TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ABRÉVIATIONS ET DES SIGLES

ANN *Artificial Neural Network* (réseau de neurones artificiel)

CBO *Coupling Between Objects* (couplage entre les objets)

CC *Cylomatic Complexity* (complexité cyclomatique)

HySOM *Hybrid Self-Organizing Map* (carte auto adaptative hybride)

LOC *Lines of Code* (nombre de lignes de code)

MRL *Multiple Risk Levels* (niveaux de risque multiples)

RFC *Response For a Class* (réponse pour une classe)

SLOC *Source Lines of Code* (nombre de lignes de code source)

SOM *Self-Organizing Map* (carte auto adaptative)

TOp *Total Operators* (nombre d'opérateurs total)

TOpnd *Total Operands* (nombre d'opérandes total)

UOp *Unique Operators* (nombre d'opérateurs uniques)

UOpnd *Unique Operands* (nombre d'opérandes uniques)

WMC *Weighted Methods per Class* (nombre de méthodes pondérées par classe)

CHAPITRE 1

INTRODUCTION

De nos jours, beaucoup de systèmes logiciels orientés-objet existent et sont développés chaque jour. La programmation orientée-objet a remplacé l'utilisation de la programmation fonctionnelle afin de produire un code source mieux structuré. Cependant, cela ne garantit pas la qualité d'un logiciel, loin de là.

Ces programmes doivent être testés comme tous les autres et doivent suivre certains critères de qualité. Avec la forte compétition des différentes entreprises au niveau du logiciel, la qualité du logiciel est un aspect très important de nos jours. Certains de ces systèmes peuvent même être critiques, c'est-à-dire que s'ils devaient connaître un dysfonctionnement, cela pourrait entraîner d'importantes pertes, matérielles ou humaines. Il est donc crucial que ces programmes soient d'une qualité très élevée et minimisent la probabilité de voir survenir des bogues lors de leur utilisation.

Cependant, afin d'être certain qu'un logiciel ne contienne aucun bogue, il faut tester le système en entier. Cela peut être réalisé pour des systèmes de très petite taille. Cependant, pour la plupart des systèmes, l'explosion combinatoire résultante du nombre de chemins d'exécution possibles du programme rend les tests exhaustifs impossibles. C'est pourquoi les équipes de développement vont souvent concentrer leurs efforts de test sur les parties les plus critiques du système. Néanmoins, le travail d'analyse nécessaire pour connaître les parties les plus critiques du programme peut s'avérer difficile, long et donc très coûteux.

C'est pour accélérer ce travail d'analyse ardu que certains chercheurs développent des modèles de prédiction des fautes. Ces modèles tentent de prédire à partir d'informations diverses les endroits dans le code source du système où se produiront les fautes. Ces informations doivent aider les développeurs et testeurs à déterminer quelles sont les parties les plus critiques du système.

Dans ce chapitre d'introduction, la problématique que la prédiction de fautes tente de résoudre et l'approche utilisée pour y arriver sont présentées. La structure du mémoire sera également présentée à la fin du présent chapitre.

1.1 Problématique

La plupart des modèles de prédiction des fautes utilisent des approches dites supervisées, c'est-à-dire que le modèle est construit à partir d'informations sur les fautes déjà détectées dans le système. Par contre, ces informations ne sont pas toujours disponibles, comme lorsqu'un nouveau développement est commencé ou dans le cas d'un système légataire où aucune donnée sur les fautes n'a jamais été conservée (Catal, 2014). Dans ces cas, les modèles supervisés peuvent être utilisés en utilisant des données d'autres systèmes. On essaie évidemment de choisir des données sur des systèmes semblables à celui à tester (même langage de programmation, même équipe de développement, etc.). Cependant, les analyses à faire pour déterminer les données d'entraînement du modèle peuvent être longues et difficiles. Cela rend ces modèles moins accessibles et limite leur utilisation.

Bien sûr, certains chercheurs ont proposé des modèles semi-supervisés et non-supervisés pour résoudre ce genre de problématique. Les modèles semi-supervisés tentent de prédire la localisation des fautes dans le code source, mais à partir d'une quantité de données très limitée sur les fautes du système (Catal, 2014). Pour ce qui est des modèles non-supervisés, ceux-ci tentent de résoudre la même problématique sans aucune utilisation des données sur les fautes pour construire le modèle de prédiction, les rendant donc utilisables en tout temps (Catal, Sevim, & Diri, 2009a).

Cependant, malgré toutes ces propositions offertes dans la littérature, aucune méthode de prédiction des fautes n'est largement utilisée. Aussi, ces modèles ne sont pas toujours simples à mettre en oeuvre et à interpréter pour leurs utilisateurs. Ils peuvent également nécessiter un lourd travail pour être construits, ce qui les rend difficilement utilisables

dans des environnements de développement itératifs. Étant donné que de nos jours, le développement itératif (agile) prend de l'ampleur, le temps de construction important du modèle peut être un important frein pour certains. Aussi, la plupart de ces modèles ne donnent qu'une indication à savoir si une partie du code est critique (risque de contenir des fautes) ou non, ce qui laisse le soin à l'utilisateur du modèle de décider quelles classes critiques il doit d'abord tester. L'utilisateur n'a donc qu'une information très limitée pour diriger son effort de test. Nous pensons que l'adaptation d'un modèle de prédiction des fautes existant ou encore que la proposition d'un nouveau modèle pourrait corriger ces défauts freinant l'utilisation de ce type de modèle. C'est d'ailleurs ce qui a motivé la recherche réalisée dans le cadre de ce mémoire.

1.2 Approche

Dans le présent mémoire, nous tentons donc de résoudre cette problématique en proposant un modèle de prédiction des fautes validé à l'aide d'une analyse empirique. Cette analyse est composée de plusieurs expérimentations, dans le but de résoudre cette problématique ou encore de renforcer la pertinence de notre modèle.

Tout d'abord, notre étude se concentre sur les modèles non-supervisés utilisant les valeurs seuils de métriques de code pour déterminer si une classe est potentiellement fautive ou non. Ces modèles simples sont faciles à mettre en place et à comprendre. Cependant, le calcul des valeurs seuils peut causer des difficultés. C'est pourquoi trois techniques de calcul des valeurs seuils sont d'abord analysées dans le cadre de la prédiction de fautes.

Ensuite, le modèle de prédiction des fautes non-supervisé HySOM (Abaei, Selamat, & Fujita, 2014) est présenté et analysé. Deux algorithmes distincts composent ce modèle : SOM (*Self-Organizing Map*) et ANN (*Artificial Neural Network* ou réseau de neurones artificiels). HySOM se base sur la valeur des métriques de code au niveau des fonctions et leurs valeurs seuils afin de prédire si une classe est fautive ou non. Ce modèle com-

plexe est étudié en premier pour tenter d'en améliorer la performance de prédiction et de l'adapter à nos besoins. Cependant, après avoir testé le modèle original et l'avoir adapté pour une utilisation au niveau des classes, la performance de prédiction du modèle n'était pas suffisante pour répondre à nos attentes. Celui-ci ne marquait pas comme étant fautives beaucoup de parties du code qui auraient dû l'être.

Suite à ce constat, nous avons travaillé sur le développement d'un nouveau modèle. Nous proposons donc un nouveau modèle de prédiction des fautes non-supervisé, le modèle MRL (*Multiple Risk Levels*). Malgré tout, nous réutilisons une partie du travail que nous avons effectué sur HySOM dans ce nouveau modèle. Plus particulièrement, nous réutilisons le calcul des valeurs seuils fait dans l'adaptation de HySOM pour fonctionner à une granularité niveau classe plutôt qu'à une granularité fonction. Ce nouveau modèle a été testé sur 12 jeux de données publics souvent utilisés lors d'études sur la prédiction des fautes. L'utilisation de notre approche présente de nombreux avantages qui sont décrits tout au long du présent mémoire et sont ici résumés :

- Le modèle MRL est complètement non-supervisé. C'est-à-dire qu'aucune donnée sur les fautes n'est utilisée dans la construction du modèle. Même les valeurs seuils des métriques de code, éléments sur lesquels repose cette approche, sont calculées sans les données sur les fautes. Cette caractéristique importante du modèle le rend utilisable dans n'importe quel projet de développement. De plus, il peut être utilisé dans n'importe quel processus de développement, que l'on soit dans un processus de développement en cascade ou encore dans un développement de type itératif (agile). La rapidité de mise en place et d'exécution de notre modèle fait de lui un candidat idéal pour une utilisation dans un processus de développement itératif.
- Le modèle MRL donne à son utilisateur une bonne indication sur l'endroit où investir son effort d'implémentation des tests en catégorisant les classes dans cinq niveaux de risque. De plus, il peut également donner à l'utilisateur une bonne idée de la rai-

son pour laquelle certaines classes sont plus critiques que d'autres. Par exemple, une classe peut être considérée comme ayant un risque très élevé de contenir des fautes étant donné qu'elle a une complexité et un couplage trop importants. Notre approche est donc simple d'utilisation et de compréhension, aspect très important pour qu'elle soit utilisée.

- Une analyse du lien entre le niveau de risque de contenir des fautes donné par le modèle MRL et la sévérité des fautes a également été réalisée. Les résultats montrent que les niveaux de risque plus élevés ont plus de chance de cerner des fautes critiques. Cela démontre l'importance de bien tester les classes catégorisées dans les niveaux de risque plus élevés de notre modèle.
- La performance du modèle MRL a été comparée à deux algorithmes de prédiction des fautes supervisés (Bayes Network et ANN). Les résultats démontrent que notre modèle donne une performance de prédiction semblable et parfois même meilleure que ces modèles supervisés. De plus, il nécessite un travail d'analyse moindre, en évitant le travail ardu requis pour obtenir des données de qualité sur les fautes (Lu, Cukic, & Culp, 2012). De plus, la performance du modèle MRL s'avère meilleure que celle donnée par le modèle initialement envisagé, HySOM.

1.3 Organisation du mémoire

Ce mémoire est divisé en 8 chapitres. Le premier et présent chapitre a introduit le sujet de la prédiction de fautes et présente ce qui a été réalisé dans le cadre de la maîtrise.

Ensuite, le second chapitre présente les différents modèles de prédiction de fautes existants. Les modèles supervisés, semi-supervisés et non-supervisés sont abordés, afin de bien cerner la problématique et comprendre le fonctionnement de chacun d'eux. Cependant, un accent particulier sera mis sur les approches non-supervisées utilisant les métriques de code, étant donné que le mémoire porte principalement sur celles-ci.

Le troisième chapitre présente la prédiction de fautes basée sur les valeurs seuils. Trois méthodes de calcul des valeurs seuils seront présentées afin d'introduire le contenu du chapitre suivant.

Le quatrième chapitre traite des deux articles rédigés dans le cadre de la maîtrise portant sur l'analyse des différentes méthodes de calcul des valeurs seuils (voir annexes A et C). Une présentation des objectifs des articles, de la méthodologie suivie et des résultats obtenus est effectuée dans ce chapitre.

Le cinquième chapitre porte sur le modèle de prédiction de fautes non-supervisé HySOM (Abaei et al., 2014). Ce chapitre montre le fonctionnement original du modèle afin d'introduire le chapitre suivant.

Le sixième chapitre présente l'adaptation faite du modèle HySOM pour une prédiction des fautes au niveau des classes d'un système logiciel. Cette adaptation a également été réalisée dans un article rédigé dans le cadre de la maîtrise (voir annexe B).

Le septième chapitre présente le modèle MRL proposé en présentant l'article rédigé à ce sujet dans le cadre de la maîtrise (voir annexe D). De façon semblable aux autres articles présentés, les objectifs, la méthodologie suivie et les résultats obtenus de cet article seront présentés.

Le huitième et dernier chapitre conclura le présent mémoire et résumera son contenu et ses apports. Il présentera également les possibilités de travaux futurs.

CHAPITRE 2

LES MODÈLES DE PRÉDICTION DE FAUTES

La prédiction de fautes est un sujet important dans le domaine d'étude du génie logiciel. L'importance de ce sujet d'étude réside dans l'aide qu'il peut apporter aux développeurs et testeurs de systèmes logiciels, en identifiant les parties du code source à tester en priorité. L'utilisation de modèles de prédiction de fautes peut épargner beaucoup de temps aux développeurs et testeurs. Par conséquent, cela entraîne une réduction des coûts en temps et en argent pour l'entreprise. Avec ces avantages marqués, on se demande pourquoi toutes les entreprises n'utilisent pas les modèles de prédiction de fautes. En fait, la réponse est simple : tous ces modèles ne sont pas simples à mettre en place et chacun présente ses avantages et ses inconvénients.

Pour bien comprendre en quoi consistent les modèles de prédiction de fautes, il est important de bien comprendre ce qu'est une faute. Selon Avizienis, Laprie, et Randell (2001), une faute est la cause d'une erreur dans un système logiciel. Une erreur (ou bogue) se produit lorsqu'une partie du système est dans un état invalide, ce qui à son tour peut entraîner une altération du service fourni par le système. Une faute n'est donc pas toujours facilement détectable. Tant qu'une faute ne provoque pas d'erreur, on dit qu'elle est dormante, sinon on la considère comme active. Souvent, les fautes sont détectées dans un système logiciel via la manifestation d'erreurs (ou bogues). Les modèles de prédiction de fautes tentent donc de prédire et détecter où se situent les fautes dans le système, afin de prévenir l'apparition d'erreurs.

Avant de comprendre le fonctionnement interne des modèles de prédiction de fautes, il faut comprendre qu'ils sont divisés en trois grandes catégories : les approches supervisées, semi-supervisées et non-supervisées. Chaque type d'approche présente des avantages et des inconvénients qui lui sont propres. Dans la suite du présent chapitre, chaque catégorie sera expliquée et détaillée, tout en insistant sur les approches non-supervisées, plus direc-

tement liées au travail réalisé. La méthode d'évaluation de la performance des modèles de prédiction de fautes sera également présentée dans le présent chapitre.

2.1 La prédiction de fautes supervisée

À la base, la prédiction de fautes supervisée est très simple et semble très prometteuse. Cependant, elle a aussi ses inconvénients pouvant être des freins majeurs à son adoption. Les approches supervisées utilisent l'historique des fautes détectées dans un ou plusieurs systèmes logiciels afin d'entraîner un modèle à reconnaître les fautes selon certaines caractéristiques prédéfinies. Par exemple, les algorithmes d'apprentissage automatique et modèles statistiques sont souvent utilisés pour réaliser ce travail. Les caractéristiques utilisées pour cibler les fautes dans le système logiciel sont très variées : métriques du code source (Catal, Diri, & Ozumut, 2007 ; Hong, 2012 ; Malhotra, 2012 ; Gondra, 2008), métriques d'inspection du code (Cong Jin & Jing-Lei Guo, 2013), détection d'anti-patrons de conception (Jaafar, Gueheneuc, Hamel, & Khomh, 2013), etc. Par contre, la plupart du temps, les métriques de code sont utilisées pour leur simplicité de compréhension et de calcul, en plus de leur lien déjà validé avec la prédisposition aux fautes (Isong & Obeten, 2013).

Ces caractéristiques du logiciel sont données en entrée au modèle de prédiction, qui est d'abord entraîné avec les données sur les fautes afin de détecter les relations avec celles-ci. Une fois entraîné, le modèle est appliqué sur le système à analyser pour extraire les parties du logiciel les plus à risque de contenir des fautes.

Malgré leur simplicité, les modèles de prédiction de fautes présentent un désavantage majeur, soit le besoin de données sur les fautes du système pour fonctionner. Effectivement, ces données ne sont pas toujours disponibles, par exemple pour un nouveau développement logiciel ou encore pour un système légataire dont les données sur les fautes n'ont pas été répertoriées de façon standardisée (Catal, Sevim, & Diri, 2009b). De plus, acquérir

et conserver des données de qualité sur les fautes d'un logiciel peut s'avérer coûteux, très long et nécessiter certains experts (Abaei et al., 2014 ; Lu et al., 2012). Le besoin important de données sur les fautes fait en sorte que les approches supervisées sont difficiles à mettre en oeuvre. C'est pourquoi des approches semi-supervisées et non-supervisées ont également été proposées dans la littérature.

2.2 La prédiction de fautes semi-supervisée

Étant donné qu'il est difficile de recueillir et conserver un historique de toutes les fautes d'un système logiciel, les approches semi-supervisées ont été pensées afin de fonctionner avec quelques données sur les fautes seulement. Elles fonctionnent donc lorsqu'on détient seulement une quantité limitée de données sur les fautes d'un système. Cela les rend plus facilement utilisables que les approches supervisées. Mis à part qu'elles utilisent une quantité limitée de données afin de parvenir à faire de la prédiction de fautes, ces approches fonctionnent de façon très semblable aux approches supervisées. C'est-à-dire qu'un entraînement du modèle est fait à l'aide de caractéristiques du logiciel et des données sur les fautes afin d'avoir un modèle utilisable. Voyons maintenant quelques exemples d'approches semi-supervisées présentées dans la littérature.

Dans deux études produites par Lu et al. (2012) ; Lu, Cukic, et Culp (2014), un modèle utilisant l'algorithme d'apprentissage automatique *Random Forest* et des techniques de réduction de dimensions des données a été étudié pour faire de la prédiction de fautes semi-supervisée. Les auteurs ont découvert que réduire le nombre de dimensions des données permettait d'améliorer de façon significative la prédiction faite par le modèle semi-supervisé. Ils ont également découvert que l'algorithme *Random Forest* donnait une meilleure prédiction lorsqu'il était utilisé de façon semi-supervisée (avec leur approche) que de façon supervisée. Ils ont également comparé leurs résultats avec ceux d'autres études sur les modèles de prédiction de fautes semi-supervisés, montrant que leur approche donnait de meilleurs résultats.

Aussi, Catal (2014) a étudié plusieurs algorithmes semi-supervisés de prédiction de fautes. Les approches comparées, au nombre de quatre, sont : *Low-Density Separation*, *Support Vector Machine*, *Expectation-Maximization* et *Class Mass Normalization*. Suite à son analyse, Catal a conclu que le modèle *Low-Density Normalization* donnait les meilleurs résultats. Il pouvait être utilisé autant sur des jeux de données de grande taille que de petite taille.

Malgré le fait que les approches semi-supervisées utilisent des algorithmes d'apprentissage supervisés dans la construction de leur modèle, ces algorithmes ont été conçus afin de faire de la classification de données non-balancées (Lu et al., 2014). Cela signifie que la prédiction de fautes n'est pas une prédiction (ou classification) dite balancée, c'est-à-dire qui prédit environ 50% du système comme non fautif et l'autre 50% comme fautif. En fait, la plupart du temps, un logiciel contiendra plus de code source non fautif que de code source fautif. Cela fait en sorte que les algorithmes d'apprentissage semi-supervisés, faits pour fonctionner avec ce genre de données, parviennent très bien à prédire les fautes malgré la quantité restreinte de données sur celles-ci.

Malgré leurs avantages, les approches semi-supervisées nécessitent tout de même un minimum de données sur les fautes. Par contre, ces données ne sont pas toujours disponibles, ce qui peut parfois occasionner un frein à l'utilisation de ces approches. C'est pourquoi des approches non-supervisées ont également été proposées, afin de remédier à ce problème.

2.3 La prédiction de fautes non-supervisée

Le désavantage majeur des approches supervisées et semi-supervisées est qu'elles nécessitent des données sur les fautes afin de construire le modèle de prédiction. Cependant, les approches non-supervisées ont été conçues afin de remédier à ce problème, en n'utilisant aucune donnée antérieure sur les fautes pour prédire où celles-ci se trouvent dans le

logiciel. Étant donné que les modèles présentés dans ce mémoire sont non-supervisés, une emphase particulière est mise sur ce type de modèle.

Catal et al. (2009a) ont déjà présenté un modèle de prédiction de fautes non-supervisé qui utilise différentes métriques de code et des valeurs seuils pour déterminer si une classe est potentiellement fautive ou non. Les valeurs seuils ont été calculées à partir de l'outil PREDICTIVE, qui n'est plus disponible à l'heure actuelle (Catal et al., 2009a). Les auteurs ont tenté de prédire si les fonctions du code source étaient fautives ou non pour trois systèmes différents écrits en C : AR3, AR4 et AR5. Leur étude portait sur deux expérimentations différentes. La première expérimentation prenait les métriques de code de chaque fonction et considérait la fonction comme potentiellement fautive si au moins une métrique dépassait sa valeur seuil correspondante. La seconde expérimentation regroupait les fonctions présentant des métriques semblables en utilisant l'algorithme des K-moyennes. Une fois les groupes formés, les métriques centroïdes de chaque groupe étaient comparées aux valeurs seuils. Un peu comme la première expérimentation, dès qu'une métrique dépassait sa valeur seuil, les fonctions du groupe étaient marquées comme potentiellement fautives. En plus de ces deux expérimentations, Catal et al. ont comparé leurs analyses à celle d'un modèle supervisé basé sur un algorithme de réseaux bayésiens naïfs. Ils ont conclu que leur approche était plus facilement automatisable que l'approche supervisée, tout en donnant une performance de prédiction acceptable.

Dans une étude ultérieure, les mêmes auteurs ont fait une nouvelle expérimentation semblable aux précédentes, mais en utilisant l'algorithme des X-moyennes (plutôt que les K-moyennes) pour regrouper les fonctions semblables (Catal, Sevim, & Diri, 2010). L'algorithme des X-moyennes a l'avantage d'avoir un nombre de regroupements qui n'est pas fixé d'avance, contrairement à l'algorithme des K-moyennes. Cela fait en sorte que l'approche proposée est encore plus simple à automatiser, tout en conservant une performance de prédiction acceptable.

Bishnu et Bhattacherjee (2012) ont utilisé une approche semblable à Catal et al. (2009a, 2010) pour construire un modèle de prédiction non-supervisé. Leur modèle utilise également l'algorithme des K-moyennes, mais les centroïdes de ce dernier sont initialisés en utilisant l'algorithme *Quad-Tree* avec un algorithme génétique. Cela permettrait d'avoir des centroïdes mieux initialisés qu'en utilisant l'algorithme des K-moyennes directement. De plus, selon les auteurs, la performance du modèle suggéré est comparable à celle de modèles de prédiction de fautes supervisés.

Dans une autre étude, cette fois réalisée par Abaei, Rezaei, et Selamat (2013), l'algorithme SOM (*Self-Organizing Map*) a été utilisé, de façon semblable aux algorithmes des K-moyennes et X-moyennes, afin de regrouper les fonctions présentant des métriques de code semblables. Les mêmes valeurs seuils de métriques de code que les études présentées précédemment ont été utilisées pour déterminer si les fonctions d'un groupe étaient potentiellement fautives ou non. Selon les auteurs, l'algorithme SOM offre une meilleure performance de prédiction que l'algorithme des K-moyennes, tout en étant moins propice à trouver un optimum local. De plus, le nombre de neurones (ou groupes) peut être dynamiquement calculé (à l'aide d'une fonction déterminée) pour l'algorithme SOM, contrairement à l'algorithme des K-moyennes. Les résultats présentés dans cette étude sont bons et même meilleurs que ceux présentés dans les études de Catal et al. (2009a, 2010) ou encore de Bishnu et Bhattacherjee (2012).

Cependant, Abaei et al. (2014) ont présenté le modèle HySOM dans une étude ultérieure, donnant de meilleurs résultats encore que leur étude précédente. Ce modèle hybride utilise encore l'algorithme SOM, mais en conjonction avec un réseau de neurones artificiels pour déterminer la prédisposition aux fautes des fonctions du code source. Les données de sortie du SOM sont comparées à des valeurs seuils et ensuite passées au réseau de neurones pour procéder à l'entraînement de ce dernier. Dans l'étude Abaei et al. (2014), il est dit que c'est un modèle semi-supervisé, car un algorithme supervisé (le réseau de

neurones) est utilisé. Cependant, suite à d'autres lectures sur les modèles semi-supervisés et comme présentés dans la section 2.2, les modèles semi-supervisés utilisent des données limitées sur les fautes pour être construits, tandis que le modèle HySOM n'en utilise pas. C'est pourquoi il devrait plutôt être considéré comme un modèle non-supervisé. Le modèle HySOM est plus détaillé et expliqué dans le chapitre 5 du présent mémoire.

Une autre étude récente faite par Erturk et Akcapinar Sezer (2016) propose l'utilisation d'un modèle de prédiction non-supervisé, utilisé avec un modèle supervisé lorsque des données sur les fautes d'un système sont disponibles. Pour arriver à ce modèle, les auteurs ont utilisé les systèmes à inférence floue (*Fuzzy Inference Systems*) et l'expertise d'un expert pour prédire la prédisposition aux fautes des classes de systèmes orientés-objet. Lorsque des données sur les fautes étaient disponibles pour les versions précédentes du système pour lequel les fautes étaient prédites, un réseau de neurones artificiels ainsi qu'un système à inférence neuro-floue adaptatif (*Adaptive Neuro Fuzzy Inference System*) étaient utilisés pour prédire les fautes dans la version actuelle du logiciel. Cette approche permettait également de faire ressortir trois niveaux de risque de prédisposition aux fautes pour les classes du logiciel. Ce modèle, bien qu'étant très attrayant, a deux désavantages majeurs. Premièrement, l'approche n'est pas entièrement automatisée. Deuxièmement, l'approche requiert l'intervention d'un expert en systèmes à inférence floue. Pour la plupart des entreprises, ce type d'expert n'est pas disponible, ce qui rend le modèle de prédiction difficile d'approche.

En résumé, trois catégories d'approches de prédiction de fautes ont été présentées : supervisées, semi-supervisées et non-supervisées. Les approches supervisées utilisent des données sur les fautes pour entraîner un modèle à reconnaître les fautes dans le logiciel. Le modèle semi-supervisé agit de façon similaire, mais lorsque les données sur les fautes du système sont limitées. Les modèles non-supervisés, quant à eux, n'utilisent aucune donnée sur les fautes présentes dans le système pour prédire où se trouvent les fautes.

2.4 Évaluation de la performance d'un modèle de prédiction de fautes

Afin d'évaluer et de comparer la performance de différents modèles de prédiction de fautes, une matrice de confusion (ou table de classification) est utilisée. Cette matrice simple est composée de 4 cellules, chacune présentant les valeurs véridiquement ou faussement positives ou négatives. Voir le tableau I pour la présentation de la structure d'une matrice de confusion.

Tableau I

Structure d'une matrice de confusion

	Valeur réelle	
Valeur prédite	Fautif	Non fautif
Fautif	Vrais positifs (*True Positives* ou TP)	Faux positifs (*False positives* ou FP)
Non fautif	Faux négatifs (*False negatives* ou FN)	Vrais négatifs (*True negatives* ou TN)

Une matrice de confusion se compose de vrais positifs, de faux positifs, de faux négatifs et de vrais négatifs. Dans le cadre de la prédiction de fautes, une valeur positive signifie que la partie du code source est fautive et une valeur négative signifie que la partie est non fautive. Un vrai positif ou négatif signifie que la partie du code a correctement été prédite comme étant fautive ou non fautive respectivement. Un faux positif ou négatif signifie que la partie du code a incorrectement été prédite comme étant fautive ou non fautive respectivement. Pour chaque partie du code source sur laquelle une prédiction est faite, on incrémente de 1 la cellule de la matrice correspondant au résultat de la prédiction.

À partir de la matrice de confusion produite, plusieurs métriques de performance peuvent être calculées (à ne pas confondre avec les métriques de code source). Ces métriques sont nombreuses, mais seules les plus utilisées en prédiction de fautes sont présentées. Tout d'abord, les métriques FPR (*False Positive Rate* ou taux de faux positifs), FNR

(*False Negative Rate* ou taux de faux négatifs) et le taux d'erreur (*error rate*) sont très utilisées pour décrire la prédiction de fautes (Catal et al., 2009b, 2010 ; Abaei et al., 2014 ; Zhong, Khoshgoftaar, & Seliya, 2004). Voici comment ces trois métriques sont calculées :

$$FPR = \frac{FP}{FP + TN} \tag{2.1}$$

$$FNR = \frac{FN}{FN + TP} \tag{2.2}$$

$$\text{Taux d'erreur} = \frac{FP + FN}{FP + FN + TP + TN} \tag{2.3}$$

Pour ces trois métriques de performance, chaque valeur est meilleure lorsqu'elle est basse. Évidemment, étant donné que la prédiction de fautes n'est jamais parfaite, on cherche à équilibrer les valeurs des métriques FPR et FNR étant donné que la plupart du temps, lorsque l'une augmente, l'autre diminue.

En plus des ces 3 métriques, une autre métrique est souvent utilisée, soit g-mean (*geometric mean* ou moyenne géométrique) (Shatnawi, 2010 ; Malhotra & Bansal, 2015). L'avantage de cette métrique est qu'elle convient bien lorsque la prédiction est dite non balancée, c'est-à-dire que la prédiction ne qualifie pas environ 50% des éléments comme appartenant à une catégorie et l'autre 50% à une autre catégorie (Shatnawi, 2010). Dans le cas de la prédiction de fautes, la prédiction est la plupart du temps non équilibrée, car il est censé y avoir beaucoup plus de parties du code non fautives que fautives (Shatnawi, 2012 ; Mende & Koschke, 2010). Un autre avantage de la métrique g-mean est qu'à elle seule, elle décrit bien la prédiction, simplifiant ainsi la comparaison des résultats.

La métrique g-mean est calculée à partir de deux autres métriques : TPR (*True Positive Rate* ou la précision des positifs) et TNR (*True Negative Rate* ou la précision des négatifs). Contrairement aux trois métriques de performance présentées précédemment (FPR,

FNR et le taux d'erreur), TPR, TNR et g-mean sont considérées meilleures plus elles sont élevées. Voici comment ces trois métriques de performance sont calculées :

$$TPR = 1 - FNR = \frac{TP}{TP + FN} \tag{2.4}$$

$$TNR = 1 - FPR = \frac{TN}{TN + FP} \tag{2.5}$$

$$g\text{-}mean = \sqrt{TPR * TNR} \tag{2.6}$$

Les métriques FPR, FNR et g-mean sont beaucoup utilisées dans les articles rédigés dans le cadre de la maîtrise (donnés en annexe), afin de rendre les articles produits facilement comparables aux études existantes.

2.5 Conclusion

Dans le présent chapitre, une présentation de l'état de l'art de la prédiction de fautes a été présentée. Celle-ci peut être effectuée à partir de modèles supervisés, semi-supervisés ou non-supervisés. Ces différents types de modèles utilisent ou non des données existantes sur les fautes du système logiciel.

Un accent particulier a été mis sur les modèles de prédiction non-supervisés, étant donné que les modèles étudiés et présentés dans le présent mémoire sont principalement non-supervisés. Le prochain chapitre présent notamment l'utilisation de valeurs seuils pour effectuer de la prédiction de fautes non-supervisée.

CHAPITRE 3

LA PRÉDICTION DE FAUTES NON-SUPERVISÉE BASÉE SUR LES SEUILS

Comme mentionné dans la section 2.3 du présent mémoire, la plupart des approches non-supervisées utilisent des valeurs seuils sur les métriques de code afin de prédire où se situent les fautes dans le logiciel. Cette méthode simple permet de rendre l'approche facilement automatisable, ne nécessitant pas d'expert pour catégoriser le code source comme étant fautif ou non. Par exemple, Zhong et al. (2004) ont utilisé les algorithmes de regroupement des K-moyennes et de gaz neuronal pour regrouper les fonctions présentant des métriques semblables dans le logiciel. Un expert classifiait ensuite les regroupements comme étant possiblement fautifs ou non. L'utilisation de valeurs seuils permettrait l'automatisation de cette approche en éliminant la nécessité d'un expert.

Les modèles de prédiction de fautes basés sur les seuils fonctionnent très simplement. Un algorithme de regroupement peut être utilisé ou non pour regrouper les fonctions ou classes semblables du logiciel (selon leurs métriques de code). Ensuite, les valeurs seuils sont comparées aux valeurs des métriques de code correspondantes pour chacun des groupes, fonctions ou classes. Si un nombre x de métriques de code excèdent leur valeur seuil, l'instance (groupe, fonction ou classe) est considérée comme fautive. Sinon, elle est considérée comme non fautive. Cela permet une prédiction des fautes très simple et facile à comprendre pour ses utilisateurs.

Une composante critique des approches basées sur les valeurs seuils est le calcul de ces valeurs seuils utilisées. Étant donné que toute la prédiction dépend de ces calculs, ceux-ci doivent être pertinents et surtout donner de bonnes prédictions.

Plusieurs méthodes de calcul de seuils existent, mais trois pouvant potentiellement être utilisées pour la prédiction de fautes ont été retenues et analysées. Ces trois méthodes sont : la méthode des courbes ROC (Shatnawi, Li, Swain, & Newman, 2010), la méthode VARL

(*Value of an Acceptable Risk Level*) (Bender, 1999) et la méthode des niveaux d'Alves (*Alves Rankings*) (Alves, Ypma, & Visser, 2010). Ces trois méthodes ont été analysées dans deux articles rédigés dans le cadre de la maîtrise (Boucher & Badri, 2016, 2018). Les résultats de ces analyses seront donnés dans le chapitre 4.

Le présent chapitre présente les trois approches de calcul de seuils qui ont été utilisées et analysées dans le cadre de la maîtrise et des articles produits (Boucher & Badri, 2016, 2018).

3.1 Méthode des courbes ROC

La méthode des courbes ROC pour calculer des valeurs seuils est très simple et a été proposée par Shatnawi et al. (2010). Elle consiste d'abord à représenter la performance de prédiction d'une valeur seuil sur un graphique, appelé courbe ROC (*Receiver Operating Characteristic*). Cette courbe tente de maximiser la sensibilité (*sensitivity*) et la spécificité (*specificity*), deux indicateurs de performance de la prédiction. Ces deux indicateurs sont calculés à partir d'une matrice de confusion (voir le tableau I pour un exemple) et sont donnés par les deux formules suivantes :

$$Sensibilité = TP/(TP + FN) \tag{3.1}$$

$$Spécificité = 1 - FP/(FP + TN) \tag{3.2}$$

Pour construire la courbe ROC, toutes les valeurs seuils possibles pour une métrique de code dans un système logiciel donné sont testées une après l'autre. La performance de prédiction est ensuite calculée selon la sensibilité et la spécificité. La courbe ROC est ensuite construite, ou chaque couple (x, y) correspond à un couple $(sensibilité, 1 - spécificité)$ (Shatnawi et al., 2010). Une valeur seuil est ensuite retenue, simplement en

prenant le couple du graphique maximisant la somme des deux composantes.

Cette méthode de calcul des valeurs seuils est très simple, mais comporte un désavantage majeur. Elle peut être considérée comme une approche supervisée, car elle nécessite d'avoir les données réelles sur les fautes pour produire les valeurs seuils. Ces données sont nécessaires pour produire la matrice de confusion utilisée pour calculer les composantes de sensibilité et de spécificité. Cependant, elle pourrait être utilisée pour calculer les seuils sur un système semblable à un autre, et les valeurs seuils pourraient être réutilisées pour un autre système logiciel.

3.2 Méthode VARL

La méthode VARL (*Value of an Acceptable Risk Level* ou valeur d'un niveau de risque acceptable) a été proposée par Bender (1999), mais pas pour le calcul de valeurs seuils pour les métriques de code. En fait, cette technique visait plutôt une utilisation pour des études épidémiologiques, mais certains chercheurs l'ont adaptée pour le génie logiciel (Shatnawi, 2010 ; Malhotra & Bansal, 2015 ; Singh & Kahlon, 2014).

Cette méthode simple utilise l'analyse de régression logistique univariée pour calculer des valeurs seuils. Cette analyse statistique utilise des variables indépendantes (dans notre cas les métriques de code) pour prédire une variable dépendante (dans notre cas la prédisposition aux fautes d'une partie du code source). Les équations suivantes sont utilisées pour construire le modèle de régression logistique univariée :

$$P(x) = \frac{e^{g(x)}}{1 + e^{g(x)}} \tag{3.3}$$

$$g(x) = \alpha + \beta x \tag{3.4}$$

Dans ces équations, $P(x)$ représente la probabilité pour une partie du code source d'être considérée comme fautive en utilisant la métrique x (Malhotra & Bansal, 2015). Tant qu'à $g(x)$, c'est le logarithme naturel ln des chances qu'un événement survienne (la prédisposition aux fautes d'une partie du code) (Malhotra & Bansal, 2015). Dans l'équation 3.4, α donne l'ordonnée à l'origine (ou constante) et β donne la pente (ou le coefficient estimé) (Malhotra & Bansal, 2015). Les valeurs de α et β sont toutes deux utilisées dans le calcul des seuils avec la méthode VARL.

La valeur seuil donnée par VARL se calcule selon l'équation 3.5, où p_0 est le niveau de risque acceptable défini par l'utilisateur. Le niveau de risque acceptable peut être interprété comme étant la probabilité maximale qu'une faute survienne dans une partie du code où la valeur de la métrique de code est plus petite que sa valeur seuil. En variant ce paramètre, on peut donc obtenir des valeurs seuils différentes.

$$VARL = p^{-1}(p_0) = \frac{1}{\beta}\left(\log\left(\frac{p_0}{1-p_0}\right) - \alpha\right) \tag{3.5}$$

Un inconvénient majeur de cette approche est que comme la méthode des courbes ROC, elle est supervisée, c'est-à-dire que les données sur les fautes du logiciel sont nécessaires pour calculer les valeurs seuils.

3.3 Méthode des niveaux d'Alves

La méthode des niveaux d'Alves (*Alves Rankings*) a été proposée par Alves et al. (2010) pour calculer des valeurs seuils afin de décrire la qualité du code source. Étant donné que la méthode n'avait pas déjà été validée pour la prédiction de fautes, une analyse a été menée afin de valider ou réfuter si cette méthode peut être utilisée dans ces circonstances. La méthode n'ayant pas de nom, elle a été référencée comme étant la méthode des

niveaux d'Alves dans les articles rédigés (Boucher & Badri, 2016, 2018).

Afin de calculer les valeurs seuils, la méthode des niveaux d'Alves passe par 6 étapes. Cependant, les étapes 4 et 5 sont facultatives et sont nécessaires seulement si on calcule les valeurs seuils à partir de multiples systèmes logiciels.

La première étape consiste à extraire les métriques de code source du système logiciel. Dans cette même étape, le poids de chaque partie du code est également calculé. Dans l'étude d'Alves et al. (2010), le poids est donné par la métrique SLOC (*Source Lines of Code* ou nombre de lignes de code source). Ce poids sera réutilisé dans les étapes ultérieures.

La seconde étape de la méthodologie consiste à calculer le ratio des poids de chaque partie du code. Pour ce faire, le poids (métrique SLOC) de chaque partie du code est divisé par le poids total du système logiciel (somme de toutes les métriques SLOC). Cela donne la proportion de code source que chaque partie du code représente.

La troisième étape du calcul consiste à agréger les poids de chaque partie du code source par valeur de la métrique de code. Cela donne donc le pourcentage du système logiciel représenté par la valeur donnée d'une métrique de code. Par exemple, on pourrait savoir que 5% du code source est représenté par une métrique de couplage CBO de 6 après avoir réalisé cette étape.

Les quatrième et cinquième étapes, comme mentionné précédemment, permettent d'obtenir le même résultat qu'à l'étape trois, mais en utilisant les données de multiples systèmes logiciels. Les détails de cette méthodologie sont donnés dans l'étude d'Alves et al. (2010) mais sont exclus du présent mémoire, car ces étapes ne sont pas utilisées dans les articles produits et présentés.

Pour ce qui est de la sixième et dernière étape du calcul, des valeurs seuils sont calculées en choisissant un pourcentage du code source que l'on veut cibler. Par exemple, si on choisit de cibler 80% du code source, on pourrait avoir une valeur seuil de 30 pour la métrique de code CBO. 20% du code aurait donc une valeur de couplage CBO supérieure à 30 et serait donc ciblé lorsque la valeur seuil de 30 serait utilisée.

L'avantage majeur de cette technique sur les méthodes de calcul des courbes ROC et de VARL est qu'elle ne nécessite aucune donnée sur les fautes pour calculer les valeurs seuils. Seule la distribution des différentes métriques de code est utilisée à des fins de calcul.

3.4 Conclusion

Trois méthodes de calcul des valeurs seuils ont été présentées : les courbes ROC, VARL et les niveaux d'Alves. Les méthodes des courbes ROC et de VARL utilisent des données sur les fautes d'un ou plusieurs systèmes logiciels pour générer des valeurs seuils. Cependant, la méthode des niveaux d'Alves peut calculer des valeurs seuils sans l'utilisation de données existantes sur les fautes.

Dans la suite du mémoire, les trois techniques de calcul de valeurs seuils sont utilisées pour faire de la prédiction de fautes. Cependant, un accent particulier est mis sur la technique des niveaux d'Alves, étant donné qu'elle s'applique facilement dans un contexte non-supervisé.

CHAPITRE 4

ANALYSE DES MÉTHODES DE CALCUL DES VALEURS SEUILS

Dans le cadre de la maîtrise, une analyse a été menée sur les modèles de prédiction de fautes non-supervisés et plus particulièrement sur ceux utilisant des valeurs seuils sur les métriques de code. C'est pourquoi une analyse des différentes façons de calculer les métriques de code a été menée et présentée dans 2 articles (un pour la conférence ACIT 2016 (Boucher & Badri, 2016) et un autre pour le journal IST (Boucher & Badri, 2018)). Étant donné que l'article d'IST (Boucher & Badri, 2018) étend celui d'ACIT (Boucher & Badri, 2016), seulement celui-ci sera présenté dans le présent mémoire.

Dans ce chapitre, différents éléments de l'article rédigé (Boucher & Badri, 2018), dont ses objectifs, la méthodologie suivie, les résultats obtenus et leur discussion seront résumés.

4.1 Objectifs

Les objectifs de l'article rédigé étaient d'abord de comparer les différentes méthodes de calcul de valeurs seuils présentées au chapitre 3 (courbes ROC, VARL et les niveaux d'Alves), afin de déterminer laquelle ou lesquelles donnaient les meilleurs résultats de prédiction de fautes dans les systèmes orientés-objet. Pour ce faire, nous avons posé et répondu à 6 questions de recherche.

Question #1 : Est-ce que la méthode des courbes ROC peut produire des valeurs seuils pour d'autres jeux de données que ceux utilisés dans l'étude originale (Shatnawi et al., 2010) et donner de bons résultats ?

Question #2 : Est-ce que la méthode VARL peut donner des valeurs seuils permettant une bonne prédiction de fautes ?

Question #3 : Est-ce que la méthode des niveaux d'Alves peut donner des valeurs seuils permettant une bonne prédiction de fautes ?

Question #4 : Laquelle des trois méthodes de calcul de seuils étudiées donne les meilleurs résultats pour la prédiction de fautes ?

Question #5 : Est-ce que les modèles de prédiction de fautes basés sur les valeurs seuils offrent une performance comparable aux modèles supervisés ? Si les modèles basés sur les valeurs seuils sont combinés avec un algorithme de regroupement ou d'apprentissage automatique, est-ce qu'ils performent mieux ?

Question #6 : Est-ce que les valeurs seuils calculées pour un système logiciel ou pour une version de ce dernier peuvent être réutilisées pour d'autres systèmes ou versions d'un même système et offrir une bonne prédiction de fautes ? Est-ce que cette réutilisation des valeurs seuils performe mieux que la réutilisation de modèles supervisés sur différents systèmes ou différentes versions d'un système ?

Cette étude a été menée car aucune étude précédente n'a été trouvée comparant ces méthodes de calcul de valeurs seuils. De plus, la technique de calcul de valeurs seuils des niveaux d'Alves n'avait jamais été utilisée pour faire de la prédiction de fautes, la rendant très intéressante à analyser.

4.2 Méthodologie

Trois techniques de calcul des valeurs seuils ont été analysées et comparées : la méthode des courbes ROC, VARL et la méthode des niveaux d'Alves. Les valeurs seuils ont été calculées en utilisant chaque technique pour 12 jeux de données distincts. Ces jeux de données ont été construits pour les systèmes logiciels Apache ANT (1.3, 1.4, 1.5, 1.6 et 1.7), Apache IVY 2.0, Apache LUCENE 2.4, Apache POI, Apache TOMCAT, KC1, JEdit

et Eclipse JDT Core. Les valeurs seuils ont été calculées pour quatre métriques de code choisies suite à une analyse de régression logistique univariée faite sur tous les jeux de données analysés. Les métriques de code retenues suite à cette analyse sont :

- **SLOC :** *Source Lines of Code* ou nombre de lignes de code source ;
- **CBO :** *Coupling Between Objects* ou couplage entre les objets ;
- **RFC :** *Response For a Class* ou réponse d'une classe ;
- **WMC :** *Weighted Methods per Class* ou somme de la complexité cyclomatique des méthodes d'une classe.

Le nombre de fautes répertoriées dans chacune des classes de chaque système était disponible. Nous avons donc dupliqué les valeurs des métriques de code de chaque classe selon le nombre de fautes présentes dans celle-ci. Cette méthodologie a déjà été utilisée par Yuming Zhou et Hareton Leung (2006) ainsi que Shatnawi (2012) pour tenir compte du nombre de fautes dans les prédictions.

Les algorithmes de réseaux bayésiens (*Bayes Network*), de réseaux de neurones artificiels (ANN), C4.5 et de *Support Vector Machine* ont été analysés comme modèles de prédiction de fautes supervisés. De plus, les algorithmes de regroupement SOM (*Self-Organizing Map* ou carte auto-adaptative) et des K-moyennes ont également été analysés comme modèles supervisés. Les résultats obtenus ont été comparés à ceux obtenus avec les valeurs seuils pour répondre à la question de recherche #5.

Aussi, certaines valeurs seuils calculées sur certains systèmes logiciels ont été utilisées pour faire de la prédiction de fautes dans d'autres systèmes, afin de voir si les valeurs seuils pouvaient être réutilisées. La même procédure a été suivie en testant avec des versions antérieures du même système logiciel, afin de vérifier si les valeurs seuils pouvaient être réutilisées dans les versions ultérieures d'un même logiciel. La même méthodologie a été

suivie pour construire les modèles supervisés sur des jeux de données différents et les réutiliser sur d'autres. Ces réutilisations des valeurs seuils et des modèles supervisés ont ensuite été comparées.

4.3 Résultats et discussion

En résumé, les résultats de prédiction de fautes ont montré que les méthodes de calcul de valeurs seuils des courbes ROC et des niveaux d'Alves étaient plus performantes que VARL. De plus, VARL ne pouvait pas calculer certaines valeurs seuils pour certains systèmes logiciels. La méthode des courbes ROC a donné des résultats significativement supérieurs à ceux des niveaux d'Alves. Par contre, la méthode des niveaux d'Alves a tout de même bien performé et elle est tout de même considérée comme étant une des meilleures techniques à utiliser. Aussi, étant donné que la méthode des niveaux d'Alves ne nécessite pas de données sur les fautes pour fonctionner, cela la rend plus simple d'utilisation que la technique des courbes ROC. De plus, cette technique permet de calculer différents seuils pour une seule et même métrique de code.

Parmi les modèles de prédiction de fautes supervisés construits à partir des algorithmes d'apprentissage automatique et de regroupement, l'algorithme des réseaux bayésiens est celui ayant donné les meilleurs résultats. Sa performance est comparable à celle donnée par les modèles construits à l'aide des valeurs seuils calculées via les courbes ROC. Aussi, la combinaison des algorithmes de regroupement et d'apprentissage automatique avec les valeurs seuils n'a pas amélioré la performance des modèles de prédiction basés sur les valeurs seuils.

Pour ce qui est des valeurs seuils réutilisées sur des systèmes logiciels différents, il semblerait que les valeurs seuils calculées pour un système logiciel peuvent être réutilisées pour d'autres systèmes semblables. Cependant, les résultats sont meilleurs lorsque les valeurs seuils sont calculées spécifiquement pour le système en question.

Lorsque les valeurs seuils sont calculées à partir des versions précédentes d'un logiciel, il semblerait que la prédiction donne de bons résultats pour le système étudié (Apache ANT). Par contre, la performance de prédiction n'est pas supérieure à celle donnée lorsque les valeurs seuils calculées pour la version courante du logiciel sont utilisées.

Lorsque le modèle supervisé basé sur les réseaux bayésiens (algorithme d'apprentissage automatique ayant le mieux performé) est construit sur certains systèmes et testé sur d'autres, il donne tout de même de bons résultats. Lorsqu'il utilise les données de fautes des versions précédentes d'un logiciel, il donne de bons résultats surtout si on utilise toutes les versions précédentes dans la prédiction. Cela s'explique par le fait qu'étant un algorithme d'apprentissage automatique, plus il y a de données d'entraînement, plus le modèle devient précis dans ces prédictions.

4.4 Conclusion

En résumé, les résultats obtenus dans cet article démontrent bien que les approches de prédiction de fautes non-supervisées basées sur les valeurs seuils donnent de bons résultats. Leur performance est comparable à celle des approches supervisées étudiées. Idéalement, les valeurs seuils devraient être calculées et testées sur la même version du même système logiciel lorsque c'est possible.

Les méthodes des courbes ROC et des niveaux d'Alves sont celles étant les plus prometteuses en terme de performance de prédiction. De plus, ces deux méthodes permettent de calculer des valeurs seuils à partir de n'importe-quel système logiciel, contrairement à VARL. C'est pourquoi seulement celles-ci sont utilisées pour le reste du mémoire.

CHAPITRE 5

LE MODÈLE DE PRÉDICTION DE FAUTES HYSOM

Le modèle de prédiction de fautes HySOM, proposé par Abaei et al. (2014), est un modèle non-supervisé, malgré qu'il soit présenté comme étant semi-supervisé dans l'article original. Il est présenté comme étant semi-supervisé car il utilise un algorithme supervisé dans la construction du modèle, mais il n'utilise aucun historique des fautes pour construire le modèle, le rendant donc non-supervisé selon les critères établis (voir sections 2.2 et 2.3).

Le modèle HySOM est ici présenté, car il a été considéré pour une adaptation visant à atteindre les objectifs de recherche. Le modèle a été adapté pour une utilisation au niveau des classes plutôt qu'au niveau des fonctions, afin d'améliorer sa performance de prédiction (Boucher & Badri, 2017a). Dans ce chapitre, le modèle HySOM original tel que présenté par Abaei Abaei et al. (2014) est expliqué. Le chapitre suivant présentera ce qui a été réalisé avec le modèle HySOM dans l'article pour la conférence QRS 2017 (Boucher & Badri, 2017a).

5.1 Les métriques de code source utilisées

Le modèle HySOM utilise des métriques de code sur les fonctions et des valeurs seuils afin de déterminer si une fonction est potentiellement fautive ou non. En fait, il utilise les six métriques de code suivantes :

- ***Lines of Code* (LOC) :** Le nombre de lignes de code, incluant les lignes vides et les commentaires ;
- ***Cyclomatic Complexity* (CC) :** La complexité cyclomatique, définie par le nombre de chemins indépendants dans le graphe de flot de contrôle ;
- ***Unique Operators* (UOp) :** Le nombre d'opérateurs uniques dans le code ;

- ***Unique Operands* (UOpnd) :** Le nombre d'opérandes uniques dans le code ;
- ***Total Operators* (TOp) :** Le nombre d'opérateurs total dans le code ;
- ***Total Operands* (TOpnd) :** Le nombre d'opérandes total dans le code.

Quatre de ces métriques de code (UOp, UOpnd, TOp et TOpnd) proviennent de la suite de métriques de code d'Halstead (1977). Ces métriques permettent de décrire la taille et la complexité des différentes fonctions présentes dans le code source du logiciel.

5.2 Les valeurs seuils utilisées

Avec les six métriques de code présentées, le modèle HySOM utilise également pour chacune d'entre elles une valeur seuil, déterminant si une fonction est fautive ou non selon cette métrique. Dans leur article sur HySOM, Abaei et al. (2014) utilisent des valeurs seuils précédemment utilisées dans d'autres études antérieures (Catal et al., 2009a, 2010 ; Bishnu & Bhattacherjee, 2012).

Chaque valeur seuil utilisée a préalablement été calculée en utilisant l'outil PREDICTIVE, développé par Integrated Software Metrics (ISM) (Abaei et al., 2014). Cependant, aucune information n'a été trouvée mentionnant comment les valeurs seuils étaient calculées avec cet outil. Les articles utilisant les mêmes seuils (Catal et al., 2009a, 2010 ; Bishnu & Bhattacherjee, 2012 ; Abaei et al., 2014) ne donnent aucun indice à cet égard et le site web d'ISM [1] est hors ligne pour le moment, tel que mentionné précédemment par Catal et al. (2009a).

Malgré l'absence d'informations sur la méthode de calcul des valeurs seuils, celles-ci sont données dans l'article (Abaei et al., 2014) et sont les suivantes : 65 pour LOC, 10 pour CC, 25 pour UOp, 40 pour UOpnd, 125 pour TOp et 70 pour TOpnd. Le détail sur comment ces valeurs seuils sont utilisées dans le modèle HySOM est présenté plus loin

1. Integrated Software Metrics inc. - `http://www.ismwv.com`

lors de l'explication du fonctionnement du modèle (voir section 5.4).

5.3 Les jeux de données utilisés

Dans les différentes études de prédiction de fautes, il est important d'évaluer la performance des modèles de prédiction à l'étude. Pour ce faire, on doit non seulement avoir les métriques de code nécessaires pour construire le modèle, mais également avoir les données sur les fautes du système. Les données sur les fautes sont utilisées afin de valider que les parties du code source prédites comme étant fautives le sont et que celles prédites comme étant exemptes de fautes le sont bien également.

Obtenir ces données via le code source des programmes à l'étude et via les répertoires de bogues peut s'avérer long et fastidieux. C'est pourquoi la plupart des études utilisent des jeux de données publics, contenant toutes ces informations pour certains systèmes logiciels (Isong & Obeten, 2013). En plus de faciliter l'acquisition des données, l'utilisation de tels jeux de données facilite également la comparaison et la réplication des études.

Dans l'étude sur le modèle HySOM, ce sont huit jeux de données publics qui sont utilisés. Tous les systèmes logiciels sur lesquels sont construits ces jeux de données sont écrits en C, excepté pour deux d'entre-eux (KC1 et KC2) qui sont programmés en C++. Trois jeux de données sont construits sur des logiciels turcs (AR3, AR4 et AR5) et cinq jeux de données sont construits sur des systèmes de la NASA (CM1, KC1, KC2, MW1 et PC1). L'ensemble de ces jeux de données sont disponibles en ligne sur le répertoire PROMISE (Menzies, Krishna, & Pryor, 2016).

5.4 Architecture et fonctionnement du modèle

Le modèle HySOM est composé de deux parties principales : l'algorithme SOM et un réseau de neurones artificiels (ANN ou perceptron à couches multiples). Tout d'abord,

l'algorithme SOM regroupe les fonctions du code source présentant des métriques de code de valeurs similaires. Cela permet de réduire le nombre de vecteurs d'entrée qui seront utilisés pour la phase d'entraînement de l'ANN. L'ANN est ensuite entraîné en utilisant la sortie de l'algorithme SOM et les valeurs seuils des métriques de code. Cette section détaille le fonctionnement du modèle tel que présenté par Abaei et al. (2014). La figure 1 représente elle aussi le processus d'entraînement du modèle HySOM (Boucher & Badri, 2017b).

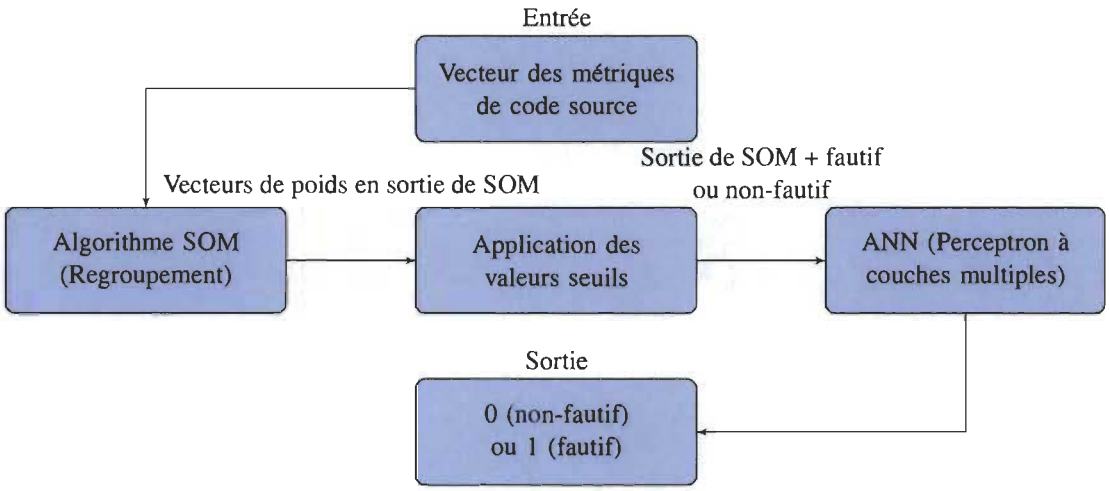


Figure 1 Processus d'entraînement du modèle HySOM

5.4.1 Première partie - SOM

L'algorithme SOM est un algorithme de regroupement, ce qui signifie qu'il est utilisé pour regrouper des vecteurs de données semblables. Il est aussi utilisé pour réduire le nombre de dimensions des données d'entrée (souvent pour avoir une représentation en deux dimensions).

