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
MARC-ANTOINE LEVASSEUR

PRÉDICTION DE L'EFFORT DE TEST UNITAIRE DES LOGICIELS ORIENTÉS OBJET: UNE APPROCHE BASÉE SUR LES MÉTRIQUES ET L'APPRENTISSAGE AUTOMATIQUE

AOÛT 2022

Résumé

Pour assurer la qualité du logiciel, fortement impactée par la taille croissante et de la complexité des logiciels, une technique couramment utilisée est l'écriture de tests unitaires. Ils permettent de s'assurer que chaque classe (unité) d'un logiciel orienté objet répond à ses spécifications et qu'elle ne contient pas de fautes. Étant donné les contraintes comme le coût important et les ressources limitées pour l'écriture des tests, le choix est souvent fait de concentrer et de prioriser les efforts sur les parties critiques du programme (qui ne sont pas toujours faciles à identifier). Plusieurs approches basées sur les métriques orientées objet ont été proposées pour identifier automatiquement les unités logicielles les plus critiques. Le travail présenté dans ce mémoire les combines aux mesures de centralité tirées de l'analyse des réseaux sociaux et aux métriques de processus (calculées à partir des métadonnées des projets). 3 approches de modélisation de l'effort de test sont présentées basées sur les algorithmes d'apprentissage automatique et de l'apprentissage du classement (Learning to Rank). Peu de travaux ont été réalisés sur ces derniers en génie logiciel. Les approches présentées réussissent à modéliser précisément les niveaux d'effort de test requis, les classes qui sont de bonnes candidates à être testées et le classement des classes selon l'effort de test requis.

Mots-clés : Prioritisation des tests - Apprentissage automatique - Métriques orientées objet - Mesures de centralité - Métriques de processus - Apprentissage du classement

Abstract

To ensure software quality, strongly impacted by the increasing size and complexity of software, a commonly used technique is unit testing. Unit tests ensure that each class (unit) of an object-oriented software meets its specifications and that it does not contain any faults. Given constraints such as the high developpemnt cost and limited resources for writing tests, the choice is often made to concentrate and prioritize the efforts on the most critical units (which are not always easy to identify). Several approaches based on object-oriented (OO) metrics have been proposed to automatically identify the most critical software units. The work presented here combines OO metrics with centrality measures from social network analysis and process metrics (computed from project metadata). 3 approaches to modeling the testing effort are presented based on Machine Learning (ML) and Learning to Rank algorithms (LTR). Only a few studies have been done on LTR algorithms in software engineering. The approaches presented here succeed in accurately modeling the levels of testing effort required, the classes that are good candidates to be tested and the ranking of the classes according to the required testing effort.

Keywords : Test prioritization - Machine learning - Object-oriented metrics - Centrality measures - Process metrics - Learning to Rank

Remerciements

Je tiens à remercier mon directeur de recherche Mourad Badri pour son support, sa patience et son dévouement à la réalisation de ce projet.

Je souhaite aussi remercier mes parents, Monique et Marc, pour leur support et leurs encouragements.

Table des matières

Liste des tableaux

Table des figures

Liste des abréviations et des sigles

10-fold CV Validation croisée 10 fois (10 fold Cross-Validation)

AUC Aire sous la courbe ROC (Area Under the ROC Curve)

CM Réfère aux mesures de centralité (Centrality Measures)

COM Ensemble de métriques combinants les métriques orientées objet et les mesures de centralité

COMMITS Ensemble de métriques de processus tiré de l'analyse des modifications (commits) d'un logiciel

COR Ensemble de métriques obtenu à l'aide de l'algorithme de sélection de variables par corrélation (Correlation Feature Selection)

G-mean Moyenne géométrique du taux de vrais positifs et du taux de vrais négatifs dans une matrice de confusion

MAP Mesure du classement : Précision moyenne (Mean Average Precision)

nDCG Mesure du classement : Gain cumulé escompté normalisé (Normalized Discounted Cumulative Gain)

OO Réfère aux métriques orientées objet

PCA Ensembles de métriques tiré de l'analyse en composantes principales (PCA : Principal Composant Analysis)

Chapitre 1

Introduction

1.1 Introduction

L'augmentation de la complexité et de la taille des systèmes logiciels orientés objet a fait apparaître de nouveaux défis en matière d'assurance qualité. Maintenir la qualité et éviter d'introduire des fautes est souvent une tâche difficile, considérant que la plupart des logiciels nécessitent des modifications durant leur cycle de vie. Différentes méthodes et outils ont été développés pour améliorer la qualité des logiciels et réduire les coûts de développement.

Une technique couramment utilisée est l'écriture de tests unitaires. Elle permet de détecter les fautes rapidement et de maintenir ainsi la qualité logicielle. Les tests unitaires permettent de vérifier que l'implémentation de chaque unité du logiciel est conforme à ses spécifications. En programmation orientée objet, les tests unitaires sont utilisés pour tester l'implémentation des méthodes d'une classe. Ils permettent de s'assurer avec des assertions que lors de l'exécution d'une méthode, on obtient exactement le résultat auquel on s'attend.

Cependant, la conception et l'écriture de ces tests sont un processus coûteux en temps et en ressources. Plusieurs développeurs font donc le choix de ne pas tester l'ensemble des unités des programmes qu'ils développent. Plusieurs travaux se sont penchés sur le problème de la priorisation des tests, qui consiste (entre autres) à identifier les unités logicielles les plus critiques à tester. On souhaite identifier et tester en priorité les endroits où le logiciel est le plus susceptible de contenir des fautes.

1.2 Problématique

L'écriture des tests unitaires ainsi que le développement d'outils pour prédire quelles classes nécessitent un effort de test plus important (permettant de réduire les coûts du processus de développement) prennent beaucoup de temps et nécessitent beaucoup de ressources.

En génie logiciel, la question de la prédiction des fautes a fait l'objet de plusieurs travaux. On utilise souvent une approche dans laquelle des métriques sont extraites du code et sont utilisées pour créer des modèles prédictifs de la propension aux fautes. Être capable de déterminer quelles classes sont sujettes à contenir des fautes permet d'augmenter la qualité logicielle. Des métriques mesurant les attributs liés à la conception orientée objet sont souvent utilisées pour ces questions. Les métriques orientées objet mesurent des attributs comme la complexité, la cohésion, le couplage et la taille.

Peu de travaux se sont toutefois penchés sur la question de la prédiction de l'effort de test en combinant les métriques orientées objet et les mesures de centralité.

Les mesures de centralité sont tirées de la théorie des graphes. Elles permettent de déterminer à quel point un nœud est central dans un graphe et leur influence sur les autres nœuds. En programmation orientée objet, on peut utiliser la relation de couplage entre les classes afin de déterminer quelles classes ont le plus d'influence sur les autres. Les classes les plus influentes sont critiques étant donné que les fautes qu'elles contiennent peuvent potentiellement se répercuter à un plus grand nombre de voisins.

1.3 Questions de recherche

Dans cette section, les questions de recherche seront présentées et groupées selon leur objectif de prédiction. Trois façons de prédire l'effort de test seront étudiées.

1.3.1 Prédiction de l'effort de test par niveau

Dans un premier temps, la prédiction des niveaux de l'effort de test sera étudiée. On souhaite prédire pour chaque classe logicielle d'un programme le niveau d'effort de test requis (ex. : faible, moyen, élevé) pour assurer la qualité logicielle. Des modèles prédictifs basés sur les algorithmes d'apprentissage automatique seront utilisés. Ceux-ci utiliseront les métriques orientées objet, les mesures de centralité et les métriques de processus afin de grouper les classes logicielles par niveau d'effort de test.

Q1 & Q2. Est-ce qu'il y a une façon de séparer les classes selon leur effort de test de façon plus optimale lors de l'entrainement des modèles prédictifs ?

Nous utiliserons des algorithmes de partitionnement (clustering) pour regrouper les classes selon leur effort de test lors de l'entrainement. Nous vérifierons s'il y a un algorithme de partitionnement plus approprié (Q1) et l'influence du nombre de partitions sur la précision des prédictions (Q2).

Q3. Quel est l'impact du choix de métriques de test sur la modélisation de l'effort de test requis ?

Nous avons collecté 4 métriques de test sur nos jeux de données. Nous vérifierons si celles-ci améliorent ou dégradent la qualité des modèles prédictifs de l'effort de test unitaire.

Q4. Est-ce que les modèles basés sur les données historiques permettent un estimé plus précis de l'effort de test que les modèles basés sur d'autres systèmes ?

Nous avons collecté les données de 5 versions d'un logiciel (ANT version 1.3 à 1.7). Nous comparerons les modèles entrainés sur ces données aux modèles entrainés sur les 7 autres logiciels open-souce (Apache IVY, Apache LUCENE, Apache IO, Apache POI, Apache MATH, JFreeChart, JODA Time) afin d'étudier l'impact d'avoir l'historique des données et la capacité des modèles prédictifs à être généralisés à d'autres projets.

Q5. Est-ce que l'ajout des mesures de centralité est bénifique pour l'estimation de l'effort de test comparé aux métriques orientées objet seules ?

On souhaite déterminer si l'ajout des mesures de centralité est bénéfique à la prédiction des niveaux de l'effort de test (en comparaison aux métriques OO seules).

Q6. Est-ce que l'ajout des métriques de processus aux données historiques est bénéfique à la prédiction de l'effort de test ?

On souhaite déterminer si l'ajout des métriques de processus (ex. : nombre de modifications sur un fichier, nombre d'auteurs ayant modifié le fichier) permet de faire des prédictions plus précises de l'effort de test requis.

Q7. Est-ce qu'il y a un algorithme d'apprentissage automatique plus approprié à ce problème ?

Nous construirons des modèles prédictifs basés sur plusieurs algorithmes d'apprentissage automatique différents qu'on retrouve dans la littérature. On souhaite identifier s'il y a un algorithme qui permet d'obtenir des résultats plus précis.

1.3.2 Prédiction des classes candidates aux tests

Le deuxième objectif est de prédire (identifier) les classes qui sont de bonnes candidates aux tests, c'est-à-dire de prédire si une classe nécessite ou non des tests unitaires. Les développeurs choisissent souvent de ne pas tester toutes les classes pour réduire le temps nécessaire pour l'écriture de tests unitaires. En comparant avec ce que l'on observe dans nos jeux de données, on peut évaluer la capacité des modèles prédictifs à prédire les classes candidates (qui devraient être testées en priorité) aux tests. On utilise ici aussi des modèles prédictifs basés sur les algorithmes d'apprentissage automatique, mais cette fois la variable prédite est binaire (0 - pas prioritaire à être testé, 1 - devrait être testée).

Q8. Est-ce que les modèles basés sur les données historiques permettent un estimé plus précis des classes candidates que les modèles basés sur d'autres systèmes ?

Nous comparerons les modèles entrainés avec les versions précédentes d'ANT (1.3 à 1.7) aux modèles entrainés sur les 7 autres logiciels open-souce afin d'étudier l'impact d'avoir l'historique des données.

Q9. Est-ce que l'ajout des mesures de centralité est bénifique pour la prédiction des classes candidates comparé aux métriques orientées objet seules ?

On souhaite déterminer si l'ajout des mesures de centralité est bénéfique ou non à la prédiction des classes candidates aux tests (en comparaison aux métriques OO seules).

Q10. Est-ce que l'ajout des métriques de processus aux données historiques est bénéfique ou non à la prédiction de l'effort de test ?

On souhaite déterminer si l'ajout des métriques de processus permet de faire des prédictions plus précises des classes candidates aux tests.

Q11. Est-ce qu'il y a un algorithme d'apprentissage automatique plus approprié à ce problème ?

Nous construirons des modèles prédictifs basés sur plusieurs algorithmes d'apprentissage automatique différents qu'on retrouve dans la littérature. On souhaite identifier s'il y a un algorithme qui permet d'obtenir des résultats plus précis.

1.3.3 Classement de l'effort unitaire

Finalement, le troisième objectif est de prédire le classement des classes logicielles par l'effort de test requis. En identifiant les classes nécessitant un effort de test plus important, on peut guider et simplifier le processus de test. Pour cette partie, les algorithmes d'apprentissage du classement (Learning to Rank) seront utilisés. Ces algorithmes sont tirés du domaine de la recherche d'information et des moteurs de recherche. Ils permettent de classer des documents selon leurs attributs.

Q12. Est-ce que les modèles basés sur les données historiques permettent un estimé plus précis de l'ordonnancement des classes à tester que les modèles basés sur d'autres systèmes ?

Nous comparerons les modèles entrainés avec les versions précédentes d'ANT (1.3 à 1.7) aux modèles entrainés sur les 7 autres logiciels open-souce afin d'étudier l'impact d'avoir l'historique des données.

Q13. Est-ce que l'ajout des mesures de centralité est bénifique pour la prioritisation de l'effort de test comparé aux métriques orientées objet seules ?

On souhaite déterminer si l'ajout des mesures de centralité est bénéfique à la prioritisation des tests (en comparaison aux métriques OO seules).

Q14. Est-ce que l'ajout des métriques de processus aux données historiques est bénéfique à la prédiction de l'effort de test ?

On souhaite déterminer si l'ajout des métriques de processus (nombre de modification sur un fichier, nombre d'auteurs ayant modifié le fichier) permet de faire des prédictions plus précises pour le classement des classes critiques.

Q15. Est-ce qu'il y a un algorithme de « Learning to Rank » plus approprié à ce problème ?

Nous construirons des modèles prédictifs basés sur plusieurs algorithmes proposés dans la littérature. On souhaite identifier s'il y a un algorithme qui permet d'obtenir des résultats plus précis.

1.4 Organisation du mémoire

Le reste du mémoire est organisé de la façon suivante. La section 2 présente l'état de l'art de la recherche pour notre problématique. Les travaux les plus importants y seront présentés. La section 3 présente la méthodologie utilisée pour répondre aux questions de recherche. La section 4 présente les résultats des expérimentations. La section 5 présente une discussion des résultats et les conclusions des questions de recherche.

Chapitre 2

État de l'art

2.1 Introduction

Dans ce chapitre, l'état de l'art des différents aspects reliés à la prédiction de l'effort de test unitaire est présenté.

2.2 Revue de la littérature

2.2.1 Métriques orientées objet

Les métriques OO ont été utilisées dans de nombreux domaines du génie logiciel [9, 15,21,24] avec des modèles de régression, des classificateurs d'apprentissage automatique et d'autres modèles de prédiction pour différentes tâches telles que la prédiction des défauts [10]. Les métriques OO ont également été combinées avec des techniques de calcul de seuil pour prédire la prédisposition aux fautes [12, 48].

Différentes métriques ont été proposées dans la littérature. Les plus connues (utilisées) sont les métriques de Chidamber et Kemerer (CK) [23, 24]. D'autres suites de métriques ont été proposées, comme les suites MOOD et QMOOD, mais la suite CK reste la plus utilisée dans la littérature [19].

De nombreuses études ont été réalisées pour comprendre les capacités de prédiction des métriques logicielles [8, 34, 82]. La combinaison de différentes métriques liées à diverses caractéristiques telles que la taille, la complexité, le couplage et la cohésion, permet souvent des prédictions plus précises [37,54]. Par exemple, Zhou et al. ont utilisé des métriques orientées objet pour prédire les classes sujettes aux fautes et la sévérité

des fautes sur un jeu de données de la NASA [82]. Ils ont réussi à démontrer le lien entre la plupart des métriques et la sévérité des fautes.

Shatnawi a utilisé des métriques orientées objet pour classer les classes de logiciels à haut et bas niveau de risque en utilisant des valeurs seuils et des arbres de décision [61]. Ce travail a montré que des valeurs seuils des métriques CK pouvaient être calculées à l'aide d'arbre de décisions et utilisées pour la prédiction des fautes. Gyimóthy et al. les ont utilisées pour prédire la prédisposition aux fautes et ont réussi à valider le lien entre la propension aux fautes des logiciels de Mozilla et ces métriques avec des méthodes de régressions et des approches d'apprentissage automatique [34]. Briand et al. ont utilisé diverses métriques pour prédire le risque de défaillance dans les logiciels C++ [14]. Cette étude a été reproduite par Aggarwal et al. pour les programmes Java [4]. Les métriques OO ont également été utilisées dans certaines études comme guide de refactoring [5], car elles peuvent être utilisées comme de bons indicateurs de la qualité du code.

2.2.2 Mesures de centralité

Les mesures de centralité, également appelées mesures de réseau, ont souvent été étudiées pour l'analyse des réseaux sociaux. Les mesures de centralité ont également été utilisées dans quelques études d'ingénierie logicielle. Les mesures de centralité sont basées sur la théorie des graphes, et mesurent essentiellement la centralité d'un nœud dans un graphe. Dans les logiciels orientés objet, le couplage entre les classes peut être utilisé pour construire un graphe de dépendance des classes logicielles à partir duquel les mesures de centralité peuvent être extraites. Zhu et al. ont montré que les mesures de centralité sont prometteuses pour la prédiction des défauts logiciels [83].

Zimmerman et al. ont également utilisé des mesures de réseau à partir des relations de dépendance entre les classes de logiciels pour prédire les défauts [84]. Tosun et al. ont validé les résultats de Zimmerman et al. dans [64]. Bettenburg et al. ont conclu que les mesures de centralité étaient plus performantes lorsqu'elles étaient combinées à des mesures logicielles traditionnelles [11].

Ouellet et Badri ont utilisé des mesures de réseau pour prédire la vulnérabilité des logiciels Java open source avec des algorithmes d'apprentissage automatique [56]. Ils ont démontré que sur leur jeu de données, les mesures de centralité offraient de l'information complémentaire aux métriques orientées objet.

Certains travaux ont été réalisés pour utiliser les mesures de centralité pour guider l'effort de test logiciel. Par exemple, Kayes et al. ont montré que les mesures de réseau

pouvaient être utilisées pour prioriser les tests de régression en utilisant des réseaux de fautes [42]. Dans leur étude, les mesures de centralité des fautes ont permis de prioriser les tests pour couvrir plus efficacement les régressions que les méthodes de priorisation traditionnelles. Par contre, très peu de travaux ont été réalisés par rapport à la prédiction de l'effort de test.

2.2.3 Prédiction de l'effort de test

Des travaux ont porté sur la prédiction de l'effort de test dans les systèmes orientés objet. Des travaux par Bruntink et Van Deursont ont étudié le lien entre les métriques orientées objet et l'effort de test dans des programmes Java [15, 16]. Ces travaux ont démontré que les métriques orientées objet pouvaient aider à prédire la testabilité des classes. Badri et Toure ont utilisé les mesures orientées objet et la régression logistique multivariée pour classifier l'effort de test en deux niveaux [8]. Ils ont réalisé plusieurs autres travaux sur la prédiction de l'effort de test avec Lamontagne [9, 66, 67]. Ils ont aussi proposé de nouvelles métriques pour mesurer l'effort de test unitaire pour JUnit [65]. Le travail présenté dans ce mémoire est (en partie) une continuation de ces travaux.

2.2.4 Prioritisation des cas de test

La priorisation des cas de test est un autre sujet important étudié en génie logiciel. La priorisation des cas de test vise à réduire le nombre requis de tests à exécuter tout en maximisant la couverture des fautes et des régressions. Elle est similaire à bien des égards à la prédiction de l'effort de test unitaire, mais au lieu de réduire le nombre de tests à écrire, elle tente de réduire le nombre de tests à exécuter.

Différentes méthodes de hiérarchisation des tests ont été proposées dans la littérature. La priorisation des cas de test a été le plus souvent étudiée pour les tests de régression. Les approches proposées utilisent souvent les fautes, la couverture, les informations historiques et l'analyse de risques [67].

