le même langage.

Malgré l'efficacité de notre approche, nous avons constaté une certaine limite sur un système qui n'est pas développé entièrement dans le même langage que les deux autres. Il s'agit bien sûr du système « **Password Store Master** ». Il obtient un résultat moins pire quand ses données sont parfois fusionnées avec le système « **Android PDF Viewer** ». Des études supplémentaires devront être menées pour comprendre les raisons d'un tel manque d'efficacité. La validation des résultats ne peut être exhaustive et généralisée, mais reste robuste et cohérente en tenant compte de la méthodologie et de l'approche misent en place pour atteindre ces résultats.

REFERENCES BIBLIOGRAPHIQUES

[1] "Statistiques mondiales sur le mobile en 2021", https://zenuacademie.com/marketing/marketing-mobile/statistiques-mondiales-mobile visité le 05 février 2021.

[2] M. Linares-Vásquez, K. Moran, D. Poshyvanyk "Continuous, Evolutionary and Large-Scale: A New Perspective for Automated Mobile App Testing", in IEEE International Conference on Software Maintenance and Evolution (ICSME), September 2017.

[3] "IEEE Recommended Practice for Software Requirements Specifications",

https://ieeexplore.ieee.org/document/392555 visité le 08 août 2021

[4] "IEEE Standard Glossary of Software Engineering Terminology",

https://ieeexplore.ieee.org/document/159342 visité le 08 août 2021

[5] L. Cruz, R. Abreu, D. Lo "To the Attention of Mobile Software Developers: Guess What, test your App!", in Empirical Software Engineering, Volume 24, Issue 4, pp 2438–2468 August 2019.

[6] "Introduction aux stratégies de déploiement", https://ludovicwyffels.dev/deploy/ visité le 08 août 2021.

[7] H. Muccini et A. Di Francesco, P. Esposito "Software testing of mobile applications: Challenges and future research directions", in 7th International Workshop on Automation of Software Test (AST), 2012.

[8] "NIST (Natonal Institute of Stadards & Technology)" The Economic Impact of Inadequate Infrastructure for Software Testing planning, Report 2002-2003.

[9] "Statistiques système Android", https://www.lapresse.ca/techno/mobilite/201807/17/01-5189864-android-de-google-regne-sur-les-systemes-dexploitation.php visité le 08 février 2021.

[10] "Statista", https://fr.statista.com/a-propos/notre-engagement-pour-la-recherche visité le 07 février 2021

[11] "Nombre d'applications disponibles sur Google Play Store", https://fr.statista.com/statistiques/565393/nombre-d-applications-disponibles-sur-google-play-store/ visité le 08 février 2021.

[12] J. Printz, JP. Pradat-Peyre "Pratique des tests logiciels - concevoir et mettre en œuvre une stratégie de tests", préparation de la certification ISTQB, Février 2014.

[13] S. Chidamber et C. Kemerer "A metrics suite for object, oriented design", in IEEE Transactions on, Software Engineering 1994, vol. 20, no 6, p. 476-493.

[14] F. Brito et R. Carapuga "Candidate metrics for object-oriented software within a taxonomy framework", on Journal of System and Software, vol. 26, no. 1, North-Holland, Elsevier Science, July 1994.

[15] R. Martin "Object oriented design quality metrics an analysis of dependencies" published the October 28,1994.

[16] Computer Glossaries, IEEE Standard Computer Dictionary, A Compilation of IEEE Standard, IEEE Std. 610-1991.

[17] F. TOURE "Orientation de l'effort des tests unitaires dans les systèmes orientés objet: une approche basée sur les métriques logicielles". Thèse présentée à l'université de Laval en vue de l'obtention du grade de Philosophie Doctor (Ph.D.) en informatique publié en 2016.

[18] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell "A Systematic Literature Review on Fault Prediction Performance in Software Engineering", in IEEE Transactions on Software Engineering (Volume: 38, Issue: 6, Nov.-Dec. 2012).

[19] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, et B. Murphy "Cross-Project Defect Prediction: A Large-Scale Experiment on Data Vs. Domain Vs. Process" in Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, pages 91-100, August 2009.

[20] K. Kaur, Parvinder K, Amandeep B.S "Early software fault prediction using real time defect data", in Second International Conference on Machine Vision, pages 243-245, 2009.

[21] "NASA IV &V Facility, Metric Data Program", available from http://MDP.ivv.nasa.gov visité le 11 février 2021.