La structure de l'algorithme SOM peut être représentée à sa plus simple forme par une grille carrée de S par S neurones, pour une taille totale de S^2 neurones. Dans le cas du modèle HySOM, chaque vecteur d'entrée de l'algorithme SOM est composé de $p = 6$ composantes, étant donné qu'on a 6 métriques de code par fonction. Les neurones du

SOM sont également des vecteurs à p composantes et sont notés w. Chaque composante des vecteurs w est appelée poids. Dans le modèle HySOM, l'algorithme SOM utilisé peut être présenté en trois phases : l'initialisation, l'entraînement et la finalisation.

5.4.1.1 La phase d'initialisation

Pour débuter, les métriques de code source pour chaque fonction du code source sont transformées sous forme de vecteurs. Ces vecteurs seront utilisés comme données d'entrée à l'algorithme SOM. Une fois les vecteurs d'entrée définis, une formule utilisée par Abaei et al. (2014) permet de calculer la taille du SOM. Cette formule utilise le nombre de fonctions dans le système logiciel n_{rows} et la dimension des vecteurs d'entrée p pour calculer la taille S du SOM (voir l'équation 5.1).

$$S = Round\left(\sqrt{\frac{n_{rows} \cdot p}{5 \cdot (p-1)}}\right) \tag{5.1}$$

Une fois que la taille du SOM est définie, chaque neurone w est initialisé en utilisant des valeurs aléatoires situées entre 0 et 1 inclusivement. Chaque neurone servira par la suite pour représenter un groupe de fonctions présentant des métriques de code semblables.

5.4.1.2 La phase d'entraînement

La phase d'entraînement de l'algorithme SOM permet de calibrer les neurones afin de bien regrouper les vecteurs d'entrée. Pour commencer, un nombre d'itérations maximal t_{max} est défini afin de s'assurer que l'entraînement se termine. Après chaque itération, un taux d'erreur est calculé. L'entraînement se termine soit quand ce taux d'erreur est en bas d'un certain seuil ou lorsque le nombre maximal d'itérations est atteint. Au début de chaque itération, le taux d'apprentissage et le rayon d'impact sont calculés. Ces deux

valeurs sont élevées dans les premières itérations, puis diminuent à chaque itération. Ces deux paramètres sont calculés en utilisant les formules 5.2 et 5.3. Dans ces deux équations, L_0 est le taux d'apprentissage initial et R_0 est le rayon d'impact initial (à l'itération $t = 0$). Tant qu'à λ_L et λ_R, ce sont des constantes qui définissent à quel rythme le taux d'apprentissage et le rayon d'impact diminueront respectivement.

$$L(t) = L_0 \exp\left(-\frac{t}{\lambda_L}\right) \tag{5.2}$$

$$R(t) = R_0 \exp\left(-\frac{t}{\lambda_R}\right) \tag{5.3}$$

Au début de chaque itération de l'entraînement, un vecteur d'entrée est choisi aléatoirement et la distance euclidienne le séparant de chaque neurone du SOM est calculée. Une fois toutes les distances calculées, le neurone le plus « près » du vecteur d'entrée est considéré comme étant le BMU (*Best Matching Unit* ou meilleure combinaison). Le BMU est ensuite utilisé pour mettre à jour les poids de tous les neurones du SOM. Pour ce faire, la distance euclidienne $dist_w$ entre chaque neurone du SOM et le BMU est calculée. L'influence du BMU sur chaque neurone ϕ est ensuite calculée en utilisant l'équation 5.4. Selon cette équation, plus un neurone est près du BMU, plus son influence est grande. Cette influence est également déterminée en utilisant le rayon d'impact calculé au début de l'itération.

$$\phi(t, w) = \exp\left(-\frac{dist_w^2}{2R^2(t)}\right) \tag{5.4}$$

Une fois l'influence du BMU calculée pour un neurone w, les poids de ce neurone peuvent être mis à jour. Pour ce faire, chaque composante w_i du neurone w est mise à jour une après l'autre, en utilisant l'équation 5.5.

$$w_i(t+1) = w_i(t) + \phi(t,w)L(t)(BMU_i - w_i(t)) \tag{5.5}$$

L'algorithme SOM répète ce processus de mise à jour de ses neurones jusqu'à la condition d'arrêt de l'algorithme. Une fois l'entraînement terminé, chaque vecteur d'entrée appartient au groupe formé par le neurone le plus « près » de ce vecteur d'entrée. L'algorithme SOM traditionnel est terminé à ce moment dans le modèle HySOM. La phase de finalisation restante est donc propre au modèle HySOM.

5.4.1.3 La phase de finalisation

Cette phase, propre au modèle HySOM, permet de préparer la sortie de l'algorithme SOM pour être passée au réseau de neurones artificiels (ANN). Pour chaque neurone w du SOM, on considère une valeur NOH (*Number of Hits* ou nombre de correspondances), qui correspond au nombre de vecteurs d'entrée regroupés par ce neurone. Les neurones ne regroupant aucun vecteur d'entrée ($NOH = 0$) sont considérés comme « morts » et sont donc ignorés pour la suite de l'algorithme.

Une fois les neurones « morts » retirés, une valeur potentiellement fautive ou non fautive est affectée à chaque groupe (ou neurone) restant. Pour ce faire, le poids w_i de chaque neurone (correspondant à la valeur d'une métrique de code) est comparé à la valeur seuil de la métrique de code correspondante. Ensuite, si 3 composantes ou plus du neurone dépassent leurs valeurs seuils respectives, le groupe formé par le neurone est considéré comme fautif. Sinon, il est considéré comme non fautif. Enfin, l'algorithme SOM donne en sortie le poids de chaque neurone encore « vivant » et une valeur fautive ou non associée à chaque groupe. Ce sont ces données qui seront passées à l'ANN pour la suite de la construction du modèle.

5.4.2 Deuxième partie - ANN

Le réseau de neurones artificiels (ANN) utilisé dans le modèle HySOM est en fait un perceptron à couches multiples (*Multilayer Perceptron*). Cet algorithme permet de catégoriser, suite à un entraînement, des éléments dans deux catégories ou plus. Cet algorithme permet de modéliser une fonction potentiellement non linéaire, ce qui en fait un meilleur classifieur que la régression linéaire. Dans l'étude d'Abaei et al. (2014), peu de détails sont donnés sur la structure de l'ANN. Il est seulement mentionné que le réseau est minimalement composé de 2 couches, soit une couche d'entrée et une couche de sortie. Il est probablement composé d'une ou plusieurs couches cachées également, mais l'article ne le mentionne pas (Abaei et al., 2014). Aussi, c'est probablement un ANN *feedforward* utilisant l'algorithme de propagation arrière (*backpropagation*) pour la phase d'entraînement.

Un perceptron à couches multiples *feedforward* a toujours une couche d'entrée et une couche de sortie. Il peut également avoir des couches cachées, permettant de détecter encore plus de liens entre les variables. Chacune de ces couches est composée d'un ou plusieurs neurones. Par exemple, dans le modèle HySOM, l'ANN est composé de 6 neurones dans la couche d'entrée (une pour chaque métrique de code). Sa couche de sortie ne contient qu'un seul neurone, qui a une valeur près de 0 (signifiant que l'entrée est non fautive) ou de 1 (signifiant que l'entrée est potentiellement fautive). Chaque neurone peut être vu comme étant le noeud d'un graphe dirigé et les liens entre les neurones comme étant les arcs du graphe.

Aussi, chaque neurone peut être simplement considéré comme étant une fonction. Cette fonction, appelée fonction d'activation, reçoit un vecteur en entrée et calcule une valeur réelle qui est donnée en sortie par le neurone. Une valeur est également associée à chaque lien entre les neurones. Celle-ci est multipliée par la valeur en sortie du neurone de la couche précédente et donne la valeur d'une composante du vecteur d'entrée du neu-

rone de la couche suivante. Pour les neurones de la couche d'entrée, leur sortie est donnée directement par les valeurs en entrée, soit la valeur des différentes métriques de code.

Une fois que la structure du réseau de neurones a été choisie, celui-ci est entraîné en utilisant l'algorithme de rétropropagation. Expliqué simplement, cet algorithme commence avec un réseau initialisé avec des valeurs aléatoires. Ensuite, une valeur d'entrée est passée et la valeur de sortie du réseau est comparée à la valeur qu'il devrait donner (la réponse). Une valeur d'erreur est ensuite calculée, qui est l'écart entre la valeur réelle et la valeur calculée. Cette erreur est ensuite propagée progressivement dans les couches précédentes du réseau de neurones, d'où le terme de rétropropagation.

Dans le cas du modèle HySOM, les neurones de sortie de l'algorithme SOM (vecteurs de métriques de code) sont donnés en entrée à l'ANN et la valeur fautive ou non fautive est donnée comme réponse à calculer au réseau de neurones. Une fois entraîné, le réseau de neurones peut être utilisé directement en utilisant les métriques de code de n'importe quelle fonction du code source afin de prédire si celle-ci est fautive ou non.

Les résultats de prédiction donnés par le modèle HySOM sont présentés et analysés dans un article joint à ce mémoire (Boucher & Badri, 2017a).

5.5 Conclusion

Le modèle de prédiction de fautes HySOM est un modèle très intéressant pour faire de la prédiction de fautes, combinant le regroupement de données et un algorithme de réseau de neurones artificiels. Le modèle, malgré l'utilisation d'un algorithme supervisé, fonctionne de façon totalement non-supervisée. Pour ces raisons, il est étudié de façon plus approfondie dans le prochain chapitre.

CHAPITRE 6

ADAPTATION DU MODÈLE HYSOM POUR UNE UTILISATION SUR LES CLASSES

Le modèle de prédiction des fautes HySOM, tel que présenté dans le chapitre 5, utilise des métriques de code au niveau des fonctions afin d'effectuer de la prédiction des fautes. Cependant, de nos jours, la plupart des systèmes logiciels sont développés en langage orienté-objet et sont testés de façon unitaire (une unité étant une classe). De plus, nous croyions qu'une utilisation de métriques de code au niveau des classes améliorerait la performance du modèle HySOM. C'est donc pourquoi nous avons décidé de l'adapter pour une telle utilisation dans un article pour la conférence QRS 2017 (Boucher & Badri, 2017a).

Dans ce chapitre, les objectifs, la méthodologie et les résultats obtenus suite à ces recherches seront présentés.

6.1 Objectifs

L'objectif de recherche de cet article, comme précédemment mentionné, était d'adapter le modèle HySOM pour une utilisation au niveau des classes. Cette adaptation visait à simplifier la planification de l'effort de test en pointant clairement quelles classes sont à risque de contenir des fautes. Étant donné que la plupart des logiciels développés de nos jours sont écrits en langages orientés-objet et que les tests unitaires se font sur les classes, il est logique d'avoir une prédiction des fautes faite au niveau des classes. De plus, le modèle HySOM original a été analysé et sa performance de prédiction comportait quelques lacunes, comme un taux de faux négatifs (FNR) souvent élevé. Cet article visait donc à améliorer ces aspects du modèle.

De plus, le modèle HySOM était intéressant à considérer dans le cadre des objectifs

de ce mémoire, car c'est un modèle non-supervisé qui a déjà été analysé par le passé. Il aurait pu être simplement modifié et adapté pour donner plusieurs niveaux de risque en sortie plutôt que d'avoir simplement une division dichotomique entre les classes fautives et non-fautives.

6.2 Méthodologie

L'adaptation du modèle HySOM pour une utilisation au niveau des classes s'est faite en plusieurs étapes. Ces étapes sont résumées dans cette section pour décrire le processus d'adaptation. Il est important de noter que le processus décrit peut être réutilisé avec n'importe quel modèle de prédiction des fautes basé sur les valeurs seuils.

6.2.1 Choix des métriques de code

Étant donné que les métriques de code utilisées au niveau des fonctions sont différentes de celles utilisées au niveau des classes, les métriques de code utilisées pour construire le modèle ont dû être changées. Pour ce faire, une analyse de régression logistique univariée a été effectuée sur 12 systèmes logiciels différents pour lesquels les données sur les fautes étaient disponibles (Boucher & Badri, 2017a). Les métriques de code ayant été retenues par cette analyse ont été SLOC, CBO, RFC et WMC.

6.2.2 Choix des systèmes logiciels à l'étude

Dans l'article original présentant le modèle HySOM, les jeux de données utilisés étaient, pour la plupart, basés sur des systèmes logiciels écrits en C (un langage procédural et non-orienté-objet) (Abaei et al., 2014). Il était donc normal d'utiliser des systèmes logiciels différents pour analyser l'adaptation faite du modèle HySOM. Pour ce faire, 12 jeux de données ont été choisis représentant chacun un système logiciel différent (ou de version différente). Un seul des jeux de données utilisés était basé sur un même système

logiciel analysé par le modèle HySOM original, le système logiciel KC1. Ce système logiciel a pu être utilisé dans les deux études car il a été développé en C++ (un langage de programmation orienté-objet).

6.2.3 Calcul des valeurs seuils

Une fois les métriques de code et les jeux de données déterminées, il fallait calculer les valeurs seuils des métriques de code pour chacun des systèmes. Pour ce faire, nous avons utilisé les techniques de calcul de valeurs seuils des courbes ROC et des niveaux d'Alves. Avec des nouvelles métriques de code et des valeurs seuils déterminées expressément pour ces métriques, le modèle HySOM pouvait être adapté très simplement. Voir la figure 2 pour une représentation graphique de comment le modèle HySOM a été adapté.

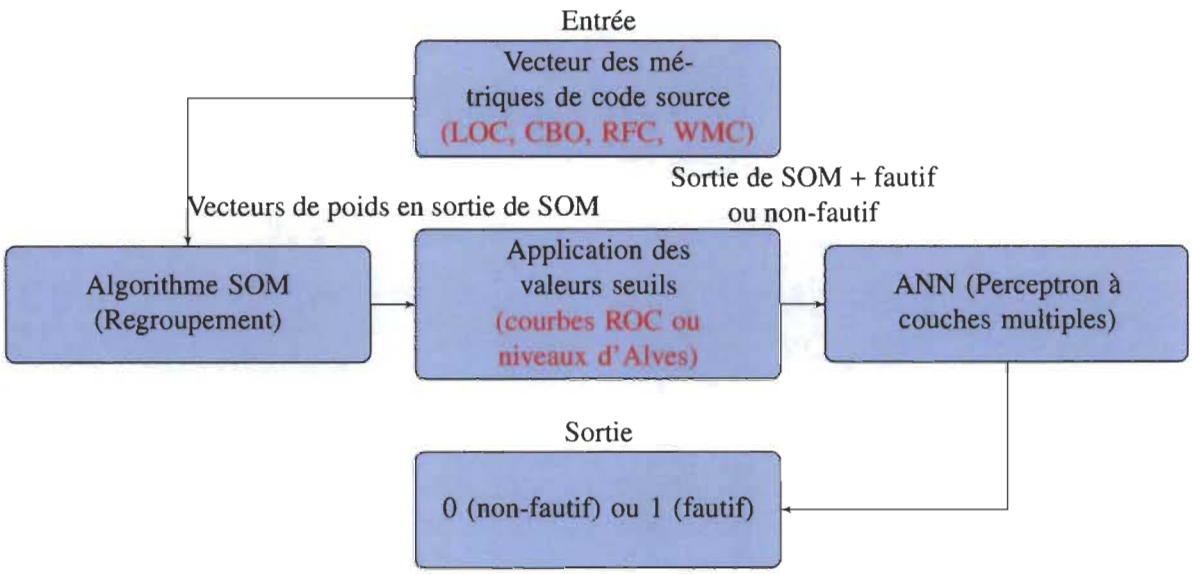


Figure 2 Adaptation du modèle HySOM pour une utilisation sur les classes

6.2.4 Comparaison de la performance avec des modèles supervisés

Afin de pouvoir comparer la performance du modèle HySOM adapté, trois modèles de prédiction des fautes ont été construits en utilisant différents algorithmes supervisés. Pour ce faire, un réseau bayésien naïf, un réseau de neurones artificiels (ANN) et un algorithme

de forêt aléatoire (*Random Forest*) ont été utilisés pour faire de la prédiction de fautes (Boucher & Badri, 2017a). Il était ainsi possible de comparer la performance du modèle adapté non-supervisé avec des modèles utilisant des données sur les fautes pour faire leurs prédictions.

6.3 Résultats et discussion

Les résultats donnés par le modèle HySOM adapté étaient très intéressants, s'étant avérés meilleurs que pour le réseau bayésien naïf et les autres algorithmes supervisés. Plus important encore, l'adaptation du modèle HySOM pour une utilisation au niveau des classes plutôt que des fonctions a donné de meilleurs résultats que l'approche originale. Aussi, l'adaptation faite au niveau des métriques de code et des valeurs seuils pourrait facilement être reproduite pour tout autre modèle de prédiction basé sur les valeurs seuils. Cependant, la performance pour certains jeux de données aurait pu être meilleure.

6.4 Conclusion

Le modèle HySOM, une fois adapté pour une utilisation au niveau des classes, donne de meilleurs résultats que le modèle original. Cependant, bien que prometteur, les résultats nous laissent croire qu'un meilleur modèle de prédiction de fautes pourrait être proposé.

C'est pourquoi le prochain chapitre présente un nouveau modèle que nous proposons, le modèle MRL (*Multiple Risk Levels*).

CHAPITRE 7

LE MODÈLE DE PRÉDICTION DE FAUTES MRL

Le modèle de prédiction de fautes HySOM a été étudié dans le cadre de la maîtrise et de la rédaction d'un article sur le sujet pour la conférence QRS 2017 (Boucher & Badri, 2017a). Par contre, ce modèle ne répondant pas aux attentes en termes de performance et de fonctionnalités, un nouveau modèle a été proposé dans un article pour le journal IST, le modèle MRL (*Multiple Risk Levels* ou niveaux de risque multiples) (Boucher & Badri, 2017b).

Dans ce chapitre, les objectifs, la méthodologie et les résultats obtenus suite à ces recherches seront présentés. Le modèle MRL y sera en même temps présenté et expliqué.

7.1 Objectifs

L'objectif de l'article sur le modèle MRL était de définir un modèle de prédiction de fautes non-supervisé donnant plusieurs niveaux de risque de prédisposition aux fautes à son utilisateur. Par exemple, au lieu de simplement dire si une classe du code source est potentiellement fautive ou non, on pourrait dire si elle a un risque élevé, moyen ou faible de contenir des fautes. En fait, l'objectif principal de la recherche rejoignait celui du présent mémoire.

Pour atteindre cet objectif, un modèle existant pouvait être adapté pour produire différents niveaux de risque. Par contre, si cela n'était pas concluant, un nouveau modèle pouvait également être proposé. Alors, le modèle HySOM a été considéré, mais ne donnant pas les résultats et la flexibilité escomptés (Boucher & Badri, 2017a), nous avons opté pour la définition d'un nouveau modèle, le modèle MRL (Boucher & Badri, 2017b). La performance du modèle MRL a ensuite été comparée à celles de deux modèles supervisés entraînés sur les versions précédentes de différents systèmes logiciels (représentant un cas

d'utilisation réel).

Un autre objectif de cette recherche était de vérifier si les différents niveaux de risque de prédisposition aux fautes donnés par le modèle étaient corrélés avec la sévérité des fautes. Si cela s'avérait le cas, les classes ayant un niveau de risque plus élevé auraient plus de chance de contenir des fautes de sévérité plus élevée. Cette corrélation permettrait de renforcer le besoin de tester en priorité les classes catégorisées dans des niveaux de risque plus élevé.

Ces différents objectifs ont été formulés sous la forme de 3 questions de recherche :

Question #1 : Est-ce que le modèle MRL proposé est plus performant que le modèle HySOM adapté pour une utilisation sur les classes ?

Question #2 : Y-a-t'il une corrélation entre les niveaux de risques donnés par le modèle MRL et la sévérité des fautes détectées ?

Question #3 : Est-ce que la prédiction de fautes supervisée basée sur les versions précédentes de systèmes logiciels peut être plus performante que le modèle MRL ?

7.2 Méthodologie

Dans l'article publié dans QRS 2017, une adaptation du modèle HySOM a été proposée pour permettre une prédiction au niveau des classes plutôt qu'au niveau des fonctions du code source (Boucher & Badri, 2017a). Cette adaptation avait été réalisée en vue de produire un modèle répondant aux attentes du présent mémoire. Cependant, nous croyions que la performance d'un modèle non-supervisé pourrait être encore meilleure et ses résultats plus facilement compréhensibles. Le processus d'adaptation fait avec le modèle HySOM est en fait facilement réalisable pour n'importe quel modèle de prédiction de fautes basé

sur les valeurs seuils. Ces travaux ont été repris et utilisés dans l'article d'IST à des fins de comparaison de performance (Boucher & Badri, 2017b).

Voyant que les résultats produits par le modèle HySOM ne satisfaisaient pas nos exigences de performance, un nouveau modèle a été proposé, soit le modèle MRL. En plus d'être mieux adapté aux objectifs de la recherche, la structure du modèle MRL est plus simple que celle du modèle HySOM. Ce modèle permet également de catégoriser des classes selon cinq niveaux de risque de contenir des fautes. La structure du modèle MRL sera expliquée plus loin (voir section 7.2.1). Le modèle MRL a été testé sur 12 jeux de données publics et chaque jeu de données a été testé deux fois. Une fois en considérant les classes du système comme fautives ou non dans la matrice de confusion et une autre fois en considérant le nombre de fautes détectées dans chaque classe.

Dans l'article rédigé (Boucher & Badri, 2017b), les niveaux de risque donnés par le modèle MRL sont ensuite comparés avec la sévérité des fautes de deux jeux de données (KC1 et Eclipse JDT Core). Cette analyse permet de voir si les classes catégorisées dans les niveaux de risque plus élevés contiennent plus de fautes critiques que les classes des niveaux de risque inférieurs. Aussi, une analyse de corrélation permet de vérifier la relation entre la sévérité des fautes et les niveaux de risque donnés par le modèle MRL.

Une fois cette étude de corrélation faite, la performance de prédiction du modèle MRL est comparée à celle de deux modèles de prédiction de fautes supervisés basés sur les réseaux bayésiens et sur un réseau de neurones artificiel (ANN). Les modèles supervisés sont entraînés sur une ou plusieurs versions précédentes du système logiciel. Cette comparaison permet de simuler des cas d'utilisation réels et de vérifier si le modèle proposé performe suffisamment bien pour être utilisé.

7.2.1 Structure du modèle MRL

La structure du modèle MRL est en fait très simple. Aucun algorithme d'intelligence artificielle ou de regroupement n'est utilisé dans son fonctionnement. Cela rend ses résultats facilement compréhensibles pour n'importe quel développeur ou testeur.

Le modèle MRL utilise quatre métriques de code source appliquées à la granularité des classes : SLOC, CBO, RFC et WMC. Ces quatre métriques de code ont été déterminées selon une analyse de régression logistique univariée appliquée à 12 jeux de données publics. Pour chaque métrique de code utilisée, plusieurs valeurs seuils sont calculées à l'aide de la méthode des niveaux d'Alves (telle que vue dans la section 3.3). Les valeurs seuils, calculées aux niveaux d'Alves de 90%, 70%, 50% et 30%, délimitent cinq niveaux de risque, soient : très élevé, élevé, moyen, faible et très faible.

Une classe est considérée comme ayant un risque très élevé d'être fautive si deux métriques ou plus dépassent les valeurs seuils de 90%. Une classe est considérée comme ayant un risque élevé d'être fautive si deux métriques ou plus dépassent les valeurs seuils de 70%. Il en va de même pour les niveaux de risque moyen et faible avec les valeurs seuils de 50% et de 30%. Toutes les classes n'étant pas considérées dans les niveaux de risque supérieurs ou égaux à faible sont considérées comme comportant un risque très faible de contenir des fautes. Pour une représentation schématique de ce fonctionnement, consultez la figure 3 (Boucher & Badri, 2017b).

Donc, en plus de catégoriser les classes par niveau de risque, une description pourrait facilement accompagner chaque classe catégorisée, indiquant pourquoi elle présente ce niveau de risque. Par exemple, prenons une classe catégorisée dans un niveau de risque très élevé. Elle pourrait être accompagnée d'une description courte indiquant qu'elle présente un risque très élevé de contenir des fautes car elle est très fortement couplée et qu'elle a une taille très élevée par rapport aux autres classes du système.

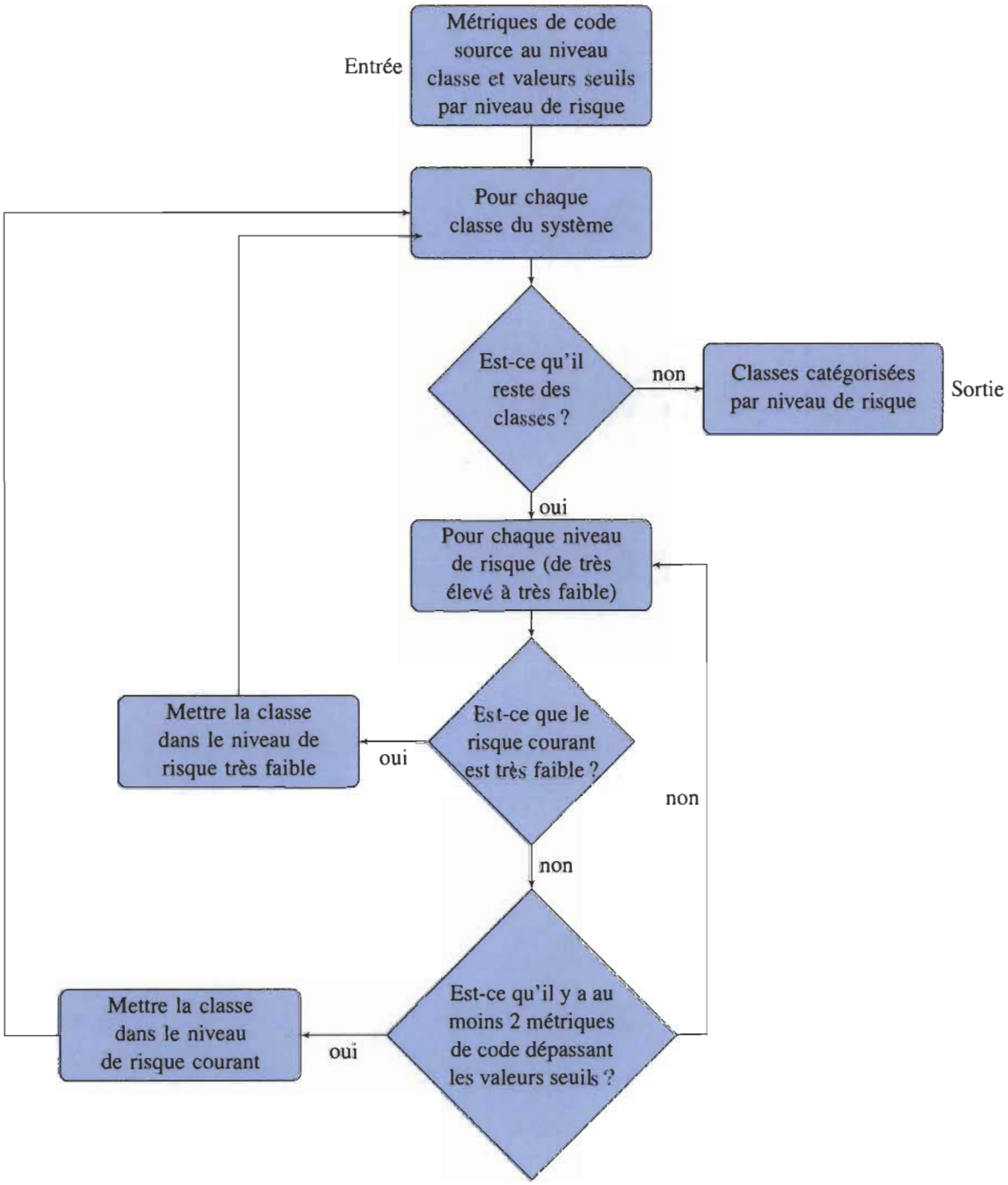


Figure 3 Étapes de construction du modèle MRL

Bref, en plus d'avoir une structure simple et facile à implémenter, le modèle MRL donne une sortie facilement compréhensible pour ses utilisateurs. De plus, les différents

niveaux de risque permettent de bien prioriser l'implémentation des tests dans le système logiciel.

7.3 Résultats et discussion

Le modèle MRL a donné une bonne performance, qui s'est avérée plus constante que celle du modèle HySOM adapté pour une utilisation au niveau des classes. Autrement dit, la performance de prédiction variait de façon moins importante d'un jeu de données à l'autre. La performance du modèle étant nettement supérieure et permettant de donner différents niveaux de risque à l'utilisateur, le modèle MRL a été retenu pour le reste des analyses.

Par la suite, l'analyse de la relation entre les niveaux de risque donnés par le modèle MRL et la sévérité des fautes a été menée. Il se trouve que les niveaux de sévérité seraient corrélés de façon significative à la sévérité des fautes. Les niveaux de risque très élevés et élevés sont ceux répertoriant le plus grand ratio de fautes critiques. Autrement dit, la plupart des fautes détectées dans ces deux niveaux sont critiques en termes de sévérité.

Ensuite, le modèle MRL a été comparé à deux modèles supervisés. Étonnamment, le modèle MRL donne une performance très semblable à celles des modèles supervisés sous étude (aucune différence statistique significative). Par contre, le modèle MRL semble plus fiable d'utilisation, car il donne une performance de prédiction plus constante d'un jeu de données à l'autre.

Suite aux résultats obtenus, il a été conclu que le modèle MRL répondait aux objectifs définis dans notre recherche. Tout d'abord, le modèle proposé permet une prédiction de fautes non-supervisée et donne plusieurs niveaux de risque pour une classe de contenir des fautes. Aussi, selon les analyses sur deux jeux de données, les niveaux de risque donnés par le modèle MRL seraient corrélés avec la sévérité des fautes contenues dans les classes.

7.4 Conclusion

Dans le présent chapitre, le nouveau modèle MRL a été présenté et analysé. Ce modèle répond aux objectifs définis dans le cadre de ce mémoire et offre plusieurs avantages comparativement au modèle HySOM. Le modèle MRL est plus simple à comprendre pour l'utilisateur en plus de données plusieurs niveaux de risque possibles. C'est pourquoi le modèle MRL est finalement retenu plutôt que le modèle HySOM.

CHAPITRE 8

CONCLUSION

Le présent mémoire a présenté le cheminement de la recherche nécessaire pour arriver à produire le modèle de prédiction de fautes non-supervisé MRL.

Pour rappel, le but de cette recherche était d'adapter ou produire un modèle de prédiction de fautes pour atteindre deux objectifs majeurs. En premier lieu, un modèle de prédiction de fautes non-supervisé était nécessaire, car bien des systèmes logiciels n'ont pas de données sur les fautes disponibles. Aussi, cela rend la construction du modèle plus simple et rapide. En second lieu, la plupart des modèles existants ne font que dire si une partie du code est potentiellement fautive ou non. Cependant, cela ne permet pas de bien prioriser l'effort de test à investir. C'est pourquoi le modèle en question devait fournir à son utilisateur plusieurs niveaux de risque de prédisposition aux fautes. C'est donc pour ces raisons que le modèle MRL a été proposé. Une fois ces objectifs atteints, le modèle proposé pourrait être utilisé avec n'importe quel système logiciel orienté-objet.

Tout d'abord, une revue de littérature sur les différents modèles de prédiction de fautes existants a été réalisée. Un accent particulier a été mis sur les modèles non-supervisés, étant donné que c'était le type de modèle recherché. Aucun de ces modèles n'était entièrement non-supervisé et ne permettait de catégoriser les parties du code en plusieurs niveaux de risque.

Étant donné que la plupart des approches non-supervisées utilisaient des valeurs seuils sur les métriques de code dans leur prédiction, le calcul des valeurs seuils a été plus rigoureusement analysé. Les méthodes des courbes ROC, VARL et des niveaux d'Alves ont été analysées pour le calcul des valeurs seuils pour la prédiction de fautes (Boucher & Badri, 2016, 2018). Après cette recherche et la rédaction de deux articles sur le sujet, les méthodes des courbes ROC et des niveaux d'Alves se sont avérées les deux meilleures

approches de calcul de valeurs seuils. La méthode des courbes ROC a donné les meilleurs résultats, mais nécessite néanmoins des données sur les fautes pour calculer les valeurs seuils. La méthode des niveaux d'Alves, quant à elle, a présenté une performance de prédiction des fautes légèrement moindre à celle des courbes ROC, mais ne nécessite cependant aucune donnée sur les fautes dans ses calculs.

Par la suite, le modèle de prédiction de fautes non-supervisé HySOM a été expliqué et analysé. Dans le cadre de la rédaction d'un article, le modèle a été adapté pour une utilisation au niveau des classes afin d'améliorer sa performance de prédiction (Boucher & Badri, 2017a). Cependant, la performance du modèle n'était pas à la hauteur de nos attentes.

C'est pourquoi dans un des articles, le modèle MRL a été proposé (Boucher & Badri, 2017b). Ce modèle utilise différentes valeurs seuils calculées à l'aide de la technique des niveaux d'Alves afin de catégoriser les classes d'un système en cinq paliers de risque. Ces cinq niveaux de risque de contenir des fautes sont : très élevé, élevé, moyen, faible et très faible. Ce modèle, en plus de répondre aux objectifs définis en début de recherche, est très simple à comprendre autant pour quelqu'un souhaitant l'implémenter ou encore l'utiliser. De plus, d'après une analyse de corrélation sur les niveaux de risque et la sévérité des fautes, il est démontré que les niveaux de risque plus élevés du modèle contiennent des fautes de sévérités plus importantes. Cela démontre qu'il est important de bien tester les classes comportant un risque élevé de contenir des fautes, car elles ont plus de chance de contenir des fautes sévères.

Ce mémoire propose donc le modèle de prédiction de fautes MRL, utilisable sur tous les systèmes logiciels orientés-objet. Ce modèle offre une bonne performance de prédiction, comparable à celles données par des modèles de prédiction supervisés basés sur les réseaux bayésiens et ANN (Boucher & Badri, 2017b).

8.1 Recommandations

Plusieurs recommandations peuvent être faites suite à la présentation de la recherche menée. Celles-ci pourraient renforcer les résultats obtenus ou encore étendre l'approche à d'autres utilisations.

Tout d'abord, le modèle de prédiction de fautes MRL pourrait être testé sur plus de systèmes logiciels, afin de généraliser ses résultats. Cela devrait être fait sur plusieurs systèmes de différents domaines et de langages de programmation différents. Cela permettrait de généraliser les résultats non seulement au niveau des systèmes logiciels, mais également au niveau des langages de programmation utilisés.

Aussi, le modèle MRL pourrait aisément être implémenté dans un outil quelconque. Par exemple, une extension de l'outil de développement Eclipse ou encore IntelliJ IDEA pourrait être développée soutenant l'approche. Cela permettrait d'utiliser facilement le modèle durant le développement d'une grande variété de systèmes.

Une autre recommandation pourrait être d'adapter le modèle MRL pour utiliser des métriques de conception seulement et non des métriques de code. Cela permettrait de faire de la prédiction de fautes avant même que le développement commence à partir de diagrammes UML, par exemple.

BIBLIOGRAPHIE

Abaei, G., Rezaei, Z., & Selamat, A. (2013). Fault prediction by utilizing self-organizing map and threshold. *2013 IEEE International Conference on Control System, Computing and Engineering*, 465–470.

Abaei, G., Selamat, A., & Fujita, H. (2014). An empirical study based on semi-supervised hybrid self-organizing map for software fault prediction. *Knowledge-Based Systems*, *74*, 28–39.

Alves, T. L., Ypma, C., & Visser, J. (2010). Deriving metric thresholds from benchmark data. *2010 IEEE International Conference on Software Maintenance*, 1–10.

Avizienis, A., Laprie, J.-C., & Randell, B. (2001). *Fundamental Concepts of Dependability* (Rapport technique).

Bender, R. (1999). Quantitative Risk Assessment in Epidemiological Studies Investigating Threshold Effects. *Biometrical Journal*, *41*(3), 305–319.

Bishnu, P. S., & Bhattacherjee, V. (2012). Software Fault Prediction Using Quad Tree-Based K-Means Clustering Algorithm. *IEEE Transactions on Knowledge and Data Engineering*, *24*(6), 1146–1150.

Boucher, A., & Badri, M. (2016). Using Software Metrics Thresholds to Predict Fault-Prone Classes in Object-Oriented Software. *Special Session of Software Engineering with Artificial Intelligence, 4th International Conference on Applied Computing & Information Technology*, 169–176.

Boucher, A., & Badri, M. (2017a). Predicting Fault-Prone Classes in Object-Oriented Software : An Adaptation of an Unsupervised Hybrid SOM Algorithm. In *2017 ieee international conference on software quality, reliability and security (qrs)* (pp. 306–317). IEEE.

Boucher, A., & Badri, M. (2017b). An Unsupervised Fault-Proneness Prediction Model Using Multiple Risk Levels For Object-Oriented Software Systems : An Empirical Study. *Submitted to Information and Software Technology*, 1–36.

Boucher, A., & Badri, M. (2018). Software metrics thresholds calculation techniques to predict fault-proneness : An empirical comparison. *Information and Software Technology*, *96*, 38–67.

Catal, C. (2014). A Comparison of Semi-Supervised Classification Approaches for Software Defect Prediction. *Journal of Intelligent Systems*, *23*(1), 75–82.

Catal, C., Diri, B., & Ozumut, B. (2007). An Artificial Immune System Approach for Fault Prediction in Object-Oriented Software. In *2nd international conference on dependability of computer systems (depcos-relcomex '07)* (pp. 238–245). IEEE.

Catal, C., Sevim, U., & Diri, B. (2009a). Clustering and metrics thresholds based software fault prediction of unlabeled program modules. *ITNG 2009 - 6th International Conference on Information Technology : New Generations*, 199–204.

Catal, C., Sevim, U., & Diri, B. (2009b). Software Fault Prediction of Unlabeled Program Modules. *Proceedings of the World Congress on Engineering*, *I*, 1–6.

Catal, C., Sevim, U., & Diri, B. (2010). Metrics-Driven Software Quality Prediction Without Prior Fault Data. In S.-I. Ao & L. Gelman (Eds.), *Electronic engineering and computing technology* (pp. 189–199). Dordrecht : Springer Netherlands.

Cong Jin, & Jing-Lei Guo. (2013). Automated software fault-proneness prediction based on fuzzy inference system. In *Proceedings of 2013 2nd international conference on measurement, information and control* (Vol. 1, pp. 482–485). IEEE.

Erturk, E., & Akcapinar Sezer, E. (2016). Iterative software fault prediction with a hybrid approach. *Applied Soft Computing*, *49*, 1020–1033.

Gondra, I. (2008). Applying machine learning to software fault-proneness prediction. *Journal of Systems and Software*, *81*(2), 186–195.

Halstead, M. H. (1977). *Elements of software science*. New York : Elsevier Science Inc.

Hong, E. (2012). Software Fault-proneness Prediction using Random Forest. *International Journal of Smart Home*, *6*(4), 147–152.

Isong, B., & Obeten, E. (2013). A Systematic Review of the Empirical Validation of Object-Oriented Metrics Towards Fault-Proneness Prediction. *International Journal of Software Engineering and Knowledge Engineering*, *23*(10), 1513–1540.

Jaafar, F., Gueheneuc, Y.-G., Hamel, S., & Khomh, F. (2013). Mining the relationship between anti-patterns dependencies and fault-proneness. *2013 20th Working Conference on Reverse Engineering (WCRE)*, 351–360.

Lu, H., Cukic, B., & Culp, M. (2012). Software defect prediction using semi-supervised learning with dimension reduction. *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*, 314.

Lu, H., Cukic, B., & Culp, M. (2014). A Semi-supervised Approach to Software Defect Prediction. *2014 IEEE 38th Annual Computer Software and Applications Confe-*

rence, 416–425.

Malhotra, R. (2012). A defect prediction model for open source software. *Proceedings of the World Congress on Engineering*, *2*, 1–5.

Malhotra, R., & Bansal, A. J. (2015). Fault prediction considering threshold effects of object-oriented metrics. *Expert Systems*, *32*(2), 203–219.

Mende, T., & Koschke, R. (2010). Effort-Aware Defect Prediction Models. *2010 14th European Conference on Software Maintenance and Reengineering*, 107–116.

Menzies, T., Krishna, R., & Pryor, D. (2016). *The Promise Repository of Empirical Software Engineering Data.* Consulté le 2015-11-29, sur `http://openscience.us/repo/`

Shatnawi, R. (2010). 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.

Shatnawi, R. (2012). Improving software fault-prediction for imbalanced data. *2012 International Conference on Innovations in Information Technology (IIT)*, 54–59.

Shatnawi, R., Li, W., Swain, J., & Newman, T. (2010). Finding software metrics threshold values using ROC curves. *Journal of Software Maintenance and Evolution : Research and Practice*, *22*(1), 1–16.

Singh, S., & Kahlon, K. S. (2014). Object oriented software metrics threshold values at quantitative acceptable risk level. *Csit*, *2*(3), 191–205.

Yuming Zhou, & Hareton Leung. (2006). Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults. *IEEE Transactions on Software Engineering*, *32*(10), 771–789.

Zhong, S., Khoshgoftaar, T., & Seliya, N. (2004). Analyzing software measurement data with clustering techniques. *IEEE Intelligent Systems*, *19*(2), 20–27.

ANNEXE A

BOUCHER & BADRI, 2016

2016 4th Intl Conf on Applied Computing and Information Technology/3rd Intl Conf on Computational Science/Intelligence and Applied Informatics/1st Intl Conf on Big Data, Cloud Computing, Data Science & Engineering

Using Software Metrics Thresholds to Predict Fault-Prone Classes in Object-Oriented Software

Alexandre Boucher
Software Engineering Research Laboratory
University of Quebec, Trois-Rivières, Canada
Email: Alexandre.Boucher2@uqtr.ca

Mourad Badri
Software Engineering Research Laboratory
University of Quebec, Trois-Rivières, Canada
Email: Mourad.Badri@uqtr.ca

***Abstract*—Most code-based quality measurement approaches are based, at least partially, on values of multiple source code metrics. A class will often be classified as being of poor quality if the values of its metrics are above given thresholds, which are different from one metric to another. The metrics thresholds are calculated using various techniques. In this paper, we investigated two specific techniques: ROC curves and Alves rankings. These techniques are supposed to give metrics thresholds which are practical for code quality measurements or even for fault-proneness prediction. However, Alves Rankings technique has not been validated as being a good choice for fault-proneness prediction, and ROC curves only partially on few datasets. Fault-proneness prediction is an important field of software engineering, as it can be used by developers and testers as a test effort indication to prioritize tests. This will allow a better allocation of resources, reducing therefore testing time and costs, and an improvement of the effectiveness of testing by testing more intensively the components that are likely more fault-prone. In this paper, we wanted to compare empirically the selected threshold calculation methods used as part of fault-proneness prediction techniques. We also used a machine learning technique (Bayes Network) as a baseline for comparison. Thresholds have been calculated for different object-oriented metrics using four different datasets obtained from the PROMISE Repository and another one based on the Eclipse project.**

***Index Terms*—Metrics Thresholds, Class-Level Metrics, Object-Oriented Metrics, Faults, Fault-Proneness Prediction, Code Quality, Object-Oriented Programming.**

I. Introduction

Nowadays, software systems must be of good quality and in certain cases, fault-free, because problems generated by faults could cause major damages and important losses of money. The problem is that it is cost prohibitive, difficult and often impossible to exhaustively test all execution paths of a complex software. In order to support developers and testers in the testing process, quality models and tools can be used for identifying poor quality and particularly fault-prone code. These models generally use source code metrics to identify fault-prone classes or methods. Many metrics-based models were suggested by different researchers in the literature. Most of them use Chidamber and Kemerer (CK) [1] object-oriented metrics suite [2–6]. Some researchers defined models based on regression analysis [3, 4, 6, 7], machine learning algorithms [4, 6, 7], threshold effect of code metrics [2, 3, 6] or even a combination of those methods. The advantages of models based on threshold effects of code metrics is that they can easily be implemented and understood by the software engineering experts or programmers. In addition, they can provide valuable and simple insights on why a specific class is classified as fault-prone, for example. However, many subjective thresholds were suggested for those metrics by software engineering experts (see [8, 9] or thresholds used in [10]). Furthermore, the suggested thresholds can't be generalized to all projects. For example, McCabe suggested a given threshold for his cyclomatic complexity metric [8], and Rosenberg suggested different thresholds for Chidamber & Kemerer metrics [9] (not directly related to fault-proneness). These threshold definition techniques were strictly subjective [8–10]. Many objective threshold definition algorithms were suggested in literature [2, 3, 6, 11, 12], but not all of them were validated as good predictors of fault-proneness.

In this paper, we evaluated and compared two of those threshold definition algorithms as fault-proneness predictors. The studied methods are the ROC (Receiver Operating Characteristic) curves method suggested by Shatnawi *et al.* [3] and the method of Alves *et al.* [11], which we'll reference to as Alves Rankings method for brevity.

This paper is organized as follows. Section II presents a summary of the selected threshold definition algorithms and clearly define the context of this study. Section III describes the methodology used to perform our study, so it is easy to reproduce it on other datasets. The datasets, data preprocessing procedure, threshold calculation techniques and a machine learning algorithm used as a baseline will be described in this section. Section IV presents, analyses and discusses the findings of this study. Section V mentions the possible threats to the validity of our study. Finally, section VI concludes this paper by summarizing the contributions of this study and suggesting some future work directions.

II. Related Work

Many studies validated relationships between object-oriented metrics and code quality. Code quality and fault-proneness are closely related to each other and are often both referred in many papers addressing software quality. This strong relationship can be defined as higher quality classes will likely be less fault-prone than poor quality classes. This section

978-1-5090-4871-7/16 $31.00 © 2016 IEEE
DOI 10.1109/ACIT-CSII-BCD.2016.41

will present some relevant papers related to the problematic addressed in this paper.

Isong & Obeten presented a systematic review of papers using object-oriented metrics for predicting fault-proneness [5]. This paper states two pertinent conclusions for our study. First, according to most papers studied in this review, SLOC (Source Lines of Code), CBO (Coupling Between Objects) [1], RFC (Response For a Class) [1] and WMC (Weighted Methods per Class) [1] are the metrics that are the most related to fault-proneness. Second, the authors stated that most studies are not replicated by other researchers, but the datasets are often reused from one study to another, meaning that we can use datasets that were already used in other researches, therefore letting other researchers compare our findings with other studies.

Considering papers that mention threshold definition algorithms, the method by Shatnawi *et al.* makes use of ROC curves to define per-project code metrics thresholds [2]. The authors performed two classification experiments on three versions of the Eclipse project using their methodology: one binary and another one ordinal. The binary classification consisted in predicting if classes were fault-prone or not, while the ordinal one tried to predict if a class had high, medium, low or no risk to be fault-prone. Shatnawi *et al.* found that the method was not viable for binary classification of classes, but found relevant threshold values for high and medium risk categories of ordinal classification. They therefore concluded that: (1) more work is needed to be done on more datasets, and (2) so far, their method was useful for ordinal, but not for binary classification of classes. Since this method has been partially validated, we decided to investigate it further on different datasets than those used by the authors in their study (different versions of Eclipse IDE).

A different algorithm presented for deriving thresholds from source code metrics is the algorithm of Alves *et al.* [11] (which we mentioned earlier as Alves Rankings algorithm for brevity). The authors used metrics values distribution in order to define thresholds for different systems. By combining a hundred different projects, they extracted one threshold value per metric applicable to all projects. Alves *et al.* calculated metrics thresholds for evaluating class quality, but did not investigate if their method could be used for predicting fault-proneness. Furthermore, according to our searches, and to the best of our knowledge, there are no studies that investigated if the Alves Rankings method is valid to use in fault-proneness prediction. This is why we decided to use this threshold definition method in our study, to verify if it can be applied in this specific context.

During our study, we even found papers using thresholds produced by a tool called PREDICTIVE, developed by Integrated Software Metrics, Inc. (ISM) [10, 13]. However, we could not find the tool mentioned in those studies, and as mentioned in [10], ISM website is no longer accessible. Since we can't know how these metrics thresholds are calculated, we won't use them in this paper. Additionally, these thresholds were used for fault-prediction of multiple different datasets. However, according to us and other studies [2, 6], metrics thresholds should be defined on a per-project basis and even on a per-version basis, as suggested by Shatnawi *et al.* [2]. Of course, thresholds defined for a dataset could be applicable to another dataset. On the other hand, each development project is done differently, by different development teams possibly using different programming patterns, which makes it difficult (or even impossible) to reuse thresholds for all systems. Furthermore, project size should be considered when validating metrics thresholds on multiple datasets, as a threshold for one project could be obsolete for another one [2, 6].

Now that the study context is set, let's summarize and define more accurately what this paper is about. Our goal is to compare the selected metric threshold definition algorithms to determine if one is more appropriate for fault-proneness prediction. Of course, studies were performed with this goal in mind, but we are using two different methods and each one is used to achieve a different goal. ROC curves method was retained because it was not tested on different systems and binary classification was not valid for the system used in the original study. For that reason, we wanted to test binary classification on different datasets, seeing if this method is valid for other datasets. Alves Rankings is interesting to validate as a good threshold definition algorithm for fault-proneness prediction, because it has only being assessed to be good at defining metrics thresholds for code quality. Since code quality and fault-proneness are closely related, we thought that this method could give good results when used as a fault-proneness prediction model and decided to investigate it as such. For both methods, we will consider the number of faults in each class when doing fault-proneness prediction. During our study, we didn't find any study testing at least one of the 2 methods presented here evaluating the model performance taking the number of faults into account. Also, in order to simplify data collection and further comparison of our results with other studies, we will use datasets used in other studies, which will be presented in section III.

III. Experimental Design

The objective of this study is to assess and compare different thresholds definition techniques for fault-proneness prediction. In order to do so, the first step is to define which software metrics will be used to predict fault-proneness. The second step of this study is to find datasets which contain the information we need to do fault-proneness prediction, which are code metrics for each class of a system and the number of faults that occurred in that class. The third step is to find the thresholds values for each metric of each system using the 2 threshold definition algorithms we decided to investigate in this study (ROC curves and Alves Rankings). The fourth step is to find and use a machine learning algorithm that will serve as a baseline method for performance comparison of threshold calculation techniques used in step three. The fifth and final step is to compare performance of each threshold calculation technique and of the machine learning model.

A. Metrics

An important part of fault-proneness prediction using threshold effect of code metrics is to choose which metrics will be used for fault-proneness prediction. Since Source Lines of Code (SLOC) and Chidamber & Kemerer metrics are widely used for fault-proneness prediction [2–6], we decided to use a subset of those metrics for fault-proneness prediction. The selected subset was determined after univariate and multivariate logistic regression analysis done on the Apache ANT 1.7 dataset, which is available on the PROMISE repository website [14]. Plus, our subset of metrics was validated as being the same as the one found in [5], which is composed of SLOC (Source Lines of Code), CBO (Coupling Between Objects), RFC (Response For a Class) and WMC (Weighted Methods per Class) metrics. According to Isong & Obeten, those metrics are the best predictors of fault-proneness according to multiple studies [5]. We therefore concluded that this choice of metrics, which is validated by other previous studies and a personal analysis, is appropriate for predicting fault-prone classes.

Each of these metrics yields different information. SLOC is a size metric, WMC is a complexity metric, CBO is a coupling metric and RFC is another size/complexity metric. SLOC gives the number of source lines of code in a given class while WMC gives the sum of the cyclomatic complexity of each method in a given class, where the cyclomatic complexity is a metric defined by McCabe giving the number of linearly independent paths in source code [8]. CBO gives the number of classes to which a given class is coupled and finally, RFC gives the number of methods that can potentially be executed when a message is received by a given class (which is the number of methods in the class added to the number of methods that can be called by those methods) [1].

B. Datasets

Fault-proneness prediction input data is composed of two different elements, the source code metrics for each of the classes of a software system and the number of faults that occurred in each particular class. In a real-life enterprise context, the source code metrics and faults information would be obtained directly from the source code and bug tracker, but since we are in a research context and want to make our results as reproducible and comparable as possible, we will use datasets which can easily be obtained online and were used in other studies. In this study, we will use five different datasets from different systems, Apache ANT, Apache IVY, KC1, JEdit and Eclipse JDT Core. ANT, IVY and JEdit are available from the PROMISE Repository, which makes available multiple datasets for fault-proneness prediction [14]. KC1 dataset is available on the PROMISE Repository of University of Ottawa [15], while the Eclipse JDT Core dataset is available from the research results of D'Ambros *et al.* [16].

The first dataset, which was built on version 1.7 of the Apache ANT system, was used in multiple studies [17, 18]. ANT is a command-line tool developed in Java mainly used for building Java applications [19]. Another dataset used was made for Apache IVY 2.0, which was also used in multiple studies [17, 20, 21]. IVY is a dependency manager developed in Java, integrated in Apache ANT [19]. KC1 [15], which was developed by the NASA with the C++ language and was used in numerous studies [6, 13, 17, 22–24], is the third system we used in our study. The fourth dataset we used was built for the JEdit 4.3 program, which is a text editor developed in Java [21]. It was used in multiple studies for fault-proneness prediction [17, 19–21]. The last dataset used is based on the Eclipse JDT Core system. It was produced after a study by D'ambros *et al.* [16] on multiple releases of the system. The JDT Core is the primary infrastructure of the Eclipse Java IDE, which includes plenty of practical features for the developers using the Eclipse Java IDE [25]. The Eclipse project was used in numerous studies [2, 3, 5, 16, 23, 24, 26]. Although the JDT Core Component wasn't used specifically in those studies, we used this dataset for the simplicity of the data acquisition and to simplify study replication.

Note that for Apache ANT, IVY and JEdit datasets, WMC metric value had to be calculated using the average cyclomatic complexity of all methods multiplied by the method count in each class. The reason we are not using the WMC metric presented in those datasets is that it only gives the method count of each class according to the study that produced those datasets [19].

C. Threshold Definition Methods

As mentioned previously, we will assess and compare two threshold definition methods for fault-proneness prediction: ROC curves and Alves Rankings. We present summarily in what follows how we used each methodology to calculate threshold values for the selected source code metrics.

1) ROC Curves: The ROC curves method, as defined by Shatnawi *et al.* [2], plots a ROC curve for each code metric and then retrieves the optimal threshold for this value, maximizing the sum of sensitivity and specificity [2]. Plotting a ROC curve consists of taking a continuous and a binary variable. For this method, the continuous variable consists of the metric value for each class of the system, while the binary variable is the presence of faults in a given class. A range of possible thresholds for the metric is then produced, varying from the minimum to the maximum possible value for this metric in the given dataset. Then, for each possible threshold value defined, a confusion matrix is built. A confusion matrix is a table that presents classification results, giving the number of true/false positives/negatives (Table I gives the structure of a confusion matrix). Each confusion matrix then outputs a point on the ROC curve plot. The X axis of the plot is mapping the 1 - specificity value, while the Y axis is mapping the sensitivity. Each 1 - specificity and sensitivity pair is obtained from the confusion matrix using equations 1 and 2. The kept threshold will be the one that maximizes both 1 - specificity and sensitivity.

$$Specificity = 1 - FP/(FP + TN) \quad (1)$$

$$Sensitivity = TP/(TP + FN) \quad (2)$$

TABLE I
CONFUSION MATRIX STRUCTURE

	Actual	
Classified	Faulty	Not-faulty
Faulty	True positives (TP)	False positives (FP)
Not faulty	False negatives (FN)	True negatives (TN)

Since the original ROC curves method only seems to consider if a class contains a fault or not and is therefore not taking into account the number of faults in a class, we decided to make use of the number of faults in each class to calculate better thresholds. To do so, we followed a simple methodology used by Zhou & Leung [6] which consists of duplicating each class containing more than one fault in the dataset. For example, if a class contains 3 faults, it will be present 3 times in the dataset, each one marked as containing a fault. This variant is useful here since the number of faults can be used to determine better threshold values without much data preprocessing.

2) Alves Rankings: The method of Alves *et al.* for calculating thresholds didn't have a name in the original paper [11], so we decided to call it Alves Rankings method for brevity. This method hasn't been investigated for defining metric threshold values for fault-proneness prediction. Alves *et al.* did use their method to find thresholds describing quality of classes, for finally categorizing them. To do so, they passed through 6 steps for calculating thresholds [11], but as we will see, only the steps 1, 2, 3 and 6 will be relevant for our study.

The first step, which they called *metrics extraction*, consists of extracting the metrics of the system [11]. Of course, code metrics of each class will be calculated in this step, but also the *weight* of each class. The weight of a class is defined by SLOC in their paper. For our study, the first step consisted of finding the datasets we decided to use.

The second step, named *weight ratio calculation*, consists of calculating the *weight ratio* of each class [11]. This ratio is calculated simply by dividing the weight of a class (SLOC) by the sum of all classes weight. The weight ratio simply represents the relative size of each class in the system. For example, if a class has a weight ratio of 0.01, this means that the class code represents 1 percent of the total code of the system.

The third step, which is called *entity aggregation*, consists of aggregating the weight of all entities (which here are classes) per metric values [11]. The result of that step is similar to a weighted histogram, giving the percentage of code of the system being represented by each metric value. For example, after that step we could say that 1% of a system's SLOC is represented by a CBO metric of 6.

The fourth and fifth steps of the method proposed by Alves *et al.* will not be used for our study. The reason of this decision is that Alves *et al.* did calculate thresholds using a hundred different software systems [11]. In our case and as mentioned in section II, we wanted to calculate for each single system one threshold value for each code metric. The fourth and fifth steps of the Alves Rankings method consisted in normalizing the weights of each system they evaluated and aggregating the metric values for those systems, getting the same output as in third step, but where the percentage of each metric represents the percentage of code across all systems.