Dans un environnement contrôlé, Rothermel et al. [58] ont analysé différentes techniques d'ordonnancement des cas de test et ont utilisé la métrique APFD (pourcentage moyen de fautes détectées) pour comparer les performances des modèles. Les auteurs ont montré que le réordonnancement des cas de test améliore le taux de détection des fautes lors des tests de régression. Yu et Lau [80] ont proposé une technique de priorisation des cas de test basée sur leur capacité de détection des fautes et d'ainsi réduire

de 28% le nombre de cas de test à exécuter pour maintenir la même couverture. La technique est basée sur les relations entre les cas de test et les fautes. Mei et al. ont proposé une approche statique sans collecte de données sur la couverture des tests [51]. Leur travail a utilisé le graphe des appels de méthodes des tests JUnit de 19 versions de 4 logiciels Java pour montrer que les approches statiques (sans couverture des tests) permettent d'obtenir des modèles similaires aux modèles dynamiques nécessitant l'exécution des tests unitaires pour calculer la couverture et de maintenir un taux similaire de la couverture des fautes.

2.2.5 Algorithmes d'apprentissage du classement (Learning to Rank)

Les algorithmes d'apprentissage du classement (Learning to Rank) ont été principalement utilisés dans le domaine de la recherche d'informations [47]. Les algorithmes d'apprentissage du classement tentent classiquement de classer au mieux des documents en fonction de leur pertinence. Les moteurs de recherche sont un exemple de leur utilisation. Trois types d'algorithmes d'apprentissage du classement ont été proposés : par points (pointwise), par paires (pairwise) et par listes (listwise) [47]. Les algorithmes par points donnent à chaque document une pertinence et effectuent le classement en utilisant cette pertinence avec la fonction de perte. Les algorithmes par paires comparent des paires de documents. Les algorithmes par liste effectuent le classement en utilisant la liste entière. Wu et al. ont proposé l'algorithme LambdaMART, une approche basée sur les arbres renforcés (boosted trees) et le LambdaRank [75]. Il s'agit d'un algorithme par paires [17, 75].

En génie logiciel, certains travaux ont utilisé les algorithmes d'apprentissage du classement pour la localisation des fautes [44, 63, 78, 79]. Par exemple, Yang et al. ont utilisé les métriques tirées d'Eclipse pour identifier avec précision les classes les plus susceptibles de contenir des fautes. Leurs résultats identifient les approches de Learning to Rank et Random Forest comme étant les plus prometteuses pour la localisation de fautes [78]. Sohn et Yoo ont notamment utilisé des mesures de taille, de changements, de couverture des tests et les taux de réussite des tests pour la localisation de fautes avec les algorithmes de Learning to Rank [63]. Leur approche leur a permis d'obtenir des modèles d'ordonnancement précis (MAP > 0.5) des classes fautives et d'identifier la majorité des classes fautives au sommet de l'ordonnancement.

Chapitre 3

Méthodologie

3.1 Introduction

Ce chapitre présente la méthodologie utilisée pour répondre aux questions de recherche. Trois types de prédiction de l'effort de test seront étudiés : la prédiction des niveaux (faible, moyen, élevé) d'effort de test, la prédiction des classes candidates aux tests et le classement des classes logicielles selon leur priorité à être testées. Dans les trois cas, des données sont collectées à partir du code source et des métadonnées du projet à partir desquelles on entraine des modèles prédictifs basés sur l'apprentissage automatique ou sur l'apprentissage du classement (Learning to Rank). On utilise ensuite une autre partie des données pour évaluer la capacité de prédiction de l'effort de test du modèle.

3.2 Jeux de données

Pour étudier le lien entre l'effort de test et la qualité du logiciel, nous avions besoin de données sur le code source et sur le code de test.

Les métriques de test ont été extraites du code source des tests unitaires. Tous les projets de cette étude utilisent JUnit comme plateforme de test. Les cas de tests JUnit effectuent des assertions pour vérifier la validité de l'implémentation d'une classe Java, dans le but d'empêcher les régressions lors de la modification du code source du programme. Les métriques ont été extraites à l'aide d'outils d'analyse statique. Un seul test unitaire valide généralement l'implémentation d'une classe, alors les métriques du code source du programme peuvent être associées aux métriques de test.

Les métriques ont été extraites de 8 projets Java open source, la plupart maintenus par la Fondation Apache. Nous avions besoin de projets open source de taille et de qualité substantielles qui utilisaient des tests unitaires. Les projets suivants ont été utilisés pour entrainer les modèles prédictifs dans cette étude : Apache ANT, Apache Lucene, Apache IO, Apache POI, Apache MATH, JFreeChart et JODA Time. Le tableau 3.1 présente la version de chaque projet dont nous avons extrait le code source et les métriques de test.

TABLE 3.1 – Logiciels du jeu de données

Projet	Version
Apache ANT	1.7.0
Apache LUCENE	7.4.0
Apache IO	2.6.0
Apache POI	3.17
Apache MATH	3.6.1
JFreeChart	1.0.19
JODA Time	2.10
IVY	2.5.0

Pour les questions de recherche en lien avec les données historiques, 5 versions du logiciel ANT ont été utilisées (1.3, 1.4, 1.5, 1.6, 1.7). Les mêmes métriques orientées objet et mesures de centralité ont été extraites, mais les métriques de processus ont été aussi ajoutées à partir des métadonnées du projet collectées à partir du système de contrôle de versions Git. Les données des auteurs et des modifications (commits) ont été extraites à partir de la version 1.2 jusqu'à 1.7 pour avoir des valeurs associées à la version 1.3.

Les classes définies à l'intérieur d'autres classes ont été exclues, car les métriques de processus doivent être associées à un fichier. Les classes faisant partie des "packages" optionnels d'ANT ont aussi été exclues. Les résultats des modèles prédictifs entrainés sur ces données seront présentés et comparés aux modèles prédictifs inter-projets.

3.3 Métriques orientées objet

Dans cette section, les métriques orientées objet utilisées dans ce travail sont présentées. Celles-ci proviennent en majorité des travaux Chidamber et Kemerer [23, 24].

Ces métriques mesurent différents attributs du code orienté objet : la complexité, le couplage, la cohésion, la taille et l'héritage.

- WMC (Weighted Method per Class) : La somme de la complexité cyclomatique de chaque méthode de la classe [23, 24].
- RFC (Response for a Class) : Le nombre de méthodes qui peuvent potentiellement être appelées lorsque la classe reçoit un message [23, 24].
- CBO (Coupling Between Objects) : Le nombre de classes auxquelles la classe est couplée [23, 24].
- LOC (Lines of Code) : Le nombre de lignes de code de la classe
- Ca (Afferent Coupling) : Le nombre de dépendances entrantes dans une classe [50].
- Ce (Efferent Coupling) : Le nombre de dépendances sortantes d'une classe [50].
- LCOM3 (Lack of Cohesion of Methods) : Calcule la cohésion avec un ratio basé sur le nombre de variables membres d'une classe et le nombres de méthodes les utilisant. La valeur de LCOM3 est un nombre réel compris entre 0,0 et 2,0, où 0 représente une classe très cohésive et 2 représente les classes les moins cohésives [37].
- DIT (Depth of Inheritance Tree) : La longueur de l'arbre d'héritage depuis la racine jusqu'à la classe en question. [23]
- NOC (Number of Children) : Le nombre de classes enfants qui héritent d'une classe. [24]

3.4 Mesures de centralité

Dans cette section, les mesures de centralités sont présentées. Peu d'études ont été faites sur la capacité des mesures de centralité à prédire la qualité logicielle. L'hypothèse est que les classes centrales fautives d'un logiciel ont plus d'impact que les autres classes fautives et sont donc prioritaires à être testées.

- EV (Eigen Vector centrality) : Mesure de centralité basée sur les valeurs propres de la matrice de couplage [59].
- LE (Leverage Centrality) : Mesure de centralité utilisée pour identifier les noeuds critiques dans un graphe [38].

- LO (Lobby Centrality) : Utiliser l'indice l d'un noeud pour évaluer sa centralité dans le graphe [18, 29].
- SLC (Semi Local Centrality) : Utilise la somme des plus proches voisins et de leurs plus proches voisins pour évaluer la centralité d'un noeud [22].
- DMNC (Density Maximum Neighborhood Component Centrality) : Mesure de centralité qui utilise la densité maximale des composantes de ses voisins [46].
- BC (Betweenness Centrality) : Mesure basée sur les chemins les plus courts entre chaque noeud [30].
- CL (Closeness Centrality) : L'inverse de la somme des chemins les plus courts vers tous les autres noeuds [31].

3.5 Métriques pour l'effort de test

JUnit est une plateforme pour l'écriture et l'exécution de tests unitaires Java. Une classe de test est écrite pour chaque classe qu'on souhaite tester. On choisit souvent de ne pas tester chaque classe, seulement celles critiques afin de réduire l'effort requis pour tester le logiciel.

Les assertions sont des déclarations qui comparent les valeurs attendues aux résultats de l'implémentation. Lors de l'exécution des tests unitaires, chaque assertion contenue dans le code de test unitaire est validée [67]. Un bon code de test tentera de couvrir toutes les branches du code afin de maximiser son effet sur la fiabilité et la qualité.

- TLOC (Test Lines of Code) : Nombre de lignes de code du test unitaire [15].
- TASSERT (Assertion in Test) : Nombre de validations d'assertion dans le test unitaire [15].
- TDATA (New Java Objects) : Nombre d'instanciations d'objet dans le test unitaire [65].
- TINVOK (Number of Invocations) : Nombre d'invocations de méthode dans le test unitaire [65].

Dans cette étude, nous avons utilisé un analyseur de code statique pour extraire les 4 métriques de test pour chaque classe testée. Les métriques TLOC et TASSERT ont été introduites par Bruntink et Van Deursont dans [15] pour mesurer deux aspects de la taille d'un cas de test JUnit. Les deux autres métriques (TDATA, TINVOK) ont

été proposées par Toure et al. [65]. TDATA est mesurée en comptant le nombre de fois qu'un objet est instancié avec le mot clé Java "new" dans la classe de test. TINVOK est mesurée en comptant nombre d'invocations de méthodes effectuées dans les tests unitaires.

3.5.1 Métriques de processus

Les métriques de processus ont été extraites du système de contrôle de versions Git. Ces métriques calculent des informations sur le nombre de changements faits à un fichier entre deux versions. L'hypothèse est que les classes d'un système orienté objet qui sont modifiées le plus souvent sont les plus critiques à tester, car leurs modifications peuvent contenir des fautes.

Ces métriques seront utilisées dans les modèles utilisant des métriques historiques pour voir leur impact sur la capacité des modèles prédictifs à identifier le niveau d'effort de test, les classes candidates aux tests et le classement des classes logicielles selon leur priorité à être testée.

Les métriques de processus utilisées dans cette étude sont les suivantes :

- numCommits : Nombre de modifications faites sur le fichier.
- numWords : Nombre de mots dans le texte descriptif des modifications.
- numAuthors : Nombre d'auteurs ayant modifié le fichier.
- insertions : Nombre de lignes de code ajoutées au fichier.
- deletions : Nombre de lignes de code supprimées au fichier.

3.6 Algorithmes d'apprentissage automatique

Dans cette étude, des classificateurs d'apprentissage automatique ont été utilisés pour prédire l'effort de test requis pour les classes Java de notre jeu de données. Nous avons utilisé l'apprentissage supervisé pour la prédiction de l'effort de test unitaire, et l'apprentissage non supervisé pour séparer les classes en groupes de niveaux similaires d'effort de test (ex : faible, moyen, élevé) avec des algorithmes de partitionnement (clustering).

Le logiciel WEKA a été utilisé pour construire et évaluer les classificateurs sur notre jeu de données. Tous les classificateurs utilisent les algorithmes et les paramètres par défaut de WEKA version 3.8 [73].

3.6.1 Régression logistique (LinearRegression)

La régression logistique a été largement utilisée pour la prédiction de défauts dans la littérature sur le génie logiciel [25]. Basili et al. [10] ont utilisé la régression logistique pour prédire les défauts des logiciels C++ à l'aide de métriques logicielles. La régression logistique a également été utilisée pour prédire l'effort de test dans certaines études. Zhou et al. [81] ont démontré que la régression logistique avec des métriques logicielles pouvait être utilisée pour guider la gestion des tests logiciels. Badri et Toure [8] ont également utilisé des métriques OO et la régression logistique pour prédire l'effort de test requis.

3.6.2 Perceptrons multicouches (MultilayerPerceptron)

Les classificateurs à perceptron multicouches sont basés sur des algorithmes de réseaux neuronaux artificiels (ANN). Il s'agit d'un réseau neuronal à action directe. Les perceptrons multicouches ont été utilisés dans la littérature dans le contexte de la prédiction des défauts et de l'effort logiciel. Par exemple, Kanmani et al. [40] ont utilisé des métriques orientées objet et des réseaux neuronaux artificiels pour prédire les défauts dans les applications Java. Aljahdali et al. [6] ont utilisé des ANN pour prédire l'effort de développement de logiciels à l'aide de métriques logicielles.

3.6.3 Bagging

L'agrégation bootstrap, ou bagging en anglais est une technique d'apprentissage automatique [60]. Cette technique fonctionne en sélectionnant différents échantillons dans l'ensemble d'apprentissage (bootstrapping) et en agrégeant les résultats de leurs prédictions. La différence entre les approches Random Forest et Bagged Trees est que cette dernière considère que chaque variable peut être choisie lors de la construction des arbres alors que la première ne permet de choisir qu'un sous-ensemble de variables. La méthode des arbres en sac (Bagged Trees) a été comparée dans d'autres études à d'autres types de classificateurs et il a été démontré qu'elle donne de bonnes performances pour la prédiction des fautes. Braga et al. ont montré qu'elle pouvait être utilisée pour prédire l'effort logiciel requis [13].

3.6.4 Random Forest

Random Forest est un algorithme de classification qui utilise des arbres de décision pour effectuer la classification. L'algorithme utilise différents sous-ensembles de variables pour construire les arbres. Chaque arbre effectue la classification et produit un vote. La classification finale est choisie en fonction de la majorité des votes. L'algorithme Random Forest s'est avéré performant dans plusieurs contextes de l'ingénierie logicielle [1, 33, 49]. Par exemple, Kaur et Malhotra ont utilisé des modèles basés sur Random Forest pour prédire les fautes dans des logiciels Java (JEdit) [41]. Guo et al. ont également utilisé des modèles basés sur Random Forest pour prédire les défauts dans les logiciels C++ [33]. Abdelali et al. [2] ont utilisé des modèles basés sur Random Forest pour l'estimation de l'effort logiciel.

3.6.5 Réseau Bayésien

Les classificateurs à réseau bayésien utilisent le théorème de Bayes pour construire un graphe de décision et effectuer la classification. [32]. Dans cette étude, nous avons configuré le classificateur pour utiliser l'algorithme K2 comme algorithme de recherche. Le réseau de Bayes a été utilisé avec succès dans le contexte de la prédiction des défauts logiciels. Challagulla et al. ont utilisé des réseaux de croyance bayésiens, parmi d'autres techniques, pour prédire les défauts des logiciels [21]. Okutan et Yıldız ont utilisé des réseaux bayésiens pour prédire les défauts des logiciels avec les données de Promise Software Engineering Data Repository [55].

3.6.6 Naives Bayes

Un classificateur Naive Bayes est un classificateur de réseau bayésien dans lequel la relation entre les variables, dans ce cas les métriques, sont supposées indépendantes. Les classificateurs Naive Bayes ont été utilisés dans le passé pour la prédiction des défauts à l'aide de métriques logicielles [68, 71]. Catal et al. [20] ont utilisé des classificateurs Naive Bayes pour prédire les défauts sur les logiciels basés sur Eclipse (Java). Dejaeger et al. [26] ont utilisé des Naives Bayes pour prédire les défauts dans des jeux de données de la NASA et de la fondation Eclipse.

3.6.7 J48 (C4.5)

L'implémentation WEKA d'un arbre de décision est basée sur l'algorithme C4.5. L'algorithme C4.5, son prédécesseur ID3, et son successeur C5 ont été utilisés pour la prédiction de fautes dans l'ingénierie logicielle. Par exemple, Wang et al. ont utilisé modèles C4.5 comprimés pour prédire les défauts sur l'IDE Eclipse (orienté objet, écrit en Java) [45, 70].

3.6.8 Machines à vecteurs de support (SVM/SMO)

Les machines à vecteurs de support (SVM) tentent de définir des limites pour séparer les classes en faisant correspondre chaque variable à un point dans un espace multidimensionnel. [76]. L'algorithme tente de trouver une fonction qui sépare correctement ces points. Les SVM ont été utilisés dans de nombreuses études pour la prédiction de la qualité des logiciels, souvent pour la prédiction des défauts. Xing et al. ont utilisé des modèles SVM pour faire des prédictions de la qualité des logiciels [76]. Singh et al. [62] ont utilisé les SVM pour prédiction de défauts les données KC1 de la NASA. Weka fournit une implémentation de l'algorithme basé sur le modèle d'optimisation séquentiel minimale (SMO) de John Platt [36, 43, 57].

3.7 Algorithmes d'apprentissage du classement (Learning to Rank)

Trois types d'algorithmes d'apprentissage du classement - par points (pointwise), par paires (pairwise) et par listes (listwise) - sont utilisés pour résoudre les problèmes de classement. La catégorisation de ces algorithmes dépend du nombre de documents utilisés dans le calcul de la fonction de perte [47]. Le classement par points utilise les caractéristiques d'un seul document pour estimer sa pertinence parmi tous les documents d'un ensemble. Les modèles de régression classiques et les classificateurs d'apprentissage automatique sont considérés comme des approches ponctuelles des problèmes de classement. Les approches par paires comparent deux documents pour identifier le plus pertinent et le processus est répété jusqu'à ce que tous les documents soient triés. Les algorithmes de classement par liste tentent de maximiser des paramètres tels que le gain cumulatif actualisé normalisé (nDCG) ou de maximiser une fonction de perte complexe. LambdaMART est un algorithme par paires proposé par Wu et al., basé sur les Boosted

Trees et LambdaRank [17,75]. AdaRank est un algorithme basé sur le boosting par liste introduit par Xu et al. [77].

3.8 Algorithmes de partitionnement

Un algorithme de partitionnement (clustering algorithm) prend des données à n dimensions et regroupe chaque observation dans un nombre déterminé de partitions de nature similaire. Deux algorithmes de partitionnement seront comparés dans ce travail : AHC et k-means. Les résultats des prédictions de l'effort de test obtenus avec les deux algorithmes seront présentés, comparés et discutés. Les données d'entrée utilisées dans le partitionnement sont les métriques de code de test TLOC, TASSERT, TINVOK et TDATA. La sortie est le niveau d'effort de test (par exemple, faible, moyen, élevé). Les expériences ont été réalisées pour différents nombres de partitions (2, 3, 4, 5).

3.8.1 K-means

Dans cette étude, nous avons utilisé l'algorithme de partitionnement k-means pour séparer les classes Java des projets en groupes ayant un effort de test similaire. Nous avons utilisé les 4 métriques de code de test comme entrée pour le processus de partitionnement : TLOC, TASSERT, TINVOK et TDATA. La sortie est le niveau d'effort de test (ex. : faible, moyen, élevé). Les partitions ont été associées aux niveaux (ex. : faible, moyen, élevé) en calculant la moyenne des valeurs des métriques des éléments de la partition et en les assignant de la plus petite moyenne à la plus grande.

L'algorithme k-means fonctionne en générant aléatoirement k centroïdes et en associant chaque observation au centroïde le plus proche, mesuré par la distance euclidienne carrée. Chaque centroïde est déplacé vers le point moyen de toutes les observations qui lui sont assignées. Tous les centroïdes sont réassignés et déplacés à nouveau pendant de nombreuses itérations, jusqu'à ce qu'ils convergent vers une position stable. Dans notre cas, les centroïdes sont générés dans l'espace des métriques de tests, et les observations sont chaque classe Java du projet ANT pour laquelle il existe des tests unitaires associés. Le résultat est k partitions de classes avec un niveau d'effort de test similaire, où la somme de la variance inter-partitions est minimisée. En d'autres termes, la somme des distances euclidiennes des métriques de test est minimisée.