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

[23] K. Aggarwal, Y. Singh, A. Kaur, et R. Malhotra "Empirical Analysis for Investigating the Effect of Object-Oriented Metrics on Fault Proneness: A

Replicated Case Study", in Software Process Improvement and Practice, vol. 14, no. 1, pp. 39-62, 2009.

[24] S. Rathore et A. Gupta "Investigating Object-Oriented Design Metrics to Predict Fault-Proneness of Software Modules, Software engineering", in CSI 6h International Conference on Software Engineering, September 2012.

[25] "The 15th International Conference on Predictive Models and Data Analytics in Software Engineering", www.promisedata.org visité le 12 février 2021.

[26] T. Gyimothy, R. Ferenc et I. Siket "Empirical Validation of Object-Oriented Metrics on Open-Source Software for Fault Prediction", in IEEE Trans. Software Eng., vol. 31, no. 10, pp. 897-910, October 2005.

[27] M. Thwin, T. Quah "Application of Neural Networks for Predicting Software Quality Using Object-Oriented Metrics", in J Systems and Software, vol. 76, non. 2, p. 147-156, 2005.

[28] G. Sharma, Sharma et Gujral "A Novel Way of Assessing Software Bug Severity Using Dictionary of Critical Terms", in Procedia Computer Science 70: 632-39, 2015.

[29] S. Bouktif "Amélioration de la prédiction de la qualité du logiciel par combinaison et adaptation de modèles". Thèse présentée à la Faculté des études supérieures en vue de l'obtention du grade de Philosophie Doctor (Ph.D.) en informatique publié en mai 2005.

[30] M. Linares-Vâsquez, S. Klock, C. McMillan, A. Sabané, D. Poshyvanyk, et G. Yann-Gaël "Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in Java mobile apps", in Proceedings of the 22nd International Conference on Program Comprehension, pages 232–243, June 2014.

[31] R. Minelli, M. Lanza "Software analytics for mobile applications insights and lessons learned", in 11th European Conference on Software Maintenance and Reengineering (CSMR), p.144- 153, March 2013.

[32] G. Recht "Détection et analyse de l'impact des failles de code dans les applications mobiles", thèse pprésentée et soutenue publiquement le 30 Novembre 2016.

[33] "Métriques logicielles", https://fr.wikipedia.org/wiki/Métrique_(logiciel) visité le 14 décembre 2021.

[34] T. McCabe "Une mesure de la complexité logicielle", in IEEE Transactions on Software Engineering, Vol 2, Numéro 4, pp 308-320, 1976.

[35] M. Bouguessa, "Forage de données", notes de cours 2015. Université du Québec à Montréal, 88 pages.

[36] "Apprentissage non supervisé" et "Apprentissage supervisé", https://dataanalyticspost.com/Lexique/apprentissage-non-supervisé/ visité le 15 février 2021.

[37] "Détection d'anomalie", https://dataanalyticspost.com/Lexique/detection-danomalie/ visité le 15 février 2021.

[38] H. Gadhvi, S. Madhu "Comparative Study of Classification Algorithms for Web Spam Detection", in International Journal of Engineering Research et Technology (IJERT), pp. 2497-2501, 2013.

[39] "Github",https://fr.wikipedia.org/wiki/GitHub visité le 16 février 2021.

[40] "IntelliJ IDEA", https://fr.wikipedia.org/wiki/IntelliJ_IDEA visité le 18 février 2021.

[41] "CodeMR", https://plugins.jetbrains.com/plugin/10811-codemr visité le 18 février 2021.

[42] "Orange", https://fr.wikipedia.org/wiki/Orange_(logiciel) visité le 18 février 2021.

[43] "Xlstat", https://www.xlstat.com/fr/ visité le 19 février 2021.

[44] "Software Engineer at Memo Bank", https://joanzapata.com/ visité le 19 février 2021.

[45] "AndroidPdfViewer", https://github.com/barteksc/AndroidPdfViewer visité le 19 février 2021.

[46] "Quran", https://github.com/quran/quran_android visité le 23 février 2021.

[47] "Android Password Store", https://github.com/android-password-store/Android-Password-Store visité le 23 février 2021.

[48] A. Moore, Professor School of Computer Science Carnegie Mellon University. "Cross-validation for detecting and preventing overfitting", at USA, October 2001.

[49 “Aire sous la Courbe”, https://lemakistatheux.wordpress.com/category/tests-statistique-indices-de-liaison-et-coefficients-de-correlation/laire-sous-la-courbe-roc/ visité le 15 septembre 2021.