The sixth step of this method, which is called *thresholds derivation*, consists of calculating the threshold values for each class. To do so, we define a percentage of code we want to represent with our threshold values. For example, choosing 80% of the overall code could output a threshold value of 30 for the CBO metric. That would mean that 20% of the poor quality code according to CBO metric would be targeted by the threshold value of 30. As an example, Alves *et al.* used threshold values defined at 70%, 80% and 90% of the metrics distributions for their final quality model, but some tests are required in order to decide if those cut-off points are valid to use for fault-proneness prediction. In the original paper, they performed this step after aggregating the metrics at system level, but for our study, we will perform it at class level of each system separately.

D. Machine Learning Algorithm

For this study, a machine learning algorithm that was used multiple times for fault-proneness prediction will serve as a baseline for comparing the threshold calculation techniques. Many machine learning algorithms were used for fault-proneness prediction, as Random Forest [7, 13, 22], Support Vector Machine [7], Multilayer Perceptron (Artificial Neural Network) [4, 7, 13], Bayes Network [17, 26] and others [4, 7, 22]. Since Bayes Network seems to yield good results for fault-proneness prediction according to [7, 26], we will use this technique as a baseline for fault-proneness prediction performance. This algorithm classifies the given instances by building a Bayesian Network (directed graph), which maps metrics as nodes and their independencies as links between the metrics to classify instances as fault-prone or not [26].

E. Model Evaluation

An important point of our study is to compare the algorithms between each other. To evaluate the prediction performance of each threshold calculation technique and of the machine learning model, we will use the FPR, FNR and error rate metrics, which can be easily calculated using the confusion matrix resulting from the classification. Those metrics are often used in other studies to evaluate performance of fault-proneness prediction models [10, 13, 27]. Here are the equations used to calculate those 3 metrics:

$$FPR = \frac{FP}{FP + TN} \tag{3}$$

$$FNR = \frac{FN}{FN + TP} \tag{4}$$

$$Error\ rate = \frac{FN + FP}{TP + FP + FN + TN} \tag{5}$$

The FPR metric gives the percentage of false positives among all the actual negative values, while the FNR gives the

TABLE II
ROC CURVES THRESHOLD VALUES

Dataset	Metric threshold value			
	SLOC	CBO	RFC	WMC
ANT	336	9	46	15
IVY	299	8	39	30
KC1	103	8	62	43
JEdit	560	16	115	30
Eclipse	166	13	86	63

percentage of false negatives among all actual positive values. The error rate gives the percentage of errors in the classification. The lower each metric is, the better is the classification. To rapidly describe the performance of a classification using a classification table, we will use the following simple levels:

- Error rate > 0.5, FPR > 0.5 or FNR > 0.5 means no good classification;
- $0.4 <$ Error rate ≤ 0.5, $0.4 <$ FPR ≤ 0.5 and $0.4 <$ FNR ≤ 0.5 means poor classification;
- $0.3 <$ Error rate ≤ 0.4, $0.3 <$ FPR ≤ 0.4 and $0.3 <$ FNR ≤ 0.4 means fair classification;
- $0.2 <$ Error rate ≤ 0.3, $0.2 <$ FPR ≤ 0.3 and $0.2 <$ FNR ≤ 0.3 means acceptable classification;
- $0.1 <$ Error rate ≤ 0.2, $0.1 <$ FPR ≤ 0.2 and $0.1 <$ FNR ≤ 0.2 means excellent classification;
- Error rate ≤ 0.1, FPR ≤ 0.1 and FNR ≤ 0.1 means outstanding classification;

These levels are similar to the ones used by Shatnawi *et al.* [2] for the Area Under Curve metric. Classification should at least be in the acceptable range to be considered usable in a fault-proneness prediction context.

IV. EXPERIMENTAL RESULTS AND DISCUSSION

In this section, we present and discuss the results obtained for each of the threshold calculation techniques and the machine learning baseline model. For each threshold calculation technique, the thresholds are calculated and presented in different tables. It is worth noting that the thresholds presented are always inclusive lower bound thresholds. Fault-proneness prediction performance will then be presented for each threshold calculation technique and for the Bayes Network baseline algorithm.

A. Threshold Values

The threshold values for each threshold calculation technique are presented in this section.

1) ROC curves: ROC curves threshold values are straightforward and easily understandable, as presented in Table II.

2) Alves Rankings: Thresholds calculated using Alves Rankings method are presented in Tables III and IV. The first table presents thresholds values calculated at 30% of the metric distribution and the second one presents those calculated at 70%. We did so because 70% is proposed as the lower bound for medium risk classes in [11] and 30% seems to give more realistic thresholds for fault-proneness, according to many tests we performed on different datasets.

TABLE III
ALVES RANKINGS THRESHOLD VALUES AT 30% OF DISTRIBUTION

Dataset	Metric threshold value			
	SLOC	CBO	RFC	WMC
ANT	327	7	40	17
IVY	411	12	59	20
KC1	252	10	34	47
JEdit	529	9	53	30
Eclipse	311	13	91	78

TABLE IV
ALVES RANKINGS THRESHOLD VALUES AT 70% OF DISTRIBUTION

Dataset	Metric threshold value			
	SLOC	CBO	RFC	WMC
ANT	1031	18	103	61
IVY	1130	30	120	65
KC1	991	17	85	153
JEdit	2224	44	156	158
Eclipse	1500	44	374	396

B. Fault-Proneness Prediction

This part presents the fault-proneness prediction results given by each of the threshold calculation techniques and by the Bayes Network baseline algorithm. The threshold values will be further discussed in this part. For each threshold calculation technique, we calculated the thresholds of each system and then produced 4 classification tables. The first classification table is constructed by classifying classes when at least one metric exceeds the obtained threshold value as fault-prone. The second one considers classes as fault-prone when at least 2 metrics exceed the threshold values, the third one when 3 metrics exceed the threshold values and the fourth one when all metrics exceed their threshold values. For brevity and understandability, all methods will be suffixed with the number of metrics that need to exceed their threshold value in order to classify a class as fault-prone. For example, ROC-3 would mean that the classification table was obtained using the ROC curves methodology, classifying classes as fault-prone when at least 3 metrics exceed their threshold values. The 3 evaluation metrics (FPR, FNR and error rate), not to confound with the code metrics, will then be calculated for each confusion matrix produced.

As mentioned earlier, we took the number of faults in each class into account when doing our classification. The goal of doing so was to give more weight to the classification of classes containing multiple faults. With this methodology, if a class contains 3 faults and is classified as not fault-prone, there will be 3 false negatives added to the confusion matrix. On the other side, if the class is classified as fault-prone, 3 true positives will be added to the confusion matrix (same methodology as Zhou & Leung in [6]).

1) ROC curves: Shatnawi *et al.* methodology for defining threshold values gives the results presented in Table V when applied to fault-proneness prediction.

The results show a logical inverse relationship between FPR and FNR. If the number of metrics exceeding threshold

TABLE V
ROC CURVES FAULT-PRONENESS PREDICTION

	ANT			IVY			KC1			JEdit			Eclipse		
Model	**Error**	**FPR**	**FNR**	**Error**	**FPR**	**FNR**	**Error**	**FPR**	**FNR**	**Error**	**FPR**	**FNR**	**Error**	**FPR**	**FNR**
ROC-1	0.288	0.406	0.086	0.427	0.494	0.054	0.062	0.388	0.021	0.363	0.364	0.333	0.235	0.263	0.176
ROC-2	0.189	0.183	0.198	0.193	0.196	0.179	0.174	0.282	0.160	0.195	0.189	0.417	0.184	0.154	0.246
ROC-3	0.165	0.114	0.251	0.160	0.147	0.232	0.257	0.094	0.278	0.103	0.096	0.417	0.154	0.095	0.278
ROC-4	0.185	0.071	0.382	0.136	0.090	0.393	0.527	0.012	0.592	0.051	0.040	0.500	0.152	0.066	0.334

values needed to classify a class as fault-prone is raised, the FPR gets lower and the FNR gets higher. That is plausible because if more metrics exceeding the threshold values are needed to consider a class as fault-prone, more modules will be classified as not fault-prone and less will be classified as fault-prone, therefore increasing false negatives and reducing false positives.

Fault-proneness prediction using ROC curves threshold values seems to be good using 2 or 3 metrics exceeding threshold values for classifying a class as fault-prone, as ROC-2 and ROC-3 experiments seem to give the best results across all datasets. On all datasets, ROC-2 experiment gives error rate, FPR and FNR below 30%, except for the FNR of JEdit. For ROC-3 experiment, all metrics values are below 30% too, except again for the FNR of JEdit. Further analysis on other datasets would be needed to see if we could use the exact same experiments (ROC-2 and ROC-3) for other datasets, but these results seem to indicate that the method would be viable for other datasets.

2) Alves Rankings: Results obtained using the threshold values of the Alves Rankings method for fault-proneness prediction are presented in Table VI. Note that only results for thresholds defined at 30% of the metric distribution are presented, because threshold values defined at 70% of the metric distribution didn't give good results. We therefore decided to only present the best results for brevity.

One observation we can make is that when using 3 metrics or more before considering a class as fault-prone, the FNR is often too high to be considered good (above 0.3). However, the model Alves-30%-2 gives good or acceptable results across all datasets. We can therefore conclude that a model could be constructed using the Alves Rankings threshold values calculation technique for certain datasets, as it gives good classification for fault-proneness prediction.

3) Baseline method - Bayes Network: The Bayes Network is used as a baseline method for fault-proneness prediction and the results given by this method are presented in Table VII. The results are computed using the BayesNet model provided in Weka tool [28] using 10-fold cross-validation and keeping all default parameters.

Performance of classification using Bayes Network is often acceptable or excellent. The fact that the 3 classification table metrics are good for most datasets could be caused by taking the number of faults into account when performing classification. Doing so results in bigger folds produced for 10-fold cross-validation, which could enhance the classification performance of the algorithm, each fold having more training entries, therefore acting as boosting. The second fact we denote by analyzing Table VII is that performance for all Apache (ANT and IVY) and Eclipse datasets is good. However, classification performance for KC1 and JEdit datasets don't yield results as good as for other datasets, having a high FPR or FNR. For KC1, the error rate and FNR are good, but the FPR is bad, which will lead people to invest testing effort on classes that do not contain faults, while that for JEdit, testing effort is wrongly invested on classes that do not contain faults, as a lot of faulty classes as classified as not fault-prone.

4) Comparison of All Methods: Each threshold calculation technique was performed and analyzed, but they were not compared to each other. In Table VIII, we present the best model that was built for each threshold calculation method and for the Bayes Network baseline algorithm. For the ROC curves method, the variant using 2 metrics exceeding threshold values before classifying a class as fault-prone was considered for most datasets, while the variant using 3 metrics was considered best for the JEdit dataset. For the Alves Rankings method, the variant considering 2 metrics exceeding threshold values is kept for most datasets, except for Eclipse where the variant with one metric is used.

The first conclusion drawn from those results is that the baseline method (Bayes Network), gives the lower and best error rate among all methods, but the FPR and FNR is sometimes better in other techniques. For most datasets (except KC1 and JEdit), FPR of Bayes Network technique is better than all other methods, but FNR is worse than all other methods (except for JEdit, where it is the best, Alves Rankings in IVY dataset and ROC curves and Alves Rankings for KC1 dataset). The baseline method gives different results for different datasets, as performance can be good or bad. The ROC curves and Alves Rankings methods seem to give acceptable or excellent results for all datasets (except for JEdit). For the JEdit dataset, where ROC curves and Alves Rankings performed worse, even the baseline method gave bad results. Maybe this dataset was not properly tested so that not most faults were discovered in the system, therefore making the classification classify classes as containing faults but not being marked as such in the datasets. ANT, IVY, KC1 and Eclipse all give at least acceptable results according to error rate, FPR and FNR when thresholds-based models are considered (ROC Curves and Alves Rankings). KC1 doesn't give acceptable FPR for baseline method (Bayes Network), which could be explained by the high percentage of faulty classes (above 40%) in this dataset. Boosting resulting

TABLE VI
ALVES RANKINGS FAULT-PRONENESS PREDICTION

	ANT			IVY			KC1			JEdit			Eclipse		
Model	**Error**	**FPR**	**FNR**	**Error**	**FPR**	**FNR**	**Error**	**FPR**	**FNR**	**Error**	**FPR**	**FNR**	**Error**	**FPR**	**FNR**
Alves-30%-1	0.334	0.485	0.074	0.326	0.356	0.161	0.206	0.459	0.173	0.560	0.565	0.333	0.218	0.223	0.209
Alves-30%-2	0.214	0.238	0.172	0.188	0.170	0.286	0.264	0.224	0.269	0.286	0.285	0.333	0.163	0.095	0.307
Alves-30%-3	0.183	0.155	0.231	0.160	0.128	0.339	0.310	0.082	0.339	0.211	0.206	0.417	0.161	0.076	0.340
Alves-30%-4	0.190	0.088	0.364	0.149	0.090	0.482	0.447	0.035	0.499	0.122	0.114	0.417	0.155	0.053	0.372

TABLE VII
BAYES NETWORK FAULT-PRONENESS PREDICTION

Model	**Error**	**FPR**	**FNR**
ANT	0.181	0.135	0.260
IVY	0.152	0.138	0.232
KC1	0.053	0.388	0.010
JEdit	0.024	0.000	1.000
Eclipse	0.163	0.101	0.294

TABLE VIII
FAULT-PRONENESS PREDICTION COMPARISON

	Error	**FPR**	**FNR**
ANT ROC-2	0.189	0.183	0.198
ANT Alves-30%-2	0.214	0.238	0.172
ANT Bayes	0.181	0.135	0.260
IVY ROC-2	0.193	0.196	0.179
IVY Alves-30%-2	0.188	0.170	0.286
IVY Bayes	0.152	0.138	0.232
KC1 ROC-2	0.174	0.282	0.160
KC1 Alves-30%-2	0.264	0.224	0.269
KC1 Bayes	0.053	0.388	0.010
JEdit ROC-3	0.103	0.096	0.417
JEdit Alves-30%-2	0.286	0.285	0.333
JEdit Bayes	0.024	0.000	1.000
Eclipse ROC-2	0.184	0.154	0.246
Eclipse Alves-30%-1	0.218	0.223	0.209
Eclipse Bayes	0.163	0.101	0.294

in the duplicated classes data could therefore make the model classify too many classes as fault-prone, therefore explaining the high FPR value. Since ROC-2 and Alves-30%-2 seem to work well on most datasets, the exact same metrics and methodologies could be used to find threshold values and perform fault-proneness prediction in a good range of datasets. The advantage of Alves Rankings method over ROC curves is that it could easily be automated to predict test effort (in order to prioritize tests on classes that need to be tested more intensively) without having any fault data history for a system, as it doesn't use faults information for finding threshold values.

V. THREATS TO VALIDITY

This study, as every other empirical software engineering study, has certain threats to validity. First, our study covers only 5 datasets from 5 different systems. This means that the findings of this study cannot be generalized to all software systems. Further tests on many other systems (from different domains and developed in different programming languages) would be needed to generalize obtained results.

Another threat to validity of our study is the way we chose to use 30% of the Alves Rankings distribution to find thresholds using this method. We chose this specific value for finding threshold values as it's the one that yielded the best results across multiple datasets. Of course, we should find a way to determine more objectively that percentage at which a threshold should be set. This methodology could give a generic percentage usable for all systems or a single one per dataset.

Also, the source code metrics used in the study could have been calculated differently for each dataset, as the tools used to calculate the metrics could be different. This could introduce differences in the results of the different datasets.

Another threat to validity is that although faults are listed in the datasets used, no data was found in these datasets defining if a class has been tested or not. Therefore, some bugs may not have been discovered in some classes because they were not tested (or not completely tested). Considering this, our thresholds could have found faults (classes classified as fault-prone) that are yet undiscovered, but were marked as false positives by the classification algorithm.

VI. CONCLUSIONS AND FUTURE WORK

In this study, we wanted to compare two different source code metric threshold calculation methods to achieve fault-proneness prediction. We did so because object-oriented metrics-based models for fault-proneness prediction can provide valuable and understandable insights to prioritize which classes to test more intensively, classes that are likely more fault-prone, in order to ensure the quality of the software system. We therefore calculated metrics thresholds using both techniques and tried to predict faults in different systems (datasets). We also used a machine learning algorithm (Bayes Network) as a baseline method to compare our results. The results obtained were good for both techniques investigated (ROC curves and Alves Rankings).

First, for the ROC curves method, we wanted to investigate it on more systems than the study stating it and wanted to check if binary classification, considered not valid for the studied system by Shatnawi *et al.* [2], could be valid for other datasets. Following the results we got, with error rate, FPR and FNR below 20% for 2 of 5 datasets and below 30% for 2 other datasets, we can conclude that the ROC curves method is not only valid for other datasets than the one studied in [2], but also that binary classification is valid for multiple datasets and often yields good results.

Second, we wanted to investigate if Alves Rankings method could give good results when applied to fault-proneness pre-

diction, as no previous studies were found assessing its validity for fault-proneness prediction. According to the results we obtained, it seems like this method could be used to perform fault-proneness prediction, as it gave acceptable results for 4 of the 5 datasets under study. The results found for Alves Rankings were close to those found for the ROC curves method. The advantage of this method, compared to ROC curves, is that it would be really easy to automate in a new or existing project with no prior fault data history. Further tests on other datasets would be required to generalize the validity of Alves Rankings method, but it seems like a valid choice so far, having tested it on 5 different datasets.

Third, our fault-proneness prediction models using source code metrics thresholds were compared to a machine learning method (Bayes Network) as a baseline. Results showed that the Bayes Network based method gave good results for most datasets. It also gave the lower and best error rate among all models investigated but the FPR and FNR were sometimes better in the other models under study.

Finally, according to this study, both ROC curves and Alves Rankings methods could be considered as good threshold calculation techniques for fault-proneness prediction on multiple datasets. Of course, further tests on other datasets are needed to generalize ROC curves and Alves Rankings method validity to all systems and for determining an objective way to set the percentage of the metric distribution used to set threshold values in the case of Alves Rankings method.

Future works based on this study could therefore consist in testing ROC curves and Alves Rankings method on more systems, but also considering Alves Rankings for building a testing effort orientation model, without using the fault data history of a system. The metrics used for fault-proneness prediction could be changed for using only code design metrics (as SLOC is not a code design metric), therefore being able to make testing effort prediction (and prioritization) based on class diagrams, even before implementation starts, using Alves Rankings method.

REFERENCES

[1] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design", *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, jun 1994.

[2] R. Shatnawi, W. Li, J. Swain, and T. Newman, "Finding software metrics threshold values using ROC curves", *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 22, no. 1, pp. 1–16, jan 2010.

[3] R. Shatnawi, "A Quantitative Investigation of the Acceptable Risk Levels of Object-Oriented Metrics in Open-Source Systems", *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 216–225, mar 2010.

[4] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction", *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 897–910, oct 2005.

[5] B. Isong and E. Obeten, "A Systematic Review of the Emprirical Validation of Object-Oriented Metrics Towards Fault-Proneness Prediction", *International Journal of Software Engineering and Knowledge Engineering*, vol. 23, no. 10, pp. 1513–1540, dec 2013.

[6] Yuming Zhou and Hareton Leung, "Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults", *IEEE Transactions on Software Engineering*, vol. 32, no. 10, pp. 771–789, oct 2006.

[7] R. Malhotra and A. Jain, "Fault Prediction Using Statistical and Machine Learning Methods for Improving Software Quality", *Journal of Information Processing Systems*, vol. 8, no. 2, pp. 241–262, jun 2012.

[8] T. McCabe, "A Complexity Measure", *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, dec 1976.

[9] L. H. Rosenberg, "Applying and Interpreting Object Oriented Metrics", in *Software Technology Conference*, apr 1998.

[10] C. Catal, U. Sevim, and B. Diri, "Clustering and metrics thresholds based software fault prediction of unlabeled program modules", *ITNG 2009 - 6th International Conference on Information Technology: New Generations*, pp. 199–204, apr 2009.

[11] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data", *2010 IEEE International Conference on Software Maintenance*, pp. 1–10, sep 2010.

[12] T. L. Alves, J. P. Correia, and J. Visser, "Benchmark-based aggregation of metrics to ratings", *Proceedings - Joint Conference of the 21st International Workshop on Software Measurement, IWSM 2011 and the 6th International Conference on Software Process and Product Measurement, MENSURA 2011*, pp. 20–29, nov 2011.

[13] G. Abaei, A. Selamat, and H. Fujita, "An empirical study based on semi-supervised hybrid self-organizing map for software fault prediction", *Knowledge-Based Systems*, vol. 74, pp. 28–39, jan 2014.

[14] T. Menzies, R. Krishna, and D. Pryor, "The Promise Repository of Empirical Software Engineering Data", 2016. [Online]. Available: http://openscience.us/repo/

[15] J. Sayyad Shirabad and T. Menzies, "The PROMISE Repository of Software Engineering Databases", 2005. [Online]. Available: http://promise.site.uottawa.ca/SERepository

[16] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches", *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pp. 31–41, may 2010.

[17] R. Malhotra and A. J. Bansal, "Fault prediction considering threshold effects of object-oriented metrics", *Expert Systems*, vol. 32, no. 2, pp. 203–219, apr 2015.

[18] L. Yu, "Using Negative Binomial Regression Analysis to Predict Software Faults: A Study of Apache Ant", *International Journal of Information Technology and Computer Science*, vol. 4, no. 8, pp. 63–70, jul 2012.

[19] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction", *Proceedings of the 6th International Conference on Predictive Models in Software Engineering - PROMISE '10*, p. 1, 2010.

[20] A. Kaur and K. Kaur, "Performance analysis of ensemble learning for predicting defects in open source software", *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pp. 219–225, sep 2014.

[21] M. Jureczko, "Significance of different software metrics in defect prediction", *Software Engineering: An International Journal*, vol. 1, no. 1, pp. 86–95, sep 2011.

[22] J. Moeyersoms, E. Junqué de Fortuny, K. Dejaeger, B. Baesens, and D. Martens, "Comprehensible software fault and effort prediction: A data mining approach", *Journal of Systems and Software*, vol. 100, pp. 80–90, feb 2015.

[23] T. Mende and R. Koschke, "Effort-Aware Defect Prediction Models", *2010 14th European Conference on Software Maintenance and Reengineering*, pp. 107–116, mar 2010.

[24] K. Dejaeger, T. Verbraken, and B. Baesens, "Toward Comprehensible Software Fault Prediction Models Using Bayesian Network Classifiers", *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 237–257, feb 2013.

[25] The Eclipse Foundation, "JDT Core Component", 2016. [Online]. Available: https://eclipse.org/jdt/core/

[26] R. Shatnawi, "Improving software fault-prediction for imbalanced data", *2012 International Conference on Innovations in Information Technology (IIT)*, pp. 54–59, mar 2012.

[27] C. Catal, U. Sevim, and B. Diri, "Software Fault Prediction of Unlabeled Program Modules", *Proceedings of the World Congress on Engineering*, vol. I, pp. 1–6, 2009.

[28] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: An Update", *ACM SIGKDD Explorations Newsletter*, vol. 11, no. 1, p. 10, nov 2009.

ANNEXE B

BOUCHER & BADRI, 2017A

2017 IEEE International Conference on Software Quality, Reliability and Security

Predicting Fault-Prone Classes in Object-Oriented Software: An Adaptation of an Unsupervised Hybrid SOM Algorithm

Alexandre Boucher
Software Engineering Research Laboratory
University of Quebec
Trois-Rivières, Canada
Email: Alexandre.Boucher2@uqtr.ca

Mourad Badri
Software Engineering Research Laboratory
University of Quebec
Trois-Rivières, Canada
Email: Mourad.Badri@uqtr.ca

***Abstract*—Many fault-proneness prediction models have been proposed in literature to identify fault-prone code in software systems. Most of the approaches use fault data history and supervised learning algorithms to build these models. However, since fault data history is not always available, some approaches also suggest using semi-supervised or unsupervised fault-proneness prediction models. The HySOM model, proposed in literature, uses function-level source code metrics to predict fault-prone functions in software systems, without using any fault data. In this paper, we adapt the HySOM approach for object-oriented software systems to predict fault-prone code at class-level granularity using object-oriented source code metrics. This adaptation makes it easier to prioritize the efforts of the testing team as unit tests are often written for classes in object-oriented software systems, and not for methods. Our adaptation also generalizes one main element of the HySOM model, which is the calculation of the source code metrics threshold values. We conducted an empirical study using 12 public datasets. Results show that the adaptation of the HySOM model for class-level fault-proneness prediction improves the consistency and the performance of the model. We additionally compared the performance of the adapted model to supervised approaches based on the Naive Bayes Network, ANN and Random Forest algorithms.**

***Index Terms*—Unsupervised Fault-Proneness Prediction, Self-Organizing Map, Multilayer Perceptron, Naive Bayes Network, Object-Oriented Metrics Threshold Values, Object-Oriented Software Systems.**

I. Introduction

Software quality is becoming more and more important in software development, since complexity, pervasiveness and criticality of software systems are constantly growing [1]. However, to ensure a system is fault-free, it would need to be exhaustively tested, which is in most cases impossible. Therefore, development teams focus their testing effort on parts of the software system which they think are likely the most critical or fault-prone. Nonetheless, this prioritization process can be very lengthy and costly if done manually, in addition to the possibility of leaving some critical parts of the software system untested or not sufficiently tested.

In order to address these problems, fault-proneness prediction models were suggested in literature by many researchers, to predict which parts of a software system are likely the most fault-prone. To do so, these models often use source code metrics as indicators of fault-prone source code. These metrics can capture various attributes of the source code, like size, complexity, coupling, etc. [2]. Many of the fault-proneness prediction models proposed in literature are based on supervised learning algorithms. These models are trained using source code metrics values and fault data. However, fault data history is not always available for a software system or can be very limited, making supervised approaches not always possible to use [3, 4]. This is why semi-supervised (using limited amounts of fault data) and unsupervised (using no fault data) fault-proneness prediction models were also suggested in literature.

In this study, we focus on one particular unsupervised fault-proneness prediction model, HySOM, as suggested by Abaei *et al.* [4]. We decided to focus our study on this particular model as it is a threshold-based approach which is simple to understand and automate. It was proved to be better than previous threshold-based approaches built using the same threshold values [5–8]. In their study, Abaei *et al.* present the HySOM model as a semi-supervised one [4]. However, it can be considered as unsupervised, as it doesn't use any fault data in its construction. Unsupervised fault-proneness prediction models are interesting mainly because they are most of the time easier to implement and automate than supervised approaches.

In this paper, we adapt the HySOM model, originally using function-level source code metrics, to predict fault-prone classes in object-oriented software systems using class-level object-oriented metrics. We think that this adaptation will improve the overall performance of the model and will be more useful for prioritizing the unit testing efforts in object-oriented software systems. We performed an empirical study using 12 public datasets. Results show that the adaptation of the approach for class-level fault-proneness prediction improves the consistency and the performance of the model. We addition-

978-1-5386-0592-9/17 $31.00 © 2017 IEEE
DOI 10.1109/QRS.2017.41

ally compared the adapted model with supervised approaches which are Naive Bayes Network, ANN and Random Forest.

The rest of the paper is organized as follows. Section II presents a summary of fault-proneness prediction models. Most studies presented in this section are works on which the original HySOM study is based on [4]. Section III presents how the original HySOM model works. Section IV presents the experimental design used to adapt the HySOM unsupervised fault-proneness prediction model to work with class-level source code metrics. Section V presents the approach used to adapt the HySOM model. Section VI presents the empirical study we performed to evaluate the prediction performance of the new model and compare it with the original one. It also presents and compares the results with baseline supervised approaches, such as Naive Bayes Network, ANN and Random Forest. Section VII discusses the possible threats of validity of our study. Finally, Section VIII summarizes and concludes this paper, in addition to giving some future work directions based on this study.

II. Related Work

Fault-proneness prediction models aim to help developers and testers focusing their testing effort on fault-prone parts of the source code. Supervised approaches use fault data history to build the prediction model. However, this data is not always available [3, 4] or of good quality, making these approaches difficult to use in many cases. However, cross-project fault-proneness prediction models try to solve this problem by training models on certain software systems and test them on other systems. In this section, a brief introduction to cross-project fault-proneness prediction is first presented. Unsupervised models, which do not use any fault history data, are then presented.

A. Cross-Project Fault-Proneness Prediction

As its name indicates, cross-project fault-proneness prediction trains a model using one or many software systems and then uses it on another system. Using this methodology, no fault data history is needed to predict faults in the system under test. Here are some examples of cross-project fault-proneness prediction models presented in literature.

Zimmermann *et al.* did perform a study in which they tested a fault-proneness prediction model in a cross-project pattern, training the model on one software system and testing it on another one [9]. The authors made several conclusions. First, it is not because a model trained on a system A gives good prediction performance when tested on a system B that a model built on system B would give good prediction performance when tested on system A. Also, they did conclude that similarity of two projects goes beyond the application domain when considering cross-project fault-proneness prediction. In fact, they considered 40 different characteristics to compare similarity between two software systems. These factors help determine if a specific software system should be used for building a fault-proneness prediction model that will be used on another specific software system. However, they also concluded that they should replicate their study with more software systems to generalize their findings, as they only used 12 software systems in their investigations.

In another study, Kamei *et al.* presented an application of cross-project fault-proneness prediction in a JIT (Just-In-Time) setting [10]. They did tests using one or many software systems for training based on their similarity with the system under test. They did conclude that cross-project fault-proneness prediction did not improve the performance of the model over within-project prediction (same training and testing system). They also concluded that using a single software system as training data for cross-project fault-proneness prediction gave significantly lower performance than within-project prediction. Furthermore, they found out that using only software systems similar to the system under test for training the model did not improve the performance, when compared to a model built using all available software systems for training. They also concluded that the model built using all available data did perform similarly to within-project prediction.

However, the use of such supervised approaches can be difficult to completely automate in most companies. The required fault data (even if collected from other software systems) needs to be of good quality, which acquisition can be expensive [11]. Furthermore, cross-project fault-proneness prediction needs to reuse models based on software systems similar to the system under test. Determining if two software systems are similar can be a difficult challenge, which makes cross-project fault-proneness prediction difficult to use in practice.

B. Unsupervised Fault-Proneness Prediction

To simplify the use of fault-proneness prediction models, unsupervised approaches are also suggested in literature and are applicable in any situation, as they don't require fault data. Furthermore, these models are often simpler to build, since they don't need to go through the bug tracker system to collect fault data. Here are some of these unsupervised approaches presented in literature.

Catal *et al.* [5] used threshold values on function-level source code metrics to categorize functions as fault-prone or not. They investigated 2 different approaches with threshold values: (1) by applying them directly on the functions source code metrics, and (2) by applying them on the centroids given by the K-means clustering algorithm. They investigated their approaches on three public datasets, AR3, AR4 and AR5. In addition, they compared their results to a Naive Bayes Network supervised algorithm and concluded that their approach gave acceptable performance. One year later, the same authors investigated the use of X-means as a clustering algorithm over K-means, so the number of clusters can vary [6]. This change of clustering algorithm made the approach easier to automate, while keeping acceptable classification performance results.

Another study by Bishnu & Bhattacherjee [7] used a very similar model to Catal *et al.* in [5, 6]. Their model used K-means clustering algorithm, the same source code metrics and threshold values for building the model. However, they used the Quad-Tree algorithm with a genetic algorithm to

initialize the clusters of the K-means algorithm. According to the authors of this study, the classification performance of their model is comparable to the one given by supervised approaches.

Abaei *et al.* [8] investigated the use of the SOM (Self-Organizing Map) algorithm over K-means and X-means used in [5–7]. However, they used the same source code metrics and threshold values in their study. According to the authors, SOM offered several advantages over K-means clustering algorithm. It has better performance, has fewer chances of finding a local optimum, and the number of clusters can automatically be determined using a defined function [8]. Results given by the model using the SOM algorithm are better than those of studies from Catal *et al.* [5, 6] and Bishnu & Bhattacherjee [7].

In another study, Abaei *et al.* suggested the HySOM model [4], using the same SOM algorithm to cluster source code functions, but additionally using an Artificial Neural Network (ANN) to categorize functions as fault-prone or not. According to the authors, results given by this updated model are better than the results of their previous experiment using SOM only [8]. This model (HySOM) is the approach that is adapted for predicting fault-prone classes in object-oriented software systems in the current study.

A more recent study by Erturk & Sezer [12] suggested an unsupervised fault-proneness prediction model used in conjunction with a supervised one, to predict fault-prone source code in iterative development processes. However, their approach has 2 main drawbacks to be used: (1) the approach can't be completely automated, since an expert is required for initializing the unsupervised model, and (2) it requires an expert with good knowledge about Fuzzy Inference Systems, which may not be available for most software development companies.

Among these unsupervised fault-proneness prediction models presented in literature, we decided to investigate the HySOM model suggested by Abaei *et al.* [4] and adapt it to work with object-oriented source code metrics at class-level granularity. We chose this model in particular because it is the one that gave the best classification performance according to the authors. We adapted the model for fault-proneness prediction in object-oriented software systems because object-oriented source code metrics were widely validated for usage in fault-proneness prediction studies [2], and that unit testing in object-oriented systems focusses testing efforts on class-level. Since most systems developed nowadays are object-oriented software systems, this unsupervised approach would be usable for most systems. Furthermore, since object-oriented source code metrics are related to fault-proneness [2], we think our adaptation will improve the classification performance of the original HySOM model [4].

III. The Original HySOM Model

In this section, we present the original HySOM model as it is presented in the study of Abaei *et al.* [4]. It is important to understand how the model works to therefore understand how it is adapted for class-level usage.

TABLE I
Source Code Metrics and Threshold Values Used in HySOM Study.

Metric	Description	Threshold
LOC	Number of lines of code, including blank and commented lines	65
CC	Cyclomatic complexity (number of independent paths in the control flow graph)	10
UOp	Number of unique operators	25
UOpnd	Number of unique operands	40
TOp	Total number of operators	125
TOpnd	Total number of operands	70

A. Prerequisites of the HySOM Model

The original HySOM model uses three main elements: function-level source code metrics, threshold values for these metrics and the values of the source code metrics for the system under investigation.

1) Source Code Metrics: In the original HySOM model, the authors used six function-level source code metrics to predict fault-prone functions (number of lines of code, cyclomatic complexity, and four other metrics from the suite of Halstead complexity measures [13]). These metrics are presented in Table I, along with the threshold values used for each one.

2) Threshold Values: One critical component of the HySOM model is the threshold values used for each of the source code metrics. Each metric has one associated threshold value. When the corresponding metric value is above the associated threshold value, the source code function is considered fault-prone according to this particular source code metric. In the HySOM model, if three or more of the six source code metrics exceed their threshold values, the function is considered as fault-prone, otherwise, it is not considered as fault-prone. See Table I for the threshold values used in the HySOM study [4].

According to Abaei *et al.*, these threshold values were calculated using the PREDICTIVE tool, developed by ISM (Integrated Software Metrics). However, there is no mention on how this tool calculated the threshold values in any of the studies stating it [3, 5–7] or on ISM website, which is down for the moment, as mentioned previously by Catal *et al.* [5][1].

3) Datasets: Most fault-proneness prediction studies use public datasets containing source code metrics values and fault data to test their models [2]. The original HySOM study did the same, by reusing three Turkish datasets, namely AR3, AR4 and AR5, and five datasets developed by NASA: CM1, KC1, KC2, MW1 and PC1 [4]. All the investigated datasets were collected from software systems written in C language, except for KC1 and KC2, which were written in C++. All these datasets can be obtained from the PROMISE repository [14], along with many other fault-proneness prediction datasets.

[1]Integrated Software Metrics Inc. - http://www.ismwv.com

B. Structure of the HySOM Model

The HySOM model is divided in two main parts (or algorithms), which are a Self-Organizing Map (SOM) and an Artificial Neural Network (ANN or Multilayer Perceptron) [4]. The first part (SOM) clusters the functions presenting similar source code metrics values, while the ANN part uses the SOM output data to train itself. The ANN is then used to predict which functions are fault-prone and which ones are not.

1) First Part - SOM: The SOM algorithm is a clustering algorithm which regroups similar vectors of data, like the well-known K-means algorithm. It is often used to reduce the number of dimensions of the data to a limited number (usually two) and can easily give a 2D graphical representation of the clusters. The SOM algorithm is initialized with a given number of neurons in height and width, producing a rectangular dimension. Each neuron (or cluster) is a vector of the same dimensions as the input vectors. Once the training phase of the SOM algorithm is done, these neurons represent the centroids of the clusters. The input vectors are then clustered to the closest neuron of each one, using a distance function like the Euclidean distance.

In the original HySOM study, the SOM algorithm is initialized with a height and width given by a specific formula, using the number of functions in the system and the number of metrics used for prediction (which is 6) [4]. The input vectors of the SOM algorithm are in fact the source code metrics values of each function of the system. In the HySOM model, once the clustering is done, dead neurons (being neurons which don't have any input vectors associated to them) are removed from the map. Neurons that are still *alive* are then considered for the rest of the training phase of the model. As these neurons' weights represent source code metrics values too, these values are checked against the threshold values mentioned earlier. If three metrics or more exceed their threshold values, the cluster is marked as fault-prone. Otherwise, it is marked as not fault-prone.

2) Second Part - ANN: The ANN (Artificial Neural Network or Multilayer Perceptron) is an algorithm representing a non-linear function using a directed graph. It therefore yields a better classification potential than linear regression. In the original HySOM study, the structure of the ANN used is not detailed much [4]. However, from what is said in the paper [4], it is probably a feedforward Multilayer Perceptron using the backpropagation algorithm for the training phase. It is, however, known that the network uses 6 neurons in the input layer and only one output layer neuron, probably using the sigmoid activation function. However, the number of hidden layers used by the model is not mentioned.

In the HySOM model, the ANN uses the weight vectors outputted by the SOM algorithm and the predicted fault-proneness given by threshold values as training data. Each weight of a SOM cluster (neuron) is used as an input neuron of the ANN algorithm, while the predicted fault-proneness is the target result used for training the ANN. Once the training of the ANN is completed, the HySOM model is considered trained and the ANN can then be used to directly classify functions as fault-prone or not.

TABLE II
CONFUSION MATRIX STRUCTURE.

	Actual	
Classified	Faulty	Not-faulty
Faulty	True positives (TP)	False positives (FP)
Not faulty	False negatives (FN)	True negatives (TN)

C. HySOM Evaluation Method

In the study describing HySOM, the authors used a confusion matrix (classification table) structured as presented in Table II to describe the prediction performance [4]. This matrix gives the number of true and false positives and negatives, where positives are functions classified as fault-prone and negatives are functions classified as not fault-prone. In fact, they used three classification metrics calculated using this matrix, which are: FPR (False Positive Rate), FNR (False Negative Rate) and Error Rate. These metrics are calculated as follows:

$$\text{Error Rate} = \frac{FP + FN}{FP + FN + TP + TN} \tag{1}$$

$$FPR = \frac{FP}{FP + TN} \tag{2}$$

$$FNR = \frac{FN}{FN + TP} \tag{3}$$

For all three classification metrics used, the lower each one is, the better the prediction is. In the original study, the model is tested by using a 66% stratified approach, which means that 66% of a dataset is randomly chosen as training data and the remaining 34% is used as testing data [4].

1) G-mean Evaluation Metric: Another classification metric that is used in fault-proneness prediction (but not in the original HySOM study) is the g-mean metric (geometric mean) [15, 16]. This evaluation metric is useful in these studies because the fault-proneness prediction datasets are often imbalanced (not having approximately 50% of faulty instances and another 50% of not faulty instances) [15]. In a software system, there should be a lot more not faulty instances than faulty ones [15]. This evaluation metric can also be used alone to compare prediction performance between different models. We present this evaluation metric because it is later used in the paper to describe the performance of the investigated models.

The g-mean evaluation metric uses two other evaluation metrics for its calculation, which are the accuracy of positives (TPR) and the accuracy of negatives (TNR) [16]. These three metrics are calculated as follows:

$$TPR = 1 - FNR = \frac{TP}{TP + FN} \tag{4}$$

TABLE III
CLASSIFICATION PERFORMANCE OF THE ORIGINAL HYSOM MODEL.

Dataset	Error Rate	FPR	FNR
AR3	0.1481	0.1501	0.1111
AR4	0.1250	0.1001	0.2000
AR5	0.1267	0.1690	0.0000
CM1	0.0810	0.0500	0.6250
KC1	0.1847	0.1069	0.6166
KC2	0.1666	0.1393	0.2647
MW1	0.0875	0.0547	0.4285
PC1	0.1325	0.0921	0.7138

TABLE IV
REPRODUCED HYSOM MODEL PERFORMANCE USING 10-FOLD CROSS-VALIDATION.

Dataset	FPR	FNR	g-mean
AR3	0.291	0.250	0.729
AR4	0.115	0.450	0.698
AR5	0.357	0.125	0.750
CM1	0.150	0.667	0.532
KC1	0.038	0.852	0.377
KC2	0.043	0.632	0.593
MW1	0.083	0.613	0.596
PC1	0.106	0.711	0.509
Mean	0.148	0.537	0.598

$$TNR = 1 - FPR = \frac{TN}{TN + FP} \quad (5)$$

$$g\text{-}mean = \sqrt{TPR * TNR} \quad (6)$$

Contrarily to FPR, FNR and Error Rate metrics, the higher these three metrics are, the better it is. When evaluating models, we used the FPR, FNR and g-mean metrics to describe the prediction performance. We dropped the Error Rate evaluation metric as it doesn't give valuable insights about the prediction performance in imbalanced datasets, contrarily to the g-mean metric [15, 16].

D. HySOM Prediction Results

Abaei *et al.* compared the HySOM performance with five unsupervised and three supervised approaches [4]. They used standalone SOM, two-stage, one-stage, Quad-Tree based K-means and two-stage X-means approaches as unsupervised comparison references. As supervised approaches, they used Naive Bayes Network, Random Forest and ANN. However, for brevity, only results considering the HySOM model are presented in our paper (see Table III). In their paper, the authors concluded that the HySOM model was the one that performed best for most datasets [4].

However, we think that these results may be improved, since the FNR may sometimes be high (as for CM1, KC1, MW1 and PC1 datasets). Furthermore, we think that using 10-fold cross-validation instead of a 66% stratified approach for testing the model would give more relevant results. The stratified approach performance highly depends on which 66% of the dataset is used for training data, which may impact results. However, the 10-fold cross-validation approach reduces this impact by using 90% of the dataset as training data and the remaining 10% as testing data. The experiment is then executed 9 other times, each time using a different 10% testing set, ultimately using the whole dataset as training and testing data.

We therefore decided to reproduce the original HySOM model and test it using 10-fold cross-validation. This reproduction of the HySOM model is also later used as a comparison baseline with the HySOM model adapted for class-level prediction. Table IV presents the results we obtained from this experiment.

As we can see from the reproduced HySOM model performance, the FNR is always high for NASA datasets (CM1, KC1, KC2, MW1 and PC1). As to the FPR, it is always low. However, the goal of fault-proneness prediction and classification in general is to get a good balance between false positives and negatives, which is not the case here. This is part of the reason why we decided to adapt the model to use object-oriented source code metrics, to investigate if it could improve its performance.

IV. EXPERIMENTAL DESIGN

In this section, we present the experimental design used for testing the proposed adapted HySOM model.

A. Choice of Class-Level Source Code Metrics

In order to adapt the HySOM model to work with object-oriented metrics at class-level granularity, the source code metrics used for constructing the model needed to be changed. We therefore chose four source code metrics to construct the model, based on two main elements.

First, we performed an univariate logistic regression analysis on the Apache ANT 1.7 fault-proneness prediction dataset. This dataset is widely used in other fault-proneness prediction studies [16–19] and is easy to obtain via the PROMISE Repository [14]. The analysis was performed to find relationships between the source code metrics and the faultiness of classes. This analysis to validate the relationship between source code metrics and fault-proneness was not performed in the original study [4]. For the analysis, we considered the SLOC metric and Chidamber & Kemerer source code metrics [20].

In our analysis, we wanted to consider the number of faults in each class. To do so, we followed a simple methodology previously used in other studies [21, 22], consisting in duplicating the classes in the datasets which contain more than one fault. For example, if a class contains 3 faults, it would be duplicated so the class would be present in the dataset 3 times. The advantage of this approach is that the analysis algorithm is much more accurate, since a class containing 10 faults does not have the same impact on the analysis result. For example,

TABLE V
UNIVARIATE LOGISTIC REGRESSION ANALYSIS RESULTS FOR APACHE ANT 1.7.

Metric	p-value	Wald Chi-square	R^2	AUC
SLOC	**< 0.0001**	179.021	0.509	0.886
CBO	**< 0.0001**	38.306	0.112	0.794
RFC	**< 0.0001**	206.692	0.539	0.886
WMC	**< 0.0001**	153.591	0.434	0.856
LCOM	**< 0.0001**	89.958	0.306	0.837
DIT	0.126	2.338	0.003	0.541
NOC	0.434	0.612	0.001	0.471

if one very faulty class (containing 10 faults) is considered as not fault-prone, this would make 10 false negatives considered instead of simply one. This simple process was applied on both training and testing data.

Table V presents the results obtained from the univariate logistic regression analysis performed using the XLSTAT tool[2]. The AUC column gives the Area Under Curve given by the ROC Curve plotted using the univariate logistic regression model (as calculated by XLSTAT). It is used to check if the source code metric alone gives a good prediction potential.

The first conclusion drawn from these results is that SLOC, CBO, RFC, WMC and LCOM metrics are considered relevant according to their *p-value*, because it is below the 5% confidence level. They are also good predictors of fault-proneness according to their AUC value. However, LCOM value was not kept, even if it is considered relevant. We chose to do so because it is a source code metric with many known variants, which give very different results for the same metric. It is therefore difficult to assess that LCOM will be calculated in the same way for all datasets chosen for investigation. Furthermore, in a study by Isong & Obeten [2] considering many fault-proneness prediction studies on object-oriented systems, LCOM was not retained as a metric relevant to fault-proneness prediction according to most studies revised. In the univariate logistic regression results, we see that the CBO metric has a R^2 that could be considered low (0.112), but we decided to keep it anyway, since its *p-value* is below the confidence level set and its Area Under Curve is good.

To summarize, four metrics were retained as good predictors of fault-proneness, which are SLOC (Source Lines of Code), in addition to CBO (Coupling Between Objects), RFC (Response For a Class) and WMC (Weighted Methods per Class) [20]. Each metric of the retained subset yields different information. SLOC is a size metric, WMC a complexity metric, CBO a coupling metric and RFC is another complexity/coupling metric. SLOC gives the number of source lines of code in a given class while WMC gives the sum of the cyclomatic complexity of each method in a given class, where the cyclomatic complexity is a metric defined by McCabe giving the number of linearly independent paths in source code [23]. CBO gives the number of classes to which a given class is coupled and finally, RFC gives the number of methods that can potentially be executed when a message is received by a given class (which is the number of methods in the class added to the number of methods that can be called by those methods) [20].

[2]XLSTAT - https://www.xlstat.com

B. Choice of Datasets

In order to test our HySOM model adaptation, we needed new fault-proneness prediction datasets, as the ones used in the original HySOM study contained function-level source code metrics. We tried to reuse datasets built on the same software systems, but unfortunately, since most of the eight original datasets used were written in a procedural programming language (six software systems written with C programming language), only one was found containing class-level data. The KC1 dataset, which was written in C++, was found with the class-level data we seek. However, we don't know if the function-level dataset and the class-level one were both produced from the same version of the KC1 software system, as we did not find any information mentioning this.

Since only one of the original software systems investigated was found having class-level fault-proneness data, we needed to choose other datasets too. We therefore used twelve datasets in total built on eight different object-oriented software systems. These datasets are Apache ANT (versions 1.3, 1.4, 1.5, 1.6 and 1.7), Apache IVY 2.0, Apache Lucene 2.4, Apache POI 3.0, Apache TOMCAT 6.0, KC1, JEdit 4.3 and Eclipse JDT Core. ANT (all versions), IVY, LUCENE, POI, TOMCAT and JEdit are all available from the PROMISE Repository [14]. KC1 dataset with class-level metrics can be found on the University of Ottawa PROMISE Repository [24] and Eclipse JDT Core dataset is available from the study of D'Ambros *et al.* [25]. All of these software systems were written with the Java programming language, except KC1, which was written with C++. All of these datasets were used in previous fault-proneness prediction studies and are available online.

V. PROPOSED APPROACH

In this section, the approach used to adapt the HySOM model for working with class-level object-oriented source code metrics is presented. Note that even if the approach is specifically presented for the HySOM model, it can be used with any other thresholds-based approach to change the source code metrics and threshold values used.

Figure 1 shows how the adapted HySOM model is trained and which parts of it are edited for object-oriented class-level fault-proneness prediction (marked in red). We remark that the source code metrics used are changed for the four newly selected ones (LOC, CBO, RFC, WMC) and that the associated threshold values changed. The other parts of the model keep the exact same behavior. The rest of this section will further explain how the adaptation of the model is performed.

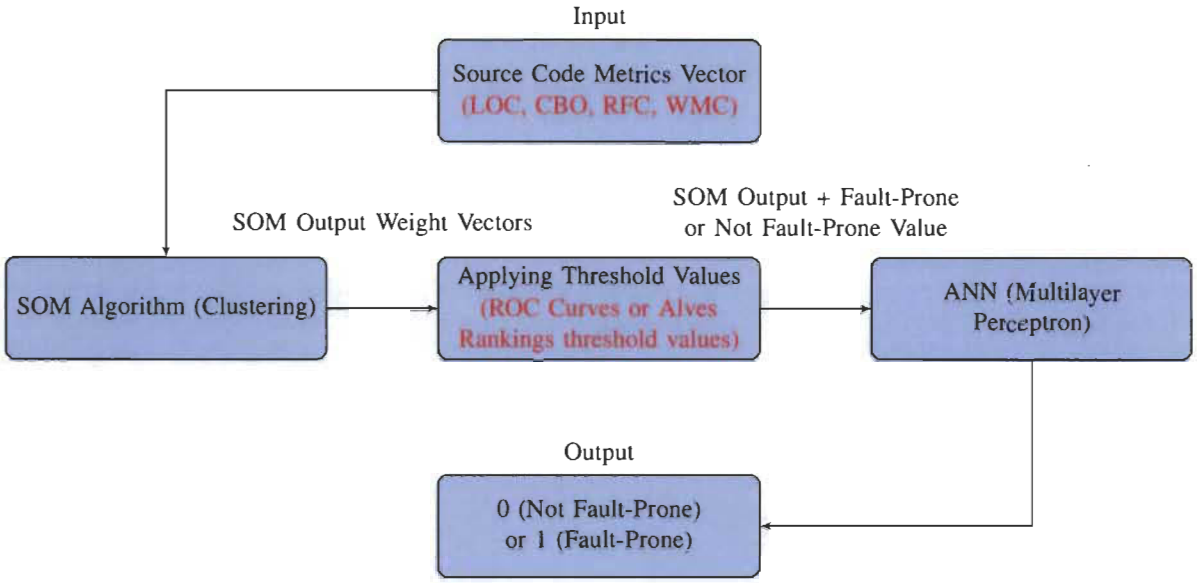


Fig. 1. Adapted HySOM Training Workflow.

A. *Why Adapt the HySOM Model for Class-Level Prediction?*

We decided to adapt the HySOM fault-proneness prediction model to work with object-oriented source code metrics at class-level for two main reasons.

First, the original HySOM model using function-level and non object-oriented source code metrics did give high FNR but low FPR in its prediction results. Since object-oriented source code metrics were validated as related to fault-proneness in many studies [2, 15, 16, 21, 26], we thought it could be a good way to improve the model's performance.

The second reason for adapting the model is that nowadays, object-oriented programming is widely used for new software system development. Object-oriented software systems are often tested using unit tests, which test units of the software system, which are classes in object-oriented software systems. This justifies the need to perform the prediction at class-level, since classes are directly tested in object-oriented software systems, not methods.

B. *Choice of Threshold Values*

The threshold values play an important role in the prediction performed by the HySOM model. However, we can't calculate the threshold values for the newly selected metrics the same way Abaei *et al.* calculated them, as they used the PREDICTIVE tool [4], which is no longer available. We therefore needed other techniques to calculate threshold values for source code metrics and we investigated two of them: ROC Curves and Alves Rankings. In a previous study we made, we investigated both techniques for fault-proneness prediction and both were considered good [27]. The ROC Curves method gave slightly better results than Alves Rankings, but probably because it is a supervised approach, which considers fault data history for calculating threshold values [27].

Using these techniques, we calculated different threshold values for each source code metric and project, contrarily to the original HySOM study, which used the same threshold values for different software systems [4]. In fact, we decided to calculate threshold values on a system and even on a version basis. We did so because threshold values calculated from one software system may not be applicable for another one, as mentioned by Shatnawi *et al.* [28] and Zhou & Leung [21]. Furthermore, Shatnawi *et al.* mentioned that threshold values could be different from one version of a software system to another [28].

1) ROC Curves: The ROC Curves technique to calculate threshold values was suggested by Shatnawi [28]. This simple technique works as a supervised one, as it needs the fault data of each class to calculate the threshold values. Threshold values could be calculated from one dataset and then applied on another one to work in an unsupervised context, but such experiment is outside the scope of this paper. However, it is very simple to use, as it only consists of plotting a ROC (Receiver Operating Characteristic) Curve and taking the threshold value maximizing both 1 - specificity and sensitivity [28].

In our study, we calculated the threshold values using ROC Curves technique with a little variant. We considered the number of faults when plotting the ROC Curve, as previously done in the univariate logistic regression analysis performed to determine the source code metrics to use (and as done in [21, 22]).

2) Alves Rankings: Alves *et al.* suggested a technique to calculate threshold values in an unsupervised way to describe quality of classes in object-oriented software systems [29]. Since the method didn't have a name in the paper [29], we will refer to it as Alves Rankings for brevity. This simple method

uses the relative weight of a class in a system (the weight being given by the SLOC metric) to calculate threshold values. A percentage of the code to represent with a threshold value is determined by the user of the technique. Different threshold values can then be calculated using different percentages of the code to target.

In the original study, they used the ones given at 70%, 80% and 90% of the source code metrics distribution to describe different quality levels of classes. However, in our study, after multiple tests done with Alves Rankings threshold values, we decided to keep the ones calculated at 30% of the source code metrics distribution. We determined that the threshold values given by this percentage gave the best fault-proneness prediction performance results for most datasets.

VI. Experimental Results and Discussion

In this section, we present the fault-proneness prediction performance results given by our adaptation of the HySOM model and discuss the results obtained. Results given by the supervised approaches (Naive Bayes Network, ANN and Random Forest) are also presented and compared in this section.

A. Results

To calculate results of the adapted HySOM model, we considered two distinct experiments, one considering the threshold values given by the ROC Curves technique (see Table VI) and another one considering the ones given by Alves Rankings (see Table VII). For each threshold calculation technique, we present the results when classes are classified as fault-prone when at least one source code metric exceeds threshold values (HySOM-class-1), when at least two source code metrics exceed threshold values (HySOM-class-2), along with HySOM-class-3 and HySOM-class-4.

We did so to investigate how many source code metrics should exceed threshold values before considering a class as fault-prone. In the original HySOM study [4], the authors mention that they chose 3 as the number of metrics to exceed threshold values after having run multiple tests, although the results are not presented. However, in our study, we decided to include the results of these tests. All results were calculated using one run of 10-fold cross-validation.

As mentioned earlier, we also investigated different supervised models based on the Naive Bayes Network, ANN (Multilayer Perceptron) and Random Forest algorithms to compare our results. We reused the same object-oriented datasets and source code metrics to run the supervised approaches. The supervised models were built using the Weka tool[3], which can run different machine learning and data mining algorithms [30]. These approaches were chosen because they are compared to the HySOM model in the original study [4]. Additionally, we used the Bayes Network (not the naive one) in a previous study on fault-proneness prediction [27] and it yielded good results. We therefore think Naive Bayes Network can give good results too, and also decided to investigate ANN and Random Forest. The results we got using the supervised algorithms and 10-fold cross-validation are presented in Table VIII.

[3]Weka - http://www.cs.waikato.ac.nz/ml/weka/

B. Discussion

The first observation that we can make is that results are better using HySOM-class-1 and HySOM-class-2 experiments according to the g-mean values obtained. This observation is true for both threshold calculation techniques investigated. When using HySOM-class-3 and HySOM-class-4 experiments, the performance is globally less desirable, since the g-mean value is lower. Furthermore, the FNR is globally higher and significantly higher than the FPR metric in HySOM-class-3 and HySOM-class-4 experiments. This results in a prediction performance which is not desirable, because a lot of false negatives and very few false positives are present. The FPR and FNR metrics should be more balanced, as in the HySOM-class-1 experiment (using either ROC Curves or Alves Rankings threshold values).

Considering the KC1 dataset, which is present in the original HySOM study and our adapted model, we see that the results we got with ROC Curves threshold values are very good (when looking at HySOM-class-1 and HySOM-class-2). The FNR is much lower than the one obtained from the 10-fold cross-validation experiment performed on the original model. Even if the FPR is a bit higher than in the original model, the prediction performance is still good, with a high g-mean value. As to Alves Rankings technique, it gave a higher FNR than the ROC Curves threshold values, but the g-mean is still higher than the original HySOM model for HySOM-class-1 experiment (which is good).