3.8.2 Partitionnement hierachique agglomératif (AHC)

Le deuxième algorithme de partitionnement étudié est le partitionnement hiérarchique agglomératif (AHC - Agglomerative Hierachical Clustering). Il fonctionne en regroupant les données à l'aide d'une fonction de distance [52]. Dans ce travail, la distance euclidienne est utilisée comme fonction de distance, et la méthode de Ward est utilisée comme critère de liaison (linkage method). Le résultat est k partitions (clusters) de classes Java contenant des classes logicielles avec des niveaux similaires d'effort de test. Une étiquette est ensuite attribuée à chaque partition (par exemple, faible = 0, moyen = 1, élevé = 3) en fonction de la moyenne des métriques de test contenues dans cette partition.

3.9 Évaluation des modèles d'apprentissage automatique

Les modèles d'apprentissage automatique sont entrainés sur un ensemble de données et évalués sur un autre ensemble de données. Dans cette étude, les performances des classifieurs ont été validées en utilisant la validation croisée 10 fois, la validation inter-versions et la validation inter-projets. Ces techniques d'évaluation seront présentées dans ce qui suit. Dans cette étude, les valeurs utilisées pour comparer les prédictions résultantes sont basées sur la matrice de confusion. Nous utilisons deux mesures différentes couramment utilisées dans l'évaluation des performances des classificateurs d'apprentissage automatique : l'aire sous la courbe ROC (AUC) et le g-mean (moyenne géométrique du taux de vrai positif et du taux de vrai négatif). Ces mesures seront présentées dans ce qui suit.

3.9.1 10-fold cross validation

Pour évaluer les prédictions des classifieurs d'apprentissage automatique sur les mêmes données que celles utilisées pour entrainer les modèles, nous utilisons la méthode de la 10-fold cross validation (validation croisée 10 fois). La validation croisée divise les données en k plis. k - 1 plis sont utilisés pour entraîner le modèle tandis que le dernier pli est utilisé pour évaluer le modèle. Ce processus est répété k fois, de sorte que chaque sous-ensemble est évalué une fois. Cela donne une représentation plus précise des résultats de la classification que de sélectionner un pourcentage des données

à utiliser comme données d'évaluation. La validation croisée 10 fois sera utilisée lors de l'évaluation des modèles sans données historiques.

3.9.2 Validation inter-versions

Les modèles inter-versions sont entrainés pour utiliser une ou plusieurs versions d'un projet. Les modèles entrainés à partir de k versions sont évalués sur la version exclue. En général, les versions précédentes sont utilisées pour prédire une version plus récente. Puisque l'une des questions de recherche est d'étudier l'impact de l'utilisation de données historiques dans les approches basées sur l'apprentissage automatique, nous comparerons les résultats des modèles entrainés avec une seule version et avec plusieurs versions. Nous comparerons également les résultats avec les modèles entrainés à partir d'autres logiciels open source.

3.9.3 Matrice de confusion

La performance de la classification est évaluée en mesurant les valeurs g-mean et AUC calculées à partir de la matrice de confusion. En comparant les valeurs prédites avec les valeurs observées pour les données d'évaluation, nous obtenons une matrice contenant les vrais positifs, les vrais négatifs, les faux positifs et les faux négatifs. En utilisant cette matrice, nous pouvons calculer différentes mesures, par exemple le taux de vrais positifs et le taux de faux négatifs. La matrice de confusion est calculée pour chaque valeur possible de la variable prédite (classe de prédiction). Par exemple, si nous prédisons le niveau d'effort de test avec trois niveaux différents (faible, moyen, élevé), nous aurons 3 matrices de confusion. Le tableau 3.2 présente ces mesures.

Table 3.2 – Matrice de confusion

		Observé	
		Positif	Négatif
Prédiction	Positif	Vrai positifs (TP)	Faux positifs (FP)
	Négatif	Faux négatifs (FN)	Vrai négatifs (TN)

3.9.4 G-mean

À partir de la matrice de confusion, nous pouvons calculer une valeur numérique pour chaque classe de prédiction. Dans cette étude, une valeur à partir de laquelle nous

évaluons la performance est la moyenne géométrique (g-mean) du taux de vrais positifs (TPR) et du taux de vrais négatifs (TNR). La valeur g-mean est calculée comme suit :

$$TPR = 1 - FNR = \frac{TP}{TP + FN} \tag{3.1}$$

$$TNR = 1 - FPR = \frac{TN}{TN + FP} \tag{3.2}$$

$$gmean = \sqrt{TNR * TPR} \tag{3.3}$$

Le TPR est calculé comme le rapport entre les vrais positifs (TP) et toutes les vraies valeurs (TP + FN).

Le TNR est calculé comme le rapport entre les vrais négatifs (TN) et toutes les fausses valeurs (FP + TN).

Pour la classification avec plus de deux classes de prédiction, la g-moyenne pondérée est utilisée comme définie dans [28]. Elle tient compte du déséquilibre des classes (niveaux de test, classes testées ou non) que l'on trouve dans nos jeux de données : la plupart des classes logicielles n'ont pas de tests et celles qui en ont sont souvent courts.

Les valeurs de g-mean sont comprises entre 0 et 1, où [12] :

- g-mean $<$ 0,5 signifie pas de bonne classification,
- $0,5 \leq$ g-mean $< 0,6$ signifie une mauvaise classification,
- $0,6 \leq$ g-mean $< 0,7$ signifie une classification passable,
- $0,7 \leq$ g-mean $< 0,8$ signifie une bonne classification,
- $0,8 \leq$ g-mean $< 0,9$ signifie une excellente classification,
- et $g - mean \geq 0,9$ signifie une classification exceptionnelle.

3.9.5 AUC

L'aire pondérée sous la courbe ROC (Area under the ROC curve AUC) est également utilisée pour mesurer la classification de nos modèles de prédiction. Elle est utilisée pour donner un aperçu de la qualité de la classification. Les valeurs de l'AUC vont également de 0,5 à 1, où l'on considère que : [12].

- AUC $<$ 0,5 signifie pas de bonne classification,
- $0,5 \leq AUC < 0,6$ signifie une mauvaise classification,
- $0,6 \leq AUC < 0,7$ signifie une classification passable,

- $0,7 \leq AUC < 0,8$ signifie une bonne classification,
- $0,8 \leq AUC < 0,9$ signifie une excellente classification,
- et $AUC \geq 0,9$ signifie une classification exceptionnelle.

3.10 Évaluation des modèles du classement

Nous avons évalué la performance du classement obtenu avec l'analyse de corrélation et deux métriques d'apprentissage du classement (Learning to Rank). Ces deux approches permettent de comparer le classement prédit au classement observés à partir de différents aspects (classement général, classement des classes logicielles les plus critiques). Le coefficient de corrélation de rang de Spearman et le coefficient de corrélation de Kendall ont été utilisés pour évaluer le classement de façon générale. Ces mesures sont souvent utilisées dans le domaine de la recherche d'information pour évaluer les méthodes de classement. Dans cette section, les deux métriques spécifiques aux problèmes d'apprentissage du classement (Learning to Rank) sont présentées et expliquées.

3.10.1 Normalized Discounted Cumulative Gain (nDCG@k)

La métrique nDCG@k (Normalized Discounted Cumulative Gain for k observations) a également été utilisée pour évaluer le classement des classes logicielles selon l'effort de test requis. Le gain cumulatif actualisé (DCG) est une mesure couramment utilisée en recherche d'information (Information Retrieval) pour évaluer la qualité d'un classement [69, 72]. La version normalisée du DCG a une valeur comprise entre 0 et 1. La métrique nDCG est calculée sur k observations, où k n'est pas toujours l'ensemble des données. Dans notre cas, nous étions intéressés par l'évaluation du classement entier, donc le nombre d'évaluations jugées (k) est le nombre d'éléments dans l'ensemble de données d'évaluation. Cette valeur permet d'identifier la capacité des modèles à identifier les classes les plus critiques.

3.10.2 Précision moyenne (MAP)

La précision moyenne (Average Precision - AP) est l'aire sous la courbe rappel-précision. La précision moyenne (Mean Average Precision - MAP) est la moyenne de la AP pour chaque document. Les valeurs se situent entre 0 et 1, mais des études en génie logiciel considèrent les valeurs autour de 0.5 comme précises [44].

3.11 Tests statistiques

Le test de Friedman et le test post-hoc de Nemenyi ont été utilisés pour détecter les différences statistiques pour chaque expérimentation. Le test de Friedman est un test non paramétrique utilisé pour comparer plus de deux essais. Il est bien adapté à la comparaison des résultats des classificateurs d'apprentissage automatique [27].

Le test de Nemenyi compare les paires d'échantillons. Lorsque la valeur p (p-value) pour deux combinaisons de tests est inférieure à la différence critique (dans notre cas $\alpha = 0.05$), il existe une différence statistiquement significative entre les deux échantillons. Le test compare le rang moyen de chaque échantillon pour chaque expérience, et calcule la valeur p pour chacun d'eux. Si la valeur p entre deux tests est inférieure à 0.05, alors les tests sont significativement différents. Les résultats peuvent être ensuite groupés à partir de la matrice des valeurs p pour créer des groupes semblables. Les paires ne faisant pas partie d'un même groupe sont différentes de façon statistiquement significative.

Chapitre 4

Expérimentations et résultats

4.1 Introduction

Dans ce chapitre, les résultats des expérimentations sont présentés et discutés. La section 4.2 présente les statistiques descriptives des 8 systèmes open source pour les métriques orientées objet, les mesures de centralité et les métriques de test. Les métriques de processus d'ANT y sont aussi présentées. La section 4.3 présente les corrélations de la même façon. La section 4.4 présente les résultats de l'analyse en composantes principales. La section 4.5 présente les ensembles de métriques utilisés dans les expérimentations. La section 4.6 présente les résultats des expérimentations sur les niveaux de test unitaire et les conclusions aux questions de recherche qui y sont reliées. La section 4.7 présente les résultats des expérimentations sur les classes candidates et les conclusions aux questions de recherche correspondantes. Finalement, la section 4.8 présente les résultats des algorithmes d'apprentissage du classement (Learning to Rank) et les conclusions aux questions de recherche correspondantes.

4.2 Statistiques descriptives

Dans cette section, nous présentons les statistiques descriptives pour les 8 systèmes open-source. Afin de comprendre la relation entre l'effort de test et les métriques logicielles, nous avons calculé les statistiques descriptives pour les métriques orientées objet, les mesures de centralité et les métriques de test à partir des données des 8 systèmes étudiés.

Le tableau 4.1 présente le nombre de classes dans chaque projet ainsi que le nombre

TABLE 4.1 – Nombre de classes par projet

	Avec Test	Total	Avec Test %
Apache ANT	159	493	32.25
Apache IO	62	117	52.99
Apache IVY	103	486	21.19
JFreeChart	336	610	55.08
Joda Time	59	165	35.76
Apache LUCENE	147	711	20.68
Apache Math	442	918	48.15
Apache POI	314	1205	26.06
Total	1622	4705	34.47

de classes testées et le pourcentage que ces dernières représentent.

Le tableau 4.2 présente les statistiques descriptives des métriques orientées objet. À partir de ces statistiques, nous pouvons observer que la valeur LOC médiane (117) est plus petite que la moyenne (302), ce qui indique qu'il y a quelques classes très grandes parmi plusieurs moyennes et petites. On note aussi que les classes ont généralement une faible complexité (médiane WMC de 5, écart type de 15, mais maximum de 231) et ont un faible couplage (moyenne Ca et Ce de 2 et 4, maximums élevés). Les minimums sont à 0 pour la plupart des métriques, car l'outil utilisé pour calculer les métriques orientées objet (CKJM-extended) mesure les métriques à partir du code Java compilé (bytecode). Quelques classes (ex. : org.jfree.chart.encoders.ImageFormat) ne contiennent pas de variables membres ni de méthodes et se retrouvent donc avec plusieurs métriques à 0. CKJM-extended calcule notamment la métrique LOC à partir de la somme du nombre d'instructions pour chaque méthode d'une classe et du nombre de variables membres. Le maximum de 151 pour NOC provient de la classe org.apache.poi.hssf.record.StandardRecord qui est héritée par un très grand nombre de classes dans le projet POI.

Le tableau 4.3 présente les statistiques descriptives des métriques orientées objet pour les classes ayant un test unitaire associé. Nous pouvons observer que la taille moyenne, la complexité et le couplage augmentent tous pour les classes qui ont été testées unitairement.

Le tableau 4.4 présente les statistiques descriptives des mesures de centralité pour toutes les classes Java présentes dans le jeu de données. Nous pouvons observer que la plupart des classes Java ne sont pas centrales dans le système global (EV moyen de 0,07,

TABLE 4.2 – Statistiques descriptives : Métriques orientées objet (8 systèmes) - Toutes les classes

	WMC	RFC	CBO	LCOM3	DIT	CA	CE	NOC	LOC
Nombre d'observations	4705	4705	4705	4705	4705	4705	4705	4705	4705
Minimum	0	0	0	0	1	0	0	0	0
Maximum	231	464	325	2	8	324	175	151	33563
1er Quartile	4	8	4	0.607143	1	1	2	0	32
Médiane	7	18	7	0.842105	1	2	4	0	117
3e Quartile	13	35	14	2	1	5	8	0	313
Moyenne	11.40043	28.47333	12.46185	1.056911	1.506908	6.554942	6.278215	0.560043	302.2087
Écart type	15.57523	36.96979	19.63885	0.651828	1.082485	18.07849	8.097577	3.139449	818.0489

TABLE 4.3 – Statistiques descriptives : Métriques orientées objet (8 systèmes) - Classes testées

	WMC	RFC	CBO	LCOM3	DIT	CA	CE	NOC	LOC
Nombre d'observations	1622	1622	1622	1622	1622	1622	1622	1622	1622
Minimum	0	0	0	0	1	0	0	0	0
Maximum	231	464	312	2	8	304	175	40	18283
1er Quartile	6	16	5	0.583333	1	0	3	0	109.25
Médiane	10	29	9	0.772393	1	1	6	0	251
3e Quartile	19	51	16	0.99401	2	4	11	0	530.75
Moyenne	16.07152	42.09864	14.9852	0.871054	1.738594	6.784217	8.64365	0.371147	489.3126
Écart type	19.62187	46.43437	23.09011	0.510782	1.173206	20.91073	9.722492	1.851783	961.2786

médiane de 0,04), mais qu'il existe des classes critiques, avec de nombreux voisins, et faisant partie de sous-graphes fortement connectés (en regardant les valeurs maximales pour SLC, BC, CL, DMNC).

TABLE 4.4 – Statistiques descriptives : Mesures de centralité (8 systèmes) - Toutes les classes

	EV	SLC	LE	LO	BC	CL	DMNC	DC
Nombre d'observations	4705	4705	4705	4705	4705	4705	4705	4705
Minimum	0	0	-0.9919	0	0	0	0	0
Maximum	1	2744668	0.932062	35	512028.2	193.2129	8.857143	325
1er Quartile	0.010403	25288	-0.61258	4	0	2.125	0	4
Médiane	0.039261	102894	-0.36831	7	12.88526	11.91383	0	8
3e Quartile	0.0977	250376	-0.06373	11	478.2098	25.08984	1.333333	14
Moyenne	0.074619	173155.1	-0.31751	7.984272	3346.507	17.51495	0.786326	12.83316
Écart type	0.108344	204212.5	0.395672	5.636616	24315.37	20.56577	1.124855	20.05795

Le tableau 4.5 présente les statistiques descriptives des mesures de centralité uniquement pour les classes auxquelles sont associées des classes de test JUnit. En comparant les moyennes de ce tableau au précédent, nous pouvons observer que les classes testées sont plus centrales dans leur projet logiciel que les classes non testées. Cela confirme que les développeurs choisissent souvent de tester les classes qui interagissent avec de

nombreuses autres classes, et qui sont centrales dans le projet.

TABLE 4.5 – Statistiques descriptives : Mesures de centralité (8 systèmes) - Classes testées

	EV	SLC	LE	LO	BC	CL	DMNC	DC
Nombre d'observations	1622	1622	1622	1622	1622	1622	1622	1622
Minimum	0	0	-0.99083	0	0	0	0	0
Maximum	1	2589527	0.919283	35	512028.2	193.2129	8.857143	314
1er Quartile	0.02167	66017.75	-0.58949	5	0	5.132813	0	5
Médiane	0.06303	189512.5	-0.36119	8	16.10022	15.14661	0	9
3e Quartile	0.127674	374074.5	-0.05945	12	543.9941	30.33496	1.333333	16
Moyenne	0.098174	248051.9	-0.30449	9.217633	4574	21.19338	0.871855	15.42787
Écart type	0.123379	244020.3	0.396518	6.091639	32018.16	22.88257	1.208514	23.68198

Le tableau 4.6 présente les statistiques descriptives calculées à partir des données combinées des 8 systèmes open-source, mais uniquement pour les classes ayant une classe de test unitaire associée. Les classes qui n'ont pas été testées n'ont pas été incluses dans le calcul des statistiques descriptives ici. Seules 1622 classes sur les 4705 avaient une classe de test associée (34.47%). Nous pouvons également noter que le cas de test moyen a 123.5 lignes de code et 27 assertions, mais que le cas de test médian est plus petit avec 68 LOC et 11 assertions, ce qui signifie que la plupart des classes de test sont petites et que seules quelques-unes sont grandes.

TABLE 4.6 – Statistiques descriptives : Métriques de test unitaire (8 systèmes) - Classes testées

	TLOC	TASSERT	TINVOK	TDATA
Nombre d'observations	1622	1622	1622	1622
Minimum	5	0	1	0
Maximum	2735	567	2178	746
1er Quartile	33	4	20	4
Médiane	68	11	44	12
3e Quartile	134	26	100.75	26
Moyenne	123.5536	26.80086	101.3113	27.10049
Écart type	174.5002	50.66363	179.2119	58.74675

Le tableau 4.7 présente les statistiques descriptives pour les métriques de processus extraites des 5 versions d'ANT. Les statistiques présentées dans ce tableau sont celles des classes testées et non testées. Les classes internes (définies à l'intérieur d'autres classes) ne sont pas incluses pour ce jeu de données, mais sont prises en considération

TABLE 4.7 – Statistiques descriptives : Métriques de processus (ANT 1.3 à 1.7) - Toutes les classes

	numCommits	numWords	numAuthors	insertions	deletions
Nombre d'observations	1440	1440	1440	1440	1440
Minimum	0	0	0	0	0
Maximum	175	3059	17	3962	2595
1er Quartile	3	38	2	92	7
Médiane	9	104	4	167	50
3e Quartile	14.28472	201.4944	4.638889	300.0653	127.8083
Moyenne	315.0099	81928.6	9.27743	158534.7	57925.48
Écart type	18	239.25	6	327.25	131.25

TABLE 4.8 – Statistiques descriptives : Métriques de processus (ANT 1.3 à 1.7) - Classes testées

	numCommits	numWords	numAuthors	insertions	deletions
Nombre d'observations	455	455	455	455	455
Minimum	0	0	0	0	0
Maximum	175	3059	17	3962	2595
1er Quartile	6	69	3	131.5	18
Médiane	13	169	5	257	84
3e Quartile	28	389	8	522	219
Moyenne	21.17802	310.789	5.778022	440.1912	201.6747
Écart type	23.63892	386.1444	3.482676	527.6195	321.3012

pour le calcul des mesures de centralité, car il est difficile de déterminer si une modification a été faite sur la classe interne (inner-class) ou la classe qui la contient pour le calcul des métriques de processus. Le tableau 4.8 présente les mêmes statistiques, mais seulement pour les classes avec un test unitaire associé. On dénote que les moyennes du nombre de modifications (numCommits), du nombre d'auteurs ayant modifié le fichier, du nombre de lignes de code modifiées (insertions, deletetions) sont plus élevées pour les fichiers avec des tests unitaires. Cela confirme que les développeurs ajoutent des tests lors des modifications du logiciel et de la correction de fautes. Plusieurs projets (incluant plusieurs de la fondation Apache) requièrent ou suggèrent fortement de tester unitairement les classes lors de la correction de fautes pour éviter les régressions [7]. Il est aussi parfois demandé de tester l'ensemble des classes ajoutées.