If we look at the other datasets results, we see that the FNR is much lower than the original HySOM approach, especially when looking at ROC Curves HySOM-class-1, ROC Curves HySOM-class-2 and Alves Rankings HySOM-class-1 experiments. Even with this lower FNR, the FPR is not too high, making the prediction more balanced between false positives and false negatives than the original approach. We can also see that when considering these 3 experiments, the g-mean metric doesn't get as low as the original approach (0.377 for KC1 dataset).

Results obtained using the adapted HySOM model are also more consistent, with less important differences between results from one dataset to another. For example, the original approach g-mean values ranged from 0.377 to 0.750, while results for the adapted model ranged from 0.547 to 0.736 for the HySOM-class-1 and from 0.569 to 0.755 for the HySOM-class-2 models using ROC Curves threshold values. The adapted approach using Alves Rankings threshold values also gives more consistent results than the original HySOM model, with g-mean values ranged from 0.497 to 0.707 for HySOM-class-1 and from 0.504 to 0.758 for HySOM-class-2. The prediction performance results given by the adapted model do not get as low as the ones sometimes given by the original HySOM approach.

TABLE VI
HYSOM MODEL PERFORMANCE USING CLASS-LEVEL SOURCE CODE METRICS AND ROC CURVES THRESHOLD VALUES.

	HySOM-class-1			HySOM-class-2			HySOM-class-3			HySOM-class-4		
Dataset	**FPR**	**FNR**	**g-mean**	**FPR**	**FNR**	**g-mean**	**FPR**	**FNR**	**g-mean**	**FPR**	**FNR**	**g-mean**
ANT 1.3	0.429	0.300	0.632	0.124	0.400	0.725	0.124	0.450	0.694	0.171	0.350	0.734
ANT 1.4	0.457	0.450	0.547	0.348	0.500	0.571	0.268	0.650	0.506	0.239	0.675	0.497
ANT 1.5	0.364	0.406	0.615	0.261	0.563	0.569	0.134	0.781	0.435	0.054	0.813	0.421
ANT 1.6	0.313	0.304	0.691	0.232	0.326	0.720	0.139	0.315	0.768	0.131	0.500	0.659
ANT 1.7	0.254	0.380	0.680	0.173	0.446	0.677	0.133	0.440	0.697	0.111	0.560	0.625
IVY	0.433	0.225	0.663	0.141	0.425	0.703	0.157	0.375	0.726	0.077	0.575	0.626
LUCENE	0.409	0.493	0.548	0.263	0.542	0.581	0.175	0.665	0.526	0.088	0.813	0.413
POI	0.360	0.153	0.736	0.298	0.285	0.709	0.230	0.327	0.720	0.186	0.452	0.668
TOMCAT	0.270	0.455	0.631	0.152	0.390	0.719	0.151	0.468	0.672	0.085	0.636	0.577
KC1	0.353	0.200	0.719	0.224	0.267	0.755	0.306	0.450	0.618	0.082	0.800	0.428
JEdit	0.279	0.455	0.627	0.179	0.455	0.669	0.116	0.545	0.634	0.054	0.818	0.415
Eclipse	0.244	0.437	0.652	0.149	0.408	0.710	0.095	0.447	0.708	0.057	0.568	0.638
Mean	0.347	0.355	0.645	0.212	0.417	0.676	0.169	0.493	0.642	0.111	0.630	0.559

TABLE VII
HYSOM MODEL PERFORMANCE USING CLASS-LEVEL SOURCE CODE METRICS AND ALVES RANKINGS THRESHOLD VALUES.

	HySOM-class-1			HySOM-class-2			HySOM-class-3			HySOM-class-4		
Dataset	**FPR**	**FNR**	**g-mean**	**FPR**	**FNR**	**g-mean**	**FPR**	**FNR**	**g-mean**	**FPR**	**FNR**	**g-mean**
ANT 1.3	0.371	0.250	0.687	0.324	0.150	0.758	0.257	0.250	0.746	0.181	0.400	0.701
ANT 1.4	0.449	0.375	0.587	0.355	0.575	0.524	0.239	0.600	0.552	0.239	0.625	0.534
ANT 1.5	0.598	0.156	0.583	0.276	0.406	0.656	0.245	0.406	0.669	0.211	0.563	0.588
ANT 1.6	0.378	0.315	0.652	0.181	0.413	0.693	0.147	0.533	0.632	0.151	0.522	0.637
ANT 1.7	0.368	0.247	0.690	0.188	0.464	0.660	0.155	0.446	0.684	0.083	0.608	0.599
IVY	0.260	0.325	0.707	0.157	0.400	0.711	0.119	0.475	0.680	0.074	0.575	0.627
LUCENE	0.336	0.360	0.652	0.153	0.700	0.504	0.109	0.719	0.500	0.102	0.714	0.506
POI	0.280	0.335	0.692	0.118	0.573	0.614	0.056	0.676	0.553	0.081	0.719	0.508
TOMCAT	0.305	0.299	0.698	0.147	0.429	0.698	0.134	0.494	0.662	0.117	0.558	0.625
KC1	0.188	0.500	0.637	0.082	0.600	0.606	0.106	0.583	0.610	0.106	0.617	0.585
JEdit	0.457	0.545	0.497	0.249	0.455	0.640	0.200	0.545	0.603	0.183	0.636	0.545
Eclipse	0.211	0.451	0.658	0.085	0.524	0.660	0.078	0.510	0.672	0.044	0.631	0.594
Mean	0.350	0.347	0.645	0.193	0.474	0.644	0.154	0.520	0.630	0.131	0.597	0.588

Both ROC Curves and Alves Rankings techniques calculated threshold values which seem to give good prediction performance, even if the ROC Curves approach gives better results in the HySOM-class-2 experiment. However, the ROC Curves technique is supervised while Alves Rankings approach is not, which could explain this small difference in prediction performance.

If we consider the results we got using the supervised approach Naive Bayes Network, we first see that results are better when using the adapted HySOM model. The FNR value given by the Naive Bayes Network model is often high, therefore giving a poor overall classification performance. In fact, the Naive Bayes Network results presented in the original HySOM study [4] somewhat gives the same results, with an often high FNR value. Results of the Naive Bayes Network algorithm are close to the ones obtained by the original HySOM model, but the supervised model gives more consistent results, with less variations from one dataset to another. More importantly, when considering the adapted HySOM model (especially HySOM-class-1 and HySOM-class-2 experiments), the results are generally better using the unsupervised approach with either ROC Curves or Alves Rankings threshold values.

When considering results from the supervised ANN model, we first denote that prediction performance is not good for

TABLE VIII
PERFORMANCE OF THE DIFFERENT SUPERVISED MODELS.

	Naive Bayes			ANN			Random Forest		
Dataset	**FPR**	**FNR**	**g-mean**	**FPR**	**FNR**	**g-mean**	**FPR**	**FNR**	**g-mean**
ANT 1.3	0.086	0.650	0.566	0.076	0.650	0.569	0.095	0.650	0.563
ANT 1.4	0.326	0.550	0.551	0.036	1.000	0.000	0.145	0.775	0.439
ANT 1.5	0.077	0.500	0.679	0.046	0.656	0.573	0.046	0.625	0.598
ANT 1.6	0.085	0.543	0.646	0.116	0.402	0.727	0.100	0.413	0.727
ANT 1.7	0.071	0.578	0.626	0.079	0.548	0.645	0.074	0.518	0.668
IVY	0.074	0.575	0.627	0.019	0.775	0.470	0.032	0.725	0.516
LUCENE	0.095	0.660	0.555	0.372	0.300	0.663	0.460	0.227	0.646
POI	0.081	0.730	0.499	0.379	0.167	0.719	0.323	0.160	0.754
TOMCAT	0.068	0.623	0.593	0.001	0.974	0.161	0.013	0.857	0.376
KC1	0.094	0.600	0.602	0.200	0.417	0.683	0.271	0.350	0.689
JEdit	0.025	0.727	0.516	0.002	1.000	0.000	0.006	1.000	0.000
Eclipse	0.038	0.650	0.580	0.046	0.592	0.624	0.067	0.529	0.663
Mean	0.093	0.616	0.587	0.114	0.624	0.486	0.136	0.569	0.553

ANT 1.4 and JEdit datasets, for which the FNR metric is at its maximum value (which is 1). Furthermore, the FNR is generally high and the FPR generally low. The overall prediction is not that good for most datasets, but for other ones like ANT 1.6 and POI, the prediction is not that bad. However, Naive Bayes gives better results than ANN for fault-proneness prediction.

As to the supervised Random Forest model, it performed better than ANN but Naive Bayes generally gave better prediction, according to the average g-mean value. However, the FNR is generally lower for the Random Forest algorithm, with generally reasonable FPR values. The prediction was very unbalanced for TOMCAT and JEdit datasets, with very high FNR. Still, Random Forest gave better results than ANN but did not outperform Naive Bayes in general.

Following the results obtained using the adapted HySOM model with class-level and object-oriented source code metrics, we can conclude that it gives more consistent and better results than the original HySOM model. When considering the three investigated supervised models, we remark that the Naive Bayes Network algorithm gave the best prediction results. It additionally gave more consistent results than ANN and Random Forest supervised algorithms. Finally, we can conclude that the adapted HySOM model gives better results than the simple supervised models investigated.

VII. THREATS TO VALIDITY

Like other empirical software engineering studies, our study has certain threats to validity. First, we tested the adapted HySOM model on different software systems than in the original study, which may impact the results. However, we tried to find datasets with class-level source code metrics built on the same software systems, but only one built on KC1 was found. We therefore tested the model on 12 different software systems to alleviate this problem and test the consistency of the results.

Another threat to validity is that there could be differences in the way source code metrics are calculated in each dataset investigated. However, we tried to reduce these differences by making sure the same metrics have the same meaning in each dataset. For example, we recalculated the WMC metric in certain datasets, because it was considered as the number of methods in certain ones and not the sum of the cyclomatic complexity of all methods, as we wanted it. We further reduced that risk by reusing widely used public datasets. There could also be errors in these fault-proneness datasets, but this is outside of our control and this threat is present in all other studies on fault-proneness prediction.

One other threat to validity is that there could be differences in our implementation of the HySOM model and the one used in the original HySOM study. However, we tried to minimize such differences by closely reproducing all mentioned elements in the study. In order to compare the adapted model results, we reproduced the original approach with the same source code metrics, datasets and parameters using 10-fold cross-validation. Comparisons between the original and adapted models were therefore made on the same implementation of the HySOM model.

VIII. CONCLUSIONS AND FUTURE WORK

In our study, we adapted the HySOM model [4], originally working at function-level granularity, to work at class-level granularity with object-oriented source code metrics. We did so to improve the prediction performance of the model and because object-oriented software systems are usually tested at class-level.

To adapt the model, we first reproduced the HySOM model of the original study, reusing the same datasets and parameters.

In order to have a common comparison baseline, we used 10-fold cross-validation to test the original model and the adapted one. This also gave more stable results than the original stratified approach at 66%, which gave highly variating results depending on which part of the dataset was used for training. We suggested a way to adapt the HySOM model that can easily be reproduced and generalized for use with other source code metrics than the ones chosen in this study (SLOC, CBO, RFC and WMC). The threshold values of each source code metric were calculated for each investigated software system, using two different techniques: ROC Curves [28] and Alves Rankings [29].

Our results showed that the ROC Curves method gave slightly better results than Alves Rankings, but it has the downside that it needs datasets with fault data to calculate threshold values. On the other side, Alves Rankings can easily calculate threshold values for any datasets without fault data information, therefore keeping the completely unsupervised nature of the model. For the KC1 dataset, which was investigated with the original and adapted HySOM models, performance was much better using the adapted one. As to the 11 other datasets investigated with the adapted model, results were found to be more consistent and better using the adapted approach than the original one, as presented in Section VI.

We also compared the adapted HySOM model with three supervised approaches, namely Naive Bayes Network, ANN and Random Forest. We found out that the adapted approach gave better results than the supervised approaches, whether ROC Curves or Alves Rankings threshold values were used.

Finally, our study proposes a possible adaptation of the HySOM model for usage at class-level granularity, using object-oriented source code metrics. The suggested adaptation method can be generalized for usage with other source code metrics or other threshold values based fault-proneness prediction models.

As a future work, it would be interesting to try the adapted HySOM model on more datasets, to better generalize its prediction potential. Different variants of the HySOM model could also be produced, to try to improve its performance. It would also be interesting to adapt the HySOM model to work with design source code metrics only, making it usable even before the software system implementation starts.

As other future works, it would be interesting to reuse elements from other studies in the adapted HySOM model. For example, Nam & Kim did suggest the CLAMI unsupervised fault-proneness model, using metrics and instances selection processes to build the training data [31]. These processes could be integrated in the HySOM model before the ANN training and could therefore enhance its prediction performance. Another study by Zheng *et al.* did suggest using a spectral clustering algorithm instead of a distance-based classifier (like K-means or SOM algorithms) [32]. The HySOM model could be adapted to use a spectral clustering algorithm instead of SOM.

Furthermore, the HySOM model could incorporate some elements present in effort-aware fault-proneness prediction models. Effort-aware fault-proneness prediction predicts fault-prone source code, but further improve it by considering the effort required to test the faulty source code in the prediction [33]. By doing so, effort-aware approaches can further prioritize which parts of the source code developers and testers have to focus their testing efforts on. Of course, these are only some examples of improvements that could enhance the prediction performance and usability of the HySOM model.

References

[1] A. Bertolino, "Software Testing Research: Achievements, Challenges, Dreams", *Future of Software Engineering (FOSE '07)*, no. September, pp. 85–103, may 2007.

[2] B. Isong and E. Obeten, "A Systematic Review of the Empirical Validation of Object-Oriented Metrics Towards Fault-Proneness Prediction", *International Journal of Software Engineering and Knowledge Engineering*, vol. 23, no. 10, pp. 1513–1540, dec 2013.

[3] C. Catal, U. Sevim, and B. Diri, "Software Fault Prediction of Unlabeled Program Modules", *Proceedings of the World Congress on Engineering*, vol. I, pp. 1–6, 2009.

[4] G. Abaei, A. Selamat, and H. Fujita, "An empirical study based on semi-supervised hybrid self-organizing map for software fault prediction", *Knowledge-Based Systems*, vol. 74, pp. 28–39, jan 2014.

[5] C. Catal, U. Sevim, and B. Diri, "Clustering and metrics thresholds based software fault prediction of unlabeled program modules", *ITNG 2009 - 6th International Conference on Information Technology: New Generations*, pp. 199–204, apr 2009.

[6] C. Catal, U. Sevim, and B. Diri, "Metrics-Driven Software Quality Prediction Without Prior Fault Data", in *Electronic Engineering and Computing Technology*, ser. Lecture Notes in Electrical Engineering, S.-I. Ao and L. Gelman, Eds. Dordrecht: Springer Netherlands, 2010, pp. 189–199.

[7] P. S. Bishnu and V. Bhattacherjee, "Software Fault Prediction Using Quad Tree-Based K-Means Clustering Algorithm", *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 6, pp. 1146–1150, jun 2012.

[8] G. Abaei, Z. Rezaei, and A. Selamat, "Fault prediction by utilizing self-organizing map and threshold", *2013 IEEE International Conference on Control System, Computing and Engineering*, pp. 465–470, nov 2013.

[9] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: a large scale experiment on data vs. domain vs. process", in *ESEC/FSE 2009*. New York, New York, USA: ACM Press, 2009, pp. 91–100.

[10] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models", *Empirical Software Engineering*, vol. 21, no. 5, pp. 2072–2106, oct 2016.

[11] H. Lu, B. Cukic, and M. Culp, "Software defect prediction using semi-supervised learning with dimension reduction", *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*, p. 314, 2012.

[12] E. Erturk and E. Akcapinar Sezer, "Iterative software fault prediction with a hybrid approach", *Applied Soft Computing*, vol. 49, pp. 1020–1033, dec 2016.

[13] M. H. Halstead, *Elements of software science*. New York: Elsevier Science Inc., 1977.

[14] T. Menzies, R. Krishna, and D. Pryor, "The Promise Repository of Empirical Software Engineering Data", 2016. [Online]. Available: http://openscience.us/repo/

[15] R. Shatnawi, "A Quantitative Investigation of the Acceptable Risk Levels of Object-Oriented Metrics in Open-Source Systems", *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 216–225, mar 2010.

[16] R. Malhotra and A. J. Bansal, "Fault prediction considering threshold effects of object-oriented metrics", *Expert Systems*, vol. 32, no. 2, pp. 203–219, apr 2015.

[17] L. Yu, "Using Negative Binomial Regression Analysis to Predict Software Faults: A Study of Apache Ant", *International Journal of Information Technology and Computer Science*, vol. 4, no. 8, pp. 63–70, jul 2012.

[18] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction", *Proceedings of the 6th International Conference on Predictive Models in Software Engineering - PROMISE '10*, p. 1, 2010.

[19] M. Jureczko, "Significance of different software metrics in defect prediction", *Software Engineering: An International Journal*, vol. 1, no. 1, pp. 86–95, sep 2011.

[20] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design", *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, jun 1994.

[21] Yuming Zhou and Hareton Leung, "Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults", *IEEE Transactions on Software Engineering*, vol. 32, no. 10, pp. 771–789, oct 2006.

[22] R. Shatnawi, "Improving software fault-prediction for imbalanced data", *2012 International Conference on Innovations in Information Technology, IIT 2012*, pp. 54–59, mar 2012.

[23] T. McCabe, "A Complexity Measure", *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, dec 1976.

[24] J. Sayyad Shirabad and T. Menzies, "The PROMISE Repository of Software Engineering Databases", 2005. [Online]. Available: http://promise.site.uottawa.ca/SERepository

[25] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches", *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pp. 31–41, may 2010.

[26] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction", *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 897–910, oct 2005.

[27] A. Boucher and M. Badri, "Using Software Metrics Thresholds to Predict Fault-Prone Classes in Object-Oriented Software", *Special Session of Software Engineering with Artificial Intelligence, 4th International Conference on Applied Computing & Information Technology*, pp. 169–176, dec 2016.

[28] R. Shatnawi, W. Li, J. Swain, and T. Newman, "Finding software metrics threshold values using ROC curves", *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 22, no. 1, pp. 1–16, jan 2010.

[29] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data", *2010 IEEE International Conference on Software Maintenance*, pp. 1–10, sep 2010.

[30] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: An Update", *ACM SIGKDD Explorations Newsletter*, vol. 11, no. 1, p. 10, nov 2009.

[31] J. Nam and S. Kim, "CLAMI: Defect Prediction on Unlabeled Datasets (T)", in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, nov 2015, pp. 452–463.

[32] F. Zhang, Q. Zheng, Y. Zou, and A. E. Hassan, "Cross-project defect prediction using a connectivity-based unsupervised classifier", in *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. New York, New York, USA: ACM Press, 2016, pp. 309–320.

[33] Y. Yang, M. Harman, J. Krinke, S. Islam, D. Binkley, Y. Zhou, and B. Xu, "An empirical study on dependence clusters for effort-aware fault-proneness prediction", in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*. New York, New York, USA: ACM Press, 2016, pp. 296–307.

ANNEXE C

BOUCHER & BADRI, 2017B

Information and Software Technology xxx (xxxx) xxx–xxx

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Software metrics thresholds calculation techniques to predict fault-proneness: An empirical comparison

Alexandre Boucher, Mourad Badri*

Software Engineering Laboratory, Department of Mathematics and Computer Science, University of Quebec, Trois-Rivières, Canada

ARTICLE INFO

Keywords:
Metrics thresholds
Class-level metrics
Object-oriented metrics
Faults
Fault-proneness prediction
Machine learning
Clustering
Cross-validation
Code quality
Object-oriented programming

ABSTRACT

Context: Nowadays, fault-proneness prediction is an important field of software engineering. It can be used by developers and testers to prioritize tests. This would allow a better allocation of resources, reducing testing time and costs, and improving the effectiveness of software testing. Non-supervised fault-proneness prediction models, especially thresholds-based models, can easily be automated and give valuable insights to developers and testers on the classification performed.

Objective: In this paper, we investigated three thresholds calculation techniques that can be used for fault-proneness prediction: ROC Curves, VARL (Value of an Acceptable Risk Level) and Alves rankings. We compared the performance of these techniques with the performance of four machine learning and two clustering based models.

Method: Threshold values were calculated on a total of twelve different public datasets: eleven from the PROMISE Repository and another based on the Eclipse project. Thresholds-based models were then constructed using each thresholds calculation technique investigated. For comparison, results were also computed for supervised machine learning and clustering based models. Inter-dataset experimentation between different systems and versions of a same system was performed.

Results: Results show that ROC Curves is the best performing method among the three thresholds calculation methods investigated, closely followed by Alves Rankings. VARL method didn't give valuable results for most of the datasets investigated and was easily outperformed by the two other methods. Results also show that thresholds-based models using ROC Curves outperformed machine learning and clustering based models.

Conclusion: The best of the three thresholds calculation techniques for fault-proneness prediction is ROC Curves, but Alves Rankings is a good choice too. In fact, the advantage of Alves Rankings over ROC Curves technique is that it is completely unsupervised and can therefore give pertinent threshold values when fault data is not available.

1. Introduction

Nowadays, software systems must be of good quality and, in certain cases, fault-free. Indeed, problems generated by faults could cause major damage and important losses of money. Having a high quality software can prevent faults, therefore reducing costs incurred for their correction. The problem is that it is cost prohibitive, difficult and often impossible to exhaustively test all execution paths of a complex software to ensure that it is fault-free. In order to support developers and testers in the testing process, quality models and tools can be used for identifying poor quality and particularly fault-prone code. These models generally use source code metrics to identify fault-prone classes or methods.

Many metrics-based fault-proneness prediction models were proposed by different researchers in the literature. The Chidamber and Kemerer (CK) [1] object-oriented metrics suite has been widely used [2–11]. Some researchers defined models based on statistical regression analysis [3,4,6,10–13], machine learning algorithms [4,6,7,10,12,14–17], threshold effect of code metrics [2,3,6–8,14,15,18,19] or even different combinations of those methods. Models based on the threshold effect of code metrics consider classes as fault-prone when the values of their metrics exceed certain thresholds. The advantage of these specific models is that they can easily be implemented and understood by software engineering experts and developers. In addition, they can provide valuable and simple insights on why a specific class is classified as fault-prone by indicating which metrics have problematic values and need to be adjusted. Subjective threshold calculation techniques (values) were, however, suggested for

* Corresponding author.
E-mail addresses: alexandre.boucher2@uqtr.ca (A. Boucher), mourad.badri@uqtr.ca (M. Badri).

https://doi.org/10.1016/j.infsof.2017.11.005
Received 4 November 2016; Received in revised form 9 October 2017; Accepted 8 November 2017

different metrics by software engineering experts (see [20] or thresholds used in [15]). As an example, McCabe suggested a given threshold for his cyclomatic complexity metric [20], and Rosenberg proposed different thresholds for CK metrics [21] (not directly related to fault-proneness). These thresholds definition techniques were indeed strictly subjective [15,20,21], as they didn't consider objective data related to fault-proneness. Furthermore, the proposed thresholds can't be generalized to all projects, as different programming styles and system sizes will probably yield different source code metrics distributions, therefore making some threshold values obsolete for certain systems. Of course, threshold values of some systems could be reused for other systems, but this cannot be generalized. According to some studies [2,6], metrics' thresholds should be calculated for each project and even for each version of a system. Despite these affirmations, another study by Malhotra and Bansal investigated cross-project fault-proneness prediction obtaining acceptable results [7]. There is still a need for objective threshold calculation techniques (and values). Therefore, many thresholds definition algorithms were proposed in literature [2,22–28]. However, not all of them were validated as being good predictors of fault-proneness.

In this paper, we evaluated and compared the metrics' threshold values of three software metrics' thresholds calculation techniques as fault-proneness predictors. The first studied method is the ROC (Receiver Operating Characteristic) Curves method proposed by Shatnawi [3]. The second studied method is VARL (Value of an Acceptable Risk Level), proposed by Bender [22]. It is based on univariate logistic regression. This method does not, in fact, concern software engineering directly, but was studied as such by several studies [3,7,8]. The third one is the method of Alves et al., which we'll reference to as Alves Rankings method [23].

The whole study aims at answering the six following research questions:

RQ1: Can the ROC Curves method produces threshold values for other datasets than the ones investigated in the original study and achieves good binary fault-proneness prediction performance?
The ROC Curves approach was only validated on the Eclipse software system in the original study [2]. Other studies investigated this method using other datasets, making some modifications to the original approach [25,29]. Furthermore, the original study that proposed this method concluded that binary classification was not valid for the investigated software system (Eclipse) [2]. We therefore wanted to investigate if this conclusion holds for other software systems and datasets as well.

RQ2: Can the VARL methodology be considered as a good methodology to calculate threshold values to do fault-proneness prediction?
Different studies investigated the VARL methodology for fault-proneness prediction and different contradictory conclusions came up, some validating it and others stating that it can't be used on all software systems [3,7,8].

RQ3: Can the Alves Rankings thresholds calculation technique produces threshold values achieving good fault-proneness performance?
Since the Alves Rankings methodology hasn't been investigated yet for fault-proneness prediction, we decided to investigate if it can achieve good performance. Furthermore, we wanted to compare its performance to already validated approaches like ROC Curves and VARL.

RQ4: Which thresholds calculation technique (considering ROC Curves, VARL and Alves Rankings) performs the best for fault-proneness prediction?
Two of the three methods (ROC Curves and VARL) were already validated for fault-proneness prediction and according to the best of our knowledge, no previous study compared at least 2 out of the 3 investigated approaches in a fault-proneness prediction context.

RQ5: Can thresholds-based fault-proneness prediction models achieve similar performance to supervised models? When combined with a machine learning or clustering based model, do thresholds-based models achieve better performance?
After investigating thresholds-based models, we wanted to investigate if they can achieve similar performance than supervised approaches, which are more complex. We also wanted to investigate if combining both methodologies could lead to a better fault-proneness prediction. To do so, we investigated four machine learning based models, which are Bayes Network [7,17], ANN (Artificial Neural Network) [4,10,16], C4.5 [4,12,30] and Support Vector Machine [7,30], and two clustering algorithms, which are K-means [11,15,16,18] and SOM (Self-Organizing Map) [11,16]. We chose these methods because they have been widely used in other fault-proneness prediction studies. Moreover, they can easily be performed using the Weka tool (a data-mining and classification tool) [31], which we used in our study.

RQ6: Can threshold values calculated for one software system or different versions of it be reused for another system or version and still achieve good fault-proneness performance? How does that compare to cross-project or cross-version supervised fault-proneness prediction?
Sometimes, it could be interesting to reuse threshold values from one project to another, especially for methods like ROC Curves or VARL, which need fault data history to calculate threshold values. We also wanted to perform this cross-project and cross-version prediction with supervised approaches, to investigate which methodology should be used in a real-life context.

The rest of the paper is organized as follows. Section 2 presents different studies related to fault-proneness prediction, thresholds definition algorithms, machine learning and clustering models. Section 3 describes the research background needed to understand the study and the methodology followed. It presents some generic background knowledge on fault-proneness prediction, the threshold definition methods and the (machine learning and clustering) algorithms used. Section 4 presents the methodology we followed in this study. It presents how the source code metrics were chosen, how the different models were evaluated and how each experiment was conducted. Section 5 presents the different results obtained. It presents how the source code metrics have been selected for the study and the results of the different fault-proneness prediction experiments. Section 6 mentions the possible threats to validity of our study. Finally, Section 7 concludes this paper by summarizing the major contributions of this study and giving some future work directions.

2. Related work

In this section, we present different research studies related to our study.

2.1. Relationship between source code metrics and fault-proneness

Fault-proneness prediction can help developers and testers to focus their testing effort on classes that are considered more fault-prone, which have to be tested more intensively. This crucial issue was widely studied by different researchers, but no one gave a final and absolute model to use. It is, in fact, a difficult and challenging task. Models defined for this purpose can be based on statistical regression analysis [3,4,6,10], machine learning algorithms [4,6,10], threshold effect of code metrics [2,3,6] or complex combinations of those models. Many studies that have addressed source code quality measurement can be reused for fault-proneness prediction, because the two concepts are closely related. This strong relationship can be defined as higher quality classes are likely less fault-prone than poor quality classes. Also, many

studies validated the relationships between object-oriented metrics and code quality (or fault-proneness) [2–7,9,10,13]. In fact, a large part of the proposed fault-proneness prediction models use source code metrics (especially SLOC and CK metrics suite [1]). Using source code metrics makes it easy to build fault-proneness prediction models, as they are based on numeric values. Furthermore, these models can easily give pertinent information about different source code attributes, like size, complexity, coupling, cohesion, etc. This section presents some relevant papers related to the problematic of fault-proneness prediction, and especially metrics-based models and threshold values definition algorithms.

Many source code metrics exist in the literature. The most used ones are from the metrics suite of Chidamber and Kemerer [1], which includes the well-known source code metrics CBO (Coupling Between Objects), RFC (Response For a Class), WMC (Weighted Methods per Class), LCOM (Lack of Cohesion in Methods), DIT (Depth of Inheritance Tree) and NOC (Number of Children). These metrics evaluate different aspects of the source code, like coupling, size, complexity, cohesion and inheritance. Other metrics suites exist. For example, the QMOOD [32] metrics suite includes 11 design metrics that measure additional elements of object-oriented programming, like encapsulation, composition and polymorphism.

Multiple studies investigated the relationships between source code metrics and fault-proneness by combining different source code metrics and statistical analysis. For example, Jureczko [9] used correlation analysis on many releases of 22 development projects to investigate the relationships between different source code metrics and faults. His study investigated the relationships between various metrics such as LOC (Lines of Code), RFC, CBO, AMC (Average Method Complexity, which is a proposed extension to CK metrics [33]) and CAM (Cohesion Among Methods, which is a QMOOD metric [32]). He concluded that these metrics were related to fault-proneness, but also that other metrics could be interesting to use. He also concluded that the performed correlation analysis may not be sufficient to find all metrics related to fault-proneness.

Additionally, Malhotra and Jain used both univariate and multivariate logistic regressions to extract relationships between source code metrics (Chidamber and Kemerer metrics [1], QMOOD metrics [32], afferent and efferent couplings) and faults [10]. They concluded that out of the 20 investigated metrics, 16 were good at predicting faults according to univariate logistic regression. Note that only DIT, NOC, MFA (Measure of Functional Abstraction, which is a QMOOD metric [32]) and afferent coupling were rejected by their study. They also constructed a multivariate logistic regression model using forward stepwise selection, where DIT, RFC and CBM (Coupling Between Methods, which is used in multiple studies but no reference is given on the study that defined it) were used to construct the model.

Another important and recent study performed by Isong and Obeten [5] investigated the relationships between metrics and faults. They presented a systematic review of papers using object-oriented metrics for predicting fault-proneness. Their paper states two pertinent conclusions for our study. The first conclusion is that, according to most papers studied in this review, SLOC (Source Lines of Code, which is the same as LOC, but considering only executable source code lines), CBO, RFC and WMC are the metrics that are the most related to fault-proneness. The second interesting conclusion is that, according to Isong and Obeten, most of the studies are not replicated by other researchers. The authors observed, however, that the datasets used are often reused from one study to another, meaning that we should use datasets that were already used in other researches, therefore letting researchers compare our findings with others.

2.2. Threshold values calculation approaches

Considering the obvious relationships between object-oriented source code metrics and faults in software systems, many threshold values were defined for these metrics. These threshold values or techniques defining them are used to build qualitative models to describe the quality or fault-proneness of different elements of the code (mostly classes). Using these thresholds, many studies constructed thresholds-based models to predict faults or assess quality of classes in object-oriented systems. These models heavily rely on the threshold values for having a good accuracy, therefore making determination of the threshold values a main challenge of these approaches. To do so, many approaches exist, where some determine threshold values in a subjective way, according to software engineering experts experience and knowledge. For example, McCabe suggested a threshold value for his cyclomatic complexity metric in [20], Catal et al. in [15] used thresholds defined with a tool called PREDICTIVE, which is no longer available and no documentation on the way it calculated thresholds can be found at the moment, or the study of Rosenberg [21], which proposed some thresholds for the CK metrics [1] derived from her descriptive statistical analysis and experience. The other main family of approaches determining threshold values is based on statistical analysis or the distribution of the metrics. These objective approaches are interesting, as they don't require a software engineering expert to determine threshold values for different projects. Furthermore, the thresholds calculated using these techniques should be more relevant, as they are often determined considering fault data expressly to perform fault-proneness prediction.

Considering papers about thresholds definition algorithms, the method by Shatnawi et al. makes use of ROC Curves to define per-project code metrics thresholds [2]. A ROC curve tries different possible threshold values for an independent variable to determine its classification performance at different values. A plot is then produced using sensitivity and specificity (2 classification performance metrics) at the different threshold values investigated. The authors performed two classification experiments on three versions of the Eclipse project using their methodology: one binary and another one ordinal. The binary classification consists in predicting if classes are fault-prone or not, while the ordinal one tries to predict if a class has high, medium, low or no risk to be fault-prone. Shatnawi et al. found that the method was not viable for binary classification of classes. They however found relevant threshold values for high and medium risk categories of ordinal classification. They therefore concluded that: (1) more work is needed to be done on more datasets, and (2) so far, their method was useful for ordinal, but not for binary classification of classes.

Another algorithm proposed for thresholds calculation is the VARL (Value of an Acceptable Risk Level) method presented by Bender [22]. This method consists of translating an univariate logistic regression model into threshold values by using a specific mathematical formula. Even if it wasn't originally produced for software engineering, it was considered for fault-proneness prediction by some studies [3,7,8]. Results of those studies mention that the VARL method gives some acceptable threshold values [8], while others mention that for a given dataset (or system), no valid threshold values were found when using this method [3,7]. Results are therefore mitigated on the usefulness of this method for fault-proneness prediction, since it can't be used on all software systems.

A different algorithm has been proposed by Alves et al. [23] (Alves Rankings) for deriving thresholds from source code metrics. The authors used metrics values distribution in order to define different thresholds for each project. By combining a hundred different projects, they extracted one threshold value per metric applicable for all projects. Alves et al. calculated metrics' thresholds for evaluating class quality, but did not investigate if their method could be used for predicting fault-proneness. Furthermore, according to our searches, and to the best of our knowledge, there are no studies that investigated if the Alves Rankings method can be used for fault-proneness prediction.

In another study, Benlarbi et al. calculated threshold values using logistic regression [24]. They concluded that using a threshold value in logistic regression do not improve the prediction results (when

compared to logistic regression without a threshold value). The threshold values investigated in this study only have been tested for usage in a logistic regression model, which is a supervised approach. Although threshold values calculated using this approach could have been investigated in our paper, we decided to concentrate our efforts on three more recent approaches.

In another study, Catal et al. reused the ROC Curves approach proposed by Shatnawi et al. [2] to calculate threshold values [25]. However, they slightly modified it so the threshold value retained from the plotted ROC curve is not the one maximizing sensitivity and specificity. They instead retained the one maximizing the Area Under Curve (AUC) calculated using only three coordinates on the plot. These three coordinates are the threshold value (1 - specificity, sensitivity), (0, 0) and (1, 1). The threshold values calculated were then used to do noise detection in the investigated datasets. The noisy instances fault-proneness labels were then changed to be non-noisy. A supervised fault-proneness model using Naive Bayes Network was then applied on the modified datasets to detect fault-prone instances. We didn't use this methodology to calculate threshold values, as the threshold values calculation is very similar to the one presented by Shatnawi et al. for the ROC Curves method [2].

Ferreira et al. suggested another methodology to calculate threshold values for source code metrics [26]. Their procedure consisted in fitting the source code metrics' distributions with a best-matching probability law. After this, for each source code metric, they determined three ranges: good, regular and bad. The good range contains the most frequently seen values of a source code metric. The regular range contains intermediate values that have low frequencies of appearance but are still relevant. The bad range represents values rarely seen for a specific source code metric. Although this methodology seems interesting to investigate for fault-proneness prediction, we decided not to use it in this paper. The reason why we didn't investigate it is that it is difficult to automate, as frequencies have to be manually analyzed by an expert to determine the threshold values. This manual analysis could lead to bias in the threshold values calculated. Of course, the threshold values presented could be reused, but the study misses different important source code metrics from the CK metrics suite [1], which are used in our study.

Oliveira et al. did present a methodology for calculating relative threshold values especially for source code metrics [27]. They used a statistical approach to calculate thresholds considering the distribution of the metrics. Their method assumes that each source code metric has a heavy-tailed distribution. The threshold values calculated are presented like: p% of the classes have a M metric below m (where M is the source code metric, m is the threshold value and p is the percentage of code that should respect this rule). This method, producing threshold values that are realistic according to most systems, was not tested for fault-proneness prediction. Although this seems interesting to calculate threshold values, we were not sure it would yield good results for fault-proneness prediction and therefore decided not to investigate it in this study.

In another study, Shatnawi presented a method to calculate threshold values using log transformation [28]. The presented methodology calculates threshold values by applying a logarithmic transformation on the source code metrics. The threshold value is then calculated on the transformed distribution using the mean and the standard deviation, and then transformed back by using an exponential function (therefore canceling the log transformation). This methodology aims in fact to reduce the skewness of the source code metrics' distributions when calculating threshold values. We didn't consider this methodology in the current paper, as we preferred to investigate the ROC Curves approach that Shatnawi also presented. Furthermore, the ROC Curves approach has already been used and validated in other studies [25,29]. It is therefore interesting to use it as a reference method to calculate threshold values. However, this method also looks promising and would be interesting to consider in a future work.

During our study, we found papers using thresholds produced by a tool called PREDICTIVE, developed by Integrated Software Metrics, Inc. (ISM) [15,16]. However, we could not find the tool mentioned in those studies, and as mentioned in [15], the ISM website is no longer accessible. Additionally, these thresholds were used for fault-proneness prediction on multiple different datasets. However, according to other studies [2,6], metrics' thresholds should be defined on a per-project basis and even on a per-version basis, as proposed by Shatnawi et al. [2]. Another study by Malhotra and Bansal [7] obtained acceptable results when evaluating fault-proneness performance doing cross-project prediction. Of course, source code metrics' thresholds could be reused for different datasets, but they should be applied on similar projects. For example, a model could be reused for other projects of a same organization, or projects that are of similar size and complexity, although thresholds need to be calculated for these projects on first use and for projects that are dissimilar. Furthermore, project size should be considered when validating metrics' thresholds on multiple datasets, as a threshold for one project could be obsolete for another one, therefore needing new threshold values [2,6]. Threshold values produced by the PREDICTIVE tool were not used in our study, since the tool is no longer available and we can't know how these metrics' thresholds were calculated.

For our study, three out of the mentioned thresholds definition algorithms are investigated, each one for different reasons. We decided to investigate the ROC Curves method [2], since it has been partially validated. This method was investigated on a few datasets, which are different versions of Eclipse IDE. We therefore wanted to analyze it further on multiple datasets and systems, to see if binary classification can yield better results in different projects. We also decided to investigate the VARL methodology [22], since results from different studies were different (acceptable for [8] and mitigated for [3,7]). We wanted to investigate if the method can give valuable thresholds when applied to other datasets and when considering the number of faults in each one of them. The last method we investigated is Alves Rankings from Alves et al. [23]. We thought that this method would be interesting to investigate for fault-proneness prediction since it was originally defined for describing quality of classes in object-oriented systems.

2.3. Fault-proneness prediction using machine learning and clustering

Since threshold values can be calculated for different source code metrics, many machine learning and clustering models were constructed using these threshold values. Machine learning algorithms are widely used to perform fault-proneness prediction, as models based on these are numerous. Malhotra and Bansal investigated Naive Bayes Network, Bayes Network, Random Forest, Support Vector Machine and Multilayer Perceptron (ANN or Artificial Neural Network) to perform fault-proneness prediction [7]. They performed tests applying the machine learning algorithms directly by using the source code metrics values and by binarizing the metrics values based on threshold values calculated with the VARL method. They concluded that Support Vector Machine was the best out of the 5 machine learning algorithms investigated and that binarizing the datasets before running the models yielded better results. Shatnawi in [17] studied the Naive Bayes Network, Bayes Network and Nearest Neighbor algorithms to find faults on Eclipse IDE. Gyimothy et al. used ANN and C4.5 algorithms on the Mozilla open source software [4]. C4.5 was also used in other studies [12,30]. Many other studies used source code metrics with machine learning algorithms to predict fault-proneness in software systems (see [12,16,30]).

We therefore decided to investigate Bayes Network, ANN and C4.5 machine learning algorithms as they were widely used in fault-proneness prediction. In addition, we also decided to include the Support Vector Machine algorithm as it was considered the best performing algorithm in [7] and was used in [30].

Clustering algorithms are often used for fault-proneness prediction,

as classes can be clustered based on similar source code metrics and then labeled as faulty or not using metrics' thresholds, as done by Bishnu and Bhattacherjee in [18] or Catal et al. in [15,19]. All of these studies use K-means algorithm, which is one of the most used clustering algorithms (see [11,16] for additional usages). In [19] by Catal et al., in addition to K-means, they investigated the use of Fuzzy C-means and X-means, which is basically the K-means algorithm where the number of clusters can be in a given range instead of being fixed to a specific value. In [11] by Jureczko and Madeyski, K-means and SOM (Self-Organizing Map) were used to cluster classes based on their metrics values. The SOM algorithm works differently compared to K-means, as each time a vector of data is clustered, it updates the means of the neighboring clusters. In their study, for each cluster formed, Jureczko and Madeyski applied a linear regression model and tested it on multiple releases of the same software. The clustering isn't directly used as in previously mentioned studies. It was used before running a statistical analysis. Another study by Abaei et al. [16] used the SOM algorithm before running a machine learning algorithm (ANN) to predict fault-prone classes.

After considering the different clustering algorithms used for fault-proneness prediction, we decided to investigate K-means and SOM algorithms as they are widely used and work in different ways (and should therefore yield different results).

3. Research background

The objective of this study is to assess and compare different thresholds definition techniques for fault-proneness prediction. In order to do so, several elements support the presented research. In this section, we present the background needed to realize this research.

3.1. Dependent and independent variables

In all classification and prediction experiments in general, there are dependent and independent variables. In fault-proneness prediction, the dependent variable is often binary (as in our study) and is the fault-proneness. When fault-proneness prediction is performed using source code metrics, these metrics act as independent variables.

The choice of the source code metrics used in thresholds-based fault-proneness prediction is important, since they are the basis of the whole prediction algorithm. In literature, many metrics have been proposed to describe the source code of a software system. However, Source Lines of Code (SLOC) and CK metrics have been widely used for fault-proneness prediction [2–6] (see Table 1 for a presentation of each source code metric investigated [1]). We therefore decided to consider these metrics to perform our study. However, after having performed an univariate logistic regression analysis on each of these metrics, we decided to consider a subset of these metrics. The results of these analyses and the resulting subset are presented in Section 5.1.

Table 1
Source code metrics investigated.

Metric	Description
SLOC (Source Lines of Code)	Number of source code lines in a class, excluding commented and blank ones.
CBO (Coupling Between Objects)	Number of classes to which the class is coupled.
RFC (Response For a Class)	Number of methods that can potentially be executed when the class receives a message.
WMC (Weighted Methods per Class)	The sum of the cyclomatic complexities of all methods.
LCOM (Lack of Cohesion in Methods)	Measures the lack of cohesion of a class using the similarity of the methods.
DIT (Depth of Inheritance Tree)	The depth of the class in the inheritance tree.
NOC (Number of Children)	The number of immediate subclasses to a class.

3.2. Data collection

In order to perform fault-proneness prediction, data giving both dependent (faultiness) and indepedent (source code metrics) variables is needed. In a real-life enterprise context, the source code metrics and faults information would be obtained directly from the source code and bug tracker. However, since we are in a research context and want to make our results as reproducible and comparable as possible, we used public datasets. These datasets can easily be obtained online and were used in other studies. Another reason why we chose these datasets is because they were not all investigated using the three thresholds calculation techniques presented in our study.

In this study, we used twelve different datasets from 8 different systems: Apache ANT (versions 1.3, 1.4, 1.5, 1.6 and 1.7), Apache IVY, Apache LUCENE, Apache POI, Apache TOMCAT, KC1, JEdit and Eclipse JDT Core. ANT (all versions), IVY, LUCENE, POI, TOMCAT and JEdit are available from the PROMISE Repository, which makes available multiple datasets for fault-proneness prediction [34]. KC1 dataset is available on the PROMISE Repository of University of Ottawa [35], while the Eclipse JDT Core dataset is available from the research results of D'Ambros et al. [36].

The first datasets, which were built on the Apache ANT system for versions 1.3, 1.4, 1.5, 1.6 and 1.7, were used in multiple studies [7,9,11,13]. In fact, version 1.7 was widely used, but we decided to include 4 previous versions as well in order to compare fault-proneness prediction among different versions of a single system. ANT is a command-line tool developed in Java and mainly used for building Java applications [11]. Another dataset used was made for Apache IVY 2.0, which was also used in multiple studies [7,9,12]. IVY is a dependency manager developed in Java, integrated in Apache ANT [11]. Apache LUCENE (version 2.4) is a text search engine library written in Java [37] and was used in some studies [9,11,12,36]. Apache POI is a library regrouping Java APIs to read or write documents following Office Open XML standards [38] and was used in multiple studies [9–12]. The last Apache project we selected is TOMCAT, which is an open source implementation of multiple Java Web server technologies [39]. This project was used in many studies related to fault-proneness prediction [7,9,11,12]. KC1 [35] was developed by the NASA with the C++ language and was used in numerous studies [6,7,14,16,30,40]. Another dataset we used was built for the JEdit 4.3 program, which is a text editor developed in Java [9]. It was used in multiple studies for fault-proneness prediction [7,9,11,12]. The last dataset used is based on the Eclipse JDT Core system. It was produced after a study by D'ambros *et al.* [36] on multiple releases of the system. The JDT Core is the primary infrastructure of the Eclipse Java IDE, which includes a compiler, a code formatter, a code assistance and other practical features for developers using the Eclipse Java IDE [41]. The Eclipse project was used in numerous studies [2,3,5,14,17,36,40]. Although the JDT Core Component wasn't used specifically in those studies, we used this dataset for the simplicity of the data acquisition and to simplify study replication.

Note that for Apache ANT (all versions), IVY, LUCENE, POI, TOMCAT and JEdit datasets, the WMC metric value had to be calculated using the average cyclomatic complexity of all methods multiplied by the method count in each class. The reason we are not using the WMC metric presented in those datasets is that it only gives the method count of each class according to the study that produced those datasets [11].

We present in Table 2 some statistics about the size and the number of faults in each system. As we can see, 3 projects contain more faults than the others (LUCENE, POI and KC1), according to the faulty class ratio. KC1 is the project containing the more faults, according to faults count. Its number of faults is about 4.5 times higher than the number of classes present in the system, which is very high. These 3 datasets are very faulty, especially LUCENE and POI, which faulty class ratio is about 60%. The high number of faults in these datasets risks giving bad

Table 2
Datasets size and fault statistics.

Dataset	Total SLOC	# of classes	# of faults	# of faulty classes	Faulty class ratio
ANT 1.3	37 699	125	33	20	16%
ANT 1.4	54 195	178	47	40	22.47%
ANT 1.5	87 047	293	35	32	10.92%
ANT 1.6	113 246	351	184	92	26.21%
ANT 1.7	208 653	145	338	166	22.28%
IVY	87 769	352	56	40	11.36%
LUCENE	102 859	340	632	203	59.71%
POI	129 327	442	500	281	63.57%
TOMCAT	300 674	858	114	77	8.97%
KC1	30 631	145	669	60	41.38%
JEdit	202 363	492	12	11	2.24%
Eclipse	224 055	997	374	206	20.66%

results, because systems usually contain a lot more non-faulty classes than faulty ones, as mentioned in [17,40]. Contrarily to these very faulty datasets, JEdit has very few faults, as 2.24% of the classes are faulty. It contains only 12 faults on 492 classes. Because of this low number of faults, fault-proneness prediction for JEdit could be difficult. A lot of classes will be surely marked as faulty when they are actually not.

3.3. Threshold definition methods

In this study, we assess and compare three thresholds definition methods for fault-proneness prediction: ROC Curves, VARL and Alves Rankings. We present in what follows how each method calculates threshold values for the source code metrics.

3.3.1. ROC Curves

The ROC Curves method, as defined by Shatnawi et al. [2], plots a ROC Curve for each code metric and then retrieves the optimal threshold value by maximizing the sum of sensitivity and specificity [2]. Plotting a ROC Curve consists in taking a continuous and a binary variable. For this method, the continuous variable consists of the metric value for each class in the system, while the binary variable is the presence of faults in a given class. A range of possible thresholds for the metric is then produced, varying from the minimum to the maximum possible value for this metric in the given dataset. Then, for each possible threshold value defined, a confusion matrix is built.

$$Specificity = 1 - FP/(FP + TN) \quad (1)$$

$$Sensitivity = TP/(TP + FN) \quad (2)$$

A confusion matrix is a table that presents classification results, giving the number of true/false positives/negatives (Table 3 gives the structure of a confusion matrix). Note that for our study and as usually done in fault-proneness prediction, a positive represents a faulty class and a negative a class that is not faulty. Each confusion matrix constructed then outputs a point on the ROC Curve plot. The X axis of the plot is mapping the 1 - specificity value, while the Y axis is mapping the sensitivity. Each 1 - specificity and sensitivity pair is obtained from the confusion matrix using Eqs. (1) and (2). The threshold value retained is the one that maximizes both 1 - specificity and sensitivity.

Table 3
Confusion matrix structure.

	Actual	
Classified	Faulty	Not-faulty
Faulty	True positives (TP)	False positives (FP)
Not faulty	False negatives (FN)	True negatives (TN)

3.3.2. VARL

VARL, which stands for *Value of an Acceptable Risk Level*, was defined by Bender to calculate threshold values in epidemiological studies [22]. There is no immediate relationship between his study and software engineering, but several studies used it to calculate threshold values for code metrics in order to do fault-proneness prediction [3,7,8].

VARL uses univariate logistic regression (which is explained in Section 4.1) to calculate threshold values. It does so by reusing two metrics outputted by logistic regression: regression coefficient value β and the constant coefficient value α [8]. VARL is calculated as presented in Eq. (3), where p_0 is the acceptable risk level and can be interpreted as follows: for the classes where the metric value is below the threshold given by VARL, the risk that a fault occurs in this class is lower than the probability p_0 [3].

$$VARL = p^{-1}(p_0) = \frac{1}{\beta}\left(\log\left(\frac{p_0}{1-p_0}\right) - \alpha\right) \quad (3)$$

3.3.3. Alves Rankings

The method of Alves et al. for calculating thresholds didn't have a name in the original paper [23]. So, we decided to name it Alves Rankings method. This method hasn't been investigated for defining metrics' threshold values for fault-proneness prediction. Alves et al. defined their method to find thresholds describing quality of classes and finally categorize them. To do so, they passed through 6 steps for calculating thresholds [23], but as we used it, only the steps 1, 2, 3 and 6 are relevant for our study.

The first step, which they called *metrics extraction*, consists of extracting the metrics of the system [23]. Of course, code metrics of each class are calculated in this step, but also the *weight* of each class. The weight of a class is defined by SLOC in their paper. For our study, the first step consisted in finding the datasets we decided to use.

The second step, named *weight ratio calculation*, consists of calculating the *weight ratio* of each class [23]. This ratio is calculated simply by dividing the weight of a class (SLOC) by the sum of all classes weights. The weight ratio simply represents the relative size of each class in the system. For example, if a class has a weight ratio of 0.01, this means that the class code represents 1 percent of the total code of the system.

The third step, which is called *entity aggregation*, consists of aggregating the weight of all entities (which here are classes) per metric values [23]. The result of that step is similar to a weighted histogram, giving the percentage of code of the system being represented by each metric value. For example, after that step we could say that 1% of a system's SLOC is represented by a CBO metric of 6.

The fourth and fifth steps of the method are not used in our study. The reason of this decision is that Alves et al. did calculate thresholds using a hundred different software systems [23]. In our case and as mentioned in Section 2, we wanted to calculate for each single system one threshold value for each code metric. The fourth and fifth steps of the Alves Rankings method consisted in normalizing the weights of each system they evaluated. Once done, they aggregated the metric values for those systems, getting a similar output as in the third step. The difference here is that the percentage of each metric represents the percentage of code across all systems, not only a single one like in the third step.

The sixth step of this method, which is called *thresholds derivation*, consists of calculating the threshold values for each class. To do so, we define a percentage of code we want to represent with our threshold values. For example, choosing 80% of the overall code could output a threshold value of 30 for the CBO metric. That would mean that 20% of the poor quality code according to CBO metric would be targeted by the threshold value of 30. As an example, Alves et al. used threshold values defined at 70%, 80% and 90% of the metrics distributions for their final quality model.

3.4. Machine learning algorithms

Machine learning algorithms are used in fault-proneness prediction to learn relationships between source code metrics and faults. These algorithms are trained using the datasets and faults. As mentioned earlier, four machine learning algorithms were used in our study to predict fault-proneness. We give in this section a brief description of each of these machine learning algorithms (Bayes Network, Multilayer Perceptron, C4.5 and Support Vector Machine).

3.4.1. Bayes Network

The Bayes Network algorithm classifies the given instances by building a Bayesian Network (directed graph), which maps metrics as nodes and their independencies as links between the metrics, to classify instances as fault-prone or not [17]. It can be used in different variants. The most popular one is the Naive Bayes Network. In our case, we used the standard Bayes Network algorithm. It was used in many studies, notably in [7,12,17] to perform fault-proneness prediction.

3.4.2. Artificial Neural Network (ANN)

The possible applications of Artificial Neural Networks (ANN) are numerous, but this algorithm is mainly used for classification, as in fault-proneness prediction. In our case, a Multilayer Perceptron (or feedforward ANN with back-propagation algorithm) is used, as in [10]. This particular ANN topology consists in having several layers of neurons, where each layer can have a different number of neurons. Each neuron of each layer is linked to the previous and next layer's neurons. The network is first trained using training data, and the back-propagation algorithm is used to update the weights of the different neurons' links. Several studies investigated this algorithm in fault-proneness prediction [4,7,12,16].

3.4.3. C4.5

The C4.5 algorithm is used for building trees. It calculates how efficiently each attribute is in splitting the data (in our case as fault-prone or not) [30]. The resulting tree is considered as a decision tree, which is easy to understand and use, as it is self-explanatory [30]. This algorithm was also used in many studies addressing fault-proneness prediction [4,12,30].

3.4.4. Support Vector Machine

The Support Vector Machine algorithm is based on the statistical learning theory, which makes it perfect for classification or regression [30]. In its learning algorithm, this model gives less weight to elements that are far from the tendency found. It is often used to classify data that do not follow a linear function [30]. It was chosen to investigate as it was considered the best machine learning algorithm to use according to Malhotra et al. [7]. It was also used in other studies related to fault-proneness prediction [10,30].

3.5. Clustering algorithms

Clustering algorithms consist in grouping similar vectors of data in groups called clusters (see [11,15] for examples using clustering algorithms in fault-proneness prediction). The two clustering algorithms used in our study, which are K-means and SOM, are presented here.

3.5.1. K-means

This clustering algorithm is very simple. It consists in building a fixed number of clusters, clustering each instance to the closest cluster, where the closest cluster is the cluster minimizing the distance between the instance (vector of source code metrics) and the cluster's mean (according to a defined distance function, like the Euclidean distance) [19]. We used this method as it was used in many studies considering fault-proneness prediction [11,15,16,19].

3.5.2. SOM (Self-Organizing Map)

The SOM algorithm usually represents clusters in a 2D space topology, named map. It iteratively clusters instances onto its map. At each iteration, a random instance is clustered to the closest cluster (or neuron), by comparing the instance data to the cluster weights. When an instance is clustered, the neighboring clusters see their weights updated to better fit the updated cluster. Since weights of each cluster are dynamically updated during the execution of the algorithm, the instances are clustered when the final weights of the map are set (using the closest cluster to each instance) [16]. The SOM clustering algorithm was used in some studies too [11,16].

4. Research methodology

In this section, we detail the methodology we used to investigate the six research questions mentioned above. We explain how we investigated thresholds-based fault-proneness prediction models and how they were compared to other existing approaches. The methodology used to perform cross-dataset fault-proneness prediction is also presented.

4.1. Choosing the source code metrics

Before performing fault-proneness prediction using source code metrics, the metrics to use need to be determined. In order to do so, logistic regression was used to determine for each source code metric if it is good at determining the fault-proneness of a class.

This statistical method uses the independent variables (in our case, source code metrics) to predict a dependent variable (in our case, fault-proneness). Logistic regression can either be univariate (with one independent variable) or multivariate (two independent variables or more). In our study, since we simply use univariate logistic regression analysis, here is the equation used in the construction of this specific model [7]:

$$P(x) = \frac{e^{g(x)}}{1 + e^{g(x)}} \tag{4}$$

$$g(x) = \alpha + \beta x \tag{5}$$

Where $P(x)$ is the probability of a class being faulty using the metric x, when applied to our specific case. $g(x)$ corresponds to the natural logarithm ln of the odds of an event [7], which is given by Eq. (5). In this equation, α gives the Y-intercept (or constant) and β gives the slope (or estimated coefficient) of the equation [7]. Note that the α and β values presented in Eq. (5) are the same used in the VARL methodology (see Eq. (3)).

Many studies on fault-proneness prediction using source code metrics didn't consider the number of faults when building their prediction model. For example, Shatnawi didn't consider the number of bugs before performing logistic regression analysis to find VARL threshold values [3]. Malhotra and Bansal performed univariate logistic regression too to select code metrics that will be used for fault-proneness prediction using the VARL methodology. However, they did not consider fault counts. In our study, we decided to make use of the number of faults in each class to get more accurate results.

To do so, we followed a simple methodology used by Zhou and Leung [6] and Shatnawi [17], which consists in duplicating each class containing more than one fault in the dataset. For example, if a class contains 3 faults, it will be present 3 times in the dataset, each one marked as containing a fault. This allows taking into account the number of faults in the statistical analysis without having to do much preprocessing. Additionally, it should give threshold values that are more representative of fault-proneness (as faults counts are considered).

In order to determine which metrics will be included in our subset, an univariate logistic regression analysis was performed on the twelve investigated datasets. The source code metrics the more related to fault-

proneness across all datasets will be conserved for fault-proneness prediction.

4.2. Performance evaluation

An important point of our study is the comparison of the different models. To do so, each model needs to be evaluated. This section presents how each model is evaluated and how it is statistically compared to the other models.

4.2.1. Evaluation method

To evaluate the prediction performance of each threshold calculation technique and of the machine learning and clustering models, we used the FPR (False Positive Rate), FNR (False Negative Rate) and geometric mean (g-mean) metrics, which can be easily calculated using the confusion matrix resulting from the classification. We used the FPR and FNR metrics since they were used in other studies to evaluate performance of fault-proneness prediction models [15,16]. We also wanted to allow an easy comparison of our results with those of other studies. These studies [15,16] also used the error rate in conjunction with the FPR and FNR metrics to evaluate the percentage of wrongly classified classes. In our study, we decided not to use it, as FPR and FNR are more important metrics, better describing the classification, especially because the data is imbalanced (the faulty and not-faulty class counts are imbalanced). Even if the classification is not good, for example classifying all classes as not fault-prone, the error rate could be good anyway because if 10% of the classes are fault-prone, the error rate will be of 10%, since 90% of the classes were correctly classified. The g-mean metric was defined especially to describe imbalanced data classification [7] and was also used in other studies on fault-proneness prediction [3,7]. Here are the equations used to calculate the FPR and FNR metrics:

$$FPR = \frac{FP}{FP + TN} \tag{6}$$

$$FNR = \frac{FN}{FN + TP} \tag{7}$$

The FPR metric gives the percentage of false positives among all the actual negative values, while the FNR metric gives the percentage of false negatives among all actual positive values. The lower each metric is, the better is the classification.

The other metric, g-mean, uses two different accuracies, which are the accuracy of positives (TPR) and the accuracy of negatives (TNR) [7] (which are the opposite metrics of FNR and FPR respectively). Contrarily to FPR and FNR, where lower is better, TPR, TNR and g-mean metrics are better the higher they are. The g-mean metric will be acceptable if both TPR and TNR are good, otherwise it won't. Having only one metric to base our comparisons makes it easier to compare datasets between each other. The reason we didn't use TPR and TNR to describe the classification performance along with g-mean is that FPR and FNR are used more often in fault-proneness prediction papers, therefore simplifying comparison of our results with other studies. In addition, since TPR and TNR can easily be calculated from FNR and FPR, we didn't see the need to include them. Here are the equations used to calculate TPR (True Positive Rate), TNR (True Negative Rate) and g-mean:

$$TPR = 1 - FNR = \frac{TP}{TP + FN} \tag{8}$$

$$TNR = 1 - FPR = \frac{TN}{TN + FP} \tag{9}$$

$$g - mean = \sqrt{TPR * TNR} \tag{10}$$

To represent g-mean values in a textual manner and therefore simplifying analysis of the results, we considered the following levels to describe the g-mean values obtained:

- g-mean $<$ 0.5 means no good classification;
- 0.5 $\leq$ g-mean $<$ 0.6 means poor classification;
- 0.6 $\leq$ g-mean $<$ 0.7 means fair classification;
- 0.7 $\leq$ g-mean $<$ 0.8 means acceptable classification;
- 0.8 $\leq$ g-mean $<$ 0.9 means excellent classification;
- g-mean $\geq$ 0.9 means outstanding classification;

Additionally, to accurately calculate prediction results, 10-fold cross-validation is used. This cross-validation method divides the investigated datasets in 10 equal parts (folds). 9 out of 10 folds are then used as training data and the remaining fold is used for testing the prediction. This is done 10 times, each time using a different fold for testing. Each classification table is summed to give a final one, giving the overall performance.

Furthermore, the fault-proneness prediction experiments presented in this study are performed twice. The first prediction is performed using the datasets containing binary information about the fault-proneness of classes. The second prediction uses the same datasets, but with classes duplicated based on the number of faults they contain. This lets us consider the number of faults in the prediction results. However, note that the number of faults in a module is not predicted for the duplicated classification, as it still remains a binary classification. Also note that, for certain experiments, only results for the duplicated datasets are presented for brevity.

In the presentation of the results and in the discussions, we put more emphasis on the results obtained for the duplicated datasets. We did so because we think the prediction results are more accurate when using the duplicated datasets. Considering the number of faults in each class makes false positives and negatives yield a bigger negative impact on the prediction. Contrarily, true positives and negatives yield a bigger positive impact on the classification performance. We think this positive and negative impact on prediction performance makes the results more accurate, as a class containing 10 faults should have more weight in the prediction results than a class containing only 1 fault.

4.2.2. Comparison method

In order to assess that one model performs better than another one, we need an objective comparison methodology. One such methodology was suggested by Demšar, to compare the performance of different models or classifiers over multiple datasets [42]. This methodology consists in using the Friedman statistical test in conjunction with a post-hoc test named Nemenyi. It was also used in other studies about fault-proneness prediction to compare the results of different models [14,40].

The Friedman test is interesting to use in this case because it is a non-parametric test, which means it does not assume that the variables follow a particular distribution. It does not evaluate the performance of the distribution. It only compares the performance of different distributions. To do so, it compares the average rank of the different models on the different datasets. The Friedman statistic is therefore calculated as follows, where k is the number of models, N the number of datasets and R_j the average rank of the model j on all datasets.

$$X_F^2 = \frac{12N}{k(k+1)}\left(\sum_j R_j^2 - \frac{k(k+1)^2}{4}\right) \tag{11}$$

The X_F^2 statistic is then compared to its critical value to check if the null hypothesis is rejected or not. The null hypothesis of the test states that there is no significant difference between the models. If the null hypothesis is rejected, there is a significant difference between at least two of the models. Demšar therefore recommends doing a post-hoc Nemenyi test to compare the performance between each pair of models [42]. According to the Nemenyi test, there is a significant performance difference between two models if the average rank CD differs by at least the critical difference (available in [42]).

$$CD = q_\alpha \sqrt{\frac{k(k+1)}{6N}} \quad (12)$$

In the above equation, q_α is based on the critical values of the Studentized range statistic divided by $\sqrt{2}$, according to [42].

In our study, we therefore decided to use the Friedman test and the post-hoc Nemenyi test to statistically compare the performance of the models. We performed the Friedman test using the g-mean performance metric, which describes the performance well. The statistical tests are performed using the XLSTAT[1] tool using 5% as the confidence level.

4.3. Thresholds-based fault-proneness prediction

The thresholds-based fault-proneness prediction is performed using the three different thresholds definition methods presented in Section 3.3 (ROC Curves, VARL and Alves Rankings) on all 12 datasets presented in Section 3.2.

The original ROC Curves method only seems to consider if a class contains a fault or not and is therefore not taking into account the number of faults in a class. We therefore decided to duplicate the classes in each dataset based on the number of faults they contain, to calculate more accurate threshold values. In fact, it's the same methodology that was used with the logistic regression analysis performed to determine the source code metrics relevant for fault-proneness prediction (see Section 4.1). We followed this methodology to calculate the threshold values using ROC Curves and VARL. The ROC Curves analysis and the logistic regression analysis used for both approaches were performed using XLSTAT. We also computed AUC (Area Under Curve) values for each source code metric for the original and duplicated datasets to see if class duplication had a positive impact on it.

For the VARL methodology, the final threshold values are obtained using the lowest p_0 risk level where all metrics are in a valid range, as done in [7]. We calculated thresholds for a maximum value of p_0 of 0.15, as done by Malhotra and Bansal [7] (Shatnawi stopping at 0.10 [3] and Singh and Kahlon stopping at 0.125 [8]). It is worth noting that even if Malhotra and Bansal calculated threshold values for IVY, KC1 and JEdit using VARL, we do not obtain the same results as theirs [7], because we duplicated entries of the datasets according to the number of faults present in each class. If not all investigated metrics fall in the validity range, we take the p_0 level where most threshold values are in a valid range. It is important to note that, for these threshold values, we rounded to the smallest following integer the resulting value. We did so since code metrics considered for this study can only be represented using integer values.

For the Alves Rankings methodology, threshold values are calculated using a single dataset, contrarily to the original approach aggregating the results of multiple datasets (as described in Section 3.3.3). We developed a small script to calculate the thresholds given by the Alves Rankings method from an Excel file, making it easy to calculate them on each dataset. One other important element to mention is that the threshold values outputted by the Alves Rankings methodology were calculated using 30% of the distribution (see Section 3.3.3 for an explanation of this percentage). We chose this specific percentage since it is the one that yielded the best results according to the tests performed on the investigated datasets. We calculated Alves Rankings threshold values at each 5% step and compared the average fault-proneness prediction results of each one, mainly comparing the g-mean metric. So, we tested threshold values calculated at 5%, 10%, 15%, 20% of the distribution and so on (until 95%). 30% of the distribution was considered giving the best threshold values, according to the average g-mean value. We therefore only present these results for brevity in this paper.

For each of the presented techniques, the fault-proneness prediction is performed four times and four classification tables are therefore produced. The first classification table is constructed by classifying classes when at least one metric exceeds the obtained threshold value as fault-prone. The second one considers classes as fault-prone when at least 2 metrics exceed the threshold values, the third one when 3 metrics exceed the threshold values and the fourth one when all metrics exceed their threshold values. In this way, we can see what is the optimal number of metrics that should exceed threshold values before considering a class as fault-prone. These experiments using thresholds-based models aim at answering RQ1, RQ2, RQ3 and RQ4.

4.4. Comparison with supervised approaches

Once thresholds-based fault-proneness prediction experiments were done, we performed experiments using machine learning and clustering algorithms. These supervised models were used for performance comparison with the thresholds-based approaches. In this way, we can assess if a thresholds-based approach has better, similar or worse performance than supervised approaches.

For the supervised experiments using machine learning algorithms, the Bayes Network, Artificial Neural Network (ANN), C4.5 and Support Vector Machine algorithms were used. The Weka tool was used to construct and test the models [31]. In fact, we developed a small tool that uses the Weka API to execute the available algorithms in Weka. This tool automatizes the calculation of the different results and outputs them directly in an Excel file.

As to the clustering experiments, the K-means and Self-Organizing Map (SOM) algorithms were used for supervised fault-proneness prediction. The Weka software system and our tool were again used to construct the models and output the results. Vectors of data representing the source code metrics of a single class were divided in 2 clusters, one that is classified as fault-prone and the other as not fault-prone. The approach is supervised, as the clusters are classified by minimizing errors using the real fault data of each dataset.

For both machine learning and clustering supervised models, two distinct experiments were performed. Firstly, results were obtained using each machine learning or clustering algorithm on the raw source code metrics values. Secondly, results were obtained using binary source code metrics values. In fact, the metrics were binarized using the threshold values given by the best performing threshold calculation technique(s) (according to thresholds-based fault-proneness prediction results). Source code metrics were converted to a value of 1 if they exceeded their threshold value, otherwise they were converted to 0. These experiments using machine learning and clustering algorithms aim at answering RQ5.

4.5. Cross-project and cross-version fault-proneness prediction

Once the thresholds-based approaches have been investigated and compared to supervised approaches, we decided to perform cross-project fault-proneness prediction. The goal of this experiment is to check if threshold values calculated and models built for one dataset can be reused for other datasets (or systems). We therefore decided to keep the best performing thresholds-based and supervised models to perform this experiment.

We also decided to perform an experiment using training data from one or many previous versions of a software system. To do so, the ANT system was investigated with the same models as for the cross-project experiment. We wanted to investigate if a model built from previous versions of a system can be applied on the next version. This would therefore simulate a real-life fault-proneness prediction application.

For the cross-version fault-proneness prediction, two distinct experiments are performed. Firstly, models are built on the immediate previous version and then tested on the next version of the software. For example, the model is built on ANT 1.3 and tested on ANT 1.4, built on ANT 1.4 and then tested on ANT 1.5, etc. Secondly, the model is built

[1] XLSTAT https://www.xlstat.com/.

using all previous versions data and then tested on the current version. For example, if we want to test the model on ANT 1.6, the model is built using the data from ANT 1.3 to 1.5.

Note that in all cross-project and cross-version experiments, no cross-validation is used, since training and testing data both come from distinct datasets. These experiments aim at answering RQ6.

5. Experimental results and discussion

In this section, the results of the different experiments are presented and discussed. The metrics choice analysis is first presented, then followed by the thresholds-based fault-proneness prediction experiments. After that, supervised, cross-project and cross-version fault-proneness prediction are presented and discussed. At the end of this section, we present a brief visual summary of how the best experiments presented in this paper performed.

5.1. Metrics choice

In order to consider only metrics related to fault-proneness in the prediction models, we performed univariate logistic regression analyses on the different metrics initially considered (SLOC, CBO, RFC, WMC, LCOM, DIT and NOC). The useful software XLSTAT was used to perform the univariate logistic regression analyses used to determine the source code metrics related to fault-proneness. Tables 4–7 present the results obtained.

The first conclusion we can draw is that SLOC, CBO, RFC, WMC and LCOM metrics are relevant for fault-proneness prediction with most of the investigated datasets, according to their *p*-value. Their *p*-value are below the .05 threshold for most datasets investigated (except for ANT 1.3 to 1.5 and JEdit). The NOC metric is only considered good for 4 out of the 12 datasets and the DIT metric for 5 out of the 12. SLOC was considered good for 10 out of 12 datasets, CBO for 9 out of 12, RFC for 11 out of 12, WMC for 10 out of 12 and LCOM for 9 out of 12. We therefore only considered metrics related to fault-proneness prediction for at least 9 datasets out of 12.

However, we decided not to use LCOM, since it generally has low R^2 and Wald Chi-square values. Furthermore, LCOM is a metric which can be calculated in many different ways [28] (different variants exist). All these variants make it difficult to assess that it is calculated the same way for each investigated dataset. This would therefore introduce one important threat to the validity of our study. Additionally, Isong and Obeten performed a systematic review of many studies on fault-proneness prediction [5]. In this review, they concluded that the LCOM metric is not relevant for fault-proneness prediction.

Additionally of being validated with several logistic regression analyses, our subset of metrics was validated as being the same that the one found in [5], which is composed of SLOC (Source Lines of Code), CBO (Coupling Between Objects), RFC (Response For a Class) and WMC (Weighted Methods per Class) metrics. Jureczko study [9] validated LOC (which is almost the same as SLOC), RFC and CBO as being related to fault-proneness. He didn't consider the WMC metric as we considered it, as in his study, WMC represented the number of methods in a class, and not the sum of the cyclomatic complexity of the methods. In his study, he also found out that LCOM was not relevant for fault-proneness prediction, therefore reinforcing the need to remove LCOM from the considered source code metrics. We therefore concluded that the subset of metrics SLOC, RFC, CBO and WMC, which is validated by other previous studies and our own analyses, is appropriate to predict fault-prone classes.

Each of these metrics yields different information: SLOC is a size metric, WMC is a complexity metric, CBO is a coupling metric and RFC is another complexity/coupling metric. This therefore means that these characteristics of the source code influence the fault-proneness of classes.

5.2. Thresholds-based results

In this section, we present and analyze the results of the thresholds-based fault-proneness prediction. The experiment was performed using the 3 threshold definition methods investigated (ROC Curves, VARL and Alves Rankings). In a first step, we present the threshold values calculated using each of the three threshold definition methods. Then, the fault-proneness prediction results using the calculated threshold values are presented and discussed.

5.2.1. Threshold values

The threshold values calculated using each threshold calculation technique are presented and discussed in this section. Table 8 presents the results obtained using all three thresholds calculation techniques. Note that the cells of the table marked with hyphens mean that no valid threshold values could be calculated for the specified metrics and datasets using the VARL technique. In fact, threshold values were calculated, but were considered invalid because they were below the minimum possible value for this metric. For example, the calculated threshold value using VARL methodology for SLOC could have been -10, which is invalid because a class has at least 1 line of code. Following the presentation of the threshold values calculated, we present a comparison of the AUC values obtained when doing the ROC analysis of each dataset.

As there is a lot of hyphens in the table, we conclude that most threshold values could not be calculated using the VARL methodology. In fact, we can already conclude that this technique didn't give any usable threshold values for 4 out of 12 datasets (ANT 1.4, LUCENE, POI and KC1) and that only 3 out of 12 datasets gave usable threshold values for all 4 selected source code metrics. We can also conclude (like Shatnawi [3]), that the VARL method is not applicable to all datasets (or software systems). According to our results, it is applicable only for 8 out of 12 investigated datasets. Also, threshold values calculated with VARL are not always relevant for fault-proneness prediction. For example, the threshold values calculated for the Eclipse dataset are mostly low, such as the calculated threshold value for the SLOC metric, which is 6. Most of the classes in this system have more than 6 lines of code, therefore making this threshold value (if used) consider almost all classes as fault-prone, which does not make sense. Another example where threshold values are not relevant is for the JEdit dataset, where a

Table 4
Univariate logistic regression analysis results for ANT 1.3, 1.4 and 1.5.

	ANT 1.3			ANT 1.4			ANT 1.5		
Metric	*p*-value	Wald Chi-square	R^2	*p*-value	Wald Chi-square	R^2	*p*-value	Wald Chi-square	R^2
SLOC	< .0001	19.018	0.27	.407	0.686	0.005	< .0001	19.588	0.145
CBO	.052	3.781	0.047	.758	0.095	0.001	.251	1.318	0.007
RFC	< .0001	22.365	0.314	.071	3.251	0.026	< .0001	37.3	0.319
WMC	.005	7.725	0.108	.557	0.346	0.003	0	13.034	0.083
LCOM	.114	2.499	0.037	.789	0.071	0.001	.013	6.176	0.042
DIT	.9	0.016	0	.041	4.158	0.033	.017	5.65	0.036
NOC	.67	0.181	0.003	.753	0.099	0.001	.73	0.119	0.001

Table 5
Univariate logistic regression analysis results for ANT 1.6, 1.7 and IVY.

	ANT 1.6			ANT 1.7			IVY		
Metric	*p*-value	Wald Chi-square	R^2	*p*-value	Wald Chi-square	R^2	*p*-value	Wald Chi-square	R^2
SLOC	< .0001	82.177	0.428	< .0001	179.021	0.509	< .0001	45.733	0.344
CBO	< .0001	40.848	0.192	< .0001	38.306	0.112	< .0001	27.993	0.164
RFC	< .0001	106.693	0.558	< .0001	206.692	0.539	< .0001	49.2	0.356
WMC	< .0001	77.13	0.372	< .0001	153.591	0.434	< .0001	44.436	0.317
LCOM	< .0001	47.389	0.331	< .0001	89.958	0.306	0	13.958	0.088
DIT	.835	0.044	0	.126	2.338	0.003	.644	0.213	0.001
NOC	.561	0.339	0.001	.434	0.612	0.001	.659	0.194	0.001

Table 6
Univariate logistic regression analysis results for LUCENE, POI and TOMCAT.

	LUCENE			POI			TOMCAT		
Metric	*p*-value	Wald Chi-square	R^2	*p*-value	Wald Chi-square	R^2	*p*-value	Wald Chi-square	R^2
SLOC	< .0001	33.56	0.161	< .0001	48.652	0.263	< .0001	105.292	0.299
CBO	< .0001	53.719	0.229	< .0001	31.937	0.169	< .0001	74.518	0.19
RFC	< .0001	56.824	0.265	< .0001	71.831	0.332	< .0001	135.058	0.361
WMC	< .0001	26.457	0.136	< .0001	50.141	0.29	< .0001	100.899	0.269
LCOM	< .0001	18.901	0.154	< .0001	26.977	0.154	< .0001	16.985	0.061
DIT	.017	5.687	0.013	0	14.175	0.032	.469	0.525	0.001
NOC	.01	6.724	0.023	.608	0.263	0.001	.042	4.121	0.008

Table 7
Univariate logistic regression analysis results for KC1, JEdit and Eclipse.

	KC1			JEdit			Eclipse		
Metric	*p*-value	Wald Chi-square	R^2	*p*-value	Wald Chi-square	R^2	*p*-value	Wald Chi-square	R^2
SLOC	< .0001	50.101	0.377	.064	3.424	0.022	< .0001	171.933	0.446
CBO	< .0001	91.793	0.354	.006	7.699	0.061	< .0001	151.275	0.359
RFC	< .0001	33.147	0.116	.001	11.909	0.087	< .0001	176.831	0.441
WMC	< .0001	45.08	0.339	.054	3.716	0.024	< .0001	171.397	0.456
LCOM	.003	8.767	0.021	.355	0.856	0.005	< .0001	78.737	0.261
DIT	.356	0.853	0.002	.161	1.965	0.018	.036	4.415	0.005
NOC	0	13.935	0.036	.523	0.408	0.009	.007	7.327	0.009

threshold value of 175 is considered for CBO and another of 1244 for RFC. These are pretty high values and the threshold value calculated for RFC doesn't even consider as fault-prone the class with the highest RFC value in JEdit (which has a RFC of 540). This threshold value makes no sense for this dataset, even if it was calculated expressly for it.

AUC comparison. We wanted to see if class duplication using the number of faults when calculating the threshold values with the ROC Curves methodology improves the prediction power of threshold values. To do so, we compared AUC values before and after class duplication. Additionally, AUC values obtained by Shatnawi et al. are also presented for comparison [2]. Those AUC values are presented in Fig. 1.

In this same study performed by Shatnawi et al., different levels of AUC (Area Under Curve) values for classifying good or bad metric classification are presented [2]. Those levels are:

- AUC = 0.5 means no good classification;
- 0.5 < AUC < 0.6 means poor classification;
- 0.6 ≤ AUC < 0.7 means fair classification;
- 0.7 ≤ AUC < 0.8 means acceptable classification;
- 0.8 ≤ AUC < 0.9 means excellent classification;
- AUC ≥ 0.9 means outstanding classification;

In the same study, the authors found AUC values for the same metrics presented here that were classified as fair or poor. However, in the current study, AUC values can be classified using the same levels as excellent and sometimes acceptable. The datasets used in our study present a stronger relationship between the used code metrics and fault-proneness than the 3 Eclipse datasets used in [2]. However, no AUC value is given in the study from Shatnawi et al. [2] for the SLOC metric and no AUC value is therefore presented in Fig. 1(a).

Comparing AUC values for duplicated and original datasets, we see that for all metrics of all datasets, except for RFC and WMC in ANT 1.3, the AUC is greater when the classes are duplicated. That means that the AUC values found in our study are better than in [2] because: the classes were duplicated according to the number of faults they contain, and we didn't use the same datasets. AUC values, even for the non-duplicated datasets are mostly excellent, otherwise acceptable with some exceptions that are simply fair. The important information to retain here is that, according to AUC, the code metrics used seem to be good classification predictors and that duplication of classes has improved the fault-proneness prediction performance.

5.2.2. Fault-proneness prediction

This section presents the fault-proneness prediction results given by each of the threshold calculation techniques when applied directly on the datasets. The threshold values are further discussed in this part. For brevity and understandability, all methods are suffixed with the number of metrics that need to exceed their threshold value in order to classify a class as fault-prone. For example, ROC-3 would mean that the classification table was obtained using the ROC Curves method, classifying classes as fault-prone when at least 3 metrics exceed their threshold value. The 3 evaluation metrics (FPR, FNR and g-mean), not to confound with the code metrics, are then calculated for each confusion matrix produced.

Table 8
Threshold values calculated using all three methodologies.

	ROC Curves				VARL				Alves Rankings (30%)			
Dataset	SLOC	CBO	RFC	WMC	SLOC	CBO	RFC	WMC	SLOC	CBO	RFC	WMC
ANT 1.3	449	7	47	23	225	–	30	1	354	7	37	13
ANT 1.4	398	7	32	20	–	–	–	–	396	7	41	16
ANT 1.5	603	8	69	17	104	–	33	1	377	6	36	14
ANT 1.6	395	8	35	12	16	–	18	–	413	7	39	17
ANT 1.7	336	9	46	15	121	–	22	4	327	7	40	17
IVY	299	8	39	30	118	2	25	8	411	12	59	20
LUCENE	192	10	26	11	–	–	–	–	394	7	28	15
POI	96	6	19	9	–	–	–	–	331	6	31	17
TOMCAT	386	9	44	31	213	4	36	16	573	6	46	24
KC1	103	8	62	43	–	–	–	–	252	10	34	47
JEdit	560	16	115	30	–	175	1244	–	529	9	53	30
Eclipse	166	13	86	63	6	4	22	16	311	13	91	78

As mentioned earlier, two distinct outputs are produced for some datasets (not for all for brevity and easiness for the reader to understand the results): one output considering the original datasets and another considering the duplicated datasets (based on the number of faults in each class). For tables not containing both outputs, only the output considering the number of faults in each class (duplicated) is presented. Performance metrics (FPR, FNR and g-mean) are suffixed with either *-B* (for binary classification) or *-D* (for duplicated classification). The *-B* suffix is used for classification using the regular (binary) dataset, and the suffix *-D* when the number of faults in each class is considered by using the duplicated datasets. For brevity, the binary classification is only presented for ANT 1.7, IVY, KC1 and Eclipse datasets for each of the 3 threshold calculation techniques.

ROC Curves results. Shatnawi et al. methodology for defining threshold values gave the results presented in Tables 9–12 when applied to fault-proneness prediction.

The results show a logical inverse relationship between FPR and FNR. If the number of metrics exceeding threshold values needed to classify a class as fault-prone is raised, the FPR gets lower and the FNR gets higher. This is plausible because if more metrics exceeding the threshold values are needed to consider a class as fault-prone, more modules are classified as not fault-prone and less are classified as fault-prone, therefore increasing false negatives and reducing false positives.

Fault-proneness prediction using ROC Curves threshold values seems to be acceptable using 2 or 3 metrics exceeding threshold values for classifying a class as fault-prone, as ROC-2 and ROC-3 experiments gave the best results across all datasets when the number of faults is taken into account (also noted by the Friedman analysis). What's interesting is that when the number of faults per class is considered, FPR stayed the same for each model constructed. On all datasets where binary and duplicated fault-proneness prediction was performed, the g-mean metric was higher and the FNR lower for the duplicated one, making it better than binary fault-proneness prediction.

This conclusion is confirmed by the Friedman test, which indicates a significant difference between prediction results of original and duplicated datasets. This test gave a p-value of .002 for the comparison of models predicting binary fault-proneness. For the experiments on duplicated datasets, we obtained a p-value less than .0001. Finally, when comparing ROC-2 and ROC-3 for both binary and duplicated experiments, we obtained a p-value of .001. The Nemenyi test concluded that duplicated experiments achieved better performance than binary ones. This shows that the classification using duplicated datasets affects the performance positively.

Also, ROC-2 experiment gave a g-mean value above 0.7 for all datasets, except for ANT 1.4 and JEdit. In this experiment, 5 out of 12 datasets have a g-mean value above 0.8, which is excellent. As to the ROC-3 experiment, all g-mean values are above 70% too, except for the binary classification of KC1 and the duplicated classification of ANT 1.4. In this experiment, 7 out of 12 datasets present a g-mean over 0.8. ANT 1.4 seems to be problematic for fault-proneness prediction, as no models using ROC Curves gave at least acceptable results. This could also explain why no threshold values were found for any of the 4 source code metrics using VARL methodology, because all 4 metrics gave p-value above the 5% confidence level. According to the Friedman test, there is no significant difference between ROC-2 and ROC-3 for duplicated datasets. However, the Nemenyi post-hoc test shows that ROC-3 gave slightly better performance than ROC-2.

Further analysis on other datasets would be needed to see if we could use the exact same experiments (ROC-2 and ROC-3) for other datasets. But, these results seem to indicate that the method would be viable for other datasets.

According to the results obtained in this experiment, we can answer positively to RQ1, which was:

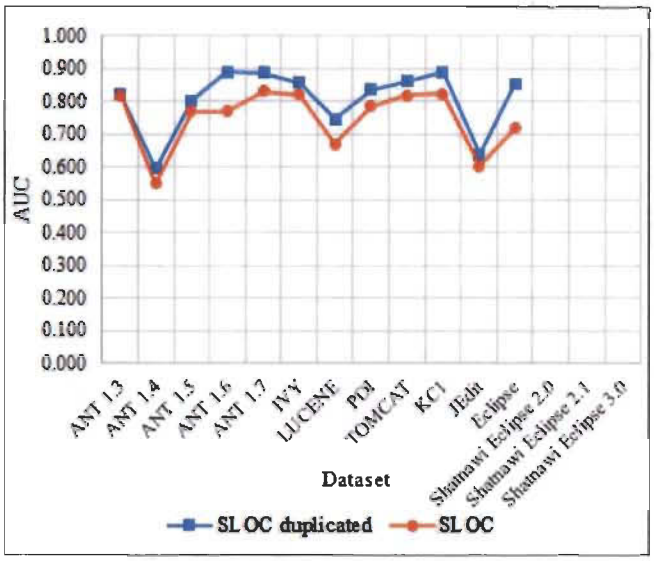
RQ1: Can the ROC Curves method produces threshold values for other datasets than the ones investigated in the original study and achieves good binary fault-proneness prediction performance?

The results obtained clearly show that the ROC Curves method calculates threshold values giving good fault-proneness prediction performance for the investigated datasets. We can therefore conclude that Shatnawi et al. experiment for binary classification is applicable for other software systems than Eclipse [2].

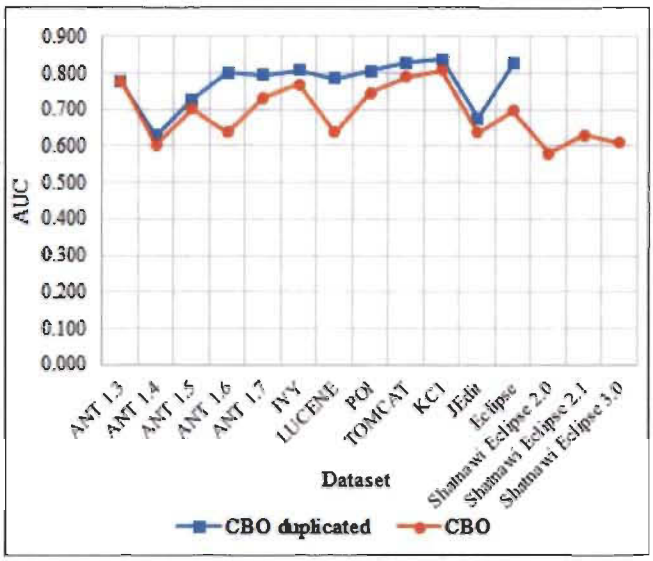
VARL results. VARL threshold calculation method gave the results presented in Tables 13–16 when predicting fault-proneness. There are no results for ANT 1.4, LUCENE, POI and KC1 because no valid threshold values could be obtained for these datasets using this method. In fact, all threshold values calculated for these datasets were below the minimum value of each metric they represented. Applying the model on these datasets would have considered all classes as fault-prone, which is not relevant.

Given those results, several observations can be made. First, the VARL-1 experiment for all datasets (except for JEdit) has a high FPR and low FNR, which explains the low g-mean values obtained. This can be explained by the high number of classes classified as fault-prone. For datasets like ANT 1.3, 1.5, 1.6, 1.7, IVY, TOMCAT and Eclipse, this behavior can easily be explained by the fact that a single or more threshold values given for a certain metric are very close to the minimum possible value of these metrics. This situation makes the classification consider most classes of each system as fault-prone, as almost all of them have at least one source code metric exceeding its threshold value.

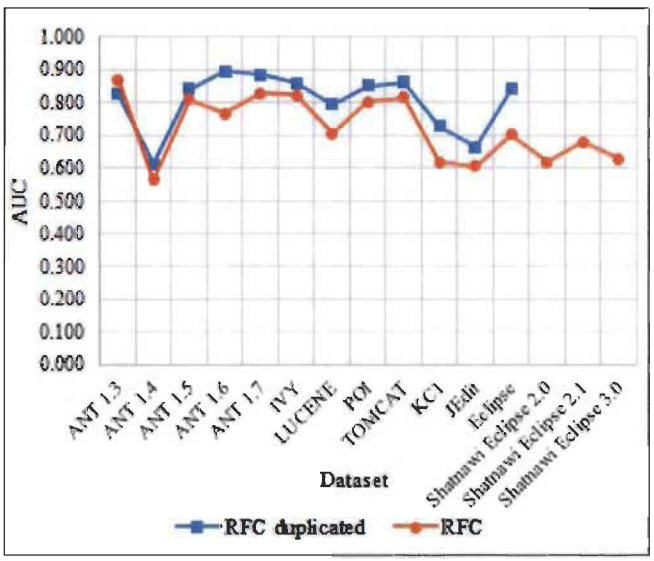
Secondly, we can see that for half of the datasets considered for VARL (ANT 1.3, ANT 1.5, ANT 1.7 and TOMCAT), the VARL-3 model gave the best results. For IVY and Eclipse, VARL-4 gave the best results.



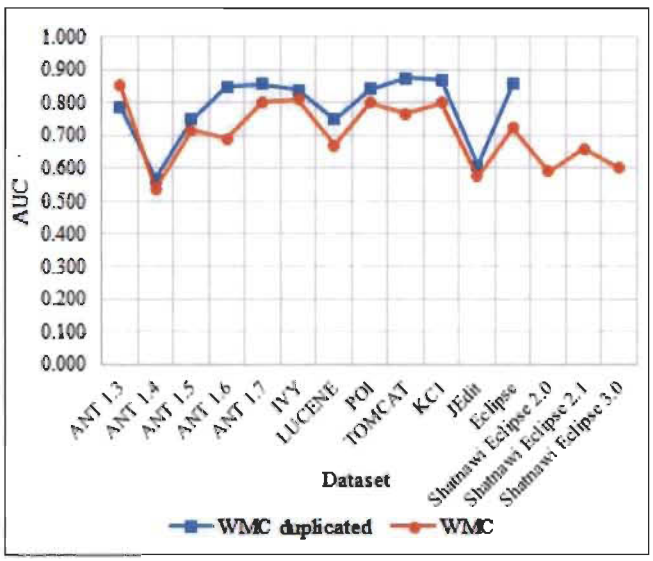
(a) Area Under Curve for SLOC metric



(b) Area Under Curve for CBO metric



(c) Area Under Curve for RFC metric



(d) Area Under Curve for WMC metric

Fig. 1. Area under curve for different source code metrics.

VARL-2 was considered best for ANT 1.6 and VARL-1 for JEdit. For JEdit, the prediction performance was not good at all, having a maximum g-mean value of 0.288. This could be explained by the fact that JEdit contains a lot of classes but only 12 have faults in the whole dataset. The results of this experiment were sometimes acceptable, fair or not good at all. The FPR was often a bit too high (which means that too many classes were classified as fault-prone), but the FNR was mostly acceptable. Out of the 8 datasets considered for VARL, five of them, which are ANT 1.3, ANT 1.5, ANT 1.7, IVY and TOMCAT, can be considered giving acceptable classification, with g-mean above or equal to 0.7. All of them fall in the acceptable range when considering the VARL-3 experiment.

Again, classification using the duplicated classes datasets gave better results than when using the original datasets. The resulting g-

Table 9
ROC Curves fault-proneness prediction performance for ANT 1.7 and IVY.

	ANT 1.7						IVY					
Model	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
ROC-1	0.406	0.157	0.708	0.406	0.086	0.737	0.494	0.075	0.684	0.494	0.054	0.692
ROC-2	0.183	0.307	0.752	0.183	0.198	0.809	0.196	0.250	0.777	0.196	0.179	0.813
ROC-3	0.114	0.416	0.720	0.114	0.251	0.814	0.147	0.300	0.773	0.147	0.232	0.809
ROC-4	0.071	0.542	0.652	0.071	0.382	0.758	0.090	0.475	0.691	0.090	0.429	0.721

Table 10
ROC Curves fault-proneness prediction performance for KC1 and Eclipse.

	KC1						Eclipse					
Model	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
ROC-1	0.388	0.050	0.762	0.388	0.021	0.774	0.263	0.286	0.725	0.263	0.176	0.779
ROC-2	0.282	0.283	0.717	0.282	0.160	0.776	0.154	0.374	0.728	0.154	0.246	0.799
ROC-3	0.094	0.467	0.695	0.094	0.278	0.809	0.095	0.427	0.720	0.095	0.278	0.808
ROC-4	0.012	0.817	0.426	0.012	0.592	0.635	0.066	0.485	0.693	0.066	0.334	0.789

Table 11
ROC Curves fault-proneness prediction performance for ANT 1.3 to 1.6.

	ANT 1.3			ANT 1.4			ANT 1.5			ANT 1.6		
Model	FPR-D	FNR-D	g-mean-D	FPR-D	FNR-D	g-mean-D	FPR-D	FNR-D	g-mean-D	FPR-D	FNR-D	g-mean-D
ROC-1	0.543	0.030	0.666	0.609	0.213	0.555	0.464	0.143	0.678	0.483	0.049	0.701
ROC-2	0.162	0.182	0.828	0.341	0.383	0.638	0.211	0.371	0.704	0.251	0.098	0.822
ROC-3	0.105	0.333	0.773	0.246	0.447	0.646	0.077	0.371	0.762	0.147	0.190	0.831
ROC-4	0.067	0.515	0.673	0.145	0.596	0.588	0.038	0.400	0.760	0.069	0.304	0.805

Table 12
ROC Curves fault-proneness prediction performance for LUCENE, POI, TOMCAT and JEdit.

	LUCENE			POI			TOMCAT			JEdit		
Model	FPR-D	FNR-D	g-mean-D	FPR-D	FNR-D	g-mean-D	FPR-D	FNR-D	g-mean-D	FPR-D	FNR-D	g-mean-D
ROC-1	0.409	0.171	0.700	0.503	0.062	0.683	0.342	0.096	0.771	0.364	0.333	0.651
ROC-2	0.219	0.334	0.721	0.304	0.116	0.784	0.198	0.175	0.813	0.189	0.417	0.688
ROC-3	0.131	0.413	0.714	0.205	0.144	0.825	0.138	0.254	0.802	0.096	0.417	0.726
ROC-4	0.058	0.593	0.619	0.099	0.270	0.811	0.077	0.377	0.758	0.040	0.500	0.693

mean value was better or equal in all experiments for the duplicated datasets. Note that for the experiment using VARL threshold values, the Friedman test could not be performed to compare the models' performance, as results are not available for all datasets using all models.

In summary, VARL-3 experiment seems to give acceptable results across most datasets for which VARL can be experimented (with 5 out of 8 datasets). Nevertheless, 4 datasets didn't give any usable threshold values and threshold values calculated can often be strange. This was observed previously with the RFC metric for JEdit which is above the maximal metric value for this dataset. Therefore, ROC Curves method seems to be a better choice to calculate threshold values and perform fault-proneness prediction. Since valid threshold values could not be calculated for all datasets and that fault-proneness prediction results are poor for certain datasets, we can answer negatively to RQ2, which was:

RQ2: Can the VARL methodology be considered as a good methodology to calculate threshold values to do fault-proneness prediction?

Alves Rankings results. Results obtained using the threshold values of the Alves Rankings method for fault-proneness prediction are presented in Tables 17–20.

The first conclusion we can make is that when using 3 metrics or more before considering a class as fault-prone, the FNR is often high (above 0.3). However, the model Alves-2 is better for most datasets than the variant using 1 or 3 metrics before classifying a class as fault-prone, as it yields a higher g-mean in most cases (for all datasets except ANT 1.3, ANT 1.5, ANT 1.7 and KC1, where the variant using 3 metrics is better). Results are also better when using duplicated datasets, as seen for other thresholds definition techniques, when fault-proneness is directly done using the threshold values calculated. This performance improvement denoted for duplicated datasets is also confirmed by the Friedman test.

The Friedman test showed that Alves-2 and Alves-3 experiments gave significantly better performance than Alves-1 and Alves-4 experiments (for experiments on both binary and duplicated datasets). However, the Friedman and Nemenyi tests showed no significant difference between the results obtained using Alves-2 and Alves-3. We can therefore assert that these two models are equivalent, even if Alves-3 seems to give high FNR. However, the Nemenyi test showed that Alves-2 is slightly better in performance than Alves-3, but not significantly.

The Friedman test gave a p-value of .014 when considering the models tested on binary datasets. As to the models tested on duplicated datasets, it gave a p-value of .002. Finally, when considering Alves-2 and Alves-3 for both binary and duplicated experiments, it gave a p-value of 0.

Following these results for Alves Rankings threshold values, we can answer positively to RQ3, which was:

RQ3: Can the Alves Rankings thresholds calculation technique produces threshold values achieving good fault-proneness performance?

Not only Alves Rankings gave good results for fault-proneness prediction, it also gave similar results to the ROC Curves method and clearly outperformed the VARL threshold values.

Following the results we got, we conclude that a model could be constructed using the Alves Rankings thresholds calculation technique for certain datasets, as it gave excellent, acceptable or fair fault-proneness prediction results. In fact, Alves-2 and Alves-3 seem to be the best models choice for Alves Rankings methodology.

Summary of results. Thresholds definition techniques gave acceptable results when applied to fault-proneness prediction, especially ROC Curves and Alves Rankings methods. In Table 21, we present a summary of the performance of fault-proneness prediction when using threshold values directly on the datasets to predict faulty classes. Each dataset results are summarized for the 2 best models constructed using each thresholds calculation technique (which are always for duplicated classification, as it gave better results than the

Table 13
VARL fault-proneness prediction performance for ANT 1.7 and IVY.

	ANT 1.7						IVY					
Model	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
VARL-1	0.708	0.030	0.532	0.708	0.018	0.535	0.990	0.000	0.098	0.990	0.000	0.098
VARL-2	0.489	0.090	0.682	0.489	0.062	0.692	0.494	0.075	0.684	0.490	0.054	0.694
VARL-3	0.392	0.163	0.713	0.392	0.101	0.739	0.388	0.150	0.721	0.388	0.107	0.739
VARL-4	–	–	–	–	–	–	0.304	0.250	0.722	0.304	0.196	0.748

Table 14
VARL fault-proneness prediction performance for Eclipse.

Model	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
VARL-1	0.947	0.019	0.228	0.947	0.019	0.228
VARL-2	0.750	0.097	0.475	0.750	0.061	0.485
VARL-3	0.558	0.165	0.608	0.558	0.104	0.630
VARL-4	0.442	0.199	0.668	0.442	0.126	0.698

binary one). To read the table, know that ++ means excellent classification (0.9 > g-mean ≥ 0.8), + means acceptable classification (0.8 > g-mean ≥ 0.7), 0 means fair classification (0.7 > g-mean ≥ 0.6), - means poor classification (0.6 > g-mean ≥ 0.5) and – means no good classification (g-mean < 0.5). Some cells of the table are marked as NA (Not available), since results using VARL were not available for this model.

From Table 21, the first observation we can make is that VARL methodology did not give results for 4 out of 12 investigated datasets. The ones that gave results were outperformed by ROC Curves and Alves Rankings methodologies, which gave results for all 12 datasets. The other observation we can make is that ROC Curves is the best of the 3 methods when thresholds are applied directly on the datasets, as the results are equal or better than those given by the VARL and Alves Rankings methodologies. This conclusion therefore answers RQ4, which was:

RQ4: Which thresholds calculation technique (considering ROC Curves, VARL and Alves Rankings) performs the best for fault-proneness prediction?

However, Alves Rankings performed well too, with results equal or a rank lower than those given by the ROC Curves method. The Friedman analysis determined that ROC Curves performed significantly better than Alves Rankings threshold values (with a p-value of 0). Of course, VARL threshold values were excluded from the Friedman analysis, since they didn't give results for all models and datasets.

However, the big advantage the Alves Rankings methodology has over ROC Curves and VARL is that it doesn't require fault data to calculate threshold values. This means that a completely non-supervised model could be built using this methodology. Most of the time, since fault data is not available in a real enterprise context, this is a major advantage, especially since results given are mostly acceptable and equivalent to ROC Curves. Furthermore, collection of quality fault data history can be expensive and time consuming [43]. Since ROC Curves and VARL use fault data history, they greatly depend on the quality and accuracy of this data to calculate relevant threshold values.

5.3. Supervised approaches results

This section presents results obtained for fault-proneness prediction using machine learning (Bayes Network, Artificial Neural Network, C4.5 and Support Vector Machine) and clustering (K-means and Self-Organizing Map) algorithms.

5.3.1. Machine learning algorithms results

Here are presented the fault-proneness prediction results for each of the machine learning algorithms. As previously mentioned, results are presented for two different experiments. First, the investigated algorithm is trained on the raw source code metrics to predict fault-prone code. The second experiment uses source code metrics binarized using threshold values (with either ROC Curves, VARL or Alves Rankings).

Bayes Network results. The Bayes Network fault-proneness prediction results on datasets using raw source code metrics are presented in Table 22.

The first conclusion that we can draw from these results is that the Bayes Network model performs better on duplicated datasets than on the original datasets, except for ANT 1.3 and ANT 1.5. This conclusion is validated by the Friedman analysis. The fact that duplicated classification gave better results than the binary one could be caused by the bigger folds produced for 10-fold cross-validation, which could enhance the classification performance of the algorithm, each fold having more training entries, therefore acting as boosting. The second fact we denote by analyzing Table 22 is that performance was acceptable or excellent for all datasets when using duplicated classification, except for ANT 1.4 and JEdit, for which performance was not good at all. In fact, for ANT 1.4 and JEdit, all classes were classified as not fault-prone, making the classification not useful at all. For LUCENE dataset, performance was close to the lower limit to be acceptable, as the FPR is somewhat high. This will make users invest testing effort on classes that are likely not fault-prone. For ANT 1.5, which g-mean was close to the lower limit of acceptable too, the FNR was high, which will make developers not investing testing effort on classes that are likely fault-prone, therefore leaving potential faulty classes untested (or not intensively tested). Since results were at least acceptable for 10 out of 12 datasets, this machine learning model could be considered for building acceptable models without using threshold values.

Table 23 presents results for the Bayes Network algorithm executed on the binarized datasets using ROC Curves thresholds, Table 24 presents results using VARL threshold values and Table 25 presents results using Alves Rankings thresholds at 30% of the distribution. Note that for VARL, metrics that don't have valid threshold values were excluded in the construction of the model instead of all setting them to 1.

The first observation that we can make is that classification using the ROC Curves threshold values are close to those obtained using Bayes Network on the raw source code values, with 10 out of 12 datasets giving at least acceptable results. Results are even better for half of the datasets and exactly the same for 2 of them (ANT 1.4 and JEdit), for which results are exactly the same.

As to VARL, this thresholds calculation method gave acceptable results for 3 datasets (ANT 1.7, IVY and TOMCAT), but gave fair or not good results for the other datasets. Since out of 12 datasets, VARL gave at least acceptable results for only 3 datasets, it doesn't seem to be an acceptable choice to be used for fault-proneness prediction, even when using a machine learning algorithm like Bayes Network.

Alves Rankings gave results close to those given by Bayes Network applied alone and to those using ROC Curves thresholds. Out of 12 datasets, 9 gave at least acceptable results and one gave fair ones. As when considering thresholds-only based models, ROC Curves seems to

Table 15
VARL fault-proneness prediction performance for ANT 1.3, 1.5 and 1.6.

	ANT 1.3			ANT 1.5			ANT 1.6		
Model	FPR-D	FNR-D	g-mean-D	FPR-D	FNR-D	g-mean-D	FPR-D	FNR-D	g-mean-D
VARL-1	0.971	0.000	0.169	0.973	0.000	0.164	0.849	0.000	0.388
VARL-2	0.419	0.091	0.727	0.556	0.057	0.647	0.517	0.049	0.678
VARL-3	0.343	0.182	0.733	0.280	0.257	0.731	–	–	–

Table 16
VARL fault-proneness prediction performance for TOMCAT and JEdit.

	TOMCAT			JEdit		
Model	FPR-D	FNR-D	g-mean-D	FPR-D	FNR-D	g-mean-D
VARL-1	0.625	0.053	0.596	0.002	0.917	0.288
VARL-2	0.323	0.140	0.763	0.000	1.000	0.000
VARL-3	0.237	0.184	0.789	–	–	–
VARL-4	0.152	0.272	0.786	–	–	–

be the best choice, but Alves Rankings gave acceptable results too.

When comparing all models built using the Bayes Network algorithm, the best models retained were the ones performed on the raw source code metrics and on the binarized datasets using ROC Curves threshold values. Both models performed significantly better than the others. According to the Nemenyi test, the model built using the binarized datasets with ROC Curves threshold values performed slightly better than the one using the raw source code metrics.

The Friedman test gave a p-value of .014 when considering the Bayes Network applied on binary datasets. When considering duplicated datasets, it gave a p-value of .014. When considering the 2 best models of both binary and duplicated datasets results (applied on raw source code metrics and on binarized datasets using ROC Curves), we obtained a p-value of .005.

Artificial Neural Network results. The Artificial Neural Network algorithm, or more precisely, Multilayer Perceptron, is provided by the Weka tool and was executed keeping all parameters to their default values. The results obtained on raw source code metrics from this experiment are presented in Table 26.

The first conclusion is that, again, duplicated fault-proneness prediction gave better or equal results than the binary one, except for LUCENE and POI. The second observation we denote is that performance was lower than Bayes Network, as only 5 datasets gave at least acceptable results. Two other datasets gave fair results, and the others were considered worse. The problem with most datasets is that either FPR or FNR is usually too high to be considered acceptable. Maybe Weka's ANN default configuration is not optimal for fault-proneness prediction. Investigations would need to be done to find out if a better configuration of the ANN could be made to get better performance. Maybe some preprocessing on the input data could also improve performance, like outliers removal.

Considering binarized datasets used with the Artificial Neural Network algorithm, Table 27 presents results using ROC Curves thresholds, Table 28 presents results using VARL threshold values and Table 29 presents results using Alves Rankings thresholds at 30% of the distribution.

One surprising fact is that ROC Curves thresholds made a big difference in fault-prediction performance, giving better results for 8 out of 12 datasets (and same results for 1 dataset) than when using the ANN algorithm on the raw values. This could be explained by the fact that the ANN algorithm better learns patterns when data is binarized than when it is normalized (as done by default for the ANN algorithm for Weka). ROC Curves thresholds gave at least acceptable results for 8 out of 12 datasets, which is better than when ANN is applied on the raw source code metrics, where only 5 offered at least the same performance. Nonetheless, the ANN algorithm seems to give inconsistent results, as some non-duplicated datasets had a g-mean of 0 for binary classification and got fair and even acceptable results for the duplicated classification.

As for results using the VARL threshold values, we can conclude that duplicated classification is better than the binary one, probably because duplication of entries acts as boosting. Nevertheless, results are not good, with a g-mean of 0 for all binary results and for 5 out of 8 datasets of duplicated classification. It gave acceptable results for a single dataset, which doesn't make it an acceptable choice to be used in conjunction with ANN.

Alves Rankings thresholds used with ANN gave at least acceptable results for 5 datasets, fair results for 1 dataset and the 6 others are not considered acceptable. This is close to the performance of the ANN algorithm applied on the raw source code metrics.

When performing the Friedman test, several models were considered significantly better than others. The best results were obtained on the duplicated datasets using binarized source code metrics with ROC Curves and Alves Rankings threshold values, along with the ANN algorithm applied on the raw metric values. The binarized datasets using Alves Rankings threshold values also gave good results. Following the Nemenyi test, the model built on duplicated datasets binarized using Alves Rankings threshold values performed slightly better than the other models.

When performing the Friedman analysis with the models applied on the binary datasets, we obtained a p-value of .249, which indicates that the difference between the results is not statistically significant. For the duplicated datasets tests, the p-value obtained is 0.035 and the results are therefore different. However, the Nemenyi test concludes that the models do not show a significant statistical difference. As to the Friedman test applied on all the models using ANN, we obtained a p-value of .001, which shows a significant difference between the models. According to the Nemenyi test, the best models using ANN are the ones using duplicated datasets.

C4.5 results. The C4.5 algorithm is provided by the Weka tool API and

Table 17
Alves Rankings fault-proneness prediction performance for ANT 1.7 and IVY.

	ANT 1.7						IVY					
Model	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
Alves-1	0.485	0.127	0.671	0.485	0.074	0.690	0.356	0.225	0.707	0.356	0.161	0.735
Alves-2	0.238	0.265	0.748	0.238	0.172	0.794	0.170	0.350	0.735	0.170	0.286	0.770
Alves-3	0.155	0.373	0.727	0.155	0.231	0.806	0.128	0.425	0.708	0.128	0.339	0.759
Alves-4	0.088	0.500	0.675	0.088	0.364	0.762	0.090	0.550	0.640	0.090	0.482	0.687

Table 18
Alves Rankings fault-proneness prediction performance for KC1 and Eclipse.

	KC1						Eclipse					
Model	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
Alves-1	0.459	0.267	0.630	0.459	0.173	0.669	0.223	0.335	0.719	0.223	0.209	0.784
Alves-2	0.224	0.433	0.663	0.224	0.269	0.753	0.095	0.456	0.702	0.095	0.307	0.792
Alves-3	0.082	0.583	0.618	0.082	0.339	0.779	0.076	0.495	0.683	0.076	0.340	0.781
Alves-4	0.035	0.733	0.507	0.035	0.499	0.695	0.053	0.544	0.657	0.053	0.372	0.771

Table 19
Alves Rankings fault-proneness prediction performance for ANT 1.3 to 1.6.

	ANT 1.3			ANT 1.4			ANT 1.5			ANT 1.6		
Model	FPR-D	FNR-D	g-mean-D	FPR-D	FNR-D	g-mean-D	FPR-D	FNR-D	g-mean-D	FPR-D	FNR-D	g-mean-D
Alves-1	0.648	0.030	0.585	0.594	0.213	0.565	0.621	0.114	0.580	0.483	0.049	0.701
Alves-2	0.343	0.182	0.733	0.304	0.404	0.644	0.326	0.257	0.708	0.205	0.152	0.821
Alves-3	0.219	0.242	0.769	0.254	0.574	0.564	0.203	0.314	0.739	0.124	0.234	0.820
Alves-4	0.124	0.424	0.710	0.181	0.596	0.575	0.130	0.400	0.722	0.069	0.321	0.795

Table 20
Alves Rankings fault-Proneness prediction performance for LUCENE, POI, TOMCAT and JEdit.

	LUCENE			POI			TOMCAT			JEdit		
Model	FPR-D	FNR-D	g-mean-D	FPR-D	FNR-D	g-mean-D	FPR-D	FNR-D	g-mean-D	FPR-D	FNR-D	g-mean-D
Alves-1	0.474	0.155	0.666	0.379	0.112	0.743	0.426	0.079	0.727	0.565	0.333	0.538
Alves-2	0.175	0.402	0.702	0.106	0.364	0.754	0.206	0.184	0.805	0.285	0.333	0.690
Alves-3	0.109	0.528	0.648	0.081	0.454	0.708	0.133	0.316	0.770	0.206	0.417	0.681
Alves-4	0.029	0.641	0.591	0.031	0.584	0.635	0.090	0.360	0.764	0.114	0.417	0.719

Table 21
Summary of fault-proneness performance for thresholds-based models.

Dataset	ROC-2	ROC-3	VARL-2	VARL-3	Alves-2	Alves-3
ANT 1.3	+ +	+	+	+	+	+
ANT 1.4	0	0	NA	NA	0	–
ANT 1.5	+	+	0	+	+	+
ANT 1.6	+ +	+ +	0	NA	+ +	+ +
ANT 1.7	+ +	+ +	0	+	+	+ +
IVY	+ +	+ +	0	+	+	+
LUCENE	+	+	NA	NA	+	0
POI	+	+ +	NA	NA	+	+
TOMCAT	+ +	+ +	+	+	+ +	+
KC1	+	+ +	NA	NA	+	+
JEdit	0	+	—	NA	0	0
Eclipse	+	+ +	—	0	+	+

Table 22
Bayes Network fault-proneness prediction performance using raw datasets.

Dataset	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
ANT 1.3	0.114	0.250	0.815	0.152	0.303	0.769
ANT 1.4	0.000	1.000	0.000	0.000	1.000	0.000
ANT 1.5	0.065	0.438	0.725	0.077	0.457	0.708
ANT 1.6	0.162	0.326	0.751	0.151	0.190	0.829
ANT 1.7	0.168	0.319	0.753	0.135	0.260	0.800
IVY	0.144	0.350	0.746	0.138	0.232	0.814
LUCENE	0.387	0.350	0.631	0.365	0.222	0.703
POI	0.280	0.178	0.770	0.273	0.130	0.795
TOMCAT	0.131	0.416	0.713	0.123	0.333	0.765
KC1	0.424	0.100	0.720	0.388	0.010	0.778
JEdit	0.000	1.000	0.000	0.000	1.000	0.000
Eclipse	0.096	0.461	0.698	0.101	0.294	0.797

was therefore performed as is, keeping all default parameters for the execution of the classification algorithm. All results for this experiment when applied on the raw source code metrics are presented in Table 30.

The C4.5 performed a bit better than the ANN algorithm, having

Table 23
Bayes Network fault-proneness prediction performance using datasets binarized with ROC Curves thresholds.