4.3 Corrélations

Pour avoir une vue complète des métriques, une étude de corrélation a été effectuée pour étudier les liens entre les métriques orientées objet, les mesures de centralité, les métriques de processus et les métriques du code de test. Seules les classes avec des tests JUnit ont été considérées pour l'analyse de corrélation. Le test de corrélation de Spearman a été utilisé, car un test de normalité (Shapiro-Wilk) a montré qu'aucune métrique ne suivait une distribution normale. Les couleurs des cases des tableaux avec les indices de corrélations suivent une échelle linéaire où les valeurs vertes sont les plus élevées (corrélations positives) et les valeurs rouges sont les moins élevées (corrélations négatives) selon l'indice de corrélation maximum et minimum présent dans le tableau.

Le tableau 4.9 présente les corrélations pour les métriques orientées objet. Ces corrélations ont été calculées à partir des données des 8 systèmes de notre jeu de données. Les corrélations du jeu de données contenant les données historiques des 5 versions d'ANT sont disponibles en annexe dans le tableau 1. Dans les deux cas, on note que WMC, RFC, CBO, CE, et LOC ont une corrélation significative avec les métriques de test.

Le tableau 4.10 présente les corrélations pour les mesures de centralité. Ces corrélations ont été calculées à partir des données des 8 systèmes de notre jeu de données. Les corrélations du jeu de données contenant les données historiques des 5 versions d'ANT sont disponibles en annexe dans le tableau 2. Dans les deux cas, on note que EV, SLC, LO et DC sont corrélées avec les mesures de test. Les métriques orientées objet de couplage sortant (CBO, Ce) sont très fortement corrélées aux mesures de centralité. Plusieurs mesures de centralité ont aussi de fortes corrélations entre elles (ex : DC et LO).

Le tableau 4.11 présente les corrélations pour les métriques de test. Ces corrélations ont été calculées à partir des données des 8 systèmes de notre jeu de données. Les corrélations du jeu de données contenant les 5 versions d'ANT sont disponibles en annexe dans le tableau 3. Dans les deux cas, on note que les métriques de test sont fortement corrélées entre elles.

Le tableau 4.12 présente les corrélations pour les métriques de processus. Cette fois, il est a noté que les corrélations ont été calculées pour les 5 versions d'ANT et non pas pour les 8 systèmes étant donné que les métriques de processus ne seront qu'étudiées que dans le cas des modèles prédictifs basés sur les données historiques. On note que les métriques de processus sont fortement corrélées entre elles et qu'elles ont

TABLE 4.9 – Corrélations des métriques orientées objet

Variables	WMC	RFC	CBO	LCOM3	DIT	CA	CE	NOC	LOC
WMC	1	0.878	0.498	-0.291	0.083	0.157	0.520	0.183	0.782
RFC	0.878	1	0.556	-0.348	0.120	0.040	0.692	0.150	0.907
CBO	0.498	0.556	1	-0.037	0.002	0.538	0.702	0.287	0.446
LCOM3	-0.291	-0.348	-0.037	1	-0.034	0.147	-0.215	-0.008	-0.387
DIT	0.083	0.120	0.002	-0.034	1	-0.143	0.056	0.030	0.146
CA	0.157	0.040	0.538	0.147	-0.143	1	-0.040	0.344	-0.028
CE	0.520	0.692	0.702	-0.215	0.056	-0.040	1	0.118	0.612
NOC	0.183	0.150	0.287	-0.008	0.030	0.344	0.118	1	0.104
LOC	0.782	0.907	0.446	-0.387	0.146	-0.028	0.612	0.104	1
EV	0.493	0.542	0.735	-0.114	0.082	0.273	0.623	0.163	0.451
SLC	0.445	0.520	0.762	-0.101	0.079	0.234	0.690	0.149	0.444
LE	0.301	0.299	0.618	0.024	0.027	0.537	0.273	0.304	0.241
LO	0.492	0.553	0.988	-0.041	-0.002	0.510	0.712	0.267	0.441
BC	0.358	0.399	0.620	-0.038	-0.052	0.515	0.514	0.262	0.323
CL	0.332	0.491	0.506	-0.164	-0.018	-0.057	0.784	0.072	0.411
DMNC	0.209	0.265	0.440	-0.034	-0.090	0.311	0.383	0.121	0.202
DC	0.494	0.556	0.993	-0.055	-0.006	0.537	0.719	0.283	0.447
TLOC	0.404	0.435	0.275	-0.099	0.210	-0.065	0.331	0.002	0.445
TASSERT	0.398	0.420	0.272	-0.102	0.193	-0.040	0.315	0.003	0.428
TINVOK	0.407	0.437	0.277	-0.097	0.207	-0.060	0.331	0.002	0.445
TDATA	0.388	0.419	0.268	-0.108	0.193	-0.068	0.328	0.002	0.431

de bonnes corrélations avec la taille des tests (TLOC), les métriques de centralité et certaines métriques orientées objet de couplage sortant, de complexité, de cohésion et de taille (WMC, RFC, CBO, LCOM3, Ce, LOC). Cela indique que les classes centrales et complexes sont parmi les plus modifiées.

TABLE 4.10 – Corrélation des mesures de centralité

Variables	EV	SLC	LE	LO	BC	CL	DMNC	DC
WMC	0.493	0.445	0.301	0.492	0.358	0.332	0.209	0.494
RFC	0.542	0.520	0.299	0.553	0.399	0.491	0.265	0.556
CBO	0.735	0.762	0.618	0.988	0.620	0.506	0.440	0.993
LCOM3	-0.114	-0.101	0.024	-0.041	-0.038	-0.164	-0.034	-0.055
DIT	0.082	0.079	0.027	-0.002	-0.052	-0.018	-0.090	-0.006
CA	0.273	0.234	0.537	0.510	0.515	-0.057	0.311	0.537
CE	0.623	0.690	0.273	0.712	0.514	0.784	0.383	0.719
NOC	0.163	0.149	0.304	0.267	0.262	0.072	0.121	0.283
LOC	0.451	0.444	0.241	0.441	0.323	0.411	0.202	0.447
EV	1	0.889	0.180	0.754	0.405	0.540	0.310	0.726
SLC	0.889	1	0.144	0.782	0.435	0.568	0.283	0.752
LE	0.180	0.144	1	0.565	0.476	0.069	0.315	0.624
LO	0.754	0.782	0.565	1	0.603	0.527	0.436	0.980
BC	0.405	0.435	0.476	0.603	1	0.435	0.380	0.628
CL	0.540	0.568	0.069	0.527	0.435	1	0.472	0.536
DMNC	0.310	0.283	0.315	0.436	0.380	0.472	1	0.496
DC	0.726	0.752	0.624	0.980	0.628	0.536	0.496	1
TLOC	0.293	0.317	0.130	0.279	0.103	0.160	0.021	0.263
TASSERT	0.289	0.305	0.131	0.277	0.110	0.144	0.024	0.260
TINVOK	0.296	0.318	0.130	0.281	0.106	0.161	0.024	0.265
TDATA	0.284	0.305	0.122	0.274	0.094	0.150	0.001	0.256

TABLE 4.11 – Corrélation des métriques de test

Variables	TLOC	TASSERT	TINVOK	TDATA
WMC	0.404	0.398	0.407	0.388
RFC	0.435	0.420	0.437	0.419
CBO	0.275	0.272	0.277	0.268
LCOM3	-0.099	-0.102	-0.097	-0.108
DIT	0.210	0.193	0.207	0.193
CA	-0.065	-0.040	-0.060	-0.068
CE	0.331	0.315	0.331	0.328
NOC	0.002	0.003	0.002	0.002
LOC	0.445	0.428	0.445	0.431
EV	0.293	0.289	0.296	0.284
SLC	0.317	0.305	0.318	0.305
LE	0.130	0.131	0.130	0.122
LO	0.279	0.277	0.281	0.274
BC	0.103	0.110	0.106	0.094
CL	0.160	0.144	0.161	0.150
DMNC	0.021	0.024	0.024	0.001
DC	0.263	0.260	0.265	0.256
TLOC	1	0.964	0.998	0.958
TASSERT	0.964	1	0.968	0.941
TINVOK	0.998	0.968	1	0.957
TDATA	0.958	0.941	0.957	1

TABLE 4.12 – Corrélation des métriques de processus (ANT 1.3 à 1.7)

	numCommits	numWords	numAuthors	insertions	deletions
WMC	0.544	0.546	0.468	0.689	0.547
RFC	0.580	0.592	0.493	0.698	0.593
CBO	0.505	0.522	0.459	0.512	0.499
LCOM3	0.332	0.361	0.311	0.230	0.342
DIT	0.154	0.146	0.217	0.117	0.213
CA	0.183	0.169	0.130	0.280	0.178
CE	0.539	0.567	0.507	0.556	0.564
NOC	0.125	0.111	0.081	0.068	0.107
LOC	0.534	0.550	0.452	0.688	0.559
EV	0.282	0.342	0.256	0.193	0.282
SLC	0.682	0.585	0.695	0.539	0.646
LE	0.318	0.364	0.234	0.434	0.319
LO	0.488	0.502	0.456	0.484	0.474
BC	0.370	0.326	0.339	0.471	0.399
CL	0.656	0.580	0.664	0.564	0.657
DMNC	0.441	0.432	0.414	0.385	0.444
DC	0.509	0.525	0.463	0.522	0.506
TLOC	0.314	0.315	0.302	0.466	0.309
TASSERT	0.184	0.204	0.154	0.326	0.134
TINVOK	0.291	0.297	0.266	0.451	0.268
TDATA	0.114	0.153	0.111	0.268	0.087
numCommits	1.000	0.927	0.939	0.729	0.918
numWords	0.927	1.000	0.850	0.659	0.840
numAuthors	0.939	0.850	1.000	0.670	0.862
insertions	0.729	0.659	0.670	1.000	0.790
deletions	0.917	0.839	0.862	0.790	1

4.4 Analyse en composantes principales

L'analyse en composantes principales (ACP) [3, 74] est une technique statistique utilisée pour représenter les données sur des axes orthogonaux afin de maximiser la variance [53].

Nous utilisons ici l'ACP pour réduire la dimensionnalité de nos données en identifiant les métriques qui offrent des informations redondantes (variables colinéaires). Comme certaines métriques couvrent les mêmes attributs internes de la qualité des logiciels, comme la complexité, la taille et le couplage, la colinéarité peut influencer les capacités de prédiction des modèles de régression et des classificateurs d'apprentissage automatique. Des ensembles de métriques ont été construits en sélectionnant une métrique par axe principal. Un sous-ensemble de métriques OO, de mesures de centralité (CM) et la combinaison de ces deux groupes ont été utilisés pour être comparés à l'ensemble complet de métriques. La technique de rotation Varimax a été utilisée dans le calcul des facteurs [39], pour faciliter l'interprétation des axes principaux résultants. Afin de garder une bonne partie des aspects couverts par les métriques et les mesures, nous avons gardé les axes principaux couvrant un certain pourcentage de la variance (80%). Une valeur seuil est couramment utilisée pour sélectionner les axes principaux les plus pertinents dans les analyses en composantes principales. Par exemple, en génie logiciel, le seuil de 80% de la variance a été utilisé pour l'analyse des métriques de test par Toure et al. [65].

Le tableau 4.13 présente les résultats de l'analyse en composantes principales des métriques orientées objet. Les métriques des 6 premiers axes ont été gardées, car elles expliquaient cumulativement au moins 80% de la variance. Les métriques résultantes sont les suivantes : CA, WMC, CE, LOC, LCOM3 et DIT. Ces métriques sont complémentaires dans leur représentation de la conception orientée objet (couplage, complexité, cohésion, taille et héritage).

Le processus a été répété pour les métriques de centralité. La majorité de la variance (plus de 80%) était expliquée par les 5 premiers axes. Les métriques les plus représentatives des axes ont été sélectionnées : SLC, LE, DMNC, CL et BC. Le tableau 4.14 présente les facteurs de l'analyse en composantes principales et le pourcentage de la variance expliqué.

Étant donné l'objectif de déterminer si la combinaison des mesures de centralité et des métriques orientées objet est bénéfique pour les différents problèmes de prédiction de l'effort de test, ces deux groupes ont été joints et une ACP a été faite pour réduire

TABLE 4.13 – Analyse en composantes principales - Métriques orientées objet

	F1	F2	F3	F4	F5	F6	F7
WMC	0.14	**0.95**	0.14	0.17	-0.07	-0.01	0.01
RFC	0.11	**0.81**	**0.47**	0.22	-0.11	0.04	0.01
CBO	**0.94**	0.19	0.26	0.07	0.00	0.01	0.09
LCOM3	0.03	-0.10	-0.06	-0.04	**0.99**	-0.03	0.00
DIT	0.00	0.01	0.01	0.01	-0.03	**1.00**	0.00
CA	**0.99**	0.05	-0.09	0.02	0.03	0.00	0.10
CE	0.10	0.40	**0.90**	0.14	-0.07	0.01	0.01
NOC	0.14	0.01	0.01	-0.01	0.00	0.00	**0.99**
LOC	0.07	0.26	0.13	**0.95**	-0.04	0.01	-0.01
% Variance Expliqué	0.22	0.21	0.13	0.11	0.11	0.11	0.11
% Variance Expliqué (cummulatif)	**0.22**	**0.42**	**0.55**	**0.66**	**0.78**	**0.89**	1.00

TABLE 4.14 – Analyse en composantes principales - Mesures de centralité

	F1	F2	F3	F4	F5	F6	F7
EV	**0.54**	0.17	0.24	0.11	0.04	**0.77**	0.12
SLC	**0.94**	0.08	0.08	0.15	0.05	0.2	0.12
LE	0.13	**0.95**	0.18	0.03	0.1	0.09	0.09
LO	**0.65**	**0.5**	0.34	0.25	0.08	0.28	-0.11
BC	0.07	0.1	0.05	0.09	**0.99**	0.03	0.04
CL	0.17	0.05	0.2	**0.96**	0.09	0.07	0.02
DMNC	0.16	0.23	**0.91**	0.23	0.06	0.16	0.06
DC	**0.54**	**0.47**	0.2	0.06	0.17	0.34	**0.54**
% Variance Expliqué	0.25	0.19	0.14	0.14	0.13	0.11	0.04
% Variance Expliqué (cummulatif)	**0.25**	**0.44**	**0.58**	**0.72**	**0.85**	0.96	1.00

la colinéarité. L'exécution d'une ACP sur la combinaison de ces deux groupes a donné l'ensemble final de l'ACP : CL, LE, LOC, SLC, DIT et LCOM3. Ces ensembles de métriques seront comparés entre eux et aux groupes complets de métriques pour voir si la réduction du nombre de variables et de la colinéarité améliore la précision des modèles prédictifs. Le tableau 4.15 présente les résultats de la 3e analyse en composantes principales.

Les résultats de ces analyses seront utilisés pour former les groupes de métriques

TABLE 4.15 – Analyse en composantes principales - Métriques orientées objet et Mesures de centralité

	F1	F2	F3	F4	F5	F6	F7
CA	-0.16	0.51	0	**0.78**	-0.05	0.05	0.14
WMC	0.35	0.26	**0.66**	0.12	0	-0.14	0.06
CE	**0.78**	0.15	0.38	0.12	0.07	-0.08	0.1
LOC	0.07	0.02	**0.91**	0.08	0	-0.01	0.01
LCOM3	-0.08	0.02	-0.08	-0.01	-0.03	**0.99**	-0.01
DIT	-0.01	-0.02	0	0.05	**0.99**	-0.03	-0.03
SLC	0.37	0.03	0.23	**0.85**	0.13	-0.06	0.02
LE	0.07	**0.88**	0.18	0.15	0.03	0.01	0.13
DMNC	**0.55**	**0.63**	0.03	0.14	-0.11	0.01	-0.04
CL	**0.91**	0.03	0.06	0.04	-0.03	-0.05	0.1
BC	0.13	0.1	0.05	0.08	-0.03	-0.01	**0.97**
% Variance Expliqué	0.22	0.16	0.16	0.15	0.11	0.11	0.11
% Variance Expliqué (cummulatif)	**0.22**	**0.38**	**0.53**	**0.68**	**0.79**	**0.89**	1.00

pour vérifier si la réduction du nombre de variables et la colinéarité ont un impact sur la capacité des modèles d'apprentissage automatique (classification des niveaux de test, prédiction des classes candidates) et les algorithmes de Learning to Rank (classement des classes selon l'effort de test requis) à prédire les différents aspects de l'effort de test unitaire.

4.5 Ensembles de métriques

Un des objectifs de ce travail est d'examiner comment les métriques orientées objet, les mesures de centralité et les métriques de processus permettent, lorsqu'elles sont combinées, de prédire l'effort de test unitaire. Dans ce but, les métriques orientées objet et les mesures de centralité ont été divisées en différents ensembles.

L'ensemble de métriques COM combine l'ensemble des métriques orientées objet et des mesures de centralité. Les ensembles OO et CM contiennent leurs métriques respectives telles que définies dans la section sur les métriques logicielles.

Correlation Feature Selection (CFS) est un algorithme qui classe les attributs en utilisant une approche basée sur la corrélation avec des heuristiques. Il sélectionne les

caractéristiques qui sont « hautement corrélées avec la classe et non corrélées entre elles » [35]. Dans notre cas, l'algorithme de sélection choisit les caractéristiques parmi les mesures OO et de centralité en mesurant la corrélation entre elles et les 4 métriques de test. L'ensemble COR contient les métriques résultantes de CFS.

Les ensembles de métriques PCA (PCA_OO, PCA_CM, PCA) contiennent les métriques résultantes de l'analyse en composantes principales présentée dans la section précédente.

L'ensemble COMMITS contient les métriques de processus tirées de l'analyse du système de contrôle de versions Git du projet ANT. L'ensemble ALL combine l'ensemble des métriques orientées objet, des mesures de centralité et des métriques de processus pour les expérimentations faites sur ANT.

La table 4.16 présente les ensembles de métriques sous forme de tableau. Les ensembles marqués d'une astérisque seront présents seulement dans les expérimentations faites avec les données historiques d'ANT.

4.6 Prédiction des niveaux d'effort de test unitaire

Un algorithme de partitionnement prend des données à n dimensions et regroupe chaque observation dans un certain nombre de groupes de nature similaire. Nous avons utilisé les métriques de test comme entrées pour le processus de partitionnement. La sortie est le niveau d'effort de test (ex. : faible, moyen, élevé).

4.6.1 Q1 : Comparaison des algorithmes de partitionnement

Dans cette section, nous comparons deux algorithmes de partitionnement couramment utilisés dans la littérature par rapport à la précision des modèles prédictifs résultants. Le nombre de niveaux est fixé à 3 et les données sont celles des 8 systèmes avec ANT comme logiciel d'évaluation. Le tableau 4.17 présente le nombre de classes Java par partitions des 8 systèmes de l'algorithme k-means. Le tableau 4.18 présente le partitionnement des classes du partitionnement hierachique agglomératif (AHC). On note que k-means génère des partitions plus petites pour les classes avec les plus grands efforts de test (moyen et élevé).

La figure 4.1 présente les résultats de la comparaison de k-means et AHC. On observe que k-means donnent de meilleurs résultats dans la grande majorité des cas. Plusieurs algorithmes et classificateurs donnent de bons résultats, dont les réseaux bayé-

TABLE 4.16 – Groupes de métriques

Groupe	Description	Métriques
COM	Combinaison des métriques orientés objet et des mesures de centralité	WMC, RFC, CBO, LCOM3, DIT, CA, CE, NOC, LOC, EV, SLC, LE, LO, BC, CL, DMNC, DC
OO	Métriques orientées objet	WMC, RFC, CBO, LCOM3, DIT, CA, CE, NOC, LOC
CM	Mesures de centralité	EV, SLC, LE, LO, BC, CL, DMNC, DC
PCA_OO	Analyse en composantes principales Métriques orientées objet	WMC, CA, CE, LOC, LCOM3, DIT
PCA_CM	Analyse en composantes principales Métriques orientées objet	SLC, LE, DMNC, CL, BC
PCA	Analyse en composantes principales Combinaison OO & CM	CL, LE, LOC, SLC, DIT, LCOM3
COR	Correlation-based Feature Selection (CFS) [35]	WMC, RFC, LOC, SLC, LE, BC, DMNC
Commits*	Métriques de processus	numCommits, numWords, numAuthors, insertions, deletions
ALL*	Métriques de processus, orientés objet et mesures de centralité	WMC, RFC, CBO, LCOM3, DIT, CA, CE, NOC, LOC, EV, SLC, LE, LO, BC, CL, DMNC, DC numCommits, numWords, numAuthors, insertions, deletions

TABLE 4.17 – Partitionnement des données - k-means - 3 niveaux d'effort de test

Niveau	Entrainement	Évaluation
Faible	1158	119
Moyen	243	33
Élevé	62	7

siens (BayesNet) et RandomForest. La combinaison des métriques OO et des mesures de centralité a été bénéfique dans plusieurs cas, indépendamment de l'algorithme de partitionnement utilisé. Une comparaison des algorithmes et des classificateurs plus détaillée sera faite dans les sections suivantes afin de mieux répondre aux autres questions de recherche.

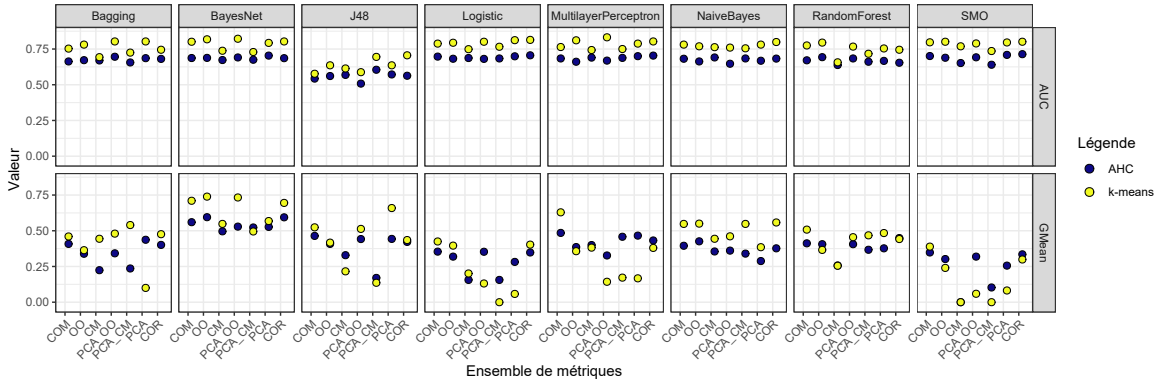
À partir de ces résultats, nous concluons à la première question de recherche (Q1) que

TABLE 4.18 – Partitionnement des données - AHC - 3 niveaux d'effort de test

Niveau	Entrainement	Évaluation
Faible	1089	70
Moyen	289	81
Élevé	85	8

le choix d'algorithme de partitionnement est un élément important pour l'entrainement des modèles prédictifs du niveau de l'effort de test. Les classes d'un système orienté objet doivent être groupées de façon précise pour que les algorithmes d'apprentissage automatique puissent apprendre et guider le processus de test unitaire.

FIGURE 4.1 – Résultats - Algorithmes de partitionnement



4.6.2 Q2 : Comparaison de l'impact du nombre de niveaux

Dans cette section, nous comparerons l'impact du nombre de niveaux dans le processus de partitionnement avec la précision des modèles de prédiction résultants. L'algorithme de partitionnement utilisé est k-means et les données sont celles des 8 logiciels avec ANT comme logiciel de validation.

Le tableau 4.17 présenté précédemment montre le partitionnement pour 3 niveaux.

Le tableau 4.19 présente le partitionnement pour 4 niveaux d'effort de test. La différence par rapport à 3 niveaux se situe principalement dans les classes avec un effort de test plus important. Dans les données d'entrainement, il y a 3 classes Java faiblement testées et 1 classe moyennement testée de moins. Les classes logicielles avec un niveau d'effort de test élevé ont été séparées en deux catégories par l'algorithme.

TABLE 4.19 – Partitionnement des données - k-means - 4 niveaux d'effort de test

Niveau	Entrainement	Évaluation
Faible	1155	119
Moyen	242	33
Élevé	55	1
Très élevé	11	6

TABLE 4.20 – Partitionnement des données - k-means - 5 niveaux d'effort de test

Niveau	Entrainement	Évaluation
Faible	1081	87
Moyen	281	46
Élevé	69	19
Très élevé	11	1
Critique	21	6

Le tableau 4.20 présente le partitionnement pour 5 niveaux d'effort de test. La différence par rapport à 4 niveaux se situe encore une fois dans les classes avec un effort de test plus important. Dans les données d'entrainement, il y a 74 classes faiblement testées de moins, mais 39 classes moyennement testées de plus. Les classes avec un niveau d'effort de test élevé ont été séparées en deux catégories par l'algorithme.

La figure 4.2 présente les résultats de la classification de l'effort de test pour 3 niveaux, 4 niveaux, et 5 niveaux. On observe que les classificateurs d'apprentissage automatique ont plus de facilité avec un nombre de niveaux plus faible, mais que les résultats restent précis même avec 5 niveaux. On remarque aussi que la combinaison des métriques orientées objet et des mesures de centralité (ensembles COM et COR) permet d'obtenir dans la plupart des cas les résultats les plus précis.

Le test statistique de Friedman sur l'AUC et le g-mean donne que la différence entre le nombre de partitions est statistiquement significative (p-value de 2.2×10^{-16}). Le test de Nemenyi montre que la différence entre chacune des paires est statistiquement significative. La table 4.21 présente la moyenne des rangs ainsi que les groupes similaires. On constate que la différence entre chaque nombre de niveaux est statistiquement significative.

En conclusion, pour répondre à la deuxième question de recherche (Q2), nous observons à partir de ces résultats que le choix du nombre de niveaux pour représenter l'effort de test a un impact sur la précision des résultats. Sur notre jeu de données,

FIGURE 4.2 – Résultats - Nombre de niveaux d'effort de test

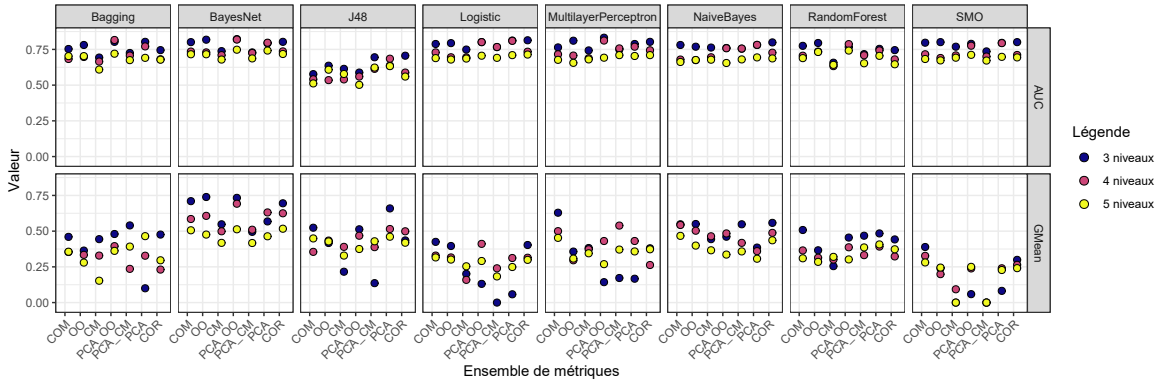


TABLE 4.21 – Test de Nemenyi - Nombre de niveaux

	n	Somme des rangs	Moyenne des rangs	Groupes
3 niveaux	112	292.000	2.607	A
4 niveaux	112	229.500	2.049	B
5 niveaux	112	150.500	1.344	C

nous observons qu'augmenter le nombre de niveaux (et par conséquent la granularité des résultats) s'avère un problème plus difficile pour les algorithmes d'apprentissage automatique, mais que les résultats obtenus restent assez précis même avec 5 niveaux et permettent d'orienter le processus de test unitaire.

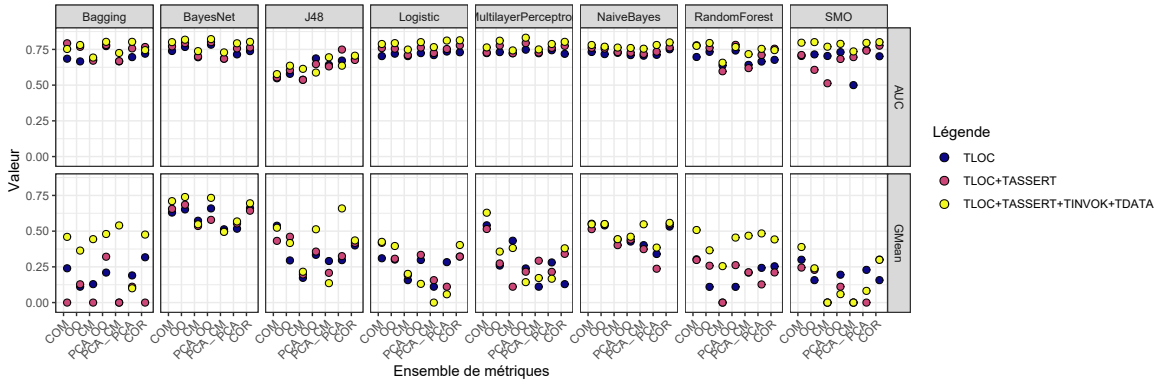
4.6.3 Q3 : Impact des métriques de test

Cette section tente de répondre à la question de recherche Q3 qui s'intéresse à la contribution des métriques de test. Les résultats des modèles prédictifs entrainés avec l'ensemble des métriques de test (TLOC, TASSERT, TINVOK et TDATA) (1) seront comparés aux modèles entrainés avec (2) TLOC et TASSERT ; et (3) TLOC seulement. L'objectif est de confirmer que les métriques de test supplémentaires sont bénéfiques à la prédiction des niveaux d'effort de test. Les niveaux de test pour les 3 expérimentations ont été obtenus avec k-means (3 partitions : faible, moyen, élevé). Le jeu de données est celui avec les 8 logiciels et l'objectif de prédiction est le projet ANT.

La figure 4.3 présente les résultats de la comparaison. On note que dans la quasi-totalité des cas, utiliser l'ensemble des métriques de test pour représenter l'effort de test et partitionner les classes selon le niveau de test (faible, moyen, élevé) est bénéfique.

Le test de Friedman sur l'AUC et le g-mean indique qu'il y a une différence statis-

FIGURE 4.3 – Résultats - Comparaison des métriques de test



tiquement significative entre les résultats (p-value $= 2.2e-16 < \alpha = 0.05$). La table 4.22 présente les résultats du test de Nemenyi. Ce tableau confirme que sur notre jeu de données, l'utilisation de toutes les métriques de test est bénéfique et que la différence est statistiquement significative, peu importe le classificateur utilisé ou l'ensemble de métriques logicielles.

TABLE 4.22 – Test de Nemenyi - Métriques de tests

	n	Somme des rangs	Moyenne des rangs	Groupes
TLOC+TASSERT+TINVOK+TDATA	112	299.000	2.670	A
TLOC+TASSERT	112	204.500	1.826	B
TLOC	112	168.500	1.504	C

En conclusion à la question Q3, nos résultats indiquent que chaque métrique de test unitaire contribue à la précision des modèles prédictifs des niveaux d'effort de test. Ajouter le nombre d'assertions contenues dans le code des tests unitaires (TASSERT) à la taille des tests (TLOC) est significativement plus précis que TLOC seule. La classification la plus précise a été obtenue en combinant ces deux métriques avec le nombre de méthodes invoquées par le code de test (TINVOK) et le nombre d'instanciations d'objets (TDATA) à l'étape de partitionnement en niveaux.

4.6.4 Q4 : Données historiques et prédictions inter-projets

Dans cette section, nous étudions l'impact de l'utilisation de données historiques sur la performance des algorithmes d'apprentissage automatique. Une comparaison est faite entre l'utilisation d'une seule version pour l'entrainement et une autre pour l'évaluation, l'utilisation de plusieurs versions pour l'entrainement et une autre pour l'évaluation, ou

l'absence de données historiques (avec une validation croisée 10 fois, qui utilise un sous-ensemble de données pour l'entrainement et le reste pour évaluer la précision de la prédiction).

1. **Validation croisée 10 fois :** Une seule version est utilisée pour l'évaluation et l'entrainement (ANT 1.7). Les données sont divisées de manière aléatoire en 10 sous-ensembles de taille similaire, en utilisant 9 sous-ensembles pour l'entrainement et le dernier pour l'évaluation. Le processus est répété 10 fois, en prenant chaque fois un ensemble différent pour l'évaluation. Cela permet de voir dans quelle mesure nous pouvons entrainer un modèle prédictif sans collecter de données historiques.
2. **Version unique :** Utiliser une seule version (ANT 1.6) pour entrainer les modèles d'apprentissage automatique et évaluer les capacités prédictives de l'effort de test unitaire sur une autre version (ANT 1.7).
3. **Versions multiples :** Utiliser plusieurs versions (ANT 1.3 à ANT 1.6) pour entraîner les modèles d'apprentissage automatique et évaluez les capacités prédictives de l'effort de test unitaire sur une autre version (ANT 1.7). Cela permet de voir l'impact d'avoir plus de données historiques pour apprendre.
4. **Validation inter-projets :** Ce sont les résultats présentés précédemment à titre comparatif (modèles entrainés à partir des autres projets open-souce - 3 niveaux avec k-means).

Les méthodes d'entrainement utilisant des données historiques incluent les métriques de processus tirées du contrôle de versions présentées précédemment.

Nous évaluerons et comparerons les 4 méthodes en utilisant l'évaluation des résultats de prédiction avec les valeurs g-mean et AUC. Les résultats seront présentés dans les sections suivantes et comparés à l'aide du test de Friedman-Nemenyi pour vérifier si la différence est statistiquement significative. L'algorithme de partitionnement utilisé ici est k-means.

La figure 4.4 présente les résultats des 4 expérimentations. En regardant les valeurs g-mean, on remarque que plusieurs combinaisons de classificateurs ont eu de la difficulté à classifier l'effort de test avec la validation croisée et avec les prédictions inter-projets, mais certaines combinaisons ont donné de bons résultats (ex : COM et OO avec les réseaux bayésiens : AUC ≈ 0.801, g-mean ≈ 0.710).

Le test de Friedman a été effectué sur l'AUC et le g-mean pour déterminer s'il y a une différence statistique significative entre les résultats des 4 façons d'entrainer. Le

FIGURE 4.4 – Résultats - Données historiques - Niveaux de test

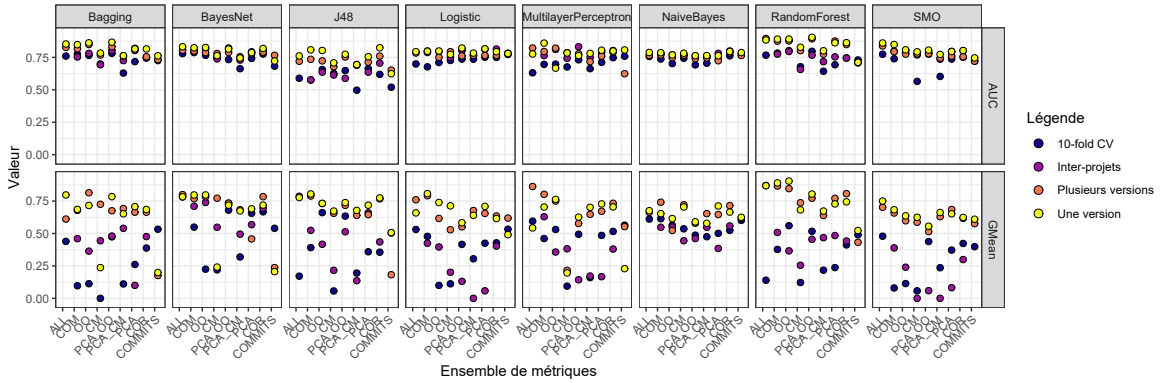


TABLE 4.23 – Test de Nemenyi - Méthodes d'entrainements (Niveaux de test)

	n	Somme des rangs	Moyenne des rangs	Groupes		
Une version	136	449.000	3.301	A		
Plusieurs versions	136	398.000	2.926	A		
Inter-projets	136	305.000	2.243		B	
10-fold CV	136	208.000	1.529			C

résultat du test de Friedman donne une p-value de 2.2×10^{-16}, ce qui signifie qu'il y a une différence statistiquement significative entre les 4 méthodes d'entrainement.

Le test de Nemenyi a été effectué pour comparer les paires de méthodes d'entrainement. La moyenne des rangs et les groupes obtenus lors du test de Nemenyi sont présentés dans le tableau 4.23. On constate que la différence entre chacune des paires est statistiquement significative à l'exception d'une version et de plusieurs versions, possiblement dû à la taille de l'échantillon. On peut donc conclure que, sur nos jeux de données, la prédiction des niveaux de test unitaire bénéficie des données historiques.

En conclusion, pour répondre à la 4e question de recherche (Q4), nous observons qu'utiliser des données historiques donne généralement des résultats plus précis que d'entrainer les modèles de prédiction sur d'autres logiciels, mais que cette seconde façon de faire donne quand même des résultats précis qui peuvent guider l'effort de test. Les résultats suggèrent donc que les modèles basés sur les métriques logicielles peuvent être généralisés d'un projet à un autre avec un faible impact sur la précision.

4.6.5 Q5 & Q6 : Mesures de centralité et métriques de processus

Afin de déterminer si la combinaison des métriques orientées objet et des mesures de centralité (Q5) est bénéfique à la prédiction des niveaux de test et si l'ajout des métriques de processus est bénéfique aux modèles entrainés avec les données historiques (Q6), nous avons effectué le test de Friedman pour comparer les ensembles de métriques à partir des résultats présentés précédemment.

TABLE 4.24 – Test de Nemenyi - Ensembles de métriques (Niveaux de test)

	n	Somme des rangs	Moyenne des rangs	Groupes	
COM	61	354.000	5.803	A	
ALL	61	352.500	5.779	A	
OO	61	345.500	5.664	A	
PCA_OO	61	323.000	5.295	A	B
COR	61	304.500	4.992	A	B
CM	61	284.500	4.664	A	B
PCA	61	276.500	4.533	A	B
COMMITS	61	264.500	4.336	A	B
PCA_CM	61	240.000	3.934		B

Le test de Friedman des valeurs du G-mean et de l'AUC révèle qu'il existe une différence statistique entre les ensembles de métriques (p-value = 2.11×10^{-4}). Afin de déterminer quelles paires sont significativement différentes, le test de Nemenyi a été effectué.

Le tableau 4.24 présente les résultats du test de Nemenyi. On constate que la combinaison des métriques orientées objet et des mesures de centralités (COM) est bénéfique à la classification des niveaux de test (plus grande moyenne des rangs), mais que la différence n'a pas pu être démontrée significative avec le test statistique. Le test révèle que peu d'ensembles de métriques se distinguent statistiquement les uns des autres, possiblement dû à la taille de l'échantillon. La variance élevée des valeurs du g-mean dans certains classificateurs (notamment avec la validation croisée) est aussi une explication possible pour expliquer ce résultat.

L'ajout des métriques de processus (ALL) a augmenté la précision de certains classificateurs (ex. : RandomForest, BayesNet, et MultilayerPerceptron lors de l'entraine-

ment avec plusieurs versions), mais dans un certain nombre de cas nuisent légèrement aux prédictions. Les groupes tirés de l'analyse par composantes principales (PCA_OO, PCA_COM et PCA) performent en moyenne de façon inférieure comparativement aux groupes de métriques avec l'ensemble des variables. La réduction de la colinéarité ne semble pas être bénéfique aux classificateurs pour l'identification des niveaux d'effort de test requis sur une classe Java.