Dataset	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
ANT 1.3	0.105	0.250	0.819	0.105	0.333	0.773
ANT 1.4	0.000	1.000	0.000	0.000	1.000	0.000
ANT 1.5	0.057	0.406	0.748	0.073	0.371	0.763
ANT 1.6	0.147	0.304	0.770	0.147	0.190	0.831
ANT 1.7	0.116	0.380	0.741	0.126	0.228	0.821
IVY	0.147	0.300	0.773	0.147	0.232	0.809
LUCENE	0.328	0.335	0.668	0.409	0.171	0.700
POI	0.304	0.174	0.758	0.304	0.118	0.783
TOMCAT	0.138	0.338	0.755	0.138	0.254	0.802
KC1	0.282	0.283	0.717	0.282	0.160	0.776
JEdit	0.000	1.000	0.000	0.000	1.000	0.000
Eclipse	0.125	0.408	0.720	0.125	0.267	0.801

Table 24
Bayes Network fault-proneness prediction performance using datasets binarized with VARL thresholds.

Dataset	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
ANT 1.3	0.143	0.800	0.414	0.333	0.303	0.682
ANT 1.5	0.000	1.000	0.000	0.000	1.000	0.000
ANT 1.6	0.000	1.000	0.000	0.517	0.049	0.678
ANT 1.7	0.392	0.163	0.713	0.392	0.101	0.739
IVY	0.304	0.250	0.722	0.304	0.196	0.748
TOMCAT	0.152	0.364	0.734	0.152	0.272	0.786
JEdit	0.000	1.000	0.000	0.000	1.000	0.000
Eclipse	0.000	1.000	0.000	0.442	0.126	0.698

classification yielding at least acceptable results for 6 out of 12 datasets and fair results for 2 others. Duplicated classification again gave better results than the binary one, except for ANT 1.5. The FPR or FNR are often too high, which makes classification bad. Maybe that some pre-processing and configuration changes, like mentioned for the ANN experiment, would make this algorithm perform better for fault-proneness

Table 25
Bayes Network fault-proneness prediction performance using datasets binarized with Alves Rankings thresholds.

Dataset	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
ANT 1.3	0.143	0.600	0.586	0.162	0.303	0.764
ANT 1.4	0.000	1.000	0.000	0.000	1.000	0.000
ANT 1.5	0.123	0.656	0.549	0.157	0.371	0.728
ANT 1.6	0.127	0.348	0.754	0.127	0.217	0.826
ANT 1.7	0.155	0.373	0.727	0.157	0.210	0.816
IVY	0.128	0.425	0.708	0.128	0.339	0.759
LUCENE	0.453	0.246	0.642	0.474	0.155	0.666
POI	0.354	0.160	0.737	0.354	0.120	0.754
TOMCAT	0.133	0.403	0.720	0.133	0.316	0.770
KC1	0.165	0.517	0.635	0.318	0.205	0.737
JEdit	0.000	1.000	0.000	0.000	1.000	0.000
Eclipse	0.095	0.456	0.702	0.095	0.307	0.792

Table 26
ANN fault-proneness prediction performance using raw datasets.

Dataset	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
ANT 1.3	0.076	0.650	0.569	0.086	0.545	0.645
ANT 1.4	0.036	1.000	0.000	0.080	0.851	0.370
ANT 1.5	0.046	0.656	0.573	0.042	0.600	0.619
ANT 1.6	0.116	0.402	0.727	0.158	0.212	0.814
ANT 1.7	0.079	0.548	0.645	0.107	0.296	0.793
IVY	0.019	0.775	0.470	0.026	0.679	0.560
LUCENE	0.372	0.300	0.663	0.788	0.038	0.451
POI	0.379	0.167	0.719	0.447	0.086	0.711
TOMCAT	0.001	0.974	0.161	0.024	0.702	0.539
KC1	0.200	0.417	0.683	0.471	0.006	0.725
JEdit	0.002	1.000	0.000	0.006	1.000	0.000
Eclipse	0.046	0.592	0.624	0.071	0.356	0.774

Table 27
ANN fault-proneness prediction performance using datasets binarized with ROC Curves thresholds.

Dataset	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
ANT 1.3	0.086	0.300	0.800	0.095	0.212	0.844
ANT 1.4	0.007	1.000	0.000	0.087	0.894	0.312
ANT 1.5	0.042	0.406	0.754	0.050	0.371	0.773
ANT 1.6	0.097	0.424	0.721	0.120	0.212	0.833
ANT 1.7	0.102	0.470	0.690	0.117	0.246	0.816
IVY	0.000	1.000	0.000	0.077	0.518	0.667
LUCENE	0.380	0.296	0.661	1.000	0.000	0.000
POI	0.230	0.206	0.782	0.292	0.124	0.788
TOMCAT	0.000	1.000	0.000	0.073	0.430	0.727
KC1	0.188	0.250	0.780	0.365	0.021	0.789
JEdit	0.000	1.000	0.000	0.000	1.000	0.000
Eclipse	0.071	0.481	0.695	0.072	0.307	0.802

prediction. According to the Friedman test, the results for duplicated datasets were not significantly better than those for the original datasets.

Considering binarized datasets used with the C4.5 algorithm, Table 31 presents results using ROC Curves thresholds, Table 32 presents results using VARL threshold values and Table 33 presents results using Alves Rankings thresholds at 30% of the distribution.

Binarization using ROC Curves thresholds for the C4.5 experiment gave better results for 9 out of 12 datasets and the same ones for 1 dataset. Some results, like for ANN, are inconsistent, since binary classification gave a g-mean of 0 and their duplicated counterpart gave fair or acceptable results, probably because of the boosting concept mentioned for each machine learning algorithm so far. Like the results given for the binarized version of the datasets using ROC Curves thresholds and the ANN algorithm, C4.5 using ROC Curves thresholds has more datasets with at least acceptable results (8 out of 12), but as

Table 28
ANN fault-proneness prediction performance using datasets binarized with VARL thresholds.

Dataset	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
ANT 1.3	0.000	1.000	0.000	0.000	1.000	0.000
ANT 1.5	0.000	1.000	0.000	0.000	1.000	0.000
ANT 1.6	0.000	1.000	0.000	0.479	0.147	0.667
ANT 1.7	0.000	1.000	0.000	0.351	0.186	0.727
IVY	0.000	1.000	0.000	0.000	1.000	0.000
TOMCAT	0.000	1.000	0.000	0.000	1.000	0.000
JEdit	0.002	1.000	0.000	0.002	1.000	0.000
Eclipse	0.000	1.000	0.000	0.250	0.465	0.633

Table 29
ANN fault-proneness prediction performance using datasets binarized with Alves Rankings thresholds.

Dataset	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
ANT 1.3	0.095	0.850	0.368	0.162	0.394	0.713
ANT 1.4	0.051	0.950	0.218	0.072	0.915	0.281
ANT 1.5	0.004	1.000	0.000	0.008	1.000	0.000
ANT 1.6	0.081	0.435	0.721	0.097	0.261	0.817
ANT 1.7	0.109	0.422	0.718	0.112	0.246	0.818
IVY	0.000	1.000	0.000	0.077	0.750	0.480
LUCENE	0.423	0.266	0.651	0.956	0.025	0.207
POI	0.342	0.171	0.739	0.354	0.126	0.751
TOMCAT	0.000	1.000	0.000	0.069	0.570	0.633
KC1	0.094	0.550	0.638	1.000	0.000	0.000
JEdit	0.000	1.000	0.000	0.000	1.000	0.000
Eclipse	0.059	0.539	0.659	0.073	0.356	0.773

Table 30
C4.5 fault-proneness prediction performance using raw satasets.

Dataset	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
ANT 1.3	0.105	0.550	0.635	0.171	0.364	0.726
ANT 1.4	0.000	1.000	0.000	0.043	0.851	0.377
ANT 1.5	0.061	0.594	0.618	0.057	0.600	0.614
ANT 1.6	0.108	0.391	0.737	0.131	0.245	0.810
ANT 1.7	0.097	0.410	0.730	0.104	0.293	0.796
IVY	0.022	0.825	0.414	0.019	0.714	0.529
LUCENE	0.482	0.192	0.647	0.555	0.036	0.655
POI	0.329	0.192	0.736	0.335	0.100	0.773
TOMCAT	0.003	0.831	0.410	0.019	0.649	0.587
KC1	0.388	0.083	0.749	0.329	0.004	0.817
JEdit	0.000	1.000	0.000	0.000	1.000	0.000
Eclipse	0.047	0.626	0.597	0.067	0.348	0.780

mentioned before, results are sometimes inconsistent.

Results given by binarized datasets using VARL threshold values gave results similar to ANN. A single dataset is considered giving acceptable classification and another one giving fair classification for duplicated classification. All the other datasets gave a g-mean value of 0, as all classes are classified as not fault-prone.

Alves Rankings methodology produced at least acceptable results for 5 datasets, but all others provided not good classification. Like other thresholds definition techniques, results are inconsistent between binary and duplicated classification, as g-mean is sometimes 0 for one and a lot higher for the other one.

After performing the Friedman test, we concluded that the models using binarized datasets with ROC Curves threshold values and the raw source code metrics performed significantly better. This is true for both original and duplicated datasets investigated with the C4.5 algorithm.

The Friedman test showed no significant statistical difference for the C4.5 models applied on binary fault-proneness datasets, with a p-value of .052. However, when C4.5 is applied on duplicated datasets, we obtained a p-value of .032 for the Friedman test. When comparing all

Table 31
C4.5 fault-proneness prediction performance using datasets binarized with ROC Curves thresholds.

Dataset	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
ANT 1.3	0.095	0.250	0.824	0.114	0.273	0.803
ANT 1.4	0.000	1.000	0.000	0.000	1.000	0.000
ANT 1.5	0.057	0.406	0.748	0.057	0.371	0.770
ANT 1.6	0.108	0.348	0.763	0.112	0.212	0.837
ANT 1.7	0.100	0.428	0.718	0.116	0.260	0.809
IVY	0.000	1.000	0.000	0.090	0.554	0.637
LUCENE	0.394	0.271	0.665	1.000	0.000	0.000
POI	0.248	0.206	0.772	0.311	0.126	0.776
TOMCAT	0.000	1.000	0.000	0.076	0.447	0.715
KC1	0.212	0.317	0.734	0.365	0.021	0.789
JEdit	0.000	1.000	0.000	0.000	1.000	0.000
Eclipse	0.071	0.476	0.698	0.081	0.329	0.785

Table 32
C4.5 fault-proneness prediction performance using datasets binarized with VARL thresholds.

Dataset	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
ANT 1.3	0.000	1.000	0.000	0.000	1.000	0.000
ANT 1.5	0.000	1.000	0.000	0.000	1.000	0.000
ANT 1.6	0.000	1.000	0.000	0.517	0.049	0.678
ANT 1.7	0.000	1.000	0.000	0.408	0.092	0.734
IVY	0.000	1.000	0.000	0.000	1.000	0.000
TOMCAT	0.000	1.000	0.000	0.000	1.000	0.000
JEdit	0.000	1.000	0.000	0.000	1.000	0.000
Eclipse	0.000	1.000	0.000	0.000	1.000	0.000

Table 33
C4.5 fault-proneness prediction performance using datasets binarized with Alves Rankings thresholds.

Dataset	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
ANT 1.3	0.000	1.000	0.000	0.190	0.333	0.735
ANT 1.4	0.000	1.000	0.000	0.000	1.000	0.000
ANT 1.5	0.000	1.000	0.000	0.000	1.000	0.000
ANT 1.6	0.069	0.424	0.732	0.108	0.239	0.824
ANT 1.7	0.112	0.392	0.735	0.119	0.246	0.815
IVY	0.000	1.000	0.000	0.051	0.875	0.344
LUCENE	0.438	0.281	0.636	1.000	0.000	0.000
POI	0.335	0.189	0.734	0.354	0.118	0.755
TOMCAT	0.000	1.000	0.000	0.051	0.746	0.491
KC1	0.071	0.633	0.584	1.000	0.000	0.000
JEdit	0.000	1.000	0.000	0.000	1.000	0.000
Eclipse	0.064	0.515	0.674	0.073	0.345	0.779

models built on binary datasets and the ones using the raw and ROC Curves binarized source code metrics (the best ones), we obtained a *p*-value of 0. The Nemenyi test showed a better performance when C4.5 is used on duplicated datasets.

Support Vector Machine results. The Support Vector Machine (SVM) algorithm is provided by the Weka tool API. It was performed keeping all default parameters for the classification algorithm. All results for this experiment when applied on the raw source code metrics are presented in Table 34.

The Support Vector Machine algorithm didn't have a good performance, as only 3 out of 12 datasets gave acceptable performance. Again, duplicated classification gave same or better results for all datasets, except for KC1, where results are a lot better for binary classification. Considering that only 3 datasets gave acceptable results, we reconsidered Malhotra study stating that Support Vector Machine gave the best results of all the machine learning algorithms they used [7]. We then noticed that on the 5 datasets they tested their models on, 3 of them didn't give results for SVM and had an AUC of 0.5, which makes us think that like us, the algorithm simply classified the totality of classes as not fault-prone or fault-prone.

Table 34
Support Vector Machine fault-proneness prediction performance using raw datasets.

Dataset	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
ANT 1.3	0.000	1.000	0.000	0.010	0.970	0.173
ANT 1.4	0.000	1.000	0.000	0.000	1.000	0.000
ANT 1.5	0.000	1.000	0.000	0.000	1.000	0.000
ANT 1.6	0.050	0.717	0.518	0.097	0.332	0.777
ANT 1.7	0.024	0.717	0.526	0.064	0.382	0.761
IVY	0.003	1.000	0.000	0.010	0.821	0.421
LUCENE	1.000	0.000	0.000	1.000	0.000	0.000
POI	1.000	0.000	0.000	1.000	0.000	0.000
TOMCAT	0.000	1.000	0.000	0.008	0.772	0.476
KC1	0.224	0.350	0.710	1.000	0.000	0.000
JEdit	0.000	1.000	0.000	0.000	1.000	0.000
Eclipse	0.010	0.850	0.386	0.034	0.481	0.708

Table 35
Support Vector Machine fault-proneness prediction performance using datasets binarized with ROC Curves thresholds.

Dataset	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
ANT 1.3	0.086	0.400	0.741	0.086	0.455	0.706
ANT 1.4	0.000	1.000	0.000	0.000	1.000	0.000
ANT 1.5	0.057	0.563	0.642	0.061	0.371	0.768
ANT 1.6	0.112	0.348	0.761	0.112	0.212	0.837
ANT 1.7	0.121	0.410	0.720	0.130	0.251	0.807
IVY	0.000	1.000	0.000	0.103	0.482	0.682
LUCENE	0.321	0.384	0.647	1.000	0.000	0.000
POI	0.261	0.206	0.766	0.304	0.122	0.782
TOMCAT	0.000	1.000	0.000	0.000	1.000	0.000
KC1	0.282	0.200	0.758	0.353	0.027	0.794
JEdit	0.000	1.000	0.000	0.000	1.000	0.000
Eclipse	0.095	0.461	0.698	0.114	0.305	0.785

Table 36
Support Vector Machine fault-proneness prediction performance using datasets binarized with VARL thresholds.

Dataset	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
ANT 1.3	0.000	1.000	0.000	0.000	1.000	0.000
ANT 1.5	0.000	1.000	0.000	0.000	1.000	0.000
ANT 1.6	0.000	1.000	0.000	0.517	0.049	0.678
ANT 1.7	0.000	1.000	0.000	0.434	0.086	0.720
IVY	0.000	1.000	0.000	0.000	1.000	0.000
TOMCAT	0.000	1.000	0.000	0.000	1.000	0.000
JEdit	0.000	1.000	0.000	0.000	1.000	0.000
Eclipse	0.000	1.000	0.000	0.000	1.000	0.000

Considering binarized datasets used with the SVM algorithm, Table 35 presents results using ROC Curves thresholds, Table 36 presents results using VARL threshold values and Table 37 presents results using Alves Rankings thresholds at 30% of the distribution.

ROC Curves thresholds binarization gave at least acceptable results for 7 out of 12 datasets, which is better than the algorithm applied directly on the class metrics. Nevertheless, this is not so good, as results are inconsistent like for other machine learning algorithms.

SVM using VARL threshold values gave similar results than with ANN and C4.5, with most datasets having a g-mean value of 0.

Alves Rankings threshold values gave at least acceptable results for 5 out of 12 datasets, which like ROC Curves is better than the machine learning algorithm applied alone. Like for ROC Curves, other datasets gave results that are not considered acceptable and are sometimes inconsistent.

Table 37
Support Vector Machine fault-proneness prediction performance using datasets binarized with Alves Rankings thresholds.

Dataset	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
ANT 1.3	0.000	1.000	0.000	0.181	0.333	0.739
ANT 1.4	0.000	1.000	0.000	0.000	1.000	0.000
ANT 1.5	0.000	1.000	0.000	0.000	1.000	0.000
ANT 1.6	0.108	0.359	0.756	0.108	0.217	0.835
ANT 1.7	0.140	0.367	0.738	0.140	0.234	0.812
IVY	0.000	1.000	0.000	0.029	0.929	0.263
LUCENE	0.380	0.315	0.652	1.000	0.000	0.000
POI	0.304	0.189	0.751	0.304	0.138	0.774
TOMCAT	0.000	1.000	0.000	0.000	1.000	0.000
KC1	0.094	0.600	0.602	1.000	0.000	0.000
JEdit	0.000	1.000	0.000	0.000	1.000	0.000
Eclipse	0.087	0.495	0.679	0.088	0.332	0.781

The Friedman test showed a significant difference between experiments using the binary fault-proneness datasets, with a p-value of .008. It noted a significant difference between SVM models applied on duplicated datasets, with a p-value of .035, but the post-hoc Nemenyi test didn't. The same conclusion comes up when comparing the best SVM models tested on binary fault-proneness datasets (using ROC Curves and Alves Rankings binarized source code metrics) and on duplicated datasets. This test gave a p-value of .037, which shows a significant difference, but the post-hoc Nemenyi test didn't. The Nemenyi test, however, showed that binarized datasets using ROC Curves (binary and duplicated) and Alves Rankings (duplicated) threshold values gave slightly better results than SVM applied on the raw source code metrics.

Summary of results. Table 38 summarizes the results obtained from the machine learning algorithms and reuses the same legend as presented for Table 21. Results for duplicated classification are presented, as they gave better results. The columns marked with RAW represent the results obtained performing the machine learning algorithm on raw source code metrics (without binarization).

The first element we notice from this table is that no good results were obtained with any machine learning algorithm for ANT 1.4 and JEdit datasets. LUCENE dataset only got acceptable results with Bayes Network algorithm applied on raw metrics and using ROC Curves thresholds. We also see that VARL thresholds didn't give acceptable results when applied before the machine learning models, as 3 results were at least acceptable for Bayes Network and a single one for each ANN, C4.5 and SVM algorithms. We also notice that ROC Curves thresholds gave better results than the machine learning algorithms applied alone. This is confirmed by the Friedman and Nemenyi tests for Bayes Network. For ANN, C4.5 and SVM, this is also true, but not significantly. Alves Rankings gave good results when used with Bayes Network, but not with the other 3 machine learning algorithms. Bayes Network seems the best machine learning algorithm for fault-proneness prediction, using thresholds or not. It gave better results than all the other 3 machine learning algorithms. Regarding thresholds definition techniques, ROC Curves is first again, while Alves Rankings is second and VARL is again last. So far, ROC Curves seems the best thresholds definition technique to use.

Table 39
K-means fault-proneness prediction performance using raw datasets.

Dataset	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
ANT 1.3	0.114	0.350	0.759	0.124	0.333	0.764
ANT 1.4	0.225	0.675	0.502	0.210	0.638	0.535
ANT 1.5	0.142	0.406	0.714	0.130	0.371	0.739
ANT 1.6	0.077	0.554	0.641	0.054	0.527	0.669
ANT 1.7	0.054	0.608	0.609	0.016	0.627	0.606
IVY	0.080	0.500	0.678	0.054	0.429	0.735
LUCENE	0.912	0.110	0.279	1.000	0.000	0.000
POI	0.994	0.040	0.077	1.000	0.022	0.000
TOMCAT	0.069	0.584	0.622	0.068	0.500	0.683
KC1	0.247	0.350	0.700	1.000	0.137	0.000
JEdit	0.013	0.800	0.444	0.019	0.818	0.422
Eclipse	0.008	0.869	0.361	0.008	0.802	0.443

5.3.2. Clustering algorithms results

In this part, fault-proneness prediction results for each of the clustering algorithms are presented. The results calculated on the raw source code metrics are presented first, shortly followed by the results on binarized datasets using threshold values.

K-means results. The results obtained by performing the K-means fault-proneness prediction on raw source code metrics are presented in Table 39. The results are computed using the SimpleKMeans algorithm model provided in the Weka tool [31], keeping all parameters at their default values.

Results are at least acceptable for only 3 out of 12 datasets and duplicated classification performed better than binary classification. ANT 1.7, LUCENE, POI, KC1 and JEdit got better results when binary classification is used. The FPR or FNR are often too high to be considered acceptable. This could be caused by the fact that 2 clusters only are built using source code metrics. Having only 2 clusters must make clusters regroup a lot of classes, having almost 50% of classes in each cluster. Maybe considering more clusters would have a positive impact on the fault-proneness prediction performance. Results vary a lot from one dataset to another.

Table 40 presents results for the K-means algorithm executed on the binarized datasets using ROC Curves thresholds, Table 41 presents results using VARL threshold values and Table 42 presents results using Alves Rankings thresholds at 30% of the distribution.

For the results obtained using ROC Curves thresholds, results are

Table 38
Summary of fault-proneness performance for machine learning models.

	Bayes Network				ANN				C4.5				SVM			
Dataset	RAW	ROC	VARL	Alves	RAW	ROC	VARL	Alves	RAW	ROC	VARL	Alves	RAW	ROC	VARL	Alves
ANT 1.3	+	+	0	+	0	++	—	+	+	++	—	+	—	+	—	+
ANT 1.4	—	—	NA	—	—	—	NA	—	—	—	NA	—	—	—	NA	—
ANT 1.5	+	+	—	+	0	+	—	—	0	+	—	—	—	+	—	—
ANT 1.6	++	++	0	++	++	++	0	++	++	++	0	++	+	++	0	++
ANT 1.7	++	++	+	++	+	++	+	++	+	++	+	++	+	++	+	++
IVY	++	++	+	+	-	0	—	—	-	0	—	—	—	0	—	—
LUCENE	+	+	NA	0	—	—	NA	—	0	—	NA	—	—	—	NA	—
POI	+	+	NA	+	+	+	NA	+	+	+	NA	+	—	+	NA	+
TOMCAT	+	++	+	+	-	+	—	0	-	+	—	—	—	—	—	—
KC1	+	+	NA	+	+	+	NA	—	++	+	NA	—	—	+	NA	—
JEdit	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Eclipse	+	++	0	+	+	++	0	+	+	+	—	+	+	+	—	+

Table 40
K-means fault-proneness prediction performance using datasets binarized with ROC Curves thresholds.

Dataset	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
ANT 1.3	0.276	0.200	0.761	0.229	0.212	0.780
ANT 1.4	0.391	0.350	0.629	0.442	0.404	0.577
ANT 1.5	0.284	0.250	0.733	0.268	0.286	0.723
ANT 1.6	0.220	0.217	0.781	0.274	0.120	0.799
ANT 1.7	0.154	0.331	0.752	0.130	0.228	0.820
IVY	0.327	0.250	0.710	0.250	0.161	0.793
LUCENE	0.197	0.522	0.619	0.190	0.378	0.710
POI	0.323	0.167	0.751	0.224	0.138	0.818
TOMCAT	0.193	0.247	0.779	0.190	0.175	0.818
KC1	0.259	0.317	0.712	0.238	0.250	0.756
JEdit	0.229	0.455	0.649	0.218	0.417	0.675
Eclipse	0.147	0.393	0.720	0.128	0.257	0.805

Table 41
K-means fault-proneness prediction performance using datasets binarized with VARL thresholds.

Dataset	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
ANT 1.3	0.362	0.100	0.758	0.362	0.152	0.736
ANT 1.5	0.330	0.500	0.579	0.273	0.829	0.353
ANT 1.6	0.477	0.185	0.653	0.579	0.130	0.605
ANT 1.7	0.489	0.090	0.682	0.489	0.062	0.692
IVY	0.388	0.150	0.721	0.388	0.107	0.739
TOMCAT	0.298	0.338	0.682	0.307	0.158	0.764
JEdit	0.000	1.000	0.000	0.002	1.000	0.000
Eclipse	0.558	0.165	0.608	0.558	0.104	0.630

Table 42
K-means fault-proneness prediction performance using datasets binarized with Alves Rankings thresholds.

Dataset	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
ANT 1.3	0.410	0.300	0.643	0.333	0.182	0.739
ANT 1.4	0.377	0.500	0.558	0.355	0.489	0.574
ANT 1.5	0.367	0.581	0.515	0.304	0.343	0.676
ANT 1.6	0.239	0.217	0.772	0.131	0.201	0.833
ANT 1.7	0.228	0.355	0.705	0.176	0.207	0.808
IVY	0.199	0.300	0.749	0.192	0.304	0.750
LUCENE	0.182	0.547	0.609	0.175	0.394	0.707
POI	0.205	0.391	0.696	0.186	0.316	0.746
TOMCAT	0.257	0.234	0.754	0.265	0.132	0.799
KC1	0.188	0.433	0.678	0.094	0.333	0.777
JEdit	0.324	0.500	0.581	0.291	0.583	0.543
Eclipse	0.134	0.413	0.713	0.083	0.334	0.781

better using duplicated classification, as 10 out of 12 datasets gave better results using duplicated classification than the binary one. Out of the 12 datasets, 10 gave acceptable or excellent results, which is good. ANT 1.4 gave poor classification, and JEdit gave fair classification. The results for clustering using K-means thresholds are more consistent than when the algorithm is applied on the raw source code metrics and are better for all datasets except one. It could therefore be considered as an acceptable solution for fault-proneness prediction.

As for VARL threshold values used in conjunction with K-means, only 3 datasets gave at least acceptable results. Other results are fair or worse, making K-means using VARL thresholds not a very desirable model, considering that ROC Curves gave acceptable results for 10 datasets.

Considering Alves Rankings thresholds, it gave at least acceptable results for 9 out of 12 datasets and fair results for another dataset. Performance is a lot better than K-means algorithm applied alone but is slightly worse than the tests performed with ROC Curves. This is confirmed by the Friedman and Nemenyi tests. Nevertheless, it can still be considered acceptable, as performance is close to the one provided by ROC Curves.

Table 43
SOM fault-proneness prediction performance using Raw datasets.

Dataset	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
ANT 1.3	0.124	0.350	0.755	0.143	0.333	0.756
ANT 1.4	0.217	0.675	0.504	0.217	0.638	0.532
ANT 1.5	0.138	0.406	0.715	0.126	0.371	0.741
ANT 1.6	0.077	0.554	0.641	0.058	0.527	0.667
ANT 1.7	0.052	0.614	0.605	0.016	0.627	0.606
IVY	0.083	0.500	0.677	0.054	0.429	0.735
LUCENE	0.912	0.119	0.278	1.000	0.000	0.000
POI	0.994	0.040	0.077	1.000	0.020	0.000
TOMCAT	0.072	0.584	0.621	0.067	0.491	0.689
KC1	0.259	0.350	0.694	1.000	0.205	0.000
JEdit	0.013	0.800	0.444	0.019	0.818	0.422
Eclipse	0.008	0.869	0.361	0.008	0.802	0.443

According to the Friedman test, the duplicated and binarized datasets using ROC Curves and Alves Rankings threshold values performed significantly better than the other models using K-means. In fact, with the Friedman test executed when the K-means clustering algorithm is applied on binary fault-proneness datasets, we obtained a p-value of 0. For the comparison of experiments using duplicated datasets, we obtained a p-value of .001. When considering the best models using K-means (ROC Curves binarized datasets on both binary and duplicated datasets and Alves Rankings binarized datasets on duplicated datasets), we obtained a p-value of .017.

SOM. Fault-proneness prediction performance using the SOM clustering algorithm is presented in Table 43. The results are obtained using Weka [31] and the SelfOrganizingMap plugin developed by John Salatas.[2]

As for K-means algorithm, FPR and FNR were often too high to be acceptable. Only 3 datasets gave at least acceptable results, as with K-means. In fact, the results are very similar to the 2 clustering algorithms and are even identical for certain cases. This could be explained by the fact that only two clusters are produced and that a very similar distance function is used to calculate the proximity between 2 source code classes. According to this, the fact that both clustering algorithms gave similar results is not surprising. Maybe their outputs would differ more by using additional clusters, as the 2 algorithms are algorithmically different. As for K-means algorithm, duplicated classification was better than binary for SOM, but some datasets follow the opposite trend (LUCENE, POI, KC1 and JEdit).

Table 44 presents results for the SOM algorithm executed on the binarized datasets using ROC Curves thresholds, Table 45 presents results using VARL threshold values and Table 46 presents results using Alves Rankings thresholds at 30% of the distribution.

Using ROC Curves thresholds with SOM algorithm yields better results than using SOM alone, as 9 out of 12 datasets gave at least acceptable classification results and the other 3 gave fair classification results. 11 out of 12 datasets gave better results using ROC Curves binarized datasets than when using the raw source code metrics for SOM clustering. ROC Curves seems a good approach to use with the SOM clustering algorithm.

VARL threshold values used with SOM gave results that are not really acceptable. On the 8 datasets investigated for VARL, 3 gave at least acceptable results, 3 others gave fair classification and the remaining 2 gave no good results.

Alves Rankings thresholds gave acceptable results for 9 out of 12 datasets, fair classification for 2 others and poor results for the remaining one. Results obtained by the Alves Rankings threshold values

[2] Self-Organizing Map plugin for Weka http://weka.sourceforge.net/packageMetaData/SelfOrganizingMap/index.html.

Table 44
SOM fault-proneness prediction performance using datasets binarized with ROC Curves thresholds.

Dataset	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
ANT 1.3	0.200	0.250	0.775	0.124	0.333	0.764
ANT 1.4	0.333	0.425	0.619	0.333	0.383	0.641
ANT 1.5	0.414	0.188	0.690	0.418	0.171	0.695
ANT 1.6	0.251	0.174	0.787	0.170	0.158	0.836
ANT 1.7	0.183	0.307	0.752	0.126	0.237	0.817
IVY	0.308	0.225	0.732	0.189	0.179	0.816
LUCENE	0.197	0.522	0.619	0.197	0.370	0.711
POI	0.273	0.181	0.771	0.255	0.138	0.802
TOMCAT	0.196	0.247	0.778	0.193	0.175	0.816
KC1	0.282	0.283	0.717	0.106	0.272	0.807
JEdit	0.341	0.364	0.648	0.341	0.333	0.663
Eclipse	0.154	0.374	0.728	0.154	0.246	0.799

Table 45
SOM fault-proneness prediction performance using datasets binarized with VARL thresholds.

Dataset	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
ANT 1.3	0.419	0.100	0.723	0.381	0.121	0.738
ANT 1.5	0.429	0.719	0.401	0.293	0.788	0.387
ANT 1.6	0.517	0.087	0.664	0.517	0.049	0.678
ANT 1.7	0.489	0.090	0.682	0.489	0.062	0.692
IVY	0.388	0.150	0.721	0.388	0.107	0.739
TOMCAT	0.273	0.247	0.740	0.273	0.184	0.770
JEdit	0.000	1.000	0.000	0.002	1.000	0.000
Eclipse	0.558	0.165	0.608	0.558	0.104	0.630

Table 46
SOM fault-proneness prediction performance using datasets binarized with Alves Rankings thresholds.

Dataset	FPR-B	FNR-B	g-mean-B	FPR-D	FNR-D	g-mean-D
ANT 1.3	0.286	0.200	0.756	0.286	0.242	0.736
ANT 1.4	0.268	0.600	0.541	0.268	0.553	0.572
ANT 1.5	0.322	0.281	0.698	0.322	0.257	0.710
ANT 1.6	0.205	0.261	0.767	0.131	0.201	0.833
ANT 1.7	0.233	0.271	0.748	0.176	0.207	0.808
IVY	0.317	0.250	0.716	0.154	0.321	0.758
LUCENE	0.168	0.547	0.614	0.124	0.505	0.659
POI	0.093	0.509	0.667	0.093	0.444	0.710
TOMCAT	0.362	0.156	0.734	0.205	0.202	0.797
KC1	0.282	0.383	0.665	0.094	0.333	0.777
JEdit	0.324	0.636	0.496	0.281	0.333	0.693
Eclipse	0.090	0.466	0.697	0.083	0.334	0.781

are close to those obtained with ROC Curves thresholds. Both approaches gave acceptable performance and better results than when SOM is used alone.

According to the Friedman and Nemenyi tests, the SOM algorithm applied on duplicated and binarized datasets using ROC Curves and Alves Rankings threshold values gave significantly better performance. The test gave a p-value of .001 when considering the SOM model applied on binary fault-proneness datasets. It also gave a p-value of .001 for SOM applied on duplicated datasets. As to the experiment considering the best models (ROC Curves and Alves Rankings binarized datasets), we obtained a p-value less than .0001.

Summary of results. A summary of the results obtained using the clustering algorithms is presented in Table 47. It reuses the same legend as presented for Table 21. Note that only results of duplicated classification are presented, as they gave better results than the binary one. As for the machine learning summary, the columns marked with RAW represent the results obtained performing the clustering algorithms on raw source code metrics.

Table 47
Summary of fault-proneness performance for clustering models.

	K-means				SOM			
Dataset	RAW	ROC	VARL	Alves	RAW	ROC	VARL	Alves
ANT 1.3	+	+	+	+	+	+	+	+
ANT 1.4	–	–	NA	–	–	0	NA	–
ANT 1.5	+	+	—	0	+	0	—	+
ANT 1.6	0	+	0	++	0	++	0	++
ANT 1.7	0	++	0	++	0	++	0	++
IVY	+	+	+	+	+	++	+	+
LUCENE	—	+	NA	+	—	+	NA	0
POI	—	++	NA	+	—	++	NA	+
TOMCAT	0	++	+	+	0	++	+	+
KC1	—	+	NA	+	—	++	NA	+
JEdit	—	0	—	–	—	0	—	0
Eclipse	—	++	0	+	—	+	0	+

The first observation we can make when looking at Table 47 is that the classification using the raw source code metrics gave the same results, according to the level of g-mean. VARL technique gave the same results too, which are not acceptable. Both K-means and SOM gave similar results for each thresholds calculation technique, making them somewhat equivalent. K-means seems to give slightly better results, as results for ROC curves and Alves Rankings gave less poor classification results. In fact, according to the Friedman and Nemenyi tests, clustering applied on binarized datasets using ROC Curves or Alves Rankings threshold values gave significantly better results than clustering applied on the raw source code metrics (with a p-value less than .0001). Therefore, the important conclusion is that classification using clustering techniques gave better results when threshold values are used to binarized datasets.

Other tests would be interesting to do using clustering techniques as part of more complex models, as done in [11,15,16,18]. We didn't perform these tests because they are often complex and require a lot more development to be put in place. Since the goal of this study is to compare different thresholds definition techniques for fault-proneness, we limited our tests to K-means and SOM applied using supervised clustering. More complex tests are therefore out of the scope of this paper.

Following the results obtained with machine learning and clustering algorithms for fault-proneness prediction, we can answer positively to RQ5, which was:

RQ5: Can thresholds-based fault-proneness prediction models achieve similar performance to supervised models? When combined with a machine learning or clustering based model, do thresholds-based models achieve better performance?

Not only the thresholds-based approaches achieved similar performance to supervised approaches, they even gave stabler performance from one dataset to another. We can also conclude that threshold values applied before running a machine learning algorithm gave slightly better results. However, for clustering algorithms, the improvement when combining the threshold values is more noticeable. Nonetheless, combining threshold values and supervised algorithms did not improve the performance of the models, when compared to thresholds-based approaches.

5.4. Cross-project results

In this part of the paper, we present results of cross-project fault-proneness prediction. Note that only the best performing models are considered, which are ROC-2, ROC-3, Alves-2, Alves-3 and Bayes Network applied on raw data. We included Bayes Network to see if a machine learning model performs similarly to thresholds-based models when training and testing data are taken from different datasets.

Table 48
Cross-project fault-proneness performance for Apache ANT 1.7.

	ROC-2			ROC-3			Alves-2			Alves-3			Bayes Network		
Dataset	FPR	FNR	g-mean	FPR	FNR	g-mean	FPR	FNR	g-mean	FPR	FNR	g-mean	FPR	FNR	g-mean
ANT 1.3	0.267	0.333	0.699	0.152	0.333	0.752	0.295	0.182	0.759	0.181	0.333	0.739	0.181	0.333	0.739
ANT 1.4	0.312	0.511	0.580	0.239	0.574	0.569	0.326	0.404	0.634	0.261	0.574	0.561	0.246	0.553	0.580
ANT 1.5	0.261	0.314	0.712	0.161	0.314	0.759	0.253	0.286	0.731	0.188	0.286	0.762	0.172	0.314	0.753
ANT 1.6	0.205	0.174	0.811	0.108	0.245	0.821	0.216	0.141	0.820	0.139	0.212	0.824	0.124	0.234	0.820
ANT 1.7	0.183	0.198	0.809	0.114	0.251	0.814	0.238	0.172	0.794	0.155	0.231	0.806	0.133	0.243	0.810
IVY	0.234	0.250	0.758	0.163	0.304	0.763	0.237	0.196	0.783	0.170	0.232	0.798	0.176	0.286	0.767
LUCENE	0.139	0.486	0.666	0.051	0.604	0.613	0.146	0.445	0.689	0.066	0.585	0.622	0.088	0.562	0.632
POI	0.112	0.470	0.686	0.075	0.568	0.632	0.106	0.430	0.714	0.075	0.520	0.667	0.081	0.548	0.645
TOMCAT	0.250	0.175	0.787	0.165	0.237	0.798	0.256	0.158	0.791	0.179	0.219	0.800	0.178	0.184	0.819
KC1	0.341	0.152	0.747	0.118	0.422	0.714	0.353	0.126	0.752	0.153	0.375	0.727	0.129	0.416	0.713
JEdit	0.385	0.333	0.641	0.285	0.333	0.690	0.405	0.333	0.630	0.301	0.333	0.682	0.314	0.333	0.676
Eclipse	0.378	0.158	0.724	0.219	0.211	0.785	0.398	0.134	0.722	0.267	0.187	0.772	0.239	0.198	0.781

Since some models built using data from other systems perform better than others at fault-proneness prediction, we wanted to investigate if the source code threshold values of these systems can be reused for other systems. In a real-life enterprise context, this type of models reuse could be done. Once a model is considered acceptable, it could be reused on similar projects to detect fault-prone code. This therefore avoids constructing the model again for specific projects, which can be time-consuming depending on the model used. Furthermore, fault data may not be available for a specific project, because it is a new one or because faults were not logged during the development and maintenance of the system. Reusing fault-proneness prediction models could therefore become handy in these contexts.

We chose to investigate the cross-project prediction using 3 reference datasets, which are: ANT 1.7, TOMCAT and Eclipse. We chose ANT 1.7 because it performed well using the best models built (with ANT 1.6, but only 1.7 was retained, as both are from the same system and are similar). We also chose the TOMCAT system, as it gave acceptable results and was the biggest system out of the 12 according to Table 2 presenting the descriptive statistics of each dataset. Using TOMCAT, we particularly investigated if performance was good on the smallest datasets (ANT 1.3, 1.4, 1.5, IVY and KC1). For both ANT 1.7 and TOMCAT cross-project experiments, a particular attention is given to other Apache datasets performance, since they are built from the same organization. The last dataset used as a reference for the cross-project prediction is Eclipse, since it performed well and is not an Apache dataset. We therefore investigated if this dataset can be reused for systems from different organizations but of similar size. Tables 48–50 give the results obtained for the different models using each reference dataset. Table 51 summarizes the results obtained in these tables in a single and clearer format while Figs. 2–4 present the results in graphics that make comparison of results easier.

Looking at the results using ANT 1.7 as the reference system, we see that the performance is at least acceptable for ANT 1.6, which seems to give results similar to ANT 1.7 for all tests using all reference datasets. Since the 2 datasets are built on the same system but with a difference of one version, these close results can be explained by the probably high similarity between the 2 datasets. Results were acceptable too for ANT 1.5, IVY, TOMCAT, KC1 and Eclipse datasets, as they are most of the time acceptable for each dataset. ANT 1.3 performed well too, with acceptable results except fair ones for ROC-2. ANT 1.4, LUCENE, POI and JEdit gave fair or not good results for most tests, regardless the reference dataset used. According to previously presented thresholds-only results, the POI dataset gave at least acceptable results for most models (see Table 48). The cross-project experiment therefore doesn't seem an acceptable choice for this specific dataset. Probably that the high number of faults in this dataset doesn't help (see Table 2), making it hard to correctly classify faulty classes. Although we have some exceptions, the cross-project prediction using ANT 1.7 seems an acceptable choice, especially on other ANT datasets, IVY and TOMCAT, which are all Apache datasets. Even KC1 yields acceptable results, which was unexpected, as it is a lot smaller in terms of LOC than ANT 1.7 and is produced by the NASA and not Apache. According to the Friedman analysis, there is no significant difference between each model (with a p-value of .559).

For the TOMCAT reference dataset, results are pretty close to those obtained with ANT 1.7 dataset. Performance is the same or similar for all datasets and tests. Results for KC1 are often below the acceptable range, probably because TOMCAT is a big dataset with more than 300 000 LOC, while KC1 is a small dataset with 30 631 LOC. We see that classification performance on smaller datasets, that we thought would be bad because of the large size of the TOMCAT dataset, wasn't that good for ANT 1.3, 1.4 and KC1, but was acceptable for ANT 1.5 and IVY. According to the Friedman analysis, there is a significant performance improvement for the Alves-2 and ROC-2 models over the other

Table 49
Cross-project fault-proneness performance for Apache TOMCAT.

	ROC-2			ROC-3			Alves-2			Alves-3			Bayes Network		
Dataset	FPR	FNR	g-mean	FPR	FNR	g-mean	FPR	FNR	g-mean	FPR	FNR	g-mean	FPR	FNR	g-mean
ANT 1.3	0.181	0.333	0.739	0.133	0.485	0.668	0.171	0.242	0.792	0.114	0.424	0.714	0.105	0.485	0.679
ANT 1.4	0.239	0.553	0.583	0.210	0.660	0.519	0.225	0.511	0.616	0.188	0.638	0.542	0.167	0.660	0.533
ANT 1.5	0.188	0.314	0.746	0.126	0.400	0.724	0.195	0.314	0.743	0.123	0.371	0.743	0.107	0.400	0.732
ANT 1.6	0.143	0.234	0.810	0.097	0.310	0.790	0.147	0.212	0.820	0.089	0.315	0.790	0.085	0.342	0.776
ANT 1.7	0.128	0.237	0.816	0.093	0.320	0.785	0.166	0.213	0.810	0.092	0.343	0.772	0.073	0.352	0.775
IVY	0.176	0.250	0.786	0.135	0.321	0.766	0.192	0.268	0.769	0.141	0.375	0.733	0.125	0.321	0.771
LUCENE	0.102	0.582	0.612	0.051	0.652	0.575	0.095	0.585	0.613	0.022	0.652	0.584	0.036	0.682	0.554
POI	0.081	0.550	0.643	0.043	0.590	0.626	0.075	0.514	0.671	0.050	0.624	0.598	0.043	0.626	0.598
TOMCAT	0.198	0.175	0.813	0.138	0.254	0.802	0.206	0.184	0.805	0.133	0.316	0.770	0.123	0.289	0.789
KC1	0.235	0.253	0.756	0.094	0.465	0.696	0.294	0.227	0.739	0.129	0.495	0.663	0.082	0.490	0.684
JEdit	0.314	0.333	0.676	0.254	0.417	0.660	0.337	0.333	0.665	0.222	0.417	0.673	0.229	0.417	0.671
Eclipse	0.287	0.171	0.769	0.197	0.222	0.790	0.360	0.139	0.742	0.257	0.201	0.771	0.149	0.241	0.804

Table 50
Cross-project fault-proneness performance for Eclipse.

	ROC-2			ROC-3			Alves-2			Alves-3			Bayes Network		
Dataset	FPR	FNR	g-mean	FPR	FNR	g-mean	FPR	FNR	g-mean	FPR	FNR	g-mean	FPR	FNR	g-mean
ANT 1.3	0.086	0.576	0.623	0.019	0.879	0.345	0.086	0.606	0.600	0.019	0.879	0.345	0.057	0.727	0.507
ANT 1.4	0.130	0.723	0.490	0.051	0.936	0.246	0.123	0.766	0.453	0.043	0.979	0.143	0.101	0.787	0.437
ANT 1.5	0.138	0.486	0.666	0.034	0.686	0.551	0.092	0.514	0.664	0.027	0.743	0.500	0.073	0.543	0.651
ANT 1.6	0.116	0.348	0.759	0.042	0.582	0.633	0.093	0.391	0.743	0.031	0.614	0.612	0.069	0.440	0.722
ANT 1.7	0.098	0.399	0.736	0.029	0.592	0.630	0.069	0.420	0.735	0.026	0.609	0.617	0.059	0.488	0.694
IVY	0.157	0.321	0.756	0.061	0.464	0.709	0.122	0.393	0.730	0.058	0.500	0.686	0.099	0.393	0.739
LUCENE	0.066	0.601	0.610	0.007	0.796	0.450	0.058	0.655	0.570	0.000	0.796	0.452	0.036	0.690	0.547
POI	0.062	0.650	0.573	0.012	0.782	0.464	0.043	0.686	0.548	0.006	0.800	0.446	0.037	0.726	0.514
TOMCAT	0.128	0.289	0.787	0.064	0.509	0.678	0.101	0.351	0.764	0.058	0.518	0.674	0.085	0.333	0.781
KC1	0.141	0.296	0.778	0.047	0.472	0.709	0.047	0.441	0.730	0.035	0.528	0.675	0.059	0.457	0.715
JEdit	0.258	0.333	0.703	0.108	0.500	0.668	0.204	0.333	0.729	0.094	0.500	0.673	0.170	0.417	0.696
Eclipse	0.154	0.246	0.799	0.095	0.278	0.808	0.095	0.307	0.792	0.076	0.340	0.781	0.099	0.286	0.802

models (with a p-value of 0).

As to the Eclipse dataset, results are not as good as with the other 2 Apache datasets used as references. Eclipse gave at least acceptable performance for all models except Alves-3 on ANT 1.7 and KC1, but for the other results performance varies from no good at all to acceptable. The fact that Eclipse is not an Apache dataset could explain results for ANT, IVY, LUCENE, POI and TOMCAT datasets. ANT 1.7 and TOMCAT present similar size characteristics to the Eclipse dataset according to Table 2, but performance obtained for TOMCAT was mitigated between fair and acceptable. The Friedman analysis shows that models based on Bayes Network, Alves-2 and ROC-2 performed significantly better than ROC-3 and Alves-3 models (with a p-value less than .0001).

To summarize cross-project fault-proneness prediction, it seems that the best performing datasets, like ANT 1.7 and TOMCAT, are acceptable choices to perform the cross-project experiment, especially when considering other Apache datasets. LUCENE and ANT 1.4 gave bad results, as when the model is built especially for these, which is not surprising. POI gave bad results too, probably because of the high number of faults in the dataset which makes it a lot different than the other ones. Results from the Eclipse dataset didn't give acceptable results, probably because no other dataset was built by the Eclipse Foundation. Performance was nor bad nor good for similar datasets like ANT 1.7 and TOMCAT (but a bit better for ANT 1.7). We also see that the non-supervised approach (Alves Rankings) yields acceptable results when cross-project fault-proneness prediction is considered. It even outperformed Bayes Network classification for thresholds calculated on TOMCAT and Eclipse datasets.

In fact, according to the Friedman and post-hoc Nemenyi tests, we conclude that thresholds-based models (especially ROC-2 and Alves-2), perform slightly better than Bayes Network for cross-project experiments. This could be explained by the fact that supervised learning learns to predict faults from its particular training data and may have difficulties to predict faults in a different system.

From the results we obtained, we can conclude that cross-project fault-proneness prediction could therefore be used in a real-life enterprise context, using a well-performing model built from one of the enterprise's systems. Although, results like POI make us think that building the model for each dataset, when it can be done, could be a better solution, also considering that performance was better when the model was built especially for each single dataset.

5.5. Cross-version results

In this part of the paper, we present the results obtained for cross-version fault-proneness prediction. Table 52 presents results obtained when training the model on the immediate previous version of a software system and testing it on the current one. Table 53 presents the results obtained when considering all previous versions for the construction of the model. An easy to understand classification performance summary is given in Table 54. Figs. 5 and 6 present the results in graphics to make comparison of results easier.

As we can see from the results obtained on immediate successive versions, the performance using ANT 1.3 models on ANT 1.4 isn't acceptable, with fair or worse results. Since ANT 1.4 didn't give good results even with the best performing models investigated, this isn't surprising. Except that all results are at least acceptable and sometimes excellent when other versions are considered. The only exception to this rule is when the Bayes Network is built using ANT 1.4 data and tested on ANT 1.5. This could be explained by the fact that ANT 1.4 seems problematic to do fault-proneness prediction, therefore making model construction problematic, especially when using a machine learning approach like Bayes Network. Thresholds-based models gave acceptable results for these cross-version tests.

Considering the performance of the models built on all previous

Table 51
Cross-project performance summary.

Reference	ANT 1.7					TOMCAT					Eclipse				
Dataset	ROC-2	ROC-3	Alves-2	Alves-3	Bayes	ROC-2	ROC-3	Alves-2	Alves-3	Bayes	ROC-2	ROC-3	Alves-2	Alves-3	Bayes
ANT 1.3	0	+	+	+	+	+	0	+	+	0	0	—	0	—	-
ANT 1.4	-	-	0	-	-	-	-	0	-	-	—	—	—	—	—
ANT 1.5	+	+	+	+	+	+	+	+	+	+	0	-	0	-	0
ANT 1.6	++	++	++	++	++	++	+	++	+	+	+	0	+	0	+
ANT 1.7	++	++	+	++	++	++	+	++	+	+	+	0	+	0	0
IVY	+	+	+	+	+	+	+	+	+	+	+	+	+	0	+
LUCENE	0	0	0	0	0	0	-	0	-	-	0	—	-	—	-
POI	0	0	+	0	0	0	0	0	-	-	-	—	-	—	-
TOMCAT	+	+	+	++	++	++	++	++	+	+	+	0	+	0	+
KC1	+	+	+	+	+	+	0	+	0	0	+	+	+	0	+
JEdit	0	0	0	0	0	0	0	0	0	0	+	0	+	0	0
Eclipse	+	+	+	+	+	+	+	+	+	++	+	++	+	+	++

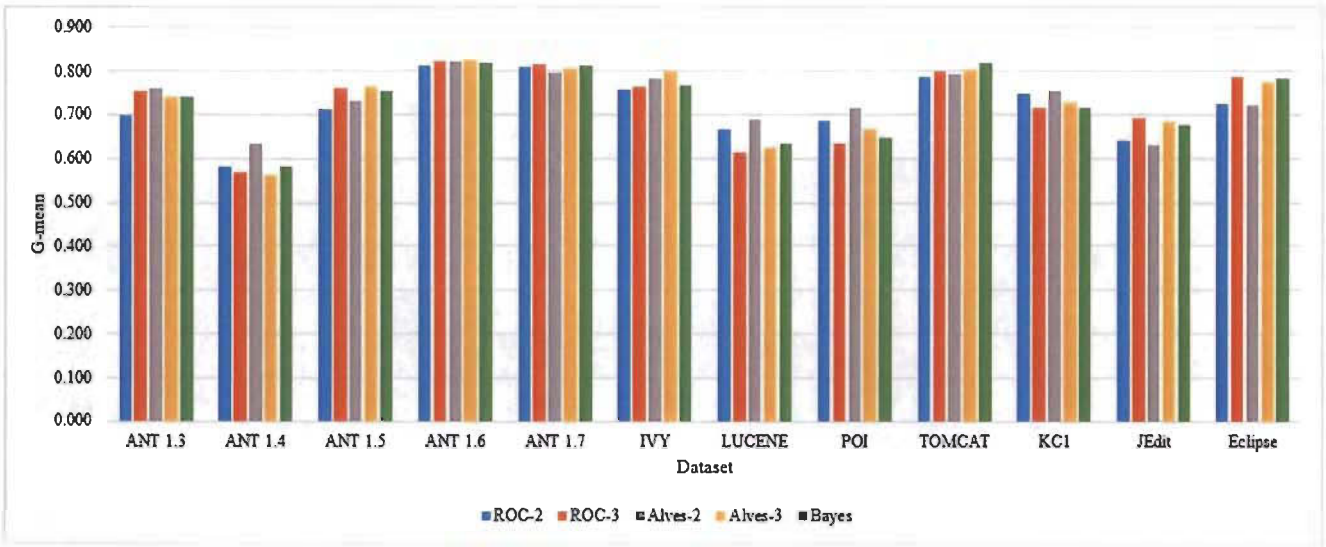


Fig. 2. Cross-project performance summary for ANT 1.7.

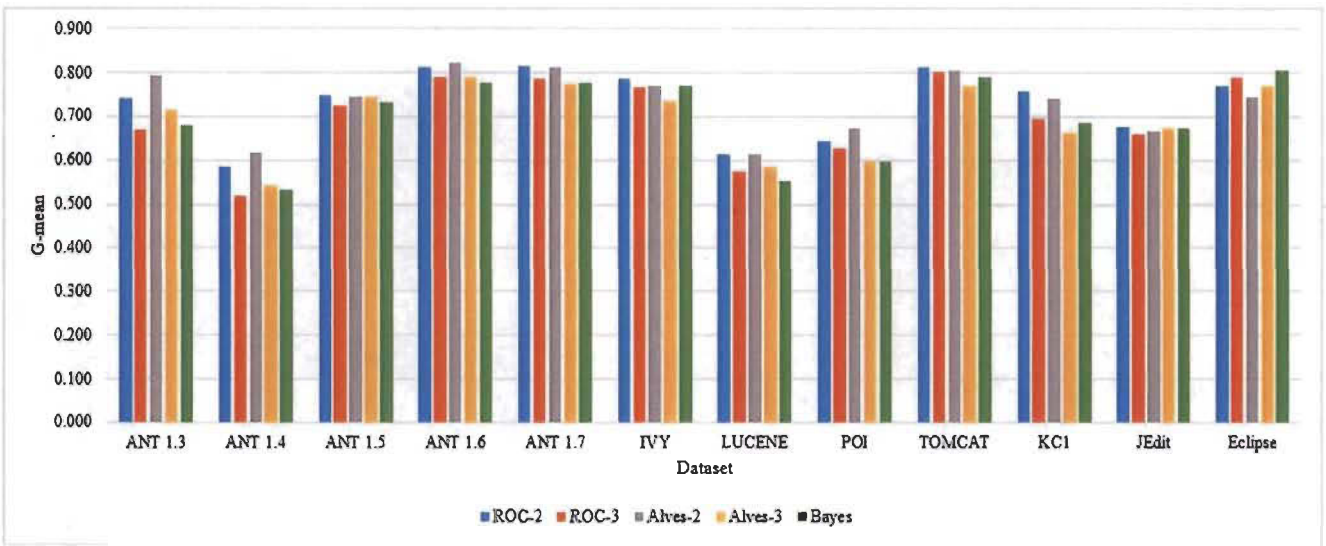


Fig. 3. Cross-project performance summary for TOMCAT.

versions of Apache ANT, the first element we denote is that results for ANT 1.4 are the same as for immediate successive versions. This is simply explained by the fact that the only previous version to ANT 1.4 is 1.3, which is exactly the same thing as taking the immediate previous version only. Except that, we can see that results are very similar to those obtained using only the previous version, especially for the Alves Rankings method, which is non-supervised. Supervised methods like ROC Curves and Bayes Network gave similar results too, but differences in the classification performance are more noticeable from one version to another. For example, the performance on ANT 1.5 using Bayes Network is a lot better when using 1.3 and 1.4 versions for training the Bayes Network than when using 1.4 version only as training data. Since most results we got for the ANT 1.4 dataset were not so good, using 1.3 version in the construction of the model seems to have helped to construct a more accurate model. For ANT 1.7, supervised approaches seem to be working better when the model is constructed on the immediate previous version only. However, this is plausible, since ANT 1.6 and ANT 1.7 were the 2 best performing models according to experiments using 10-fold cross-validation. ROC Curves approach gave different performance results for the two cross-version methodologies when ROC-3 is considered for ANT 1.6, since using all previous versions for calculating threshold values gave a much better performance. As to the Friedman analysis, it did not notice any significant performance improvement for any of the models investigated for the cross-version experiment. It gave a p-value of .663 when considering the models built using the immediate previous version and a p-value of .484 when considering all previous versions.

All results are at least acceptable and sometimes excellent, except when ANT 1.4 is concerned (which we think is problematic for fault-proneness prediction). From these results, we can conclude that building a model on one or many of the previous versions of a software and testing it on the current one seems to be an acceptable approach to do fault-proneness prediction. Considering all previous versions of a software system seems to be a good approach when using a machine learning algorithm (Bayes Network), so it has more training data to learn from than when only using the immediate previous version.

With the conclusions made for cross-project and cross-version fault-proneness prediction, we can answer positively to RQ6, which was:

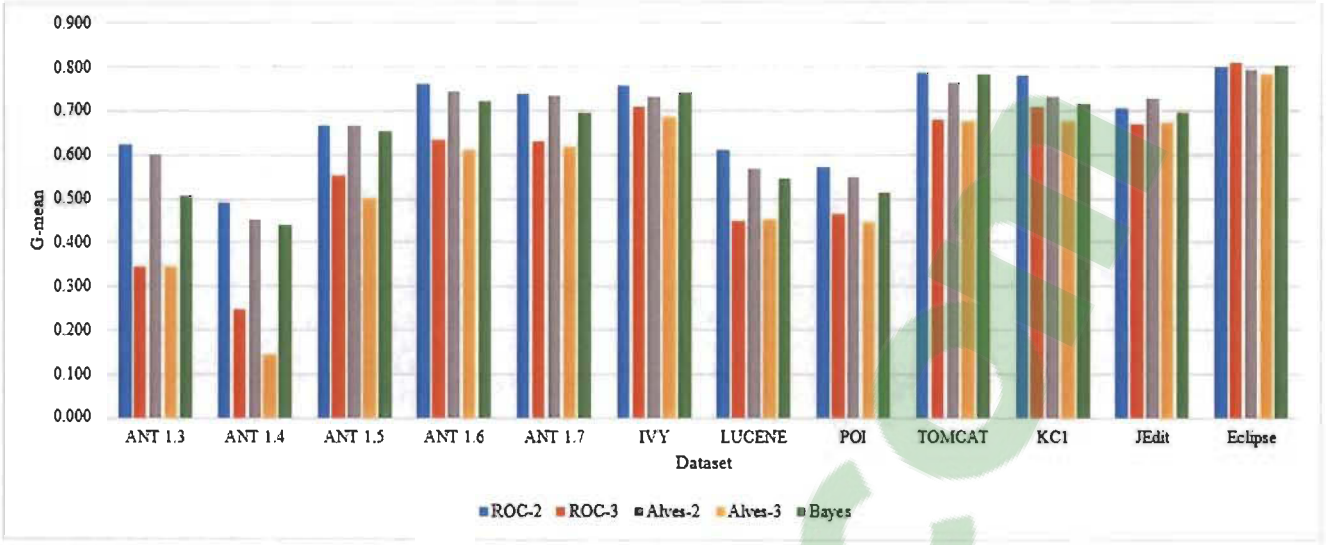


Fig. 4. Cross-project performance summary for Eclipse.

Table 52
Cross-version fault-proneness performance when the model is built on the immediate previous version.

	ROC-2			ROC-3			Alves-2			Alves-3			Bayes Network		
Dataset	FPR	FNR	g-mean	FPR	FNR	g-mean	FPR	FNR	g-mean	FPR	FNR	g-mean	FPR	FNR	g-mean
ANT 1.3 on ANT 1.4	0.217	0.468	0.645	0.210	0.617	0.550	0.370	0.404	0.613	0.290	0.532	0.577	0.217	0.532	0.605
ANT 1.4 on ANT 1.5	0.280	0.257	0.731	0.195	0.343	0.727	0.249	0.286	0.732	0.153	0.343	0.746	0.000	1.000	0.000
ANT 1.5 on ANT 1.6	0.170	0.223	0.803	0.062	0.408	0.746	0.259	0.114	0.810	0.147	0.207	0.823	0.058	0.429	0.733
ANT 1.6 on ANT 1.7	0.261	0.145	0.795	0.154	0.243	0.801	0.228	0.178	0.797	0.143	0.269	0.791	0.140	0.237	0.810

Table 53
Cross-version fault-proneness performance when the model is built on all previous versions.

	ROC-2			ROC-3			Alves-2			Alves-3			Bayes Network		
Dataset	FPR	FNR	g-mean	FPR	FNR	g-mean	FPR	FNR	g-mean	FPR	FNR	g-mean	FPR	FNR	g-mean
ANT 1.4	0.217	0.468	0.645	0.210	0.617	0.550	0.370	0.404	0.613	0.290	0.532	0.577	0.217	0.532	0.605
ANT 1.5	0.341	0.200	0.726	0.261	0.314	0.712	0.253	0.286	0.731	0.172	0.343	0.737	0.126	0.371	0.741
ANT 1.6	0.232	0.125	0.820	0.139	0.217	0.821	0.228	0.136	0.817	0.139	0.217	0.821	0.077	0.370	0.763
ANT 1.7	0.252	0.183	0.781	0.112	0.278	0.801	0.244	0.163	0.796	0.155	0.246	0.798	0.161	0.254	0.791

RQ6: Can threshold values calculated for one software system or different versions of it be reused for another system or version and still achieve good fault-proneness performance? How does that compare to cross-project or cross-version supervised fault-proneness prediction?

When considering cross-project fault-proneness prediction, results seem to indicate that it can achieve good performance. However, the training datasets should be chosen wisely, as different software systems yield different characteristics, therefore impacting prediction performance. As to cross-version prediction, it gave good results too. No significant difference was noticed when using all previous versions of a software system for building the model. However, we think that the previous versions used in the prediction should be chosen wisely, as some may impact negatively the fault-proneness prediction performance.

5.6. Summary of the best models

In this section, we present and discuss the results obtained for the best models found using ROC Curves, Alves Rankings threshold values and using Bayes Network algorithm alone. The reason we included Bayes Network is that it gave acceptable results and was the best-performing machine learning algorithm. It is also the machine learning model we will consider in this summary when applied to binarized versions of the datasets using each thresholds calculation technique, as it gave the best results again. For clustering algorithms, the algorithms using raw source code metrics won't be presented, as they didn't give acceptable results. Also, only K-means results using threshold values will be presented, as its results were slightly better than those given by SOM. All results presented are for duplicated classification, as it gave the best results among all models investigated. It also more accurately represents the classification performance. See Table 55 for the summary of the best performing models, using the same legend as for Table 21.

ROC Curves algorithm performed really well, especially ROC-2 and ROC-3 models. Its performance when used in conjunction with Bayes Network or K-means is good too. But, it is interesting to see that simpler models applying raw threshold values only to determine fault-proneness gave better results than methods using machine learning or clustering algorithms. Of course, some tuning could be done on the configurations of these models to achieve better classification performance, but thresholds-only based models still offer acceptable performance.

Bayes Network algorithm applied on the raw source code metrics gave acceptable results too. However, it gave no good classification for

Table 54
Cross-version fault-proneness performance summary.

	Immediate previous version					All previous versions				
Dataset	ROC-2	ROC-3	Alves-2	Alves-3	Bayes	ROC-2	ROC-3	Alves-2	Alves-3	Bayes
ANT 1.4	0	–	0	–	0	0	–	0	–	0
ANT 1.5	+	+	+	+	—	+	+	+	+	+
ANT 1.6	+ +	+	+ +	+ +	+	+ +	+ +	+ +	+ +	+
ANT 1.7	+	+ +	+	+	+ +	+	+ +	+	+	+

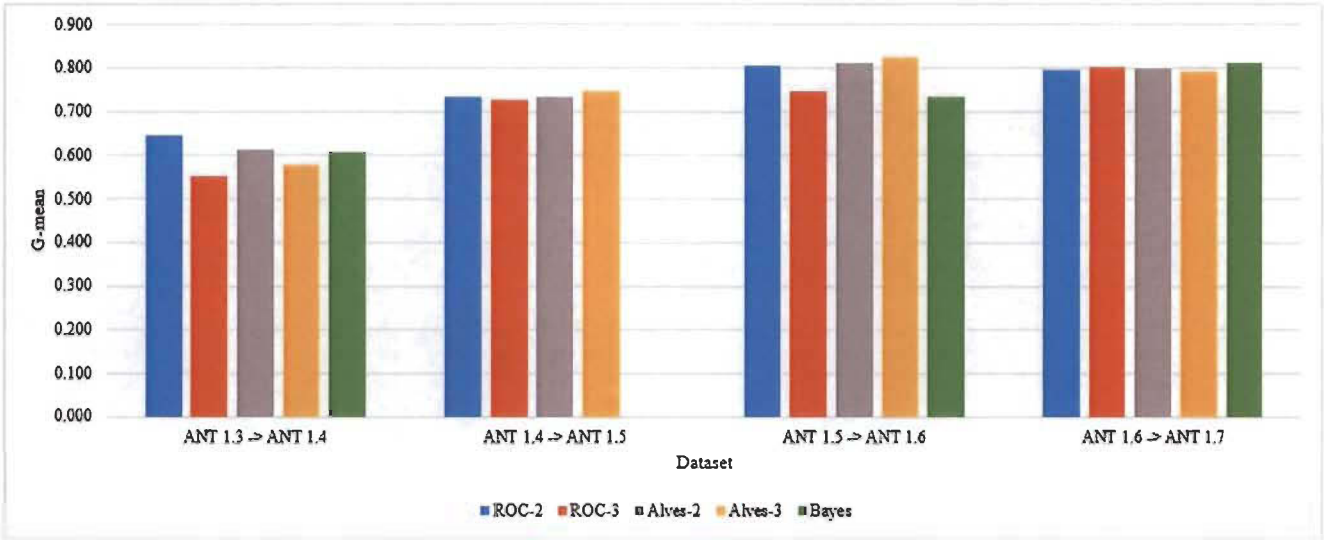


Fig. 5. Cross-version performance summary when the model is built on the immediate previous version.

2 datasets (ANT 1.4 and JEdit). This performance is worse than with ROC Curves, which gave better classification performance for both datasets. Also, results of ROC Curves using ROC-2 or ROC-3 are all better than those obtained using Bayes Network with the raw code metrics.

Alves Rankings technique gave acceptable performance, as most datasets and models gave at least acceptable results. However, it did not perform as well as ROC Curves. Alves Rankings can be considered as acceptable nonetheless, considering that its performance is pretty close to ROC Curves and that it doesn't use fault information to determine thresholds. The fact that fault data is not needed to calculate thresholds is the main advantage of using Alves Rankings. This technique could be part of an unsupervised classification model, which is not the case of ROC Curves and VARL. But still, ROC Curves and VARL could be used to calculate thresholds on previous versions of software, on which fault data exists. These thresholds could then be reused to calculate fault-proneness on the system's current version.

An important point to notice is that for thresholds-based models, models using only threshold values seem to give better results than the ones using thresholds in conjunction with machine learning or clustering algorithms. Thresholds-only based models could therefore be acceptable fault-proneness prediction solutions.

Another conclusion we can make when looking at Table 55 is that fault-proneness prediction is better performed on some datasets, which could be for various reasons. Fault-proneness prediction on ANT 1.4 and JEdit datasets seems difficult, as only ROC-3 model gave at least acceptable performance for JEdit and the best models gave fair performance for ANT 1.4. For JEdit, this classification problem was noted since the dataset was chosen, as it yielded only few faults considering it has a lot of classes. For ANT 1.4, which has 22.47% of faulty classes, the bad performance could be due to the fact that not all faulty classes were classified as such. This could be due to the fact that some critical classes were not tested or insufficiently tested. Beside these 2 exceptions, other datasets yield acceptable fault-proneness prediction performance when we consider ROC Curves, Alves Rankings or Bayes Network models. Performance was even good for LUCENE, POI and KC1 datasets, which we thought would give bad fault-proneness prediction results, as the number of faults and percentage of faulty classes in these systems is very high.

According to the Friedman test, Bayes Network applied on raw source code metrics, K-means and Bayes Network applied on binarized datasets using ROC Curves threshold values and ROC Curves thresholds-based models performed significantly better than the other models (p-value of 0). This shows that ROC Curves technique performs significantly better than the other threshold values calculation techniques investigated.

6. Threats to validity

This study, as every other empirical software engineering study, has certain threats to validity. First, our study covers only 12 datasets from 8 different systems. This means that the findings of this study cannot be generalized to all software systems, even if we investigated open and closed-source software systems. Further tests on many other systems (from different domains and developed in different programming languages) would be needed to generalize obtained results.

Another threat to validity of our study is the way we chose to use 30% of the Alves Rankings distribution to find thresholds using this method. We chose this specific value for finding threshold values as it's the one that yielded the best results across multiple datasets. However, changing it for another value could affect the results in a significant way. Of course, we should find a way to determine more objectively that percentage at which a threshold should be set. This methodology could give a generic percentage usable for all systems or a single one per dataset.

Another possible threat to validity of our study is the configuration

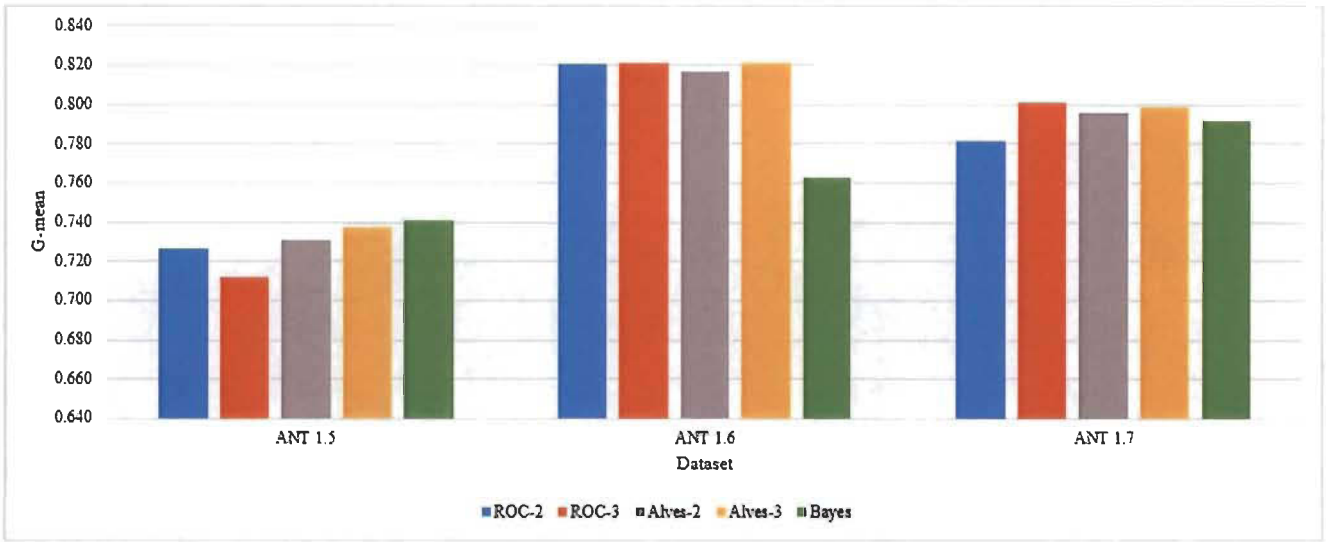


Fig. 6. Cross-version performance summary when the model is built on all previous versions.

Table 55
Performance summary of the best fault-proneness models.

	ROC Curves				VARL				Alves Rankings (30%)				Bayes Network
Dataset	ROC-2	ROC-3	Bayes	K-means	VARL-2	VARL-3	Bayes	K-means	Alves-2	Alves-3	Bayes	K-means	RAW
ANT 1.3	++	+	+	+	+	+	0	+	+	+	+	+	+
ANT 1.4	0	0	—	–	NA	NA	NA	NA	0	–	—	–	—
ANT 1.5	+	+	+	+	0	+	—	—	+	+	+	0	+
ANT 1.6	++	++	++	+	0	NA	0	0	++	++	++	++	++
ANT 1.7	++	++	++	++	0	+	+	0	+	++	++	++	++
IVY	++	++	++	+	0	+	+	+	+	+	+	+	++
LUCENE	+	+	+	+	NA	NA	NA	NA	+	0	0	+	+
POI	+	++	+	++	NA	NA	NA	NA	+	+	+	+	+
TOMCAT	++	++	++	++	+	+	+	+	++	+	+	+	+
KC1	+	++	+	+	NA	NA	NA	NA	+	+	+	+	+
JEdit	0	+	—	0	—	NA	—	—	0	0	—	–	—
Eclipse	+	++	++	++	—	0	0	0	+	+	+	+	+

used for the clustering and machine learning algorithms. Using the Weka tool, we performed the algorithms with all default parameters, but maybe some parameters for certain algorithms could have been fine-tuned to achieve higher fault-proneness prediction performance.

Also, the source code metrics used in the study could have been calculated differently for each dataset, as the tools used to calculate the metrics could be different. This could introduce differences in the results of the different datasets.

Another threat to validity is that although faults are listed in the datasets used, no data was found in these datasets defining if a class has been tested or not and how much it has been tested (in terms of testing effort and coverage). Therefore, some bugs may not have been discovered in some classes because they were not tested (or not completely tested). Considering this, our thresholds could have suggested faulty classes that are yet undiscovered, but were marked as false positives by the classification algorithm. Moreover, since we used public datasets already investigated in other studies, we supposed they were correctly built and that the fault data was accurate in each of them.

It would be interesting to reproduce the same study on systems that are more regulated, where testing coverage measures are available and where tests are built the same way (using the same methodology) for each system. The same development process could also be used for all the investigated systems and even the same development teams could have developed them. These systems could offer faults data on multiple releases, making it possible to further investigate fault-proneness prediction on successive versions of a software system. Replicating our study in such controlled environment would make our conclusions more generally applicable. It would also prevent threats to validity related to datasets testing coverage measures and source code metrics calculation.

7. Conclusions and future work

In this study, we wanted to compare three different source code metrics' threshold calculation methods to achieve fault-proneness prediction. We investigated these methods to build object-oriented metrics-based models for fault-proneness prediction. These models can provide valuable and understandable insights to prioritize classes that are likely more fault-prone and therefore need to be tested more intensively in order to ensure the quality of the software system. Since software quality is an important subject nowadays, fault-proneness prediction models can be of great use to developers and testers. Considering thresholds-based fault-proneness prediction models, we calculated metrics' thresholds using 3 different techniques (ROC Curves, VARL and Alves Rankings) and tried to predict faults in a total of 12 datasets. These different methodologies were compared using the produced threshold values alone to do fault-proneness prediction and as part of machine learning and clustering algorithms. The four investigated machine learning algorithms were Bayes Network, ANN, C4.5 and Support Vector Machine, while the two clustering algorithms were K-means and SOM.

As to the investigated thresholds calculation techniques, the results

obtained were acceptable for 2 out of 3 techniques investigated (ROC Curves and Alves Rankings). Results were mitigated for the VARL methodology, as it didn't give valid threshold values for all studied datasets and performed worse than these 2 methods.

The reason we chose to investigate ROC Curves method is because we wanted to test it on more systems than the study stating it and wanted to investigate if binary classification (considered not valid for the studied system by Shatnawi et al. [2]) could be valid for other datasets (RQ1). Following the results we got, with excellent and acceptable classification results for almost all datasets (except 2), we can conclude that the ROC Curves method is valid for other datasets than the one considered in [2]. We can also conclude that binary classification is valid for multiple datasets and often yields acceptable results. Furthermore, the 2 datasets on which this methodology failed to give good classification results didn't get acceptable classification results when used by machine learning models alone.

As to the reason why we investigated the VARL technique, it was because multiple studies came up with different conclusions about its threshold values validity for fault-proneness prediction. VARL was valid for certain datasets, but also invalid for others (ANT 1.4, LUCENE, POI and KC1) (RQ2). Although it is valid for some datasets, VARL did not give valuable threshold values for certain metrics, some of them being close to the minimum metric value and others not being valid at all. Other thresholds, like the RFC threshold value given for JEdit, were even above the maximum code metric value for the investigated system. Considering that this method has a lot of inconsistencies with the thresholds calculated and that not all datasets can provide valuable threshold values, its performance was considered worse than the ROC Curves and Alves Rankings methods. We therefore concluded that the VARL methodology is not good for fault-proneness prediction, as better alternatives exist, able to calculate threshold values for any software system.

The Alves Rankings technique was chosen to investigate if it could give acceptable results when applied to fault-proneness prediction, as previous studies only considered it for quality measurements of classes (RQ3). Since quality and fault-proneness are two closely related concepts in software engineering, we thought it would make a good choice for fault-proneness prediction. According to the results we obtained, it seems like this method could be used to perform fault-proneness prediction, as it gave at least acceptable results for 10 out of 12 datasets under study. The 2 datasets which performed worse are the same as those mentioned for ROC Curves method. The results found for Alves Rankings were close to those found for the ROC Curves method. However, ROC Curves offers significantly better classification performance, according to the Friedman and Nemenyi tests. Still, the advantage of Alves Rankings method over ROC Curves and VARL is that it is easy to automatize in a new or existing project without prior faults data history. Further tests on other datasets would be required to generalize the validity of Alves Rankings method, but it seems like a valid choice so far, having tested it on 12 different datasets coming from 8 different systems. We therefore concluded that Alves Rankings can be used for fault-proneness prediction with good fault-proneness prediction performance.

In our study, we wanted to investigate which of the three investigated threshold calculation techniques are the most relevant for fault-proneness prediction (RQ4). Of course, VARL is not one of them. In fact, we concluded that ROC Curves gave the best performance results. However, we also concluded that Alves Rankings is a good method to calculate threshold values, even if ROC Curves performed significantly better according to statistical tests. Its performance seems good enough to consider using it. In addition, the fact that it does not require any fault data history to calculate threshold values is a major positive point.

Source code metrics' thresholds calculated using the 3 techniques investigated were also used to build machine learning and clustering models. The datasets were binarized using these threshold values before running the machine learning or clustering algorithms. Additionally, these algorithms also built models using the raw source code metrics values (not binarized using threshold values). These experiments with supervised models were conducted to compare thresholds-based models' performance with supervised ones and to investigate if threshold values could improve supervised models or vice-versa (RQ5). Results showed that ROC Curves and Alves Rankings performed similarly when using these types of algorithms than when used as thresholds-only based models. However, performance was better using these simpler models (without supervised learning). Among all machine learning algorithms, Bayes Network gave the best results on both raw and binarized datasets.

As to the clustering techniques, when applied directly on raw source code metrics, they didn't give acceptable results. However, the performance was better when using datasets binarized with threshold values. Although performance using clustering with threshold values yielded better performance than clustering alone, the performance of these models were outperformed by thresholds-based models and the Bayes Network algorithm.

We therefore concluded that thresholds-based approaches based on ROC Curves threshold values yield results similar to the Bayes Network algorithm. This conclusion was validated statistically using the Friedman and post-hoc Nemenyi tests. Alves Rankings threshold values performed significantly less, but did give good results nonetheless. Furthermore, we concluded that thresholds-based models' performance is not improved when used in conjunction with supervised algorithms.

In our study, we also performed cross-project and cross-version experiments to investigate if threshold values could be calculated on one dataset and reused on another one (RQ6). We also performed the same experiment with the Bayes Network algorithm, to check if it would yield good results when used in a real-life context. Following the results obtained, we concluded that acceptable models built using certain datasets can be reused to predict fault-prone code in other systems of a same development organization. However, the performance is better when a model is built specifically for the system under test (for supervised and thresholds-based models alike). Results were acceptable when ANT 1.7 and TOMCAT were used as reference datasets, but Eclipse dataset, when used as reference, didn't give good results.

As to the tests we did reusing fault-proneness models on successive versions of the Apache ANT system, we concluded that these models can be built on one or many previous versions of a software and tested on the current version of the same system. Only ANT 1.4 dataset gave problematic results, but this dataset didn't perform well even when models were built specifically for it. For other ANT versions, results were good enough to be considered for other datasets. Considering all previous versions of a software system to construct the model seems a good option, especially for the Bayes Network algorithm. In fact, it seems to be more interesting to use with machine learning models, as more training data is available when all previous versions of the software are considered. This bigger amount of training data is supposed to improve the accuracy of the model.

Future works based on this study could therefore consist in testing ROC Curves and Alves Rankings method on more systems to validate their usefulness globally. Further tests on multiple versions of a same software could also be performed to validate that fault-proneness models can be reused on successive versions of a same software for more systems. In addition, Alves Rankings could be further investigated for building an unsupervised test effort prioritization model, without using the fault data history of a system. Moreover, the metrics used for fault-proneness prediction could be changed for design metrics only (as SLOC is a code metric). This would let users make testing effort prediction (and prioritization) based on UML class diagrams, even before implementation starts. Such unsupervised model using Alves Rankings technique could therefore be of great use for project development, giving a better idea to the development team on the testing effort to

invest in the project (better distribution of the testing effort). Uses of source code metrics' thresholds are multiple, therefore opening the way to many future work directions based on this one.

Acknowledgment

This work was partially supported by NSERC (Natural Sciences and Engineering Research Council of Canada) grant.

References

[1] S. Chidamber, C. Kemerer, A metrics suite for object oriented design, IEEE Trans. Softw. Eng. 20 (6) (1994) 476–493, http://dx.doi.org/10.1109/32.295895.
[2] R. Shatnawi, W. Li, J. Swain, T. Newman, Finding software metrics threshold values using ROC curves, J. Softw. Maint. Evol. 22 (1) (2010) 1–16, http://dx.doi.org/10.1002/smr.404.
[3] R. Shatnawi, A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems, IEEE Trans. Softw. Eng. 36 (2) (2010) 216–225, http://dx.doi.org/10.1109/TSE.2010.9.
[4] T. Gyimothy, R. Ferenc, I. Siket, Empirical validation of object-oriented metrics on open source software for fault prediction, IEEE Trans. Softw. Eng. 31 (10) (2005) 897–910, http://dx.doi.org/10.1109/TSE.2005.112.
[5] B. Isong, E. Obeten, A systematic review of the empirical validation of object-oriented metrics towards fault-Proneness prediction, Int. J. Softw. Eng. Knowl. Eng. 23 (10) (2013) 1513–1540, http://dx.doi.org/10.1142/S0218194013500484.
[6] Y. Zhou, H. Leung, Empirical analysis of object-Oriented design metrics for predicting high and low severity faults, IEEE Trans. Softw. Eng. 32 (10) (2006) 771–789, http://dx.doi.org/10.1109/TSE.2006.102.
[7] R. Malhotra, A.J. Bansal, Fault prediction considering threshold effects of object-oriented metrics, Expert Syst. 32 (2) (2015) 203–219, http://dx.doi.org/10.1111/exsy.12078.
[8] S. Singh, K.S. Kahlon, Object oriented software metrics threshold values at quantitative acceptable risk level, Csit 2 (3) (2014) 191–205, http://dx.doi.org/10.1007/s40012-014-0057-1.
[9] M. Jureczko, Significance of different software metrics in defect prediction, Softw. Eng. Int. J. 1 (1) (2011) 86–95.
[10] R. Malhotra, A. Jain, Fault prediction using statistical and machine learning methods for improving software quality, J. Inf. Process. Syst. 8 (2) (2012) 241–262, http://dx.doi.org/10.3745/JIPS.2012.8.2.241.
[11] M. Jureczko, L. Madeyski, Towards identifying software project clusters with regard to defect prediction, Proceedings of the 6th International Conference on Predictive Models in Software Engineering - PROMISE '10, (2010), p. 1, http://dx.doi.org/10.1145/1868328.1868342.
[12] A. Kaur, K. Kaur, Performance analysis of ensemble learning for predicting defects in open source software, 2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI), (2014), pp. 219–225, http://dx.doi.org/10.1109/ICACCI.2014.6968438.
[13] L. Yu, Using negative binomial regression analysis to predict software faults: a Study of Apache ANT, Int. J. Inf. Technol.Comput. Sci. 4 (8) (2012) 63–70, http://dx.doi.org/10.5815/ijitcs.2012.08.08.
[14] K. Dejaeger, T. Verbraken, B. Baesens, Toward comprehensible software fault prediction models using Bayesian network classifiers, IEEE Trans. Softw. Eng. 39 (2) (2013) 237–257, http://dx.doi.org/10.1109/TSE.2012.20.
[15] C. Catal, U. Sevim, B. Diri, Clustering and metrics thresholds based software fault prediction of unlabeled program modules, ITNG 2009 - 6th International Conference on Information Technology: New Generations, (2009), pp. 199–204, http://dx.doi.org/10.1109/ITNG.2009.12.
[16] G. Abaei, A. Selamat, H. Fujita, An empirical study based on semi-supervised hybrid self-organizing map for software fault prediction, Knowl. Based Syst. 74 (2014) 28–39, http://dx.doi.org/10.1016/j.knosys.2014.10.017.
[17] R. Shatnawi, Improving software fault-prediction for imbalanced data, 2012 International Conference on Innovations in Information Technology, IIT 2012, (2012), pp. 54–59, http://dx.doi.org/10.1109/INNOVATIONS.2012.6207774.
[18] P.S. Bishnu, V. Bhattacherjee, Software fault prediction using quad tree-based K-means clustering algorithm, IEEE Trans. Knowl. Data Eng. 24 (6) (2012) 1146–1150, http://dx.doi.org/10.1109/TKDE.2011.163.
[19] C. Catal, U. Sevim, B. Diri, Software fault prediction of unlabeled program modules, Proceedings of the World Congress on Engineering, I (2009), pp. 1–6.
[20] T. McCabe, A complexity measure, IEEE Trans. Softw. Eng. SE-2 (4) (1976) 308–320, http://dx.doi.org/10.1109/TSE.1976.233837.
[21] L.H. Rosenberg, Applying and interpreting object oriented metrics, Software Technology Conference, (1998).
[22] R. Bender, Quantitative risk assessment in epidemiological studies investigating threshold effects, Biom. J. 41 (3) (1999) 305–319, http://dx.doi.org/10.1002/(SICI)1521-4036(199906)41:3<305::AID-BIMJ305>3.0.CO;2-Y.
[23] T.L. Alves, C. Ypma, J. Visser, Deriving metric thresholds from benchmark data, 2010 IEEE International Conference on Software Maintenance, (2010), pp. 1–10, http://dx.doi.org/10.1109/ICSM.2010.5609747.
[24] S. Benlarbi, K. El Emam, N. Goel, S. Rai, Thresholds for object-oriented measures, Proceedings 11th International Symposium on Software Reliability Engineering. ISSRE 2000, IEEE Comput. Soc, 2000, pp. 24–38, http://dx.doi.org/10.1109/ISSRE.2000.885858.
[25] C. Catal, O. Alan, K. Balkan, Class noise detection based on software metrics and ROC curves, Inf. Sci. 181 (21) (2011) 4867–4877, http://dx.doi.org/10.1016/j.ins.2011.06.017.
[26] K.A. Ferreira, M.A. Bigonha, R.S. Bigonha, L.F. Mendes, H.C. Almeida, Identifying thresholds for object-oriented software metrics, J. Syst. Softw. 85 (2) (2012) 244–257, http://dx.doi.org/10.1016/j.jss.2011.05.044.
[27] F. Oliveira, M.T. Valente, F.P. Lima, Extracting relative thresholds for source code metrics, 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), IEEE, 2014, pp. 254–263.
[28] R. Shatnawi, Deriving metrics thresholds using log transformation, J. Softw. 27 (2) (2015) 95–113, http://dx.doi.org/10.1002/smr.1702.
[29] Ö.F. Arar, K. Ayan, Deriving thresholds of software metrics to predict faults on open source software: replicated case studies, Expert Syst. Appl. 61 (2016) 106–121, http://dx.doi.org/10.1016/j.eswa.2016.05.018.
[30] J. Moeyersoms, E. Junqué de Fortuny, K. Dejaeger, B. Baesens, D. Martens, Comprehensible software fault and effort prediction: a data mining approach, J. Syst. Softw. 100 (2015) 80–90, http://dx.doi.org/10.1016/j.jss.2014.10.032.
[31] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I.H. Witten, The WEKA data mining software: an update, ACM SIGKDD Explor. Newsl. 11 (1) (2009) 10, http://dx.doi.org/10.1145/1656274.1656278.
[32] J. Bansiya, C. Davis, A hierarchical model for object-oriented design quality assessment, IEEE Trans. Softw. Eng. 28 (1) (2002) 4–17, http://dx.doi.org/10.1109/32.979986.
[33] L. Etzkorn, J. Bansiya, C. Davis, Design and code complexity metrics for OO classes, J. Object-Orient. Program. 12 (1) (1999) 35–40.
[34] T. Menzies, R. Krishna, D. Pryor, The promise repository of empirical software engineering data, 2016.
[35] J. Sayyad Shirabad, T. Menzies, The PROMISE repository of software engineering databases, 2005.
[36] M. D'Ambros, M. Lanza, R. Robbes, An extensive comparison of bug prediction approaches, 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010), (2010), pp. 31–41, http://dx.doi.org/10.1109/MSR.2010.5463279.
[37] The Apache Software Foundation, Apache Lucene, 2016.
[38] The Apache Software Foundation, Apache POI - the Java API for Microsoft documents, 2016.
[39] The Apache Software Foundation, Apache Tomcat, 2016.
[40] T. Mende, R. Koschke, Effort-aware defect prediction models, 2010 14th European Conference on Software Maintenance and Reengineering, (2010), pp. 107–116, http://dx.doi.org/10.1109/CSMR.2010.18.
[41] The Eclipse foundation, JDT core component, 2016.
[42] J. Demšar, Statistical comparisons of classifiers over multiple data sets, J. Mach. Learn. Res. 7 (2006) 1–30.
[43] H. Lu, B. Cukic, M. Culp, Software defect prediction using semi-supervised learning with dimension reduction, Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012, (2012), p. 314, http://dx.doi.org/10.1145/2351676.2351734.

ANNEXE D

BOUCHER & BADRI, 2017C

An Unsupervised Fault-Proneness Prediction Model Using Multiple Risk Levels For Object-Oriented Software Systems : An Empirical Study

Alexandre Boucher[1], Mourad Badri[2,*]

University of Quebec, Trois-Rivières, Canada

Abstract

Context: Nowadays, software quality is an important subject in software engineering. Many fault-proneness prediction models (approaches) have been developed in the literature to identify fault-prone source code. However, most of these models cannot be easily automated and used in a real-life development context, since they use supervised algorithms, therefore considering that fault data is available.

Objective: Our objective is to build a fault-proneness prediction model that can easily be automated and used practically when fault data is not available (or limited). The model should also output multiple fault-proneness risk levels.

Method: We developed the MRL (Multiple Risk Levels) model, which is based on an unsupervised approach labeling classes with different risk levels. The approach uses source code metrics and threshold values to determine the risk level of each class. We compared the MRL model to an unsupervised fault-proneness prediction model (HySOM) proposed in the literature. Additionally, we investigated the correlation between its fault-proneness risk levels and faults' severity. We also compared it with two supervised algorithms (Bayes Network and ANN) trained on previous versions of different software systems.

Results: Overall, the MRL model gave better results than the HySOM model. It also performed similarly or better than the Bayes Network and ANN based approaches. Furthermore, faults' severity was more important in the higher risk levels given by the MRL model.

Conclusion: The MRL model gives good fault-proneness prediction performance. It achieves performance similar or better than the investigated supervised models, but it works even when fault data is not available. It also achieves better performance than the unsupervised HySOM model. In addition, there is a significant and strong correlation between the higher risk levels given by the MRL model and faults' severity.

Keywords: Unsupervised Fault-Proneness Prediction, Fault Severity, Risk Levels, Metric Threshold Values, Object-Oriented Metrics, Machine Learning Algorithms, Object-Oriented Software Systems, Empirical Study.

1. Introduction

Nowadays, software quality is an important subject in software development. With the complexity, pervasiveness and criticality of software growing ceaselessly, ensuring that it behaves according to the desired

*Corresponding author

Email addresses: Alexandre.Boucher2@uqtr.ca (Alexandre Boucher), Mourad.Badri@uqtr.ca (Mourad Badri)

[1]Master Student, Software Engineering Laboratory, Department of Mathematics and Computer Science

[2]Professor, Software Engineering Laboratory, Department of Mathematics and Computer Science

levels of quality becomes more crucial, increasingly difficult and expensive [1]. If a software system contains faults, this can sometimes result in major damage or important losses of money. To prevent these faults, particularly the most severe ones, and ensure having a high quality software, it needs to be rigorously tested. However, exhaustive testing is cost prohibitive and is typically not feasible except perhaps in extremely trivial cases. In addition, it is not realistic to apply equal testing effort to all components of a large and complex software system. In fact, software testing often has to be done under severe pressure due to limited resources and tight time constraints. Therefore, testing efforts have to be focused [2].

In order to address this issue, many researchers suggested, among others, fault-proneness prediction (FPP) models. These models try to predict which parts of a software system are likely more fault-prone. These models are sometimes called software quality models [3, 4], since fault-proneness and source code quality are two closely related concepts. In fact, lower quality code is likely more fault-prone and contrarily higher quality code is likely less fault-prone [4]. To predict fault-prone source code, these models often use source code metrics (or object-oriented metrics), as they evaluate different attributes of a software system source code, like size, complexity, coupling, etc. [5, 6]. However, these models can also use other indicators such as anti-patterns [7], fault data history [8], etc. In our study, we mainly focused on models using source code metrics, as they are simple to understand for developers, widely investigated in the literature and can achieve good prediction performance. These models can be used by developers and testers, among others, to prioritize the implementation of unit tests for the system under test, making it possible to focus on the most critical parts of the source code. Unit test implementation efforts should therefore be better distributed across the software system, consequently reducing its overall fault-proneness.

Although many FPP models were suggested in the literature, none is perfect and widely accepted. These models can be divided in three main categories: unsupervised, semi-supervised and supervised. Most of the proposed models are based on supervised learning approaches [9–15], using fault data to predict fault-prone source code. However, fault data is not always available or can be very limited, making supervised approaches not always possible to use [16, 17]. Additionally, it can be expensive to collect good quality fault data [18], required for these types of approaches.

Furthermore, most of the studies investigating FPP models based on supervised learning algorithms train and test their models on the same version of a software system [10–15], using some parts for training and other parts for testing. However, in a real-life development context, fault data from one or multiple previous versions of a software system will need to be used as training data [19, 20].

Different semi-supervised and unsupervised approaches were suggested in some studies these last years [3, 17, 18, 20–24], making FPP possible when fault data history is non-existent or limited. Semi-supervised approaches are mostly supervised approaches, except that they work with very limited fault data. As to unsupervised models, they don't use fault data at all to predict fault-proneness of the source code, which makes them more usable in practice. Most of the unsupervised models proposed in different studies use threshold values of source code metrics to determine if a class or a function is fault-prone or not. Thresholds-based FPP models are often simpler to understand by developers than models trained using machine learning algorithms (like supervised and semi-supervised ones), making them more accessible.

In a previous work we published [25], a particular attention was given to the study of Abaei *et al.* [17], presenting the HySOM algorithm, which is an unsupervised FPP model (even if it is presented as a semi-supervised one). It uses SOM (Self-Organizing Map), ANN (Artificial Neural Network) and source code metrics' threshold values to predict if the functions of a software system are fault-prone or not. One problem we noted with the HySOM model is the quality of its prediction, which we thought could be improved.

To solve this problem, we implemented in a first time the original algorithm and developed multiple variants of it, trying to achieve different goals while improving its overall performance. We also adapted the original HySOM algorithm to work with class-level code metrics [25]. However, we thought that the results of unsupervised FPP could be improved and propose in the current paper the MRL (Multiple Risk Levels) model. It uses class-level source code and object-oriented metrics to predict fault-prone classes. We thought, in fact, that class-level prediction could be better and more appropriate than function-level prediction, since most systems nowadays are object-oriented ones and that unit tests cover classes and not functions. This has been confirmed in the previous study we performed [25].

In the MRL model, source code metrics' threshold values were used to label system classes with 5 fault-

proneness risk levels, which are: very high, high, medium, low and very low. We used 5 levels in order to give developers and testers easily understandable information on which classes are more critical than others. We think that having 5 different risk levels is more informative to users than only splitting classes as fault-prone or not, especially when the software system investigated contains a large number of classes. Having multiple risk levels also better guide developers and testers to the most critical parts of the software system, as very high risk classes should require more testing effort than medium risk classes.

The current study aims at answering the five following research questions:

RQ1: Can the proposed MRL model outperform existing unsupervised models?
We investigated if the MRL model yields good FPP performance and outperforms an existing unsupervised model (HySOM) presented in the literature.

RQ2: Is there a relationship between the risk levels given by the MRL model and faults' severity?
The proposed MRL model categorizes classes in five levels of fault-proneness risk. We wanted to investigate if classes with higher fault-proneness risk levels are also more prone to have higher severity faults.

RQ3: Can the MRL model perform similarly or better than supervised FPP models using data from previous versions?
In a real-life context, supervised models can be trained using previous version(s) data of a software system. We investigated if the MRL model can perform similarly or better than such models, therefore making it useful even when fault data is available.

The rest of the paper is organized as follows: Section 2 presents a summary of main-related works. Section 3 presents the background of our study and the important elements that it is based on. Section 4 presents the methodology we followed to conduct our research. Section 5 presents and discusses the results we obtained. Finally, Section 6 presents the possible threats to validity of our study and Section 7 summarizes and concludes this paper, in addition to giving some future work directions.

2. Related Work

Most of FPP approaches proposed in the literature require existing fault data to train a model and then use it for prediction. Unfortunately, this data is not always available [16, 17] or of good quality, making these approaches unusable in many cases (at least their usage is limited). The required fault data needs in fact to be of good quality, which acquisition can be expensive [18]. This major flaw can prevent most companies from using these models (approaches). Over the past years, some studies suggested semi-supervised and unsupervised FPP models that can be used when fault data is limited or absent. A review of these papers is presented in this section.

2.1. Semi-Supervised Fault-Proneness Prediction Models

Several studies investigated semi-supervised FPP models, used when fault data is available but limited. These models require, indeed, less fault data than supervised ones.

In two studies by Lu *et al.* [18, 26], semi-supervised learning was considered for FPP using limited fault data, Random Forest and Dimension Reduction. They found that reducing the dimensionality of the source code metrics significantly improved the semi-supervised learning model. They also found that semi-supervised training improves the corresponding supervised learning, when the same machine learning algorithm is used for both.

Furthermore, Catal investigated different semi-supervised classification algorithms for FPP [24]. He compared four different approaches to predict fault-prone code when fault data is limited. These approaches are Low-Density Separation, Support Vector Machine, Expectation-Maximization and Class Mass Normalization. He concluded that Low-Density Normalization gave the best results for large datasets, but could also be used for smaller ones.

2.2. Unsupervised Fault-Proneness Prediction Models

Unsupervised FPP models do not require any fault data. This is the main advantage they have on the supervised and semi-supervised models suggested in the literature.

In [21], Catal *et al.* used metrics' threshold values to consider functions as fault-prone or not. The authors only focused on the function level granularity. They used the proposed unsupervised FPP model on three public datasets (namely AR3, AR4 and AR5). They investigated two variants of this model: one applying threshold values directly on the functions metrics and another one clustering the functions using K-means clustering algorithm before applying the threshold values. The obtained results were compared to a supervised approach using Naive Bayes Network algorithm. They concluded that their approach gave acceptable classification performance and could easily be automated. One year later, the same authors conducted a similar experiment with the X-means clustering algorithm [3]. The advantage of X-means over K-means is that the number of clusters does not need to be fixed before running the clustering algorithm, therefore making the approach easier to automate. They achieved acceptable classification performance too in this study.

In a study by Bishnu & Bhattacherjee [22], the authors used a similar approach as Catal *et al.* in [3, 21], using K-means clustering algorithm and the same threshold values to predict fault-prone code. However, they used the Quad-Tree algorithm in conjunction with a genetic algorithm to initialize the clusters used in the K-means algorithm. According to the authors, the classification performance of their model is comparable to the ones obtained using supervised models.

In a study by Abaei *et al.* [23], the SOM algorithm was used to cluster functions similarly to previously mentioned studies [3, 21, 22] and the same threshold values were used to discriminate source code functions as fault-prone or not. According to the authors, SOM was preferred over K-means clustering algorithm because it offers better performance, is less likely to find a local optimum and the number of neurons can automatically be determined (via a defined function). Results were found to be good, in fact better than in Catal *et al.* and Bishnu & Bhattacherjee studies [3, 21, 22]. But prediction results were found to be even better using the HySOM model suggested by Abaei *et al.* [17]. The main difference is that they used a neural network in conjunction with the SOM algorithm to classify source code functions as fault-prone or not.

In a recent study, we investigated the usage of object-oriented source code metrics and threshold values for FPP [27]. We investigated the usage of Chidamber & Kemerer source code metrics [28] and the SLOC (Source Lines of Code) metric. After an univariate logistic regression analysis performed on 12 datasets, we decided to perform FPP using SLOC, CBO (Coupling Between Objects), RFC (Response For a Class) and WMC (Weighted Methods per Class) metrics. For the calculation of threshold values, we investigated the usage of 3 different methods: ROC Curves [29], VARL [30] and Alves Rankings [31]. We concluded that VARL was not a good choice for FPP and that ROC Curves was the best for FPP among the three techniques. However, Alves Rankings also performed very well and is able to calculate threshold values without any fault data (contrarily to ROC Curves and VARL), which is interesting to perform unsupervised FPP.

In another recent study, we adapted the HySOM model for class-level FPP [25]. We decided to adapt the model for class-level usage because we thought that evaluating fault-proneness at the class granularity level is more relevant for users. Since unit testing in object-oriented software systems is performed for classes, it makes sense to perform FPP at class-level. We adapted the model by changing the source code metrics and threshold values used. For this purpose, we used the ROC Curves [29] and Alves Rankings [31] threshold values calculation techniques. We obtained better prediction results with the adapted HySOM model than the original one. We also compared the adapted model with supervised models (Naive Bayes Network, ANN and Random Forest) using 10-fold cross-validation. After comparing the supervised models with the adapted HySOM model, we concluded that the adapted model gave better results than the supervised approaches.

A recent study by Erturk & Sezer [20] suggested an unsupervised approach for FPP, along with a supervised approach using previous versions of a software system when fault data is available. They used Fuzzy Inference Systems and an expert knowledge to classify classes as fault-prone or not when no fault data is available. Once an iteration or a version of the system is finished and that fault data is available, an

Artificial Neural Network and an Adaptive Neuro Fuzzy Inference System, which are supervised approaches, are used to indicate fault-prone classes through three levels of risk for the next version of the system. This model seems giving good results overall, but the use of an expert to initialize the unsupervised algorithm therefore yields two drawbacks: (1) the approach is not completely automated, and (2) it requires someone with a good knowledge about Fuzzy Inference Systems, which may not be the case for most companies developing software systems.

3. Research Background

This section presents the background knowledge used to perform this research.

3.1. Dependent and Independent Variables

When considering classification and prediction problems, there are always dependent and independent variables. For FPP, the dependent variable is often binary and is the presence or absence of faults in a module (function, class or package). In our study, the source code metrics are used as independent variables.

Since the models investigated in this study are mainly thresholds-based models, the choice of source code metrics and threshold values is very important. In the literature, many metrics have been suggested and used to describe the source code of a software system. Source Lines of Code (SLOC) and CK metrics have been widely used for FPP [5, 9, 10, 29, 32] (see Table 1 for a presentation of each source code metric investigated [28]). We therefore decided to consider these metrics to perform our study. In fact, we chose a subset of these metrics, because not all of them are good predictors of fault-proneness (see Section 4.1 for the resulting subset).

Table 1: Source Code Metrics Investigated.

Metric	Description
SLOC (Source Lines of Code)	Number of source code lines in a class, excluding commented and blank ones.
CBO (Coupling Between Objects)	Number of classes to which the class is coupled.
RFC (Response For a Class)	Number of methods that can potentially be executed when the class receives a message.
WMC (Weighted Methods per Class)	The sum of the cyclomatic complexities of all methods.
LCOM (Lack of Cohesion in Methods)	Measures the lack of cohesion of a class using the similarity of the methods.
DIT (Depth of Inheritance Tree)	The depth of the class in the inheritance tree.
NOC (Number of Children)	The number of immediate subclasses to a class.

3.2. Data Collection

In order to perform FPP, data including both dependent (faultiness) and independent (source code metrics) variables is needed. In a real-life enterprise context, source code metrics and faults would be obtained directly from the source code and bug tracker. However, in FPP studies, this data is often taken from publicly available sources. This makes the studies easier to reproduce and compare [5].

In this study, we needed class-level fault-proneness data, since we consider class-level FPP. We therefore used twelve datasets: Apache ANT (versions 1.3, 1.4, 1.5, 1.6 and 1.7), Apache IVY 2.0, Apache Lucene 2.4, Apache POI 3.0, Apache TOMCAT 6.0, KC1, JEdit 4.3 and Eclipse JDT Core. Most of these systems are available on the PROMISE Repository (except KC1 and Eclipse JDT Core) [33] and all of them contain information on the number of faults in each of the system's classes.

The Apache ANT datasets were used in many studies [14, 34–36], especially the one built on version 1.7 of the system. ANT is a command-line tool developed in Java mainly used for building Java applications [35]. Another dataset used was made for Apache IVY 2.0, which was also used in multiple studies [13, 14, 36]. IVY is a dependency manager developed in Java, integrated in Apache ANT [35]. Apache LUCENE (version 2.4) is a text search engine library written in Java [37] and is used in some studies [13, 35, 36, 38]. Apache

POI is a library regrouping Java APIs to read or write documents following Office Open XML standards [39] and was used in multiple studies [11, 13, 35, 36]. The last Apache project we selected is TOMCAT, which is an open source implementation of multiple Java Web server technologies [40]. Many studies related to FPP use the Apache TOMCAT dataset [13, 14, 35, 36]. The KC1 [41] system was developed by the NASA with the C++ language and was used in numerous studies [10, 12, 14, 15, 17, 42]. Another dataset we used was built for the JEdit 4.3 program, which is a text editor developed in Java [36]. It was used in multiple studies for FPP [13, 14, 35, 36]. The last dataset used is based on the Eclipse JDT Core system. It was produced after a study by D'ambros *et al.* [38] on multiple releases of the system. The JDT Core is the primary infrastructure of the Eclipse Java IDE, which includes a compiler, a code formatter, a code assistance and other practical features for the developers using the Eclipse Java IDE [43]. The Eclipse project was used in numerous studies [5, 12, 29, 32, 38, 42, 44]. Although the JDT Core Component wasn't used specifically in those studies, we used this dataset for the simplicity of the data acquisition and to simplify study replication.

It is important to note that for all Apache datasets and JEdit, the WMC metric value had to be calculated, as the one provided in the datasets simply gave the number of methods in each class, according to the study that built the datasets [35]. To calculate the WMC value, we took the number of methods in each class and multiplied it with the average cyclomatic complexity of all methods in the class, therefore resulting in the sum of the cyclomatic complexity of all methods (WMC metric value).

3.3. Machine Learning Algorithms

In this paper, we use two machine learning algorithms for FPP which are presented in this section. These two algorithms Bayes Network and Artificial Neural Network (ANN).

3.3.1. Bayes Network

The Bayes Network algorithm classifies the given instances by building a Bayesian Network (directed graph). When applied to FPP, this graph maps metrics as nodes and their independencies as links between the metrics to classify instances as fault-prone or not [44]. It can be used in different variants, like the Naive Bayes Network. In our case, we used the standard Bayes Network algorithm.

3.3.2. Artificial Neural Network (ANN)

The ANN, or more precisely the Multilayer Perceptron as it is used in this study, classifies elements in 2 or more categories. It can do so by representing a potentially non-linear function, therefore having a better classification potential than linear regression. A Multilayer Perceptron is minimally composed of 2 layers, one input layer and one output layer. It also often has one or more hidden layers composed of one or more neurons. In this study, we use a feedforward neural network, which uses the backpropagation algorithm for the training phase. A feedforward Multilayer Perceptron can be considered as a directed graph, considering neurons as nodes and links between neurons as edges of the graph. Each neuron is linked to all the neurons of the next layers in a strongly connected network.

3.4. HySOM Model

The HySOM FPP model has been suggested by Abaei *et al.* [17] and does not require existing fault data to use. In a previous study, we adapted the original HySOM model to work with class-level code metrics instead of function-level ones [25].

The HySOM model clusters all functions (or classes) of a software system using the SOM (Self-Organizing Map) algorithm. Once the clustering is done, each cluster is assigned a FP (Fault-Prone) or NFP (Not Fault-Prone) value. A cluster is considered as fault-prone if at least 3 metrics exceed their corresponding threshold value. An Artificial Neural Network (ANN) algorithm is then trained using the cluster centroids and the FP or NFP values determined with the threshold values. Once the ANN is trained, it can be used directly with the functions (or classes) of the software system to predict fault-prone source code.

3.5. Alves Rankings Thresholds Definition Method

For thresholds-based FPP approaches, the choice of threshold values is important. In this study, we considered Alves Rankings method to calculate threshold values for the proposed MRL model. In a previous study, we investigated three thresholds calculation techniques, which are ROC Curves, VARL and Alves Rankings [45]. The ROC Curves and Alves Rankings approaches both gave good results for FPP. However, the Alves Rankings method is the only one which can be used in a completely unsupervised way. We therefore decided to use this technique to calculate threshold values.

The Alves Rankings method was presented by Alves *et al.* [31] to calculate threshold values on classes to describe their quality. In the original paper, threshold values are calculated by going through six steps. In our study, we only used steps 1, 2, 3 and 6. So, we only present these steps in the following. The reason we only used four of the six steps is that steps 4 and 5 aggregate the threshold values of multiple datasets together (100 in fact) [31]. Since the model is meant to be unsupervised and that threshold values could be different for different software systems, we only calculated threshold values on a per dataset basis.

The first step of this approach is called *metrics extraction* and consists in extracting the source code metrics of the system [31]. The *weight* of each class is also calculated in this step, which is in fact the SLOC value of a class (which is its size). In our study, the first step is already achieved by choosing the investigated datasets.

The second step, *weight ratio calculation*, consists in calculating the *weight ratio* of each class [31]. The calculated ratio is in fact the percentage of the source code a given class represents in the whole system, according to the SLOC metric. It is calculated by dividing each class weight by the sum of all classes weights.

The third step of the Alves Rankings methodology, *entity aggregation*, consists in aggregating the weight of all entities (or classes) per metric values [31]. The result given by this step is similar to a weighted histogram giving the percentage of the code of the system represented by each source code metric value. For example, a conclusion that could be drawn after that step is that 2% of the system consists of source code having a CBO metric value of 13.

Fourth and fifth steps of the approach were only used to output the same result as in step three, but with 100 different software systems. In our case, we skipped these steps to calculate different threshold values per system.

The sixth step of this approach, *thresholds derivation*, consists in calculating the threshold values using the output of step 5 (or step 3 in our case). To do so, we choose a percentage of code we want to represent with our threshold value. For example, if we want to target 20% of the most coupled code, we would choose a percentage of 80%. Looking at the output of step 3, we could see that 80% of the classes have a CBO metric value of 29 or less, making us choose 30 as the threshold value.

4. Research Methodology

In this section, we present how we conducted our research to answer the 3 research questions defined. We present how we chose the source code metrics we use, how we evaluated the models and how each experiment was performed.

4.1. Choosing the Source Code Metrics

In order to propose a new FPP model working at class-level granularity, we needed object-oriented and class-level source code metrics. To determine the metrics to use, we performed an univariate logistic regression analysis to investigate the relationships between class-level source code metrics and faults. The analysis was performed for the SLOC (Source Lines of Code) metric and Chidamber & Kemerer source code metrics [28].

We performed this analysis in two previous studies on FPP [25, 27]. In fact, we performed a logistic regression analysis on the Apache ANT 1.7 dataset in [25] and we performed the same analysis on a total of twelve datasets in [27] (the same datasets as presented in Section 3.2). In both papers, we concluded that the best metrics to use for class-level FPP are SLOC, CBO, RFC and WMC. Additionally, the same subset

of metrics was retained in a study from Isong & Obeten [5]. They considered that these metrics were the most relevant for FPP according to most studies they considered in their systematic review.

In our logistic regression analysis, we considered the number of faults in each class. To do so, we duplicated classes in the datasets according to the number of faults they contained. For example, if a class contained 3 faults, it would be present 3 times in the dataset instead of once, to correctly consider the 3 faults. In our research, we came by two studies, one by Zhou & Leung [10] and another one by Shatnawi [44] that considered the number of faults in this way. Considering the number of faults in each class should make the logistic regression more accurate at finding source code metrics related to fault-proneness.

4.2. Performance Evaluation

This section presents how the models' evaluation and comparison were performed in our study.

4.2.1. Evaluation Method

When evaluating classification models performance, a confusion matrix (or classification table) is built to describe the obtained results. It is a small square matrix giving the number of true positives, false positives, true negatives and false negatives obtained using the classification algorithm. In FPP, a positive is when a source code function, class or module is fault-prone. Oppositely, a negative is when this same instance is not considered fault-prone. Following this idea, a true positive is when a function is fault-prone and is classified as such and a true negative is when a function is not fault-prone and is correctly classified as such. A false positive is when a function is classified as fault-prone but is actually not. Equivalently, a false negative is when a function is not considered fault-prone but is actually fault-prone. The structure of a confusion matrix is presented in Table 2.

Table 2: Confusion Matrix Structure.

	Actual	
Classified	Faulty	Not-faulty
Faulty	True positives (TP)	False positives (FP)
Not faulty	False negatives (FN)	True negatives (TN)

Using the resulting confusion matrix, performance metrics can then be calculated to evaluate and compare the classification performance of classification models. Multiple evaluation metrics exist, but we only present and use few of them.

In many FPP papers, the Error Rate, the FPR (False Positive Rate) and the FNR (False Negative Rate) metrics are used to evaluate and compare FPP models [3, 17, 21–23]. The formulas used to calculate these classification metrics are:

$$\text{Error Rate} = \frac{FP + FN}{FP + FN + TP + TN} \tag{1}$$

$$FPR = \frac{FP}{FP + TN} \tag{2}$$

$$FNR = \frac{FN}{FN + TP} \tag{3}$$

The Error Rate represents the percentage of incorrectly classified instances. As to the FPR, it gives the percentage of actually not fault-prone instances that were classified as fault-prone. The FNR gives the percentage of actually fault-prone instances that were considered as not fault-prone by the FPP model. It is important to note that the error rate, FPR and FNR classification metrics are better the lower they are.