En conclusion à la question de recherche Q5, l'ajout des mesures de centralité aux métriques orientées objet apparaît être bénéfique dans le contexte de la prédiction des niveaux d'effort de test pour les modèles entrainés à partir d'autres logiciels et pour les modèles entrainés sur des versions précédentes d'un même logiciel. Réduire la colinéarité et le nombre de variables réduit significativement la capacité des modèles prédictifs à correctement classifier les niveaux d'effort de test.

Pour la question Q6, l'ajout des métriques de processus aux métriques logicielles (orientées objet et de centralité) a été bénéfique pour plusieurs modèles avec des données historiques sur notre jeu de données. L'ensemble de métriques combinant les métriques de processus aux autres métriques s'est classé en première position pour plusieurs classificateurs.

4.6.6 Q7 : Comparaison des algorithmes de classification

Afin de répondre à la 7e question de recherche (Q7) s'il y a un classificateur plus approprié pour la classification de l'effort de test, nous avons effectué le test de Friedman sur les résultats présentés précédemment.

La p-value du test de Friedman est de $1.57e-11$, ce qui signifie qu'il y a une différence statistique entre les classificateurs.

Le tableau 4.25 présente les résultats du test de Nemenyi. Les résultats confirment que RandomForest se distingue de la majorité des algorithmes. En deuxième place on retrouve les 2 classificateurs de réseaux bayésiens. Le test de Nemenyi montre cependant que la différence entre Random Forest et ces 2 classificateurs n'est pas statistiquement significative, possiblement due à la taille de l'échantillon.

En conclusion à la question 7 (Q7), le choix d'algorithme pour classifier les niveaux d'effort de test unitaire requis a un impact important sur la précision des prédictions. RandomForest se distingue des autres algorithmes dans plusieurs cas, particulièrement pour les prédictions avec des données historiques (ex. : AUC = 0.891, G-mean = 0.904).

Les réseaux bayésiens ont donné en moyenne les meilleurs résultats et se sont distingués pour les prédictions inter-projets (ex. : AUC = 0.818, G-mean = 0.739).

TABLE 4.25 – Test de Nemenyi - Classificateurs - Niveaux de test

	n	Somme des rangs	Moyenne des rangs	Groupes			
BayesNet	68	392.000	5.765	A			
RandomForest	68	388.000	5.706	A	B		
Logistic	68	304.500	4.478		B	C	
MultilayerPerceptron	68	304.500	4.478		B	C	
NaiveBayes	68	303.500	4.463		B	C	
Bagging	68	293.500	4.316			C	D
SMO	68	252.000	3.706			C	D
J48	68	210.000	3.088				D

4.7 Prédiction des classes candidates aux tests unitaires

Comme mentionné précédemment, le choix de ne pas tester toutes les classes d'un logiciel est souvent fait pour réduire les coûts et le temps requis du processus de test unitaire. Identifier les classes les plus critiques à tester est un problème important en génie logiciel.

Dans cette section, les résultats de la prédiction des classes qui sont de bonnes candidates aux tests sont présentés.

4.7.1 Q8 : Données historiques et prédictions inter-projets

Dans cette section, nous comparons 4 façons différentes d'entrainer les modèles à prédire les classes candidates aux tests. Le but est de déterminer si les données historiques sont bénéfiques pour le problème des classes candidates et si les modèles entrainés à partir de plusieurs projets sont généralisables à un autre. Les 4 façons d'entrainer les modèles sont les suivantes :

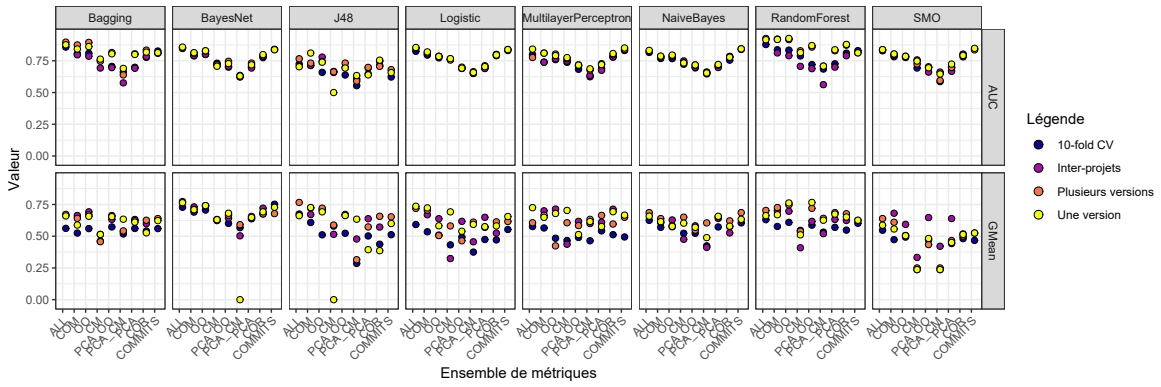
1. **Validation croisée 10 fois :** Une seule version est utilisée pour l'évaluation et l'entrainement (ANT 1.7). Les données sont divisées de manière aléatoire en 10

sous-ensembles de taille similaire, en utilisant 9 sous-ensembles pour l'entrainement et le dernier pour l'évaluation. Le processus est répété 10 fois, en prenant chaque fois un ensemble différent pour l'évaluation. Cela permet de voir dans quelle mesure nous pouvons entrainer un modèle prédictif sans collecter de données historiques.

2. **Version unique :** Utiliser une seule version (ANT 1.6) pour entrainer les modèles d'apprentissage automatique et évaluer les capacités prédictives de l'effort de test unitaire sur une autre version (ANT 1.7).
3. **Versions multiples :** Utiliser plusieurs versions (ANT 1.3 à ANT 1.6) pour entraîner les modèles d'apprentissage automatique et évaluer les capacités prédictives de l'effort de test unitaire sur une autre version (ANT 1.7). Cela permet de voir l'impact d'avoir plus de données historiques pour apprendre.
4. **Validation inter-projets :** Les modèles sont entrainés avec 7 systèmes open source (IVY, Math, JFreeChart, IO, JODA Time, Lucene, POI) et évalués sur ANT 1.7.

La figure 4.5 présente les résultats pour les 4 expérimentations. On observe des résultats précis, spécialement lorsqu'on utilise des données historiques avec plusieurs versions (ex. : AUC = 0.926, G-Mean = 0.703) ou avec une version (ex. : AUC : 0.926, g-mean 0.762).

FIGURE 4.5 – Résultats - Classes candidates



Le test de Friedman donne une p-value = 1.71×10^{-7}, ce qui indique qu'il y a une différence entre les méthodes d'entrainement des modèles et que cette différence est statistiquement significative.

La table 4.26 présente la moyenne des rangs obtenue avec le test de Nemenyi comparant les méthodes d'entrainement. On observe qu'il y a une différence importante et

statistiquement significative entre les modèles entrainés avec des données historiques (une version et plusieurs versions) et les autres (inter-projets et validation croisée).

TABLE 4.26 – Test de Nemenyi - Classes candidates

	n	Somme des rangs	Moyenne des rangs	Groupes
Une version	136	390.000	2.868	A
Plusieurs versions	136	376.000	2.765	A
Inter-projets	136	313.500	2.305	B
10-fold CV	136	280.500	2.063	B

4.7.2 Q9 & Q10 : Mesures de centralité et métriques de processus

Dans cette section, on s'intéresse aux deux questions de recherche sur les bénéfices de combiner les métriques orientées objet aux mesures de centralité (Q9) et aux métriques de processus.

Afin de déterminer si les différences observées étaient statistiquement significatives, les tests de Friedman et de Nemenyi ont été faits sur les valeurs de l'AUC et du g-mean des modèles prédictifs présentés dans la section précédente pour comparer les ensembles de métriques. Le test de Friedman donne un résultat positif quant à la différence statistique entre les ensembles de métriques (p-value = 1.13×10^{-14}). La table 4.27 présente la moyenne des rangs des ensembles de métriques pour la prédiction des classes candidates et les groupes de mesures semblables selon le test de Nemenyi.

On observe que la combinaison de toutes les métriques et mesures (ensemble ALL) se distingue des autres. Contrairement aux expérimentations sur les niveaux de test (ex. : faible, moyen, élevée), les métriques de processus seules (ensemble COMMITS) performent de manière remarquable comparativement aux autres ensembles de métriques orientées objet et de centralité. Cela est logique considérant que plusieurs projets suggèrent ou obligent les développeurs à développer des tests unitaires lors de la correction de fautes, pour éviter les régressions futures et s'assurer que la faute en question est bien corrigée. Cela fait en sorte que les classes avec des tests unitaires sont aussi les classes qui sont modifiées le plus fréquemment (*numCommits*), qui ont le plus grand nombre d'auteurs (*numAuthors*) et dont les modifications sont plus importantes en termes de

TABLE 4.27 – Test de Nemenyi - Ensembles de métriques - Classes candidates

	n	Somme des rangs	Moyenne des rangs	Groupes			
ALL	61	399.000	6.541	A			
COMMITS	61	367.000	6.016	A	B		
COM	61	357.500	5.861	A	B		
OO	61	345.000	5.656	A	B	C	
PCA_OO	61	289.000	4.738		B	C	
COR	61	285.000	4.672		B	C	
CM	61	273.500	4.484		B	C	
PCA	61	253.000	4.148			C	D
PCA_CM	61	176.000	2.885				D

lignes de code (*insertions, deletions*). La combinaison des métriques orientées objet et des mesures de centralité donne aussi de bons résultats avec plusieurs classificateurs.

On observe cependant que les différences notées précédemment ne sont pas toujours significatives statistiquement, mais, étant donnée la grande variance du g-mean de plusieurs classificateurs, la tendance présente pour plusieurs ensembles n'est pas à rejeter.

En conclusion à la question 9 (Q9), la combinaison des métriques orientées objet et des mesures de centralité est bénéfique pour plusieurs combinaisons de classificateurs sur notre jeu de données. On dénote deux cas où les classificateurs Random Forest et Bagging permettaient une classification plus précise avec seulement les métriques orientées objet (ensemble OO). Toutefois, la combinaison des deux types de métriques (ensemble COM) restait très précise et était bénéfique avec les autres classificateurs.

Pour la question 10 (Q10), ajouter les métriques de processus aux métriques logicielles semble être bénéfique quant à la précision des modèles prédictifs des classes candidates. Avec plusieurs classificateurs, les métriques de processus seules (ensemble COMMITS) permettaient de prédire précisément les classes candidates. Il faut cependant prendre en considération les politiques de test de la fondation Apache qui suggèrent pour plusieurs projets de tester unitairement les classes pour lesquelles des correctifs sont apportés. Le lien observé ici pourrait ne pas être valide pour les projets n'ayant pas une politique similaire.

4.7.3 Q11 : Comparaison des algorithmes de classification

Dans cette section, les résultats de la comparaison des algorithmes de classification pour la prédiction des classes candidates sont présentés.

Les données analysées sont celles présentées précédemment pour les questions Q8 et Q9.

Afin de déterminer si la différence observée est statistiquement significative, les tests de Friedman et de Nemenyi sont présentés. La p-value du test de Friedman est plus petite que $\alpha = 0.05$ (2.2e-16), ce qui signifie qu'il existe une différence statistiquement significative entre les classificateurs. Le tableau 4.28 présente les moyennes des rangs et les groupes obtenus lors du test de Nemenyi. Comme pour la prédiction des classes candidates, on note que RandomForest et les réseaux bayésiens se distinguent des autres algorithmes.

Table 4.28 – Test de Nemenyi - Classificateurs - Classes candidates

	n	Somme des rangs	Moyenne des rangs	Groupes				
RandomForest	68	435.000	6.397	A				
BayesNet	68	397.500	5.846	A	B			
Bagging	68	329.500	4.846		B	C		
MultilayerPerceptron	68	295.500	4.346			C	D	
NaiveBayes	68	291.000	4.279			C	D	
Logistic	68	285.000	4.191			C	D	
J48	68	217.500	3.199				D	E
SMO	68	197.000	2.897					E

On observe notamment que les différences entre les deux classificateurs les plus précis (RandomForest, BayesNet) et les autres classificateurs sont statistiquement significatives dans la grande majorité des cas.

En conclusion à la question 11 (Q11), le choix de classificateur pour identifier les classes qui sont de bonnes candidates aux tests unitaires a un impact significatif sur la précision des prédictions. Sur notre jeu de données, RandomForest et les réseaux bayésiens (BayesNet) se sont distingués des autres classificateurs de manière statistiquement significative. On note tout de même que les autres classificateurs permettent d'obtenir des prédictions de qualité des classes à tester.

4.8 Classement de l'effort unitaire (Learning to Rank)

Dans cette section, les résultats des expérimentations sur le classement des clases Java selon l'effort requis pour le test unitaire sont présentés.

4.8.1 Q12 : Données historiques et prédictions inter-projets

Pour répondre à la question de recherche Q12, nous avons entrainé des modèles de prédiction du classement (Learning To Rank) sur 7 des 8 projets (Lucene, Apache IO, etc.) et utilisé le 8e projet (ANT) pour évaluer le modèle. Nous avons choisi de mettre tous les projets sauf un dans les données d'entrainement pour maximiser la capacité des modèles à être généralisés et pour pouvoir comparer les résultats avec des modèles entrainés sur les données historiques d'ANT (version 1.3 à 1.7).

La variable dépendante que nous essayons de prédire est la combinaison des métriques de test (TLOC, TASSERT, TDATA, TINVOK) en une seule valeur normalisée pour comparer les classes par pertinence. Les valeurs des quatre métriques ont été combinées en prenant la valeur de chaque métrique et en la divisant par la valeur maximale observée dans les ensembles de données, puis en faisant la moyenne des quatre valeurs.

L'ensemble des classes (testées et non testées) sont incluses lors de l'entrainement et de l'évaluation.

Comme pour les deux autres parties sur les niveaux de test et les classes candidates, 4 façons d'entrainer les modèles sont comparées :

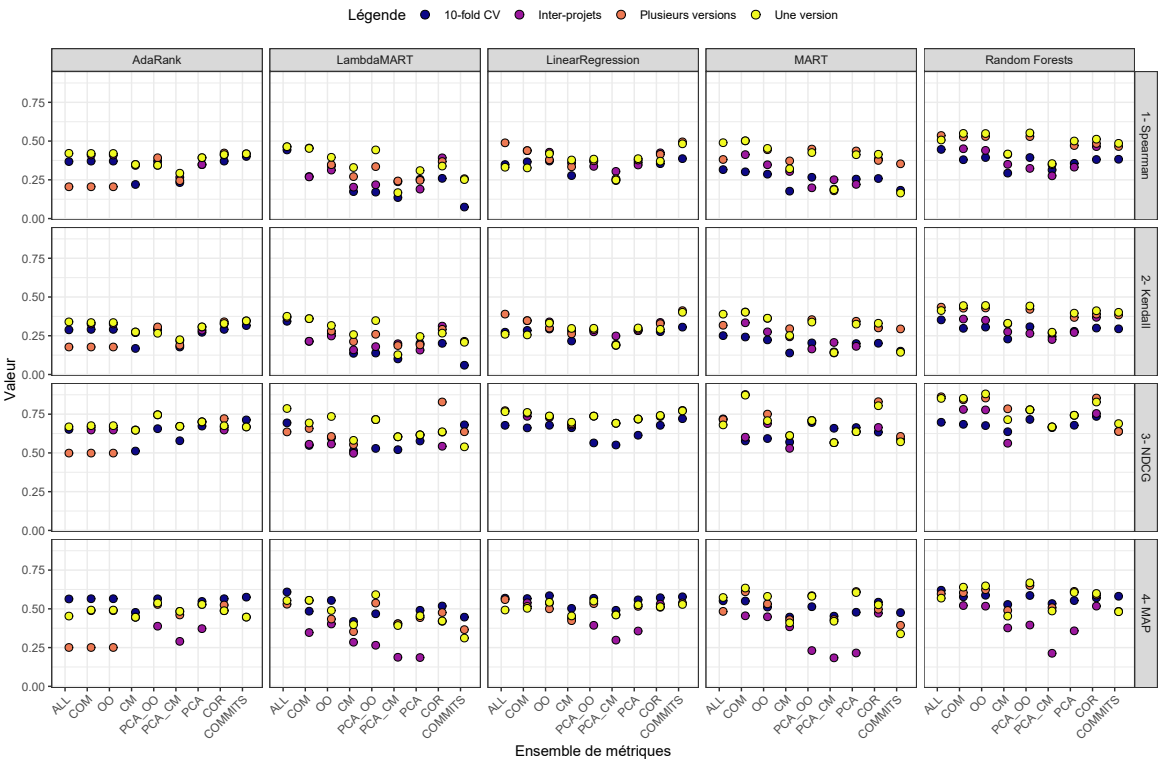
1. **Validation croisée 10 fois :** Une seule version est utilisée pour l'évaluation et l'entrainement (ANT 1.7). Les données sont divisées de manière aléatoire en 10 sous-ensembles de taille similaire, en utilisant 9 sous-ensembles pour l'entrainement et le dernier pour l'évaluation. Le processus est répété 10 fois, en prenant chaque fois un ensemble différent pour l'évaluation. Cela permet de voir dans quelle mesure nous pouvons entrainer un modèle prédictif sans collecter de données historiques.
2. **Version unique :** Utiliser une seule version (ANT 1.6) pour entrainer les modèles d'apprentissage automatique et évaluer les capacités prédictives de l'effort de test unitaire sur une autre version (ANT 1.7).

3. **Versions multiples :** Utiliser plusieurs versions (ANT 1.3 à ANT 1.6) pour entraîner les modèles d'apprentissage automatique et évaluer les capacités prédictives de l'effort de test unitaire sur une autre version (ANT 1.7). Cela permet de voir l'impact d'avoir plus de données historiques pour apprendre.
4. **Validation inter-projets :** Les modèles sont entrainés avec 7 systèmes open source (IVY, Math, JFreeChart, IO, JODA Time, Lucene, POI) et évalués sur ANT 1.7.

La figure 4.6 présente les résultats des 4 expérimentations. Plusieurs combinaisons d'algorithmes donnent d'excellentes prédictions du classement des classes selon l'effort de test requis (ex. : Random Forest atteint 0.648 de MAP et 0.881 de nDCG - Bonne classification générale et excellente capacité d'identifier les classes les plus critiques).

On note (comme pour les expérimentations sur les niveaux d'effort de test et des classes candidates) que les modèles contenant des données historiques apparaissent comme étant avantageux par rapport aux modèles inter-projets ou sans autres données (validation croisée). Les métriques de processus ne semblent cependant pas avantager ces modèles.

FIGURE 4.6 – Résultats - Classement de l'effort de test



Afin de déterminer si ces différences sont statistiquement significatives, les tests de Friedman et de Nemenyi ont été utilisés sur les valeurs des indices de performance (nDCG, MAP, Spearman et Kendall). La p-value du test de Friedman est 2.12×10^{-5} indiquant qu'il y a une différence statistiquement significative entre les méthodes d'entrainement. Le tableau 4.29 présente les résultats du test de Nemenyi pour les 4 méthodes d'entrainement et vient confirmer les observations faites précédemment : collecter des données historiques sur une version améliore dans plusieurs cas la précision des modèles de classement. La différence entre les modèles avec plusieurs versions et ceux avec seule version n'était pas statistiquement significative.

TABLE 4.29 – Test de Nemenyi - Classement de l'effort de test

	n	Somme des rangs	Moyenne des rangs	Groupes	
Une version	170	501.000	2.947	A	
Plusieurs versions	170	460.000	2.706	A	
10-fold CV	170	376.000	2.212		B
Inter-projets	170	363.000	2.135		B

En conclusion à la question Q12, l'utilisation de données historiques pour l'entrainement des algorithmes de classement de l'effort de test s'est montrée bénéfique sur notre jeu de données. Les algorithmes d'apprentissage du classement (Learning to Rank) réussissent à donner des ordonnancements précis des classes Java par rapport à leur effort de test observé. Dans les sections suivantes, nous comparerons les ensembles de métriques et les algorithmes pour vérifier leur importante respective quant à la précision des modèles de classement.

4.8.2 Q13 & Q14 : Mesures de centralité et métriques de processus

Dans cette section, nous analyserons les résultats présentés dans la section précédente pour répondre aux questions Q13 et Q14. Pour Q13, on cherche à déterminer si la combinaison des métriques orientées objet et des mesures de centralité est bénéfique dans le contexte du classement des classes Java selon l'effort de test requis. Pour Q14, on cherche à déterminer si les versions contenant des données historiques bénéficient des métriques de processus.

Le tableau 4.30 présente les résultats du test de Nemenyi des ensembles de métriques pour le classement de l'effort de test requis. On note que les groupes contenant

TABLE 4.30 – Test de Nemenyi - Ensembles de métriques - Classement de l'effort de test

	n	Somme des rangs	Moyenne des rangs	Groupes	
COMMITS	76	420.000	5.526	A	
ALL	76	415.000	5.461	A	
COM	76	413.000	5.434	A	
PCA_OO	76	400.000	5.263	A	
COR	76	396.000	5.211	A	
OO	76	381.000	5.013	A	B
PCA	76	356.000	4.684	A	B
CM	76	352.000	4.632	A	B
PCA_CM	76	287.000	3.776		B

des informations sur les modifications (ensemble COMMITS, ensemble ALL) sont en première position suivis de la combinaison des métriques orientées objet et des mesures de centralité (ensemble COM). Il est à noter que la meilleure combinaison de métriques et d'algorithmes est RandomForest et les métriques orientées objet seules (OO) avec des données historiques (1 version) (nDCG = 0.881, MAP = 0.648), mais les résultats de la moyenne des rangs montrent que ce cas particulier n'est pas représentatif de la tendance générale pour tous les classificateurs. Dans le cas où les modèles ont été entrainés avec les données d'autres logiciels open source, la combinaison des métriques OO et des mesures de centralité (ensemble COM) a mieux performé (nDCG = 0.780, MAP = 0.539) que les métriques OO seules (nDCG = 0.778, MAP = 0.537). Dans le cas de l'utilisation des données de plusieurs versions, l'avantage va à la combinaison des métriques de processus et des autres métriques logicielles (ensemble ALL) (nDCG = 0.862, MAP = 0.597) pour l'identification des classes les plus critiques.

Afin de déterminer si cette différence est statistiquement significative, les tests de Friedman et de Nemenyi ont été effectués sur les valeurs des indices de performance (nDCG, MAP, Spearman et Kendall). Le test de Friedman donne une p-value de 1.52e-6, ce qui indique qu'il y a une différence significative entre au moins une paire d'ensembles de métriques. Le test de Nemenyi présenté dans le tableau 4.30 montre qu'il y a peu de différences qui sont statistiquement significatives entre les ensembles de métriques. Ce résultat n'est pas surprenant considérant la grande variance entre la performance des ensembles de métriques d'un classificateur à un autre et d'une façon d'entrainer à une autre (données historiques, 10-fold cross-validation) et le nombre d'échantillons.

En conclusion à la question Q13, la combinaison des métriques orientées objet et

des mesures de centralité (COM) s'est globalement mieux classée que les métriques orientées objet seules (OO). Dans plusieurs cas, l'ajout des mesures de centralité aide les algorithmes d'apprentissage du classement (Learning to Rank) à générer un classement précis.

En conclusion à la question Q14, combiner les métriques logicielles classiques et les métriques de processus a été généralement bénéfique aux différents algorithmes de classement, spécifiquement lorsque les données de plusieurs versions sont utilisés. Les résultats obtenus avec une seule version étaient toutefois légèrement plus précis avec les métriques orientés seules.

4.8.3 Q15 : Algorithmes de classement

Dans cette section, on tente de répondre à la question de recherche Q15, à savoir si un algorithme d'apprentissage du classement (Learning to Rank) se distingue des autres pour la prédiction de l'effort de test. Le tableau 4.31 présente les résultats du test de Nemenyi des algorithmes à partir des résultats présentés pour la question Q12. À partir de ce tableau, on remarque que RandomForest se distingue des autres algorithmes, suivi ensuite de la régression linéaire. À partir des résultats bruts, on constate que RandomForest a produit le meilleur modèle de classement pour le cas avec une version (nDCG = 0.881, MAP = 0.648), plusieurs versions (nDCG = 0.877, MAP = 0.604) et pour le cas de l'entrainement inter-projet (nDCG = 0.780, MAP = 0.521).

Table 4.31 – Test de Nemenyi - Algorithmes de classement

	n	Somme des rangs	Moyenne des rangs	Groupes			
Random Forests	136	624.000	4.588	A			
LinearRegression	136	469.000	3.449		B		
AdaRank	136	389.000	2.860			C	
MART	136	333.000	2.449			C	
LambdaMART	136	225.000	1.654				D

Afin de déterminer si ces différences sont statistiquement significatives, les tests de Friedman et de Nemenyi ont été faits sur les valeurs des indices de performance (nDCG, MAP, Spearman et Kendall). La p-value du test de Friedman est de 2.2e-16 (significatif avec $\alpha = 0.05$). Les groupes obtenus lors du test de Nemenyi sont présentés dans la table 4.31 et montrent que toutes les différences entre les algorithmes sont statistiquement significatives, sauf la différence entre MART et AdaRank.

En conclusion à la question Q15, sur notre jeu de données, RandomForest se distingue largement des autres algorithmes pour le classement des classes Java selon l'effort de test requis. La régression linéaire a aussi donné de bons résultats, spécialement dans le cas où les modèles de classement étaient entrainés sur d'autres logiciels. Les modèles basés sur l'apprentissage du classement (Learning to Rank) apparaissent comme une solution prometteuse pour identifier les classes nécessitant un effort de test important.

Chapitre 5

Discussions et conclusions

5.1 Introduction

Dans ce chapitre, nous présentons un résumé des résultats, des réponses aux questions de recherche et une discussion dans la section 5.2. La section 5.3 présente les limites à la validité des résultats présentés. Finalement, la section 5.4 conclut le mémoire et présente des pistes de recherche intéressantes qu'il serait pertinent d'investiguer dans le futur.

5.2 Résumé des résultats et discussions

Dans cette section, nous revenons sur les résultats présentés dans le chapitre précédent et sur les questions de recherche afin d'avoir une vue d'ensemble de la problématique de l'estimation de l'effort de test requis sur les unités logicielles orientées objet.

5.2.1 Prédiction des niveaux d'effort de test unitaire

Les résultats présentés dans ce mémoire confirment que les algorithmes d'apprentissage automatique sont une approche efficace et précise pour prédire le niveau d'effort requis pour tester une classe d'un logiciel Java. Différentes métriques de code et de processus peuvent être collectées et permettent, en entrainant des modèles à partir de niveaux obtenus avec des algorithmes de partitionnement, d'obtenir un estimé précis de l'effort de test requis sur chacune des classes.

Les résultats présentés dans la section 4.6 montrent que le choix d'algorithme de partitionnement pour séparer les classes en groupes selon le niveau d'effort requis (ex. : faible, moyen, élevé) est un facteur important pour entrainer des modèles efficaces (Q1).

Les résultats indiquent qu'augmenter le niveau de granularité des résultats (ex. : de 3 à 4 niveaux, de 4 à 5) a un effet non-négligeable sur la précision des modèles prédictifs, mais que les modèles restent assez précis pour guider le processus de test unitaire et réduire les coûts et le temps associés (Q2).

Pour ce qui est de la façon de mesurer l'effort de test pour l'entrainement des modèles, nos résultats suggèrent que la taille des tests unitaires devrait être complétée par d'autres métriques. La combinaison de toutes les métriques de test (TLOC, TASSERT, TINVOK, TDATA) était significativement plus précise que TLOC et TASSERT, ou TLOC seule, indiquant que l'ajout de différents aspects des tests unitaires (taille, nombre d'assertions, nombre d'invocations de méthodes, nombre d'instanciations d'objets) permettent d'obtenir des résultats plus précis à l'aide des algorithmes d'apprentissage automatique (Q3).

L'ajout des données historiques (des versions antérieures d'un logiciel) s'est montré bénéfique pour la précision des modèles prédictifs par rapport aux modèles sans données sur les versions antérieures, mais l'ajout d'une ou de plusieurs versions n'a pas montré de différences statistiquement significatives. Les modèles entrainés sur d'autres logiciels open source restent assez précis et suggère que les modèles prédictifs des niveaux d'effort de test basés sur les métriques logicielles et de processus sont généralisables d'un projet à un autre (Q4).

La comparaison des ensembles de métriques pour cette problématique a révélé une tendance avantageuse pour la combinaison des métriques OO et des mesures de centralité (Q5). L'ajout des métriques de processus a donné des résultats similaires (certains algorithmes de classification en ont bénéficié) (Q6). La différence significative avec les métriques OO seules n'a pas pu être vérifiée statistiquement, mais la grande variance des valeurs du g-mean selon le classificateur utilisé et la taille de l'échantillon sont des explications possibles. Réduire le nombre de variables et la colinéarité des métriques avec l'analyse en composantes principales a nui aux classificateurs.

La comparaison des algorithmes de classification a montré un net avantage pour les algorithmes de RandomForest et des réseaux bayésiens. Le modèle de prédiction le plus précis a été RandomForest avec les données historiques et l'ensemble COM (OO + CM) (AUC = 0.891, G-Mean = 0.8) (Q7).

En somme, les algorithmes d'apprentissage automatique sont une solution efficace

pour la prédiction des niveaux d'effort de test unitaire requis. Les résultats présentés ici suggèrent que des outils les combinant aux métriques logicielles pourraient guider de façon précise le processus de test et identifier les classes les plus critiques à tester ; et d'ainsi potentiellement réduire les coûts et le temps requis pour le test unitaire.

5.2.2 Prédiction des classes candidates

Pour le problème de la prédiction des classes qui sont de bonnes candidates aux tests unitaires, les résultats concernant 4 questions de recherche ont été présentés. Les résultats sur notre jeu de données suggèrent qu'entrainer les modèles prédictifs avec des données historiques sur les versions précédentes d'un logiciel est bénéfique pour la prédiction des classes candidates. Les modèles entrainés sur d'autres logiciels open source était moins précis, mais réussissent quand même à guider l'effort de test efficacement (Q8). Cela suggère que, pour ce problème aussi, les modèles sont généralisables d'un projet à un autre lorsqu'ils utilisent les métriques logicielles.

Les ensembles de métriques ont été comparés dans le but de déterminer si la combinaison des métriques orientées objet et des mesures de centralité permettait aux algorithmes d'apprentissage automatique de mieux identifier les classes critiques. Comme pour la prédiction des niveaux de test, la combinaison (ensemble COM) montrait une tendance bénéfique pour plusieurs classificateurs et méthodes d'entrainement, mais le seuil pour obtenir une différence statistique était difficile à atteindre (Q9). L'ajout des métriques de processus (ensemble ALL) a cette fois dépassé les métriques tirées du code source (ensemble COM) confirmant que les classes sont souvent testées lors des modifications et des corrections de fautes dans plusieurs des projets étudiés (Q10). Les outils basés sur ces approches pourraient potentiellement aider les développeurs à identifier les parties critiques à tester, dans le cas où ils travaillent sur des projets où le test unitaire n'est pas ou peu utilisé, en collectant des données sur le nombre de modifications à partir des systèmes de contrôle de versions, mais les conclusions présentées ici pour les métriques de processus sont sujettes à un biais qui sera mentionné dans la section 5.3. La réduction de la colinéarité et du nombre de métriques n'a pas eu d'effets positifs sur la qualité des prédictions.

Les algorithmes de classifications ont aussi été comparés pour la prédiction des classes candidates. Les algorithmes RandomForest et les réseaux bayésiens se sont démarqués des autres pour cet objectif de prédiction (Q11). En conclusion, les approches basées sur l'apprentissage automatique et les métriques logicielles ont donné d'excel-

lents résultats quant à l'identification des classes qui sont de bonnes candidates à être testées.

5.2.3 Classement (Learning to Rank)

Le dernier aspect de la prédiction de l'effort de test présenté dans ce mémoire est le classement des unités logicielles (classes Java) à l'aide des algorithmes d'apprentissage du classement (Learning to Rank).

Les résultats présentés montrent que ces algorithmes réussissent à ordonner de façon précise les classes Java selon l'effort de test requis sur celles-ci. Les modèles utilisant les données de versions précédentes ont réussi à prédire plus précisément l'ordre des classes (avec une différence statistique confirmée par le test de Friedman / Nemenyi) (nDCG = 0.881, MAP = 0.648), mais les modèles basés sur d'autres logiciels ont aussi donné de bonnes prédictions (nDCG = 0.780, MAP = 0.521) (Q12).

La question Q13 cherchait à déterminer si la combinaison des métriques orientées objet et des mesures de centralité (ensemble COM) était bénéfique pour la prédiction de l'ordre des efforts de test. L'ensemble COM a donné de meilleurs résultats que les métriques OO seules dans plusieurs cas, incluant les modèles basés sur d'autres logiciels. Cependant, dans un cas précis, les métriques OO seules combinées avec l'algorithme RandomForest ont réussi à dépasser tous les autres ensembles de métriques lors de la prédiction avec les modèles entrainés sur une seule version. Le test statistique n'a pas pu confirmer qu'il y avait une différence significative entre les deux groupes (COM et OO). Pour la question Q14, les métriques de processus combinés aux autres métriques logicielles (ensemble ALL) ont mieux performé que les autres ensembles dans plusieurs cas et se sont distinguées lors de la comparaison de la moyenne des rangs.

Finalement, pour répondre à la question Q15, nous avons comparé les algorithmes de classement (Learning to Rank). RandomForest et la régression linéaire ont obtenu les meilleurs résultats sur nos jeux de données de façon statistiquement significative.

En conclusion, les algorithmes d'apprentissage du classement (Learning to Rank) sont une approche prometteuse pour l'identification des classes qui nécessitent un effort de test plus important. Les algorithmes peuvent être entrainés sur d'autres logiciels ou des versions précédentes d'un logiciel et permettent de guider l'effort de test et de réduire les coûts qui y sont associés.

5.3 Limites de validité

Dans cette section, nous présentons les limites de validité qui peuvent empêcher la généralisation des résultats présentés.

5.3.1 Externes

Pour la validité externe, la première limite de validité provient du fait que nous avons utilisé plusieurs projets provenant de la fondation Apache écrits en Java pour entrainer les modèles prédictifs. Il est possible que les algorithmes de classification et de classement performent différemment lorsqu'entrainés sur d'autres projets.

Une autre limite est que les tests unitaires sont utilisés à la discrétion des développeurs ou des chefs de projet. Les critères de sélection pour choisir les classes à tester ou non n'étaient pas présents dans notre ensemble de données. En d'autres termes, d'autres développeurs auraient pu choisir différentes classes à tester, et un ratio différent de classes testées / non testées. Le niveau d'effort de test est aussi un choix du développeur.

Plusieurs projets dans cette étude ont comme politique de tester les classes lors de la correction de fautes [7]. Les classes ayant eu des modifications ont donc par conséquent un plus grand effort de test, ce qui cause un biais avec les métriques de processus, en particulier au niveau de la prédiction des classes candidates aux tests.

5.3.2 Internes

Le choix des métriques logicielles est une autre limite à considérer. Les métriques orientées objet et les mesures de centralité ont été choisies parmi les plus utilisées dans la littérature, mais d'autres combinaisons de métriques pourraient donner des résultats différents de ceux présentés ici. Plusieurs autres métriques orientées objet, mesures de centralité et métriques de processus ont été proposées.

La transposition des modèles prédictifs d'un système à un autre est aussi limitée. Les corrélations entre les métriques de test et les métriques logicielles peuvent être élevées parmi les 8 projets sélectionnés, mais ces corrélations pourraient être plus faibles sur d'autres projets. Des classes ayant des attributs internes similaires (couplage, complexité, cohésion, etc.) pourraient nécessiter différents niveaux d'effort de test.

5.3.3 Conceptuelles

Pour la validité conceptuelle, nous avons associé les tests unitaires à leurs classes correspondantes, mais nous n'avons pas considéré les classes logicielles dont les comportements étaient testés par des dépendances transitives. Cela pourrait influencer la précision des modèles prédisant si une classe doit être testée ou non (classes candidates) et les modèles de classement (Learning to Rank).

Il faut aussi considérer la dépendance dans les tests statistiques. Une bonne partie des résultats provient de la validation sur les données d'un seul projet (avec des données historiques d'une ou de plusieurs versions). Étant donné les limites présentées précédemment, les conclusions des tests statistiques ne peuvent s'appliquer qu'aux projets présentés ici et pourraient avoir des conclusions différentes avec un plus grand échantillon de projets.

5.4 Conclusions et perspectives

En conclusion, trois approches précises pour guider l'effort de test ont été présentées dans ce mémoire. Les résultats présentés montrent que les algorithmes d'apprentissage automatique permettent de prédire l'effort de test unitaire requis sur les unités des programmes orientés objet à l'aide de différentes métriques extraites du code source (métriques orientées objet, mesures de centralité) et des métadonnées d'un projet (métriques de processus). Ils permettent donc d'identifier quelles parties d'un programme sont critiques à tester unitairement pour assurer sa qualité, minimiser les défauts et prévenir les régressions lors de la maintenance et des modifications. Ils permettraient de focaliser l'effort de test et d'ainsi réduire les coûts qui y sont associés en assurant que la couverture de test soit présente sur les parties les plus importantes.

Les résultats présentés montrent que les modèles de prédiction peuvent être entrainés avec succès à l'aide de données historiques (versions précédentes) ou de données issues d'autres projets pour prédire les niveaux d'effort de test unitaire requis (ex. : faible, moyen, élevé), suggérer les classes qui sont de bonnes candidates à être testées et classer les unités logicielles selon l'effort de test requis (Learning to Rank). L'ajout des mesures de centralité et des métriques de processus s'est montré bénéfique quant à la précision des modèles prédictifs dans plusieurs cas.

L'approche proposée pour combiner les algorithmes de classement et les métriques logicielles a permis d'obtenir des prédictions précises sur les classes les plus critiques à

tester.

Il serait intéressant de reproduire les expérimentations présentées ici sur d'autres systèmes et d'ajouter d'autres mesures pour avoir des modèles plus complets, par exemple, des données sur la couverture des tests et des données basées sur la fouille de texte (text mining) pour identifier (à partir des métadonnées des systèmes de contrôles de versions) quelles classes ont historiquement contenu des fautes. D'autres approches populaires en intelligence artificielle seraient intéressantes à explorer, comme l'apprentissage profond (Deep Learning).

Bibliographie

[1] Golnoush Abaei and Ali Selamat. A survey on software fault detection based on different prediction approaches. *Vietnam Journal of Computer Science*, 1(2) :79–95, 2014.

[2] Zakrani Abdelali, Hain Mustapha, and Namir Abdelwahed. Investigating the use of random forest in software effort estimation. *Procedia Computer Science*, 148 :343–352, 2019.

[3] Hervé Abdi and Lynne J Williams. Principal component analysis. *Wiley interdisciplinary reviews : computational statistics*, 2(4) :433–459, 2010.

[4] K K Aggarwal, Yogesh Singh, Arvinder Kaur, and Ruchika Malhotra. Empirical analysis for investigating the effect of object-oriented metrics on fault proneness : A replicated case study. *Software Process Improvement and Practice*, 14(1) :39–62, 2009.

[5] Jehad Al Dallal. Identifying refactoring opportunities in object-oriented code : A systematic literature review. *Information and Software Technology*, 58 :231–249, 2015.

[6] Sultan Aljahdali, Alaa F. Sheta, and Narayan C. Debnath. Estimating software effort and function point using regression, Support Vector Machine and Artificial Neural Networks models, 2016.

[7] Apache. On Contributing Patches, 2013.

[8] Mourad Badri and Fadel Toure. Empirical Analysis of Object-Oriented Design Metrics for Predicting Unit Testing Effort of Classes. *Journal of Software Engineering and Applications*, 05(07) :513–526, 2012.

[9] Mourad Badri, Fadel Toure, and Luc Lamontagne. Predicting unit testing effort levels of classes : An exploratory study based on Multinomial Logistic Regression modeling. *Procedia Computer Science*, 62(Scse) :529–538, 2015.