However, in the presentation of our results, we omitted the error rate metric. We did so because it is not relevant when considering FPP, because the data to classify is often imbalanced (not half of the source code modules are fault-prone and the other half not fault-prone) [44]. In fact, we replaced the error rate

metric with the geometric mean (g-mean). This metric is often used to evaluate the FPP performance of imbalanced datasets [14, 32]. Furthermore, this classification metric alone gives a good estimate of how good the classification is [32]. It therefore makes it easier to compare classification performance results with each other. However, FPR and FNR still give useful insights about what the classification did well and what it did not.

The g-mean evaluation metric uses two different accuracies, which are the accuracy of positives (TPR) and the accuracy of negatives (TNR) [14]. These metrics can be calculated from the FPR and FNR metrics, as they are their opposites. Contrarily to FPR and FNR, where lower is better, TPR, TNR and g-mean metrics are better the higher they are. The g-mean metric will give a higher value if both TPR and TNR are good, otherwise it won't. The reason we didn't use TPR and TNR to describe classification performance along with g-mean is that FPR and FNR are used more often in FPP papers, therefore simplifying comparison of our results with other studies. In addition, since TPR and TNR can easily be calculated from FNR and FPR, we didn't see the need to include them. Here are the equations used to calculate TPR (True Positive Rate), TNR (True Negative Rate) and g-mean:

$$TPR = 1 - FNR = \frac{TP}{TP + FN} \tag{4}$$

$$TNR = 1 - FPR = \frac{TN}{TN + FP} \tag{5}$$

$$g\text{-}mean = \sqrt{TPR * TNR} \tag{6}$$

In order to represent g-mean values in a textual manner and therefore simplifying analysis and interpretation of the results, we considered the following levels to describe the g-mean values obtained:

- g-mean < 0.5 means no good classification;
- $0.5 \leq$ g-mean < 0.6 means poor classification;
- $0.6 \leq$ g-mean < 0.7 means fair classification;
- $0.7 \leq$ g-mean < 0.8 means acceptable classification;
- $0.8 \leq$ g-mean < 0.9 means excellent classification;
- g-mean ≥ 0.9 means outstanding classification;

To summarize, FPR, FNR and g-mean evaluation metrics are used in the rest of the paper to evaluate and compare variants of different models together.

Additionally, when performing our experiments, we use 10-fold cross-validation. This cross-validation divides the dataset in 10 parts, where 9 out of 10 parts are used for training and one for testing. This is done 10 times, each time with a different testing part, making the whole dataset tested.

4.2.2. Comparison Method

To compare the performance of different models, we needed an objective comparison methodology. In this study, we used the methodology suggested by Demšar, to compare the performance of different models or classifiers over multiple datasets [46]. This methodology consists in using the Friedman statistical test in conjunction with the Nemenyi post-hoc test. It was also used in other studies about fault-proneness prediction to compare the results of different models [12, 42]. Furthermore, we already used this technique in a previous study on FPP [27].

The Friedman test is interesting to use for FPP because it is a non-parametric test and does not evaluate the performance of the distribution, it only compares them. To do so, it compares the average rank of the

different models on the different datasets. The Friedman statistic is therefore calculated as follows, where k is the number of models, N the number of datasets and R_j the average rank of the model j on all datasets.

$$X_F^2 = \frac{12N}{k(k+1)}\left(\sum_j R_j^2 - \frac{k(k+1)^2}{4}\right) \tag{7}$$

The X_F^2 statistic is then compared to its critical value to check if the null hypothesis is rejected or not. The null hypothesis of the test states that there is no significant difference between the models. If the null hypothesis is rejected, there is a significant difference between at least two of the models. Demšar therefore recommends doing a post-hoc Nemenyi test to compare the performance between each pair of models [46]. According to the Nemenyi test, there is a significant performance difference between two models if the average rank CD differs by at least the critical difference (available in [46]).

$$CD = q_\alpha \sqrt{\frac{k(k+1)}{6N}} \tag{8}$$

In the above equation, q_α is based on the critical values of the Studentized range statistic divided by $\sqrt{2}$, according to [46].

In our study, we therefore decided to use the Friedman test and the post-hoc Nemenyi test to statistically compare the performance of the models. We performed the Friedman test using the g-mean performance metric, which describes the performance well. The statistical tests are performed using the XLSTAT[3] tool and 5% as the confidence level.

4.3. The Proposed MRL Model

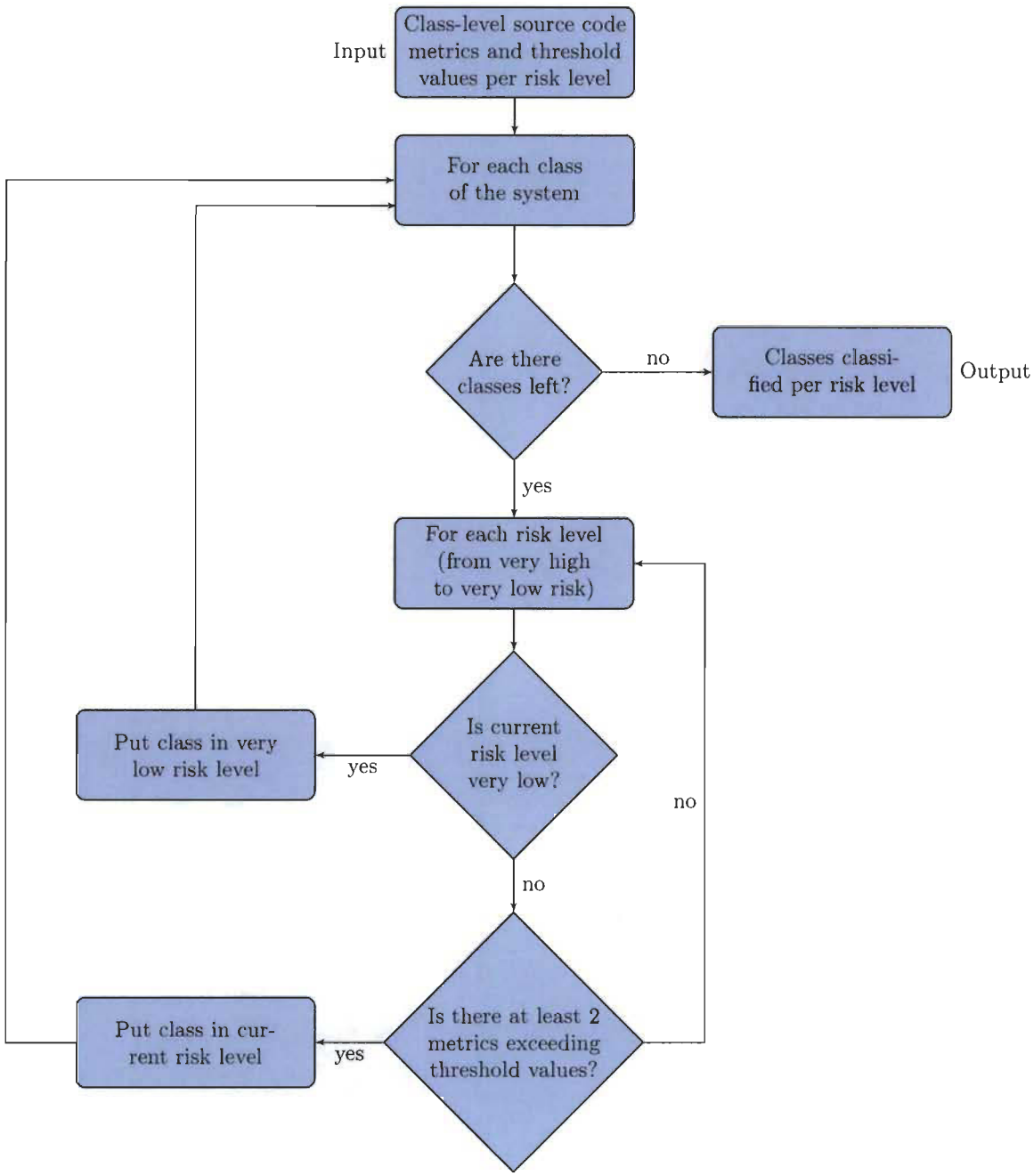
We developed and implemented a model that is completely unsupervised, outputting multiple levels of fault-proneness risk. To do so, we used the class-level source code metrics determined using the logistic regression analysis. We chose to use class-level source code metrics because they yielded significantly better results than function-level source code metrics in our previous work [25]. To define this new model, which we called the MRL (Multiple Risk Levels) model, we used threshold values calculated using the Alves Rankings method, since multiple threshold values can be calculated for the same source code metric using this approach. Furthermore, this threshold calculation technique is unsupervised and is simple to use. We then produced a completely unsupervised model using these threshold values only, which makes it very simple to use.

In the MRL model, we calculated threshold values for source code metrics using the Alves Rankings algorithm. We calculated the threshold values at 90%, 70%, 50% and 30% of the source code metrics distributions given by the Alves Rankings algorithm output. The MRL model simply considers that if two metrics or more exceed their threshold values at 90% of the Alves Rankings distribution, the corresponding class is classified in the very high fault-proneness risk level. If the class is not classified in the very high risk level, we check if two or more of its source code metrics exceed their threshold values at 70% of the Alves Rankings distribution. If this is the case, the class is classified in the high risk level. If it is not, the same algorithm goes on for threshold values defined at 50% of the Alves Rankings distribution for the medium risk level. If the class is not classified in the medium risk level, the same algorithm is applied with threshold values at 30% of the Alves Rankings distribution for the low risk level. If the class is not classified in the low risk level, it is automatically classified in the very low risk level. See Figure 1 for a visual representation of the MRL model workflow. We nicknamed this model as MRL (Multiple Risk Levels), because it considers different levels of fault-proneness risk.

The choice of the levels from which threshold values are picked (90%, 70%, 50% and 30%) and of the number of source code metrics to exceed threshold values (2 for all levels) was made following several tests we did with multiple variants of these parameters. We made these tests on the class-level investigated datasets

[3]XLSTAT `https://www.xlstat.com/`

Fig. 1. MRL Model Workflow.



and considered the average g-mean value of each variation to pick the best one. We also considered keeping a certain balance in the number of source code classes given by the different risk levels.

Note that before choosing to use five levels of fault-proneness risk in the MRL model, we investigated the MRL model with only three levels of fault-proneness risk (high, medium and low). This variant gave the

exact same classification performance (since the lowest risk level was still delimited by 30% Alves Rankings threshold values). However, we found that when using five risk levels, classes were better distributed in the different risk levels than when using three risk levels. In fact, the model seems easier to use with five risk levels, since higher risk classes are better distinguished compared to the other ones in the higher risk levels. We therefore decided to use the MRL model with five fault-proneness risk levels.

To test our model, we ran it twice on each dataset, once using the original dataset and once using the dataset containing duplicated source code classes according to the number of faults each one contains (as done to determine which class-level source code metrics to use, see Section 4.1). Testing it while considering the number of faults in each class makes the classification performance results more accurate, since a class containing 10 faults will count for 10 true positives if it is correctly classified, but 10 false negatives if it's not. This therefore gives more weight to each correctly and incorrectly classified class (or fault).

This experiment performed with the proposed MRL model aims at answering RQ1. To answer this research question, we decided to compare the MRL model results with the results given by another unsupervised model. To do so, we compared the MRL results with the class-level HySOM model we adapted in a previous study [25]. The adapted HySOM model proposed performed better than the original HySOM model proposed by Abaei *et al.*, which was found to perform better than existing unsupervised approaches [17]. Moreover, the adapted HySOM model performed better than the three supervised approaches investigated in the same study (Naive Bayes Network, ANN and Random Forest) [25]. For this reason, the MRL model is compared with the adapted HySOM model, considering that if it performs better than HySOM, it therefore outperforms the supervised models investigated. We therefore kept the four best models produced in the study on the adapted HySOM model, which are two adapted models using ROC Curves threshold values and two adapted models using Alves Rankings threshold values [25].

4.4. Investigating the Relationship Between the MRL Model and Faults' Severity

The MRL model proposed in this paper yielded good results for FPP. We wanted to investigate if higher risk levels outputted by the MRL model contain higher severity faults. The severity of a fault describes how serious is the impact of the fault on the software system. Faults' severity can be simply described as critical or noncritical, or more levels of severity can be used for further classify them. Taking example from Zhou & Leung study [10], who said that severity ratings in certain datasets could be rated from 1 to 5 (not to confound with fault-proneness risk levels used in the MRL model), 1 being that the fault is blocking the correct operation of the system and 5 being a trivial fault that does not require immediate correction. However, in our study, we simply considered classes as non-faulty, containing noncritical faults and containing critical faults, since faults' severity results from a subjective classification performed by the software development team. Other studies did likewise, like Zhou & Leung [10], which considered the NASA KC1 dataset and 5 severity ratings. They merged the severity ratings in two different rating categories, high (critical) and low (noncritical). The high level considered only the most severe faults with severity rating of 1, while the low level considered faults rated with severities 2 to 5. They did find that source code metrics were able to predict low severity (noncritical) faults well, but not high severity (critical) faults. Similarly to this study, Singh & Kahlon [47] considered three severity ratings: high, medium and low. They used this rating procedure to classify faults on three versions of the Mozilla Firefox Web browser. They, however, got better results when considering medium and high severity ratings than low severity ratings. Another study by D'ambros *et al.* [38] considering faults' severity concluded that it didn't help FPP performance.

In our study, we decided to investigate faults' severity used in conjunction with the MRL model with two datasets: KC1 and Eclipse JDT Core. It is important to note that the KC1 dataset containing faults' severity information is not the exact same KC1 dataset used previously in our study for class-level FPP. For both datasets, we associated a severity value to each class: 0 if the class is fault-free, 1 if the class contains noncritical faults and 2 if the class contains critical faults. Classes were also divided in a similar way in a previous study on faults' severity performed by Zhou & Leung in [10]. For the Eclipse dataset, we grouped certain severity ratings together to be able to categorize classes as previously mentioned. We therefore considered non-trivial, major and critical bugs as critical faults and trivial bugs as noncritical faults. Using this severity grouping methodology, there are not many bugs caused by critical faults, which is normal. According to Zhou & Leung, there should be more noncritical faults than critical ones [10].

We investigated if higher severity faults are found in higher risk levels by checking for each fault-proneness level if the classification correctly distinguished critical faults. We only investigated the prediction of critical faults, as trying to distinguish noncritical faults would almost be equivalent to considering all faults of the system (as previously done). It would almost be equivalent as classes containing critical faults almost always contain noncritical ones for both KC1 and Eclipse datasets. We therefore investigated critical faults prediction by considering classes as critically fault-prone when they exceeded the threshold values for the very high, high, medium and low risk levels of the MRL model. Each risk level result is compared with the original MRL model considering all faults of the system in order to check the percentage of faults detected that are critical ones (only true positives considered). Each experiment is performed with the original dataset (considering a faulty class as containing one fault) and with the dataset with classes duplicated based on the number of faults in each class.

To further investigate the relationship between each risk level and faults' severity, we performed a Spearman correlation analysis considering the severity of faults found in a class and the risk level predicted by the MRL model. We chose the Spearman correlation analysis over the Pearson analysis, simply because the variables did not follow a normal distribution (according to the test that we performed). The Pearson correlation is widely used in statistics to measure the degree of the relationship between linearly related variables. The variables should, however, be normally distributed (which is not the case in our study). The Spearman rank correlation is in fact a non-parametric test that is used to measure the degree of association between two variables. Spearman rank correlation test does not assume anything about the distribution of the variables.

We performed the Spearman technique on the original datasets and on datasets with duplicated class information, based on the number of faults in each class. For the correlation analysis, each class containing no fault had a 0 severity value, each class containing a fault had a 1 severity value and each class containing a critical fault had a 2 severity value. In order to run the correlation test, risk levels were also codified from 1 to 5 (1 being the very low risk level and 5 being the very high one).

These experiments comparing the MRL model's output and faults' severity aim at answering RQ2.

4.5. Comparing the MRL Model Performance With Cross-Version Supervised FPP

Supervised FPP models are commonly investigated when FPP is studied. Most of the time, they use source code metrics and fault data on one system or one previous version of a system and try to predict which classes of the new version of the software system are likely more fault-prone. We wanted to investigate if these well-studied approaches can outperform the unsupervised MRL model we proposed. To do so, we investigated the use of the Bayes Network and ANN algorithms as supervised FPP approaches. We already used the Bayes Network algorithm in a previous study on FPP and it achieved good results [45]. It was also used in other studies using supervised FPP [14, 44]. We also already used the ANN algorithm in a previous study and did find pertinent results with it [27]. Furthermore, the ANN algorithm was used in other studies on FPP [14, 17].

As previously done by Erturk & Sezer in [20], we performed two experiments training the supervised models on previous versions of a software system. We first trained each supervised algorithm on only one previous version of a software system and then on all the previous versions of the same software system. By training the algorithm on more than one previous version, more training data is available and it may therefore produce more accurate predictions [20].

For this experiment, we used datasets very similar to the other ones previously used. In fact, the datasets used were produced on the same software systems but for previous versions of it. The MRL model was also built for these models, so we can compare the results of both unsupervised and supervised approaches.

These experiments with supervised models trained using previous versions of software systems aim at answering RQ3. The Weka tool was used to build the supervised models with all default parameters set [48].

5. Experimental Results and Discussion

In this section, the results obtained from our experiments are presented and discussed.

5.1. MRL Model Results

This section presents the results obtained using the MRL model. As mentioned previously, the model is investigated twice: once using the standard datasets and once after duplication of each class based on the number of faults it contains. In the presented results, each of the models considered is either suffixed with -B (for binary) for the tests on the original dataset or -D (for duplicated) for the tests performed on the duplicated datasets.

Table 3 presents the results obtained with both tests on each dataset. In the classification performed, a class is considered as positive (fault-prone), if it is in a low risk level or above, or as negative (not fault-prone), if it is in the very low risk level. Table 4 gives the results obtained in a previous study we published adapting the HySOM model for class-level usage (which gave better results than the original HySOM model) [25]. We added these results using ROC Curves and Alves Rankings threshold values as a comparison baseline for the MRL model's performance. The suffix number of each model gives the number of source code metrics needed to exceed threshold values to consider a class as fault-prone. For example, the HySOM-ROC-2 model uses the ROC Curves threshold values to build the HySOM model and considers as fault-prone classes having at least 2 metrics exceeding threshold values.

Table 3: MRL Model Fault-Proneness Performance.

	MRL-B			MRL-D		
Dataset	**FPR**	**FNR**	**g-mean**	**FPR**	**FNR**	**g-mean**
ANT 1.3	0.343	0.150	0.747	0.343	0.182	0.733
ANT 1.4	0.304	0.450	0.619	0.304	0.404	0.644
ANT 1.5	0.326	0.281	0.696	0.326	0.257	0.708
ANT 1.6	0.205	0.250	0.772	0.205	0.152	0.821
ANT 1.7	0.238	0.265	0.748	0.238	0.172	0.794
IVY	0.170	0.350	0.735	0.170	0.286	0.770
LUCENE	0.175	0.547	0.611	0.175	0.402	0.702
POI	0.106	0.484	0.679	0.106	0.364	0.754
TOMCAT	0.206	0.260	0.767	0.206	0.184	0.805
KC1	0.224	0.433	0.663	0.224	0.269	0.753
JEdit	0.285	0.364	0.675	0.285	0.333	0.690
Eclipse	0.095	0.456	0.702	0.095	0.307	0.792
Mean	0.223	0.358	0.701	0.223	0.276	0.747

Looking at the obtained results, we can see that the average g-mean value is higher when using duplicated datasets, but also that the mean FNR is lower, which indicates that the classification is good, detecting more faults. The lower FNR means that the classification performs well when predicting fault-proneness of classes containing multiple faults. Of course, the FPR is always the same for both approaches, since duplicating classes using their number of faults won't yield more true positives or false positives (used in the FPR calculation).

Furthermore, we can see that the g-mean value is better for most investigated datasets when using the MRL model on non-duplicated datasets, when compared to the HySOM model using class-level and either ROC Curves or Alves Rankings threshold values. The average g-mean value is also higher for the MRL model. What is interesting is that our model is completely unsupervised and doesn't require any fault data, contrarily to the ROC Curves threshold values based models (two of the adapted HySOM models presented in Table 4).

We decided to compare the MRL model and the HySOM model adapted for class-level usage (see Section 2.2 for details on this study). We therefore compared the MRL model with four variants of the HySOM model adapted for class-level usage. Two out of four variants are constructed using ROC Curves threshold values, considering as fault-prone classes having at least 2 or 3 metrics exceeding threshold values. The

Table 4: HySOM Model Performance Using Class-Level Source Code Metrics and ROC Curves Threshold Values.

	HySOM-ROC-2			HySOM-ROC-3			HySOM-Alves-1			HySOM-Alves-2		
Dataset	**FPR**	**FNR**	**g-mean**	**FPR**	**FNR**	**g-mean**	**FPR**	**FNR**	**g-mean**	**FPR**	**FNR**	**g-mean**
ANT 1.3	0.124	0.400	0.725	0.124	0.450	0.694	0.371	0.250	0.687	0.324	0.150	0.758
ANT 1.4	0.348	0.500	0.571	0.268	0.650	0.506	0.449	0.375	0.587	0.355	0.575	0.524
ANT 1.5	0.261	0.563	0.569	0.134	0.781	0.435	0.598	0.156	0.583	0.276	0.406	0.656
ANT 1.6	0.232	0.326	0.720	0.139	0.315	0.768	0.378	0.315	0.652	0.181	0.413	0.693
ANT 1.7	0.173	0.446	0.677	0.133	0.440	0.697	0.368	0.247	0.690	0.188	0.464	0.660
IVY	0.141	0.425	0.703	0.157	0.375	0.726	0.260	0.325	0.707	0.157	0.400	0.711
LUCENE	0.263	0.542	0.581	0.175	0.665	0.526	0.336	0.360	0.652	0.153	0.700	0.504
POI	0.298	0.285	0.709	0.230	0.327	0.720	0.280	0.335	0.692	0.118	0.573	0.614
TOMCAT	0.152	0.390	0.719	0.151	0.468	0.672	0.305	0.299	0.698	0.147	0.429	0.698
KC1	0.224	0.267	0.755	0.306	0.450	0.618	0.188	0.500	0.637	0.082	0.600	0.606
JEdit	0.179	0.455	0.669	0.116	0.545	0.634	0.457	0.545	0.497	0.249	0.455	0.640
Eclipse	0.149	0.408	0.710	0.095	0.447	0.708	0.211	0.451	0.658	0.085	0.524	0.660
Mean	0.212	0.417	0.676	0.169	0.493	0.642	0.350	0.347	0.645	0.193	0.474	0.644

other two variants are constructed using Alves Rankings threshold values, considering as fault-prone classes having at least 1 or 2 metrics exceeding threshold values. The Friedman test showed a significant difference between the models with a p-value lower than 0.0001. The Nemenyi test showed that the MRL model tested by considering the number of faults in each dataset (duplicated experiment), performed significantly better than the adapted HySOM models. The experiment using non-duplicated datasets gave results that were not significantly different from the results given by the adapted HySOM approaches and the MRL experiment with duplicated datasets. However, according to the Nemenyi test, this MRL experiment performed slightly better than the adapted HySOM models.

Following the results obtained with the MRL model, we can answer positively to RQ1, which was:

RQ1: Can the proposed MRL model outperform existing unsupervised models?

The MRL model outperformed the HySOM model in terms of performance, significantly for the experiment with duplicated datasets and not significantly for the one using binary datasets. However, it also outperformed the HySOM model in terms of processing speed, as the adapted HySOM model has long SOM and ANN training phases to build the model. The MRL model has a very low building time, since it doesn't use any machine learning or clustering algorithm. It simply checks threshold values against source code metrics. Furthermore, the MRL model output is simpler to understand than the one of the HySOM model, since it doesn't come from a machine learning algorithm, which can sometimes give non-consistent results. The MRL model also gives indications about the risk level that a class contains faults. The HySOM model simply gives a binary output indicating if a class is fault-prone or not. Moreover, since the MRL model gives equal or better performance than the HySOM model and offers other advantages over it, we think it should be used instead of the HySOM model.

5.2. Relationship With Faults' Severity

In this section, we present the results obtained from analyzing the relationship between faults' severity and the MRL model's output.

Table 5 presents the results obtained when investigating the ratio of critical faults detected in each level of the MRL model. The number of faults presented in each risk level is cumulative from the previous level. For example, the number of faults in the medium risk level regroups the faults found in medium, high and very high risk levels. The column Critical gives the number of critical faults, the column All gives the number of noncritical and critical faults and the % column gives the percentage of critical faults detected among all faults in these risk levels.

Table 5: Class-Level Fault-Proneness Prediction Considering Faults' Severity.

	KC1 (binary)			KC1 (duplicated)			Eclipse (binary)			Eclipse (duplicated)		
Risk level	**Critical**	**All**	**%**	**Critical**	**All**	**%**	**Critical**	**All**	**%**	**Critical**	**All**	**%**
Very high	2	3	66.67	112	127	88.19	1	4	25.00	8	19	42.11
High	4	7	57.14	154	189	81.48	12	21	57.14	63	89	70.79
Medium	8	18	44.44	181	271	66.79	22	58	37.93	96	173	55.49
Low	10	34	29.41	197	408	48.28	32	112	28.57	111	259	42.86

From looking at the results obtained when comparing the number of critical faults with the total number of faults, it seems that there is a relationship between the risk levels produced and the severity of the faults. For the KC1 dataset, we can remark that the very high and high risk levels mostly detect critical faults, especially when the number of faults is considered. This therefore reinforces our conviction that classes classified in the high risk levels should be tested first and more rigorously, as they have more chances to contain critical faults. For the Eclipse dataset, the very high risk level doesn't contain a critical faults proportion as high as KC1. However, when considering classes contained in the high risk level and above, more than half of them are critical faults. As we go from high risk level to medium and low risk levels, we note that the number of critical faults goes up lower, meaning than most of them are found in the very high and high risk levels. From these results, it seems that there is a correlation between classes classified in higher risk levels and the probability that they contain critical faults.

Additionally, we visually analyzed histograms showing the number of classes per MRL's risk level without faults, with noncritical faults and critical ones. These charts are presented in Figures 2 and 3 for both KC1 and Eclipse datasets considering the number of faults in each object-oriented class (with classes source code metrics duplicated according to the number of faults in each one). The Y axis displays the number of classes contained in the risk level for each category (without faults, with noncritical faults and with critical faults).

Looking at these histograms, we can see that the very high and high risk levels for both datasets find a lot more critical faults than noncritical faults, except for the very high level of the Eclipse dataset, which contains slightly more noncritical faults. In all charts presented, we can see that the lower the risk level is, the lower is the proportion of critical faults detected. These charts also suggest that the risk levels outputted by the MRL model are related to faults' severity.

As an additional test, we performed a Spearman analysis on the original and duplicated datasets. This test analyzes the correlation between the MRL model risk levels and the severity of faults. Table 6 shows the results obtained from these correlation tests, giving p-value, Spearman coefficients and R^2 values.

Table 6: Spearman Correlation Test Results Considering Faults' Severity And MRL Model Risk Level.

	KC1 (binary)	KC1 (duplicated)	Eclipse (binary)	Eclipse (duplicated)
p-value	0	0	0	0
Spearman Coefficient	0.401	0.622	0.488	0.668
R^2	0.161	0.386	0.238	0.446

From looking at the correlation test results, we can first see that according to a 5% confidence level the MRL model output, for both KC1 and Eclipse datasets, is significantly correlated with faults' severity. The table also shows that the correlation is much higher when the duplicated datasets are used, therefore showing that the faults are correctly classified. According to the correlation levels given by Hopkins [49], correlation level is considered medium when the correlation coefficient is between 0.3 and 0.5 and considered high when the correlation coefficient is between 0.5 and 0.7. However, we got a correlation level of medium for KC1 and Eclipse when faults' count is not considered and high for both datasets when faults' count is considered. According to the multiple tests we did, we can conclude that there is a significant relationship between the risk levels outputted by the MRL model and class-level faults' severity.

Following the results obtained comparing the MRL model's output and faults' severity, we can answer

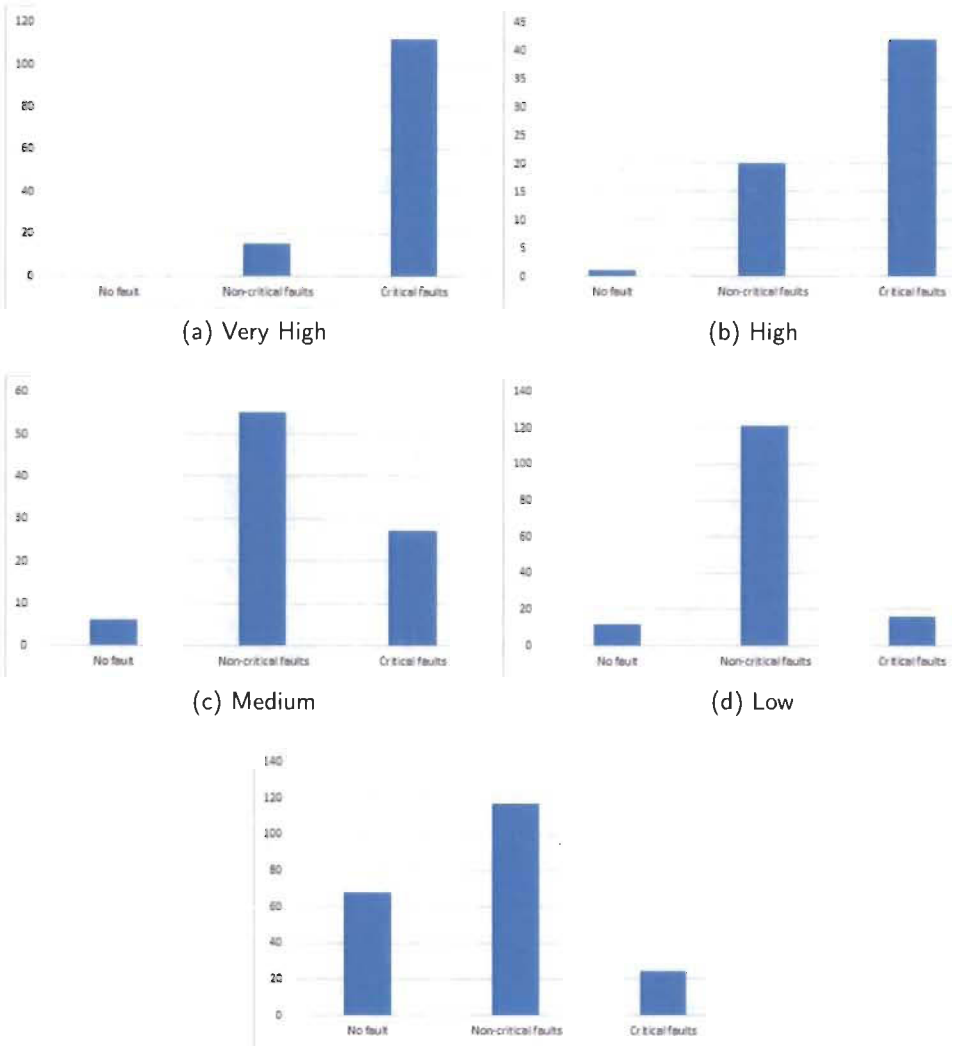


Fig. 2. Severity of Faults Detected in Each MRL's Risk Levels for the KC1 Dataset.

positively to RQ2, which was:

RQ2: Is there a relationship between the risk levels given by the MRL model and faults' severity?

According to the experiments performed with faults' severity, the higher the fault-proneness risk given by the MRL model, the more severe the faults detected are. This gives a certain advantage when using the MRL model, since testing very high risk classes first increases the chances of finding high severity faults early.

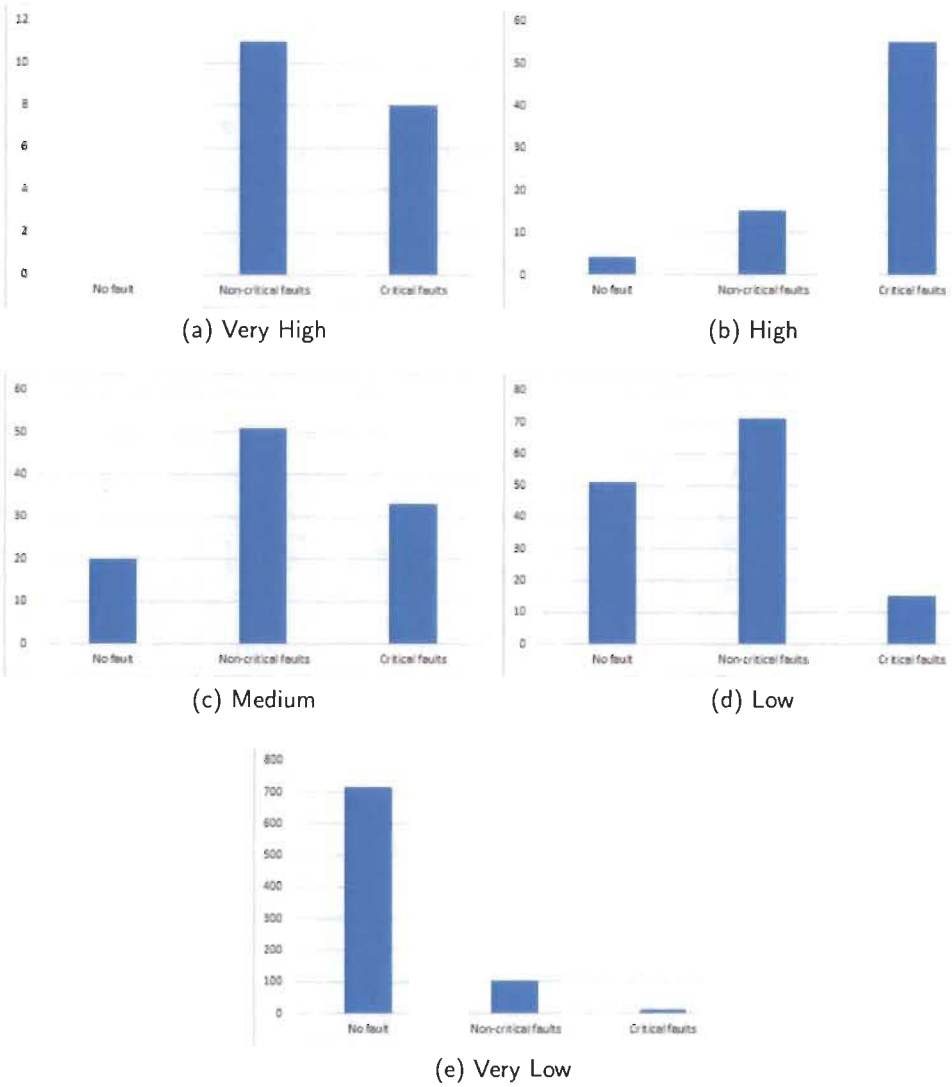


Fig. 3. Severity of Faults Detected in Each MRL's Risk Levels for the Eclipse Dataset.

5.3. Cross-Version Supervised FPP Results

We performed two cross-version supervised experiments for each investigated algorithm, one considering the previous version of a software system and another one considering all of its previous versions. Table 7 presents the results obtained by building the supervised models on the immediate previous version of a system and testing it on the next version of the same system. Table 8 presents the results when the supervised models are built on all previous versions of the system and then tested on the next one. For both tables and for a comparison purpose, we included results of the MRL model for the same datasets. Some results are unavailable with the Bayes Network and ANN algorithms (marked with a hyphen), since we didn't have fault data history for previous versions of these systems. For these experiments, we used the datasets for versions 1.3, 1.4, 1.5, 1.6 and 1.7 of Apache ANT, the datasets for versions 1.1, 1.4 and 2.0 of Apache IVY, the datasets for versions 2.0, 2.5 and 3.0 of Apache IVY, the datasets for versions 2.0, 2.2 and 2.4 of Apache LUCENE and the datasets for versions 3.2, 4.0, 4.1, 4.2 and 4.3 of JEdit. We used

Table 7: FPP Training With Machine Learning Algorithms on the Previous Version of a Software System.

	Bayes Network			ANN			MRL		
Dataset	FPR	FNR	g-mean	FPR	FNR	g-mean	FPR	FNR	g-mean
ANT 1.3	-	-	-	-	-	-	0.343	0.182	0.733
ANT 1.4	0.217	0.532	0.605	0.268	0.468	0.624	0.304	0.404	0.644
ANT 1.5	0.000	1.000	0.000	0.042	0.657	0.573	0.326	0.257	0.708
ANT 1.6	0.058	0.429	0.733	0.027	0.701	0.539	0.205	0.152	0.821
ANT 1.7	0.140	0.237	0.810	0.192	0.207	0.801	0.238	0.172	0.794
IVY 1.1	-	-	-	-	-	-	0.083	0.292	0.806
IVY 1.4	0.338	0.167	0.743	0.516	0.167	0.635	0.209	0.333	0.726
IVY 2.0	0.051	0.500	0.689	0.000	1.000	0.000	0.170	0.286	0.770
POI 1.5	-	-	-	-	-	-	0.219	0.368	0.702
POI 2.0	0.585	0.231	0.565	0.938	0.006	0.249	0.303	0.462	0.613
POI 2.5	0.036	0.883	0.336	0.015	0.919	0.282	0.263	0.466	0.628
POI 3.0	0.416	0.084	0.731	0.739	0.060	0.495	0.106	0.364	0.754
LUCENE 2.0	-	-	-				0.212	0.246	0.771
LUCENE 2.2	0.243	0.292	0.732	0.243	0.353	0.700	0.252	0.324	0.711
LUCENE 2.4	0.277	0.261	0.731	1.000	0.003	0.000	0.175	0.402	0.702
JEdit 3.2	-	-	-	-	-	-	0.110	0.264	0.809
JEdit 4.0	0.173	0.173	0.827	0.251	0.177	0.785	0.113	0.230	0.827
JEdit 4.1	0.103	0.258	0.816	0.112	0.249	0.817	0.103	0.263	0.813
JEdit 4.2	0.197	0.226	0.788	0.191	0.170	0.819	0.197	0.189	0.807
JEdit 4.3	0.195	0.417	0.685	0.067	0.667	0.558	0.285	0.333	0.690
Mean	0.202	0.379	0.653	0.307	0.387	0.525	0.211	0.299	0.741

these datasets as many versions of each one are made available online, easily obtained via the PROMISE Repository [33].

The first conclusion that we can make from the obtained results is that using previous versions of a software system for FPP seems to give good classification results with the Bayes Network algorithm. For the ANN based model, it seems like the prediction performance is a bit lower (with an g-mean average lower than with Bayes Network). We remark that the prediction does not seem better nor worse when all previous versions of the software system are used for training. However, results are more stable and the average g-mean value higher for both supervised algorithms. This can be explained by the fact that using all previous versions for building the FPP model makes use of more learning data and if some learning data is of low quality, it seems that it reduces its impact on the classification. For example, prediction for IVY 2.0 is better when all previous versions are considered (especially for the ANN based model). However, some datasets give better results when only the immediate previous version is used for building the model. This could be explained by the fact that some versions of the software contain data which is not especially good for training the FPP model, making the prediction less accurate. Overall, performance is more stable when all previous versions of the software system are considered, not having bad results like for POI 2.5 when only the immediate previous version is used.

Also, the supervised models were sometimes not able to predict fault-prone code in certain systems. For example, when only the previous version of a software system is used to train the model, the Bayes Network algorithm fails to produce pertinent prediction results for the ANT 1.5 dataset. With a FNR metric of 1 and an FPR metric of 0, this experiment considered all classes as not-fault-prone, which is not helpful. The same kind of issue applies to the ANN based model with IVY 2.0 (when only one previous version is used) and LUCENE 2.4 (when one or all previous versions of a system are used for building the model).

If we compare the results obtained with the supervised FPP models to those obtained using the unsupervised MRL model (using non-duplicated datasets), we remark that the MRL model yields better results (according to the average g-mean). In fact, according to the Friedman test, the results obtained using these

Table 8: FPP Training With Machine Learning Algorithms on All Previous Version of a Software System.

	Bayes Network			ANN			MRL		
Dataset	FPR	FNR	g-mean	FPR	FNR	g-mean	FPR	FNR	g-mean
ANT 1.3	-	-	-	-	-	-	0.343	0.182	0.733
ANT 1.4	0.217	0.532	0.605	0.268	0.468	0.624	0.304	0.404	0.644
ANT 1.5	0.126	0.371	0.741	0.038	0.657	0.574	0.326	0.257	0.708
ANT 1.6	0.077	0.370	0.763	0.027	0.755	0.488	0.205	0.152	0.821
ANT 1.7	0.161	0.254	0.791	0.069	0.385	0.757	0.238	0.172	0.794
IVY 1.1	-	-	-	-	-	-	0.083	0.292	0.806
IVY 1.4	0.338	0.167	0.743	0.516	0.167	0.635	0.209	0.333	0.726
IVY 2.0	0.215	0.304	0.740	0.199	0.214	0.793	0.170	0.286	0.770
POI 1.5	-	-	-	-	-	-	0.219	0.368	0.702
POI 2.0	0.585	0.231	0.565	0.938	0.006	0.249	0.303	0.462	0.613
POI 2.5	0.328	0.395	0.637	0.263	0.442	0.642	0.263	0.466	0.628
POI 3.0	0.149	0.314	0.764	0.540	0.066	0.655	0.106	0.364	0.754
LUCENE 2.0	-	-	-	-	-	-	0.212	0.246	0.771
LUCENE 2.2	0.243	0.292	0.732	0.243	0.353	0.700	0.252	0.324	0.711
LUCENE 2.4	0.270	0.259	0.735	1.000	0.000	0.000	0.175	0.402	0.702
JEdit 3.2	-	-	-	-	-	-	0.110	0.264	0.809
JEdit 4.0	0.173	0.173	0.827	0.251	0.177	0.785	0.113	0.230	0.827
JEdit 4.1	0.155	0.240	0.802	0.348	0.147	0.746	0.103	0.263	0.813
JEdit 4.2	0.197	0.208	0.797	0.439	0.047	0.731	0.197	0.189	0.807
JEdit 4.3	0.237	0.417	0.667	0.252	0.333	0.706	0.285	0.333	0.690
Mean	0.231	0.302	0.727	0.359	0.281	0.606	0.211	0.299	0.741

approaches are significantly different (p-value of 0.015). However, the post-hoc Nemenyi test concluded that the MRL model is not significantly different from the supervised models, but that it still performed better than those.

For the supervised models, we can observe that for some datasets, the prediction is much lower than on others, probably due to bad fault data quality. Quality fault data can be difficult to get for reasons such as high costs, lack of budget, time limitations or even unavailability of experts [17]. However, the unsupervised MRL model doesn't have these concerns and is overall simpler to use and understand, as it doesn't need to collect fault data. More importantly, it provides similar or better performance than the other models investigated.

Following the results obtained with the Bayes Network and ANN based models, we can answer positively to RQ3, which was:

RQ3: Can the MRL model perform similarly or better than supervised FPP models using data from previous versions?

The MRL model performed similarly and sometimes better than the supervised FPP models using previous versions of a software system, at least when Bayes Network and ANN are considered. Furthermore, the results given by the MRL model seem more stable from one dataset to another, as the minimum g-mean value for Bayes Network is 0, for ANN it is 0 too and for the MRL model it is 0.613. Moreover, the MRL model has the advantage to be usable even on the first versions of a software system, when fault data may not be available to train a supervised model.

6. Threats to Validity

Our study contains certain threats to validity like other empirical software engineering studies. First, we investigated several datasets from different systems, and most of them were datasets produced from NASA

or Apache software systems. A larger variety of systems could be considered for investigation to therefore generalize the results obtained in this study. We could therefore consider datasets from different domains, programming languages and environments.

Another threat to validity is that some datasets investigated could calculate the source code metrics differently than others. This could introduce differences in results, but we tried to reduce these differences by making sure the same metrics have the same meanings from one dataset to another and by correcting them if they did not. For example, we recalculated WMC for certain datasets (ANT, IVY, LUCENE, POI, TOMCAT and JEdit datasets), because it was considered as the number of methods and not as the sum of the cyclomatic complexity of the methods as we wanted to use it. Nonetheless, there could be errors in the datasets, either in source code metrics calculation or fault data collection, which is out of our control. However, we tried to reduce that risk by using public and widely used datasets.

Another threat to validity with the datasets we used is that although they were widely used in FPP studies, no data was found on which classes were completely (or partially) tested. This means that classes marked as containing no faults could be classes containing faults, but they were not properly tested and these faults were therefore undetected. However, this problem is common with most (if not all) studies considering FPP. Considering only classes that were completely tested for FPP could therefore impact the results obtained.

Although our results showed that risk levels given by the MRL model are significantly correlated with faults' severity, we think that faults' severity ratings are very subjective and should therefore be taken lightly. A lot of work would be needed on each investigated software system to effectively regroup severity ratings for the FPP models, as severity ratings are often different from one organization to another. Additionally, Ostrand *et al.* [50] considered severity ratings as highly subjective and sometimes inaccurate, because of internal political considerations. They stated that faults' severity was sometimes changed for the developers to work more intensively on certain faults than other ones.

Another threat to validity is the way the Bayes Network and ANN supervised algorithms were used using Weka. The default configuration of these algorithms were used, but some fine-tuning could have been performed, aiming to achieve better classification results.

7. Conclusion and Future Work

In our study, we wanted to investigate the use of an unsupervised FPP model outputting multiple risk levels of fault-proneness, making it more practical for developers and testers. We wanted, in fact, a model that can be used in an iterative software development process, where fault data history can be absent (at the start of a project) or very limited (after the first iterations). To achieve these objectives, we proposed the MRL model.

In a previous study, we tried to alleviate the unsupervised HySOM model performance problem by considering class-level software system data (the original model uses function-level source code metrics), still reusing public datasets [25]. This adaptation gave better results than using the original HySOM model as a function-level FPP model, but it still gave undesirable results for certain datasets. We wanted something with better performance and features, and therefore decided to propose our own model (MRL).

To be able to output multiple risk levels of fault-proneness and give better results, we decided to use source code metrics' threshold values only. The proposed MRL model uses Alves Rankings threshold values (that are calculated without using fault data history) to categorize classes of the system in 5 fault-proneness risk levels. It therefore gives an idea to developers and testers about which classes should be tested in priority and more rigorously. The MRL model is therefore simpler and gives more constant FPP performance results than the HySOM model (RQ1). When considering the number of faults in each class in the classification, we noted that the model performance was improved, correctly classifying additional faults. Another important aspect of the MRL model is that it can easily give information about why a class is fault-prone or not, since it uses simple object-oriented metrics with threshold values. It is also very fast to execute, since it only uses static data such as object-oriented metrics.

With the proposed MRL model giving good performance, we investigated if there was a correlation between classes classified in higher risk levels and the severity of the detected faults (RQ2). Results showed

that higher severity faults were generally contained in classes with higher risk of containing faults. However, only two datasets were investigated in this part since they were the only public datasets we found with usable faults' severity information. Additionally, faults' severity information can be different from one system to another and is highly subjective, making it difficult to generalize these results.

One final test we did to assess the MRL model performance was to compare its FPP results with the ones obtained with two supervised fault-proneness algorithms (Bayes Network and ANN) applied on consecutive versions of the same software system, simulating a real-life development process (RQ3). Surprisingly, the MRL model gave similar and even better results than the supervised learning models. Moreover, the MRL model gave more consistent results, not being affected by the fault data quality given by the previous versions of software systems.

Finally, our study proposes the MRL model as a completely unsupervised FPP model, outputting multiple risk levels of fault-proneness, to better guide developers and testers in the distribution of testing efforts. Our proposed model gave better results than the adapted unsupervised HySOM model for class-level FPP. It also gave results similar or better than supervised learning FPP models. Furthermore, since the MRL model only uses threshold values directly without any training phase, the time for building the model is very low. Moreover, according to tests performed on two datasets, higher fault-proneness risk levels outputted by the proposed approach contain higher severity faults. This is an important aspect to consider for developers and testers prioritizing the implementation of unit tests in their systems.

Future works based on this one could be to test the MRL model on a larger variety of open and closed-source systems, in an attempt to generalize its performance results. We also have plans to implement the proposed MRL model into a usable extension of IntelliJ IDEA or Eclipse. This tool would be very easy to use, making it also easily accessible for any developer or tester, without the need to learn how the model works. Our model could even be adapted to use design metrics only and give software engineering teams early information about which software system parts will be the more fault-prone, by applying the model on UML diagrams.

Acknowledgment

This work was partially supported by NSERC (Natural Sciences and Engineering Research Council of Canada) grant.

References

[1] A. Bertolino, Software Testing Research: Achievements, Challenges, Dreams, Future of Software Engineering (FOSE '07) (2007) 85 -103.

[2] M. Felderer, I. Schieferdecker, A taxonomy of risk-based testing, International Journal on Software Tools for Technology Transfer 16 (2014) 559–568.

[3] C. Catal, U. Sevim, B. Diri, Metrics-Driven Software Quality Prediction Without Prior Fault Data, in: S.-I. Ao, L. Gelman (Eds.), Electronic Engineering and Computing Technology, Lecture Notes in Electrical Engineering, Springer Netherlands, Dordrecht, 2010, pp. 189–199.

[4] M. K. Dhillon, P. B. Singh, P. J. Singh, Metrics Threshold Analysis On the Basis of Clustering Technique for Fault Prediction, International Journal of Science and Research (IJSR) 5 (2016) 158–162.

[5] B. Isong, E. Obeten, A Systematic Review of the Empirical Validation of Object-Oriented Metrics Towards Fault-Proneness Prediction, International Journal of Software Engineering and Knowledge Engineering 23 (2013) 1513–1540.

[6] S. S. Rathore, A. Gupta, Investigating object-oriented design metrics to predict fault-proneness of software modules, 2012 CSI Sixth International Conference on Software Engineering (CONSEG) (2012) 1–10.

[7] F. Jaafar, Y.-G. Gueheneuc, S. Hamel, F. Khomh, Mining the relationship between anti-patterns dependencies and fault-proneness, 2013 20th Working Conference on Reverse Engineering (WCRE) (2013) 351–360.

[8] T. J. Ostrand, E. J. Weyuker, R. M. Bell, Predicting the location and number of faults in large software systems, IEEE Transactions on Software Engineering 31 (2005) 340–355.

[9] T. Gyimothy, R. Ferenc, I. Siket, Empirical validation of object-oriented metrics on open source software for fault prediction, IEEE Transactions on Software Engineering 31 (2005) 897–910.

[10] Yuming Zhou, Hareton Leung, Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults, IEEE Transactions on Software Engineering 32 (2006) 771–789.

[11] R. Malhotra, A. Jain, Fault Prediction Using Statistical and Machine Learning Methods for Improving Software Quality, Journal of Information Processing Systems 8 (2012) 241–262.

[12] K. Dejaeger, T. Verbraken, B. Baesens, Toward Comprehensible Software Fault Prediction Models Using Bayesian Network Classifiers, IEEE Transactions on Software Engineering 39 (2013) 237–257.
[13] A. Kaur, K. Kaur, Performance analysis of ensemble learning for predicting defects in open source software, 2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI) (2014) 219–225.
[14] R. Malhotra, A. J. Bansal, Fault prediction considering threshold effects of object-oriented metrics, Expert Systems 32 (2015) 203–219.
[15] J. Moeyersoms, E. Junqué de Fortuny, K. Dejaeger, B. Baesens, D. Martens, Comprehensible software fault and effort prediction: A data mining approach, Journal of Systems and Software 100 (2015) 80–90.
[16] C. Catal, U. Sevim, B. Diri, Software Fault Prediction of Unlabeled Program Modules, Proceedings of the World Congress on Engineering I (2009) 1–6.
[17] G. Abaei, A. Selamat, H. Fujita, An empirical study based on semi-supervised hybrid self-organizing map for software fault prediction, Knowledge-Based Systems 74 (2014) 28–39.
[18] H. Lu, B. Cukic, M. Culp, Software defect prediction using semi-supervised learning with dimension reduction, Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012 (2012) 314.
[19] Shi Zhong, T. Khoshgoftaar, N. Seliya, Unsupervised learning for expert-based software quality estimation, Eighth IEEE International Symposium on High Assurance Systems Engineering, 2004. Proceedings. (2004) 149–155.
[20] E. Erturk, E. Akcapinar Sezer, Iterative software fault prediction with a hybrid approach, Applied Soft Computing 49 (2016) 1020–1033.
[21] C. Catal, U. Sevim, B. Diri, Clustering and metrics thresholds based software fault prediction of unlabeled program modules, ITNG 2009 - 6th International Conference on Information Technology: New Generations (2009) 199–204.
[22] P. S. Bishnu, V. Bhattacherjee, Software Fault Prediction Using Quad Tree-Based K-Means Clustering Algorithm, IEEE Transactions on Knowledge and Data Engineering 24 (2012) 1146–1150.
[23] G. Abaei, Z. Rezaei, A. Selamat, Fault prediction by utilizing self-organizing map and threshold, 2013 IEEE International Conference on Control System, Computing and Engineering (2013) 465–470.
[24] C. Catal, A Comparison of Semi-Supervised Classification Approaches for Software Defect Prediction, Journal of Intelligent Systems 23 (2014) 75–82.
[25] A. Boucher, M. Badri, Predicting Fault-Prone Classes in Object-Oriented Software: An Adaptation of an Unsupervised Hybrid SOM Algorithm, in: 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), IEEE, 2017, pp. 306–317.
[26] H. Lu, B. Cukic, M. Culp, A Semi-supervised Approach to Software Defect Prediction, 2014 IEEE 38th Annual Computer Software and Applications Conference (2014) 416–425.
[27] A. Boucher, M. Badri, Software Metrics Thresholds Calculation Techniques to Predict Fault-Proneness : An empirical comparison, Submitted to Information and Software Technology, Elsevier, (accepted for publication, in press) (2017) 1–52.
[28] S. Chidamber, C. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering 20 (1994) 476–493.
[29] R. Shatnawi, W. Li, J. Swain, T. Newman, Finding software metrics threshold values using ROC curves, Journal of Software Maintenance and Evolution: Research and Practice 22 (2010) 1–16.
[30] R. Bender, Quantitative Risk Assessment in Epidemiological Studies Investigating Threshold Effects, Biometrical Journal 41 (1999) 305–319.
[31] T. L. Alves, C. Ypma, J. Visser, Deriving metric thresholds from benchmark data, 2010 IEEE International Conference on Software Maintenance (2010) 1–10.
[32] R. Shatnawi, A Quantitative Investigation of the Acceptable Risk Levels of Object-Oriented Metrics in Open-Source Systems, IEEE Transactions on Software Engineering 36 (2010) 216–225.
[33] T. Menzies, R. Krishna, D. Pryor, The Promise Repository of Empirical Software Engineering Data, 2016. URL: `http://openscience.us/repo/`.
[34] L. Yu, Using Negative Binomial Regression Analysis to Predict Software Faults: A Study of Apache Ant, International Journal of Information Technology and Computer Science 4 (2012) 63–70.
[35] M. Jureczko, L. Madeyski, Towards identifying software project clusters with regard to defect prediction, Proceedings of the 6th International Conference on Predictive Models in Software Engineering - PROMISE '10 (2010) 1.
[36] M. Jureczko, Significance of different software metrics in defect prediction, Software Engineering: An International Journal 1 (2011) 86–95.
[37] The Apache Software Foundation, Apache Lucene, 2016. URL: `https://lucene.apache.org/core/`.
[38] M. D'Ambros, M. Lanza, R. Robbes, An extensive comparison of bug prediction approaches, 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010) (2010) 31–41.
[39] The Apache Software Foundation, Apache POI - the Java API for Microsoft Documents, 2016. URL: `https://poi.apache.org/`.
[40] The Apache Software Foundation, Apache Tomcat, 2016. URL: `http://tomcat.apache.org/`.
[41] J. Sayyad Shirabad, T. Menzies, The PROMISE Repository of Software Engineering Databases, 2005. URL: `http://promise.site.uottawa.ca/SERepository`.
[42] T. Mende, R. Koschke, Effort-Aware Defect Prediction Models, 2010 14th European Conference on Software Maintenance and Reengineering (2010) 107–116.
[43] The Eclipse Foundation, JDT Core Component, 2016. URL: `https://eclipse.org/jdt/core/`.
[44] R. Shatnawi, Improving software fault-prediction for imbalanced data, 2012 International Conference on Innovations in Information Technology (IIT) (2012) 54–59.
[45] A. Boucher, M. Badri, Using Software Metrics Thresholds to Predict Fault-Prone Classes in Object-Oriented Software,

Special Session of Software Engineering with Artificial Intelligence, 4th International Conference on Applied Computing & Information Technology (2016) 169–176.
[46] J. Demšar, Statistical Comparisons of Classifiers over Multiple Data Sets, J. Mach. Learn. Res. 7 (2006) 1–30.
[47] S. Singh, K. S. Kahlon, Object oriented software metrics threshold values at quantitative acceptable risk level, Csit 2 (2014) 191–205.
[48] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I. H. Witten, The WEKA data mining software: An Update, ACM SIGKDD Explorations Newsletter 11 (2009) 10.
[49] W. G. Hopkins, New view of statistics, 1997.
[50] T. J. Ostrand, E. J. Weyuker, R. M. Bell, Where the bugs are, ISSTA '04 Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis 29 (2004) 86–96.