[10] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10) :751–761, 1996.

[11] Nicolas Bettenburg and Ahmed E. Hassan. Studying the impact of social interactions on software quality. *Empirical Software Engineering*, 18(2) :375–431, 2013.

[12] Alexandre Boucher and Mourad Badri. Software metrics thresholds calculation techniques to predict fault-proneness : An empirical comparison. *Information and Software Technology*, 96(October 2017) :38–67, 2018.

[13] Petrônio L. Braga, Adriano L.I. Oliveira, Gustavo H.T. Ribeiro, and Silvio R.L. Meira. Bagging predictors for estimation of software project effort, 2007.

[14] Lionel C. Briand, Walcelio L. Melo, and Jürgen Wüst. Assessing the applicability of fault-proneness models across object-oriented software projects. *IEEE Transactions on Software Engineering*, 28(7) :706–720, 2002.

[15] Magiel Bruntink and Arie Van Deursen. Predicting class testability using object-oriented metrics. *Proceedings - Fourth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 136–145, 2004.

[16] Magiel Bruntink and Arie van Deursen. An empirical study into class testability. *Journal of Systems and Software*, 79(9) :1219–1232, 2006.

[17] Christopher J C Burges. From RankNet to LambdaRank to LambdaMart : An overview. *Microsoft Research Technical Report*, 11(May) :81, 2014.

[18] M.G. Campiteli, A.J. Holanda, L.D.H. Soares, P.R.C. Soles, and O. Kinouchi. Lobby index as a network centrality measure. *Physica A : Statistical Mechanics and its Applications*, 392(21) :5511–5515, 2013.

[19] Cagatay Catal and Banu Diri. A systematic review of software fault prediction studies. *Expert Systems with Applications*, 36(4) :7346–7354, 2009.

[20] Cagatay Catal, Ugur Sevim, and Banu Diri. Practical development of an Eclipse-based software fault prediction tool using Naive Bayes algorithm. *Expert Systems with Applications*, 38(3) :2347–2353, 2011.

[21] Venkata U.B. Challagulla, Farokh B. Bastani, I. Ling Yen, and Raymond A. Paul. Empirical assessment of machine learning based software defect prediction techniques. *Proceedings - International Workshop on Object-Oriented Real-Time Dependable Systems, WORDS*, pages 263–270, 2005.

[22] Duanbing Chen, Linyuan Lü, Ming Sheng Shang, Yi Cheng Zhang, and Tao Zhou. Identifying influential nodes in complex networks. *Physica A : Statistical Mechanics and its Applications*, 391(4) :1777–1787, 2012.

[23] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. *ACM SIGPLAN Notices*, 26(11) :197–211, 1991.

[24] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6) :476–493, 1994.

[25] Ana Erika Camargo Cruz and Koichiro Ochimizu. Towards logistic regression models for predicting fault-prone code across software projects, 2009.

[26] Karel Dejaeger, Thomas Verbraken, and Bart Baesens. Toward comprehensible software fault prediction models using bayesian network classifiers. *IEEE Transactions on Software Engineering*, 39(2) :237–257, 2013.

[27] Janez Demšar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7 :1–30, 2006.

[28] R.P. Espíndola and N.F.F. Ebecken. On extending F-measure and G-mean metrics to multi-class problems. In *Data Mining VI*, volume 1, pages 25–34, may 2005.

[29] Linton C. Freeman. A Set of Measures of Centrality Based on Betweenness. *Sociometry*, 40(1) :35, 1977.

[30] Linton C. Freeman. A Set of Measures of Centrality Based on Betweenness. *Sociometry*, 40(1) :35, 1977.

[31] Linton C. Freeman. Centrality in social networks conceptual clarification. *Social Networks*, 1(3) :215–239, 1978.

[32] Nir Friedman, Dan Geiger, and Moises Goldszmidt. Bayesian Network Classifiers. *Machine Learning*, 29(2-3) :131–163, 1997.

[33] Lan Guo, Yan Ma, Bojan Cukic, and Harshinder Singh. Robust prediction of fault-proneness by random forests, 2004.

[34] Tibor Gyimóthy, Rudolf Ferenc, and István Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10) :897–910, 2005.

[35] Mark Hall. Correlation-based Feature Selection for Machine Learning. *Methodology*, 21i195-i20(April) :1–5, 1999.

[36] Trevor Hastie and Robert Tibshirani. Classification by pairwise coupling. In Michael I Jordan, Michael J Kearns, and Sara A Solla, editors, *Advances in Neural Information Processing Systems*, volume 10, pages 507–513. MIT Press, 1998.

[37] Brian Henderson-Sellers. *Object-oriented Metrics : Measures of Complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[38] Karen E. Joyce, Paul J. Laurienti, Jonathan H. Burdette, and Satoru Hayasaka. A new measure of centrality for brain networks. *PLoS ONE*, 5(8) :e12200, 2010.

[39] Henry F. Kaiser. The varimax criterion for analytic rotation in factor analysis. *Psychometrika*, 23(3) :187–200, 1958.

[40] S. Kanmani, V. Rhymend Uthariaraj, V. Sankaranarayanan, and P. Thambidurai. Object-oriented software fault prediction using neural networks. *Information and Software Technology*, 49(5) :483–492, 2007.

[41] Arvinder Kaur and Ruchika Malhotra. Application of random forest in predicting fault-prone classes. In *Proceedings - 2008 International Conference on Advanced Computer Theory and Engineering, ICACTE 2008*, pages 37–43. IEEE, 2008.

[42] Imrul Kayes, Shafinaz Islam, and Jacob Chakareski. The network of faults : a complex network approach to prioritize test cases for regression testing. *Innovations in Systems and Software Engineering*, 11(4) :261–275, 2015.

[43] S. S. Keerthi, S. K. Shevade, C. Bhattacharyya, and K. R.K. Murthy. Improvements to Platt's SMO algorithm for SVM classifier design. *Neural Computation*, 13(3) :637–649, 2001.

[44] Tien Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. A learning-to-rank based fault localization approach using likely invariants. In *ISSTA 2016 - Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 177–188, New York, NY, USA, 2016. Association for Computing Machinery.

[45] Biwen Li, Beijun Shen, Jun Wang, Yuting Chen, Tao Zhang, and Jinshuang Wang. A scenario-based approach to predicting software defects using compressed C4.5 model. In *Proceedings - International Computer Software and Applications Conference*, pages 406–415. IEEE, 2014.

[46] Chung-Yen Lin, Chia-Hao Chin, Hsin-Hung Wu, Shu-Hwa Chen, Chin-Wen Ho, and Ming-Tat Ko. Hubba : hub objects analyzer–a framework of interactome hubs identification for network biology. *Nucleic acids research*, 36(Web Server issue) :W438–43, jul 2008.

[47] Tie Yan Liu. Learning to rank for Information Retrieval. *Foundations and Trends in Information Retrieval*, 3(3) :225–231, 2009.

[48] Ruchika Malhotra and Ankita Jain Bansal. Fault prediction considering threshold effects of object-oriented metrics. *Expert Systems*, 32(2) :203–219, 2015.

[49] Ruchika Malhotra and Ankita Jain. Fault prediction using statistical and machine learning methods for improving software quality. *Journal of Information Processing Systems*, 8(2) :241–262, 2012.

[50] Robert Martin. OO Design Quality Metrics. *Quality Engineering*, 8(4) :537–542, 1996.

[51] Hong Mei, Dan Hao, Lingming Zhang, Lu Zhang, Ji Zhou, and Gregg Rothermel. A static approach to prioritizing JUnit test cases. *IEEE Transactions on Software Engineering*, 38(6) :1258–1275, 2012.

[52] Fionn Murtagh and Pedro Contreras. Algorithms for hierarchical clustering : an overview. *Wiley Interdisciplinary Reviews : Data Mining and Knowledge Discovery*, 2(1) :86–97, 2012.

[53] Donald E. Neumann. An enhanced neural network technique for software risk analysis. *IEEE Transactions on Software Engineering*, 28(9) :904–912, 2002.

[54] Alberto S. Nuñez-Varela, Héctor G. Pérez-Gonzalez, Francisco E. Martínez-Perez, and Carlos Soubervielle-Montalvo. Source code metrics : A systematic mapping study. *Journal of Systems and Software*, 128 :164–197, 2017.

[55] Ahmet Okutan and Olcay Taner Yıldız. Software defect prediction using Bayesian networks. *Empirical Software Engineering*, 19(1) :154–181, 2014.

[56] Alexandre Ouellet and Mourad Badri. Empirical Analysis of Object-Oriented Metrics and Centrality Measures for Predicting Fault-Prone Classes in Object-Oriented Software. In *Communications in Computer and Information Science*, volume 1010, pages 129–143, 2019.

[57] John Platt. Fast training of support vector machines using sequential minimal optimization. In *Advances in Kernel Methods - Support Vector Learning*. MIT Press, January 1998.

[58] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization : an empirical study. *Conference on Software Maintenance*, pages 179–188, 1999.

[59] Britta Ruhnau. Eigenvector-centrality - a node-centrality. *Social Networks*, 22(4) :357–365, 2000.

[60] A. Shanthini and R. M. Chandrasekaran. Analyzing the effect of bagged ensemble approach for software fault prediction in class level and package level metrics, 2015.

[61] Raed Shatnawi. A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems. *IEEE Transactions on Software Engineering*, 36(2) :216–225, 2010.

[62] Yogesh Singh, Arvinder Kaur, and Ruchika Malhotra. Software Fault Pronennes Prediction Using Support Vector Machines. In *Procedings of the World Congress of Engineering 2009*, volume vol.1, pages 240–245. Citeseer, 2009.

[63] Jeongju Sohn and Shin Yoo. FLUCCS : Using code and change metrics to improve fault localization. *ISSTA 2017 - Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, (July) :273–283, 2017.

[64] Ayşe Tosun, Burak Turhan, and Ayşe Bener. Validation of network measures as indicators of defective modules in software systems. *ACM International Conference Proceeding Series*, 2009.

[65] Fadel Toure, Mourad Badri, and Luc Lamontagne. A metrics suite for JUnit test code : a multiple case study on open source software. *Journal of Software Engineering Research and Development*, 2(1), 2014.

[66] Fadel Toure, Mourad Badri, and Luc Lamontagne. Investigating the prioritization of unit testing effort using software metrics. In *ENASE 2017 - Proceedings of*

the 12th International Conference on Evaluation of Novel Approaches to Software Engineering, pages 69–80, 2017.

[67] Fadel Toure, Mourad Badri, and Luc Lamontagne. Predicting different levels of the unit testing effort of classes using source code metrics : a multiple case study on open-source software. *Innovations in Systems and Software Engineering*, 14(1) :15–46, 2018.

[68] Burak Turhan and Ayse Bener. Analysis of Naive Bayes' assumptions on software fault data : An empirical study. *Data and Knowledge Engineering*, 68(2) :278–290, 2009.

[69] Hamed Valizadegan, Rong Jin, Ruofei Zhang, and Jianchang Mao. Learning to rank by optimizing NDCG measure. *Advances in Neural Information Processing Systems 22 - Proceedings of the 2009 Conference*, pages 1883–1891, 2009.

[70] Jun Wang, Beijun Shen, and Yuting Chen. Compressed C4.5 models for software defect prediction. *Proceedings - International Conference on Quality Software*, 2(1) :13–16, 2012.

[71] Tao Wang and Wei Hua Li. Naïve bayes software defect prediction model. *2010 International Conference on Computational Intelligence and Software Engineering, CiSE 2010*, pages 0–3, 2010.

[72] Yining Wang, Liwei Wang, Yuanzhi Li, Di He, Wei Chen, and Tie Yan Liu. A theoretical analysis of NDCG ranking measures. *Journal of Machine Learning Research*, 30 :25–54, 2013.

[73] Ian H. Witten, Eibe Frank, and Mark A. Hall. Introduction to Weka, 2011.

[74] Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3) :37–52, 1987.

[75] Qiang Wu, Christopher J C Burges, Krysta M Svore, and Jianfeng Gao. Ranking, Boosting, and Model Adaptation. *Microsoft Research Technical Report*, pages 1–23, 2008.

[76] Fei Xing, Ping Guo, and Michael R. Lyu. A novel method for early software quality prediction based on support vector machine. *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, 2005 :213–222, 2005.

[77] Jun Xu and Hang Li. AdaRank : A boosting algorithm for information retrieval. *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR'07*, (49) :391–398, 2007.

[78] Xiaoxing Yang, Ke Tang, and Xin Yao. A learning-to-rank approach to software defect prediction. *IEEE Transactions on Reliability*, 64(1) :234–246, 2015.

[79] Xiao Yu, Kwabena Ebo Bennin, Jin Liu, Jacky Wai Keung, Xiaofei Yin, and Zhou Xu. An Empirical Study of Learning to Rank Techniques for Effort-Aware Defect Prediction. *SANER 2019 - Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution, and Reengineering*, pages 298–309, 2019.

[80] Yuen Tak Yu and Man Fai Lau. Fault-based test suite prioritization for specification-based testing. *Information and Software Technology*, 54(2) :179–202, 2012.

[81] Yue Zhou and Jinyao Yan. A logistic regression based approach for software test management, 2017.

[82] Yuming Zhou and Hareton Leung. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Transactions on Software Engineering*, 32(10) :771–789, 2006.

[83] Ling Zan Zhu, Bei Bei Yin, and Kai Yuan Cai. Software fault localization based on centrality measures, 2011.

[84] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. *Proceedings - International Conference on Software Engineering*, pages 531–540, 2008.

Annexe A : Matrices de corrélation (ANT 1.3 à 1.7)

Table 1 – Métriques orientées objet

	WMC	RFC	CBO	LCOM3	DIT	CA	CE	NOC	LOC
WMC	1.000	0.902	0.667	0.148	-0.040	0.409	0.615	0.143	0.861
RFC	0.902	1.000	0.667	0.223	0.104	0.248	0.754	0.080	0.949
CBO	0.667	0.667	1.000	0.225	0.139	0.595	0.712	0.302	0.594
LCOM3	0.148	0.223	0.225	1.000	0.085	0.087	0.213	0.047	0.178
DIT	-0.040	0.104	0.139	0.085	1.000	-0.244	0.412	0.035	0.074
CA	0.409	0.248	0.595	0.087	-0.244	1.000	0.017	0.314	0.256
CE	0.615	0.754	0.712	0.213	0.412	0.017	1.000	0.072	0.661
NOC	0.143	0.080	0.302	0.047	0.035	0.314	0.072	1.000	0.081
LOC	0.861	0.949	0.594	0.178	0.074	0.256	0.661	0.081	1.000
EV	0.502	0.597	0.740	0.143	0.235	0.171	0.732	0.179	0.536
SLC	0.435	0.495	0.634	0.227	0.284	0.115	0.699	0.085	0.402
LE	0.562	0.470	0.704	0.157	-0.234	0.703	0.310	0.352	0.428
LO	0.635	0.649	0.972	0.220	0.185	0.543	0.719	0.259	0.575
BC	0.525	0.462	0.691	0.147	0.046	0.780	0.393	0.283	0.430
CL	0.424	0.520	0.576	0.255	0.474	-0.010	0.800	0.080	0.422
DMNC	0.445	0.536	0.622	0.232	0.156	0.128	0.697	0.041	0.457
DC	0.675	0.675	0.997	0.228	0.133	0.604	0.716	0.293	0.603
TLOC	0.476	0.421	0.423	0.045	-0.036	0.439	0.277	0.076	0.437
TASSERT	0.328	0.223	0.245	0.057	-0.261	0.525	-0.013	0.038	0.255
TINVOK	0.428	0.363	0.369	0.075	-0.112	0.499	0.166	0.061	0.383
TDATA	0.376	0.284	0.291	-0.134	-0.225	0.490	0.042	0.140	0.333
numCommits	0.544	0.580	0.505	0.332	0.154	0.183	0.539	0.125	0.534
numWords	0.546	0.592	0.522	0.361	0.146	0.169	0.567	0.111	0.550
numAuthors	0.468	0.493	0.459	0.311	0.217	0.130	0.507	0.081	0.452
insertions	0.689	0.698	0.512	0.230	0.117	0.280	0.556	0.068	0.688
deletions	0.547	0.593	0.499	0.342	0.213	0.178	0.564	0.107	0.559

Table 2 – Mesures de centralité

	EV	SLC	LE	LO	BC	CL	DMNC	DC
WMC	0.502	0.435	0.562	0.635	0.525	0.424	0.445	0.675
RFC	0.597	0.495	0.470	0.649	0.462	0.520	0.536	0.675
CBO	0.740	0.634	0.704	0.972	0.691	0.576	0.622	0.997
LCOM3	0.143	0.227	0.157	0.220	0.147	0.255	0.232	0.228
DIT	0.235	0.284	-0.234	0.185	0.046	0.474	0.156	0.133
CA	0.171	0.115	0.703	0.543	0.780	-0.010	0.128	0.604
CE	0.732	0.699	0.310	0.719	0.393	0.800	0.697	0.716
NOC	0.179	0.085	0.352	0.259	0.283	0.080	0.041	0.293
LOC	0.536	0.402	0.428	0.575	0.430	0.422	0.457	0.603
EV	1.000	0.551	0.260	0.769	0.329	0.510	0.713	0.736
SLC	0.551	1.000	0.148	0.660	0.368	0.894	0.670	0.634
LE	0.260	0.148	1.000	0.608	0.646	0.109	0.239	0.715
LO	0.769	0.660	0.608	1.000	0.651	0.608	0.640	0.964
BC	0.329	0.368	0.646	0.651	1.000	0.341	0.350	0.706
CL	0.510	0.894	0.109	0.608	0.341	1.000	0.654	0.577
DMNC	0.713	0.670	0.239	0.640	0.350	0.654	1.000	0.632
DC	0.736	0.634	0.715	0.964	0.706	0.577	0.632	1.000
TLOC	0.195	0.239	0.414	0.403	0.425	0.192	0.224	0.433
TASSERT	-0.038	0.015	0.380	0.234	0.375	-0.037	0.038	0.247
TINVOK	0.108	0.159	0.408	0.353	0.433	0.110	0.159	0.375
TDATA	0.058	-0.014	0.435	0.275	0.367	-0.079	0.004	0.294
numCommits	0.282	0.682	0.318	0.488	0.370	0.656	0.441	0.509
numWords	0.342	0.585	0.364	0.502	0.326	0.580	0.432	0.525
numAuthors	0.256	0.695	0.234	0.456	0.339	0.664	0.414	0.463
insertions	0.193	0.539	0.434	0.484	0.471	0.564	0.385	0.522
deletions	0.282	0.646	0.319	0.474	0.399	0.657	0.444	0.506

TABLE 3 – Métriques de test

	TLOC	TASSERT	TINVOK	TDATA
WMC	0.476	0.328	0.428	0.376
RFC	0.421	0.223	0.363	0.284
CBO	0.423	0.245	0.369	0.291
LCOM3	0.045	0.057	0.075	-0.134
DIT	-0.036	-0.261	-0.112	-0.225
CA	0.439	0.525	0.499	0.490
CE	0.277	-0.013	0.166	0.042
NOC	0.076	0.038	0.061	0.140
LOC	0.437	0.255	0.383	0.333
EV	0.195	-0.038	0.108	0.058
SLC	0.239	0.015	0.159	-0.014
LE	0.414	0.380	0.408	0.435
LO	0.403	0.234	0.353	0.275
BC	0.425	0.375	0.433	0.367
CL	0.192	-0.037	0.110	-0.079
DMNC	0.224	0.038	0.159	0.004
DC	0.433	0.247	0.375	0.294
TLOC	1.000	0.704	0.935	0.699
TASSERT	0.704	1.000	0.863	0.575
TINVOK	0.935	0.863	1.000	0.685
TDATA	0.699	0.575	0.685	1.000
numCommits	0.314	0.184	0.291	0.114
numWords	0.315	0.204	0.297	0.153
numAuthors	0.302	0.154	0.266	0.111
insertions	0.466	0.326	0.451	0.268
deletions	0.309	0.134	0.268	0.087