UNIVERSITÉ DU QUÉBEC

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

COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN MATHÉMATIQUES ET INFORMATIQUE
APPLIQUÉES

PAR
HERIMANITRA RANAIVOSON

CLASSIFICATION DE LA SÉVÉRITÉ DES BOGUES PAR L'UTILISATION DE MÉTRIQUES TIRÉES DE L'HISTORIQUE GIT

JUIN 2019

Table des matières

Liste des Tables

Liste des Figures

RÉSUMÉ

La résolution des bogues logiciels est une activité coûteuse. Dans ce contexte, la prédiction de l'apparition des bogues en utilisant des méthodes automatisées est préférable pour le gestionnaire de projets TI afin qu'il puisse prioriser les efforts de test sur les classes logicielles les plus critiques.

Dans ce mémoire, nous proposons des algorithmes, des modèles et des métriques indépendantes du code permettant de prédire la sévérité des bogues des classes d'un système orienté objet.

Notre démarche consiste à évaluer, à partir de données collectées sur trois logiciels dont l'historique git est large, différents algorithmes, différentes métriques ainsi que différentes approches dont l'objectif est de proposer des solutions de prédiction de la sévérité des bogues indépendantes du langage de programmation du logiciel analysé.

ABSTRACT

Software testing and bugs recovery are costly and time consuming activities. In this context, bugs prediction using automated tools such as machine learning (ML) models are highly desirable. These tools may allow product managers to prioritize highly risky components that should carefully be tested.

In this paper, we suggest models and metrics to efficiently predict bugs severity among Java classes of three software projects taken from the Github of the Apache Software Foundation. These metrics are derived from GIT histories of software repositories and compared to various traditional metrics often used in the literature.

Our methodology is based on assessment and comparison of different ML models as well as metrics that are best suited to bugs severity classification.

We show that these metrics, derived from GIT histories, are better for bugs severity classification in many aspects.

Remerciements

Je tiens à exprimer ma reconnaissance envers tous ceux qui m'ont soutenu de près ou de loin à l'achèvement de ce mémoire. Elle va droit aussi au corps professoral du département de Mathématiques et Informatique (DMI). Ma reconnaissance s'adresse tout particulièrement à Monsieur Mourad Badri et Madame Linda BADRI qui ont cru en ma capacité et de surcroît ils m'ont soutenu financièrement (de différentes manières) durant ma scolarité. Aux responsables de bourses, je vous serai infiniment reconnaissant pour les décisions prises en ma faveur. Je pense également à Monsieur Daniel St-Yves, qui a mis à ma disposition une machine virtuelle sur laquelle je pouvais travailler en toute quiétude et sérénité. Ma reconnaissance intarissable s'adresse aussi à ma famille, qui me soutient moralement ainsi qu'à tous ceux qui ont travaillé dur pour me permettre de faire mes études au Canada. Enfin, un vive remerciement à Gabriel Doda qui a pris le temps de lire et corriger mon mémoire.

Chapitre 1.

INTRODUCTION

1.1 Introduction

Les fautes (bogues) sont inévitables lorsqu'on développe des logiciels. De plus, elles sont coûteuses en temps et en ressources humaines. Par conséquent, il serait particulièrement intéressant d'avoir un modèle prédictif pour leur détection précoce. Un tel modèle est d'autant plus intéressant s'il est indépendant du langage de programmation utilisé pour développer le logiciel. Ce mémoire propose et évalue deux métriques issues de l'historique Git de plusieurs dépôts publics sélectionnés et par conséquent indépendantes du logiciel étudié afin de prédire la sévérité des bogues dans les classes logicielles. Nous distinguons les sévérités suivantes : Bogues bloquants, Bogues critiques et majeurs, Bogues mineurs et Bogues triviaux. Nous comparons également les variables composées de mots, issues des « commits », par rapport aux différentes métriques de la littérature. Ce découpage de la sévérité des bogues est issu d'un logiciel de gestion de projets TI appelé JIRA, qui est largement utilisé par des compagnies de développement logiciel que ce soit dans le développement de logiciels libres ou d'entreprise.

Nous montrons que la combinaison des 02 métriques que nous avons définies est supérieure en performance aux métriques issues de la littérature (du moins parmi le grand nombre de métriques considérées) pour la prédiction de bogues logiciels. Nos critères d'évaluation sont multiples comme l'AUC, le taux de bonne classification et le « Mean Decrease Gini Score » pour évaluer la performance des modèles d'apprentissage machine. Nous proposons également différents modèles d'apprentissage dans le but de déterminer celui qui est le plus adapté pour prédire la sévérité des bogues logiciels.

Nous proposons, par la suite, l'usage des mots des « commits » pour prédire l'apparition des bogues dans les logiciels. Ces derniers sont les variables les plus efficaces dans cette tâche parmi toutes les métriques que nous considérons dans ce mémoire. De plus, couplés avec un modèle d'apprentissage profond, ils se sont avérés comme les plus performants en termes de taux de bonne classification.

1.2 Problématique

La réalisation d'un projet logiciel dénoué de bogues a toujours été le souhait de toute équipe de développement logiciel. Ce n'est, cependant, pas toujours réaliste compte tenu des différents facteurs qui contribuent à l'apparition des bogues au cours des différentes phases du processus de développement du logiciel. Il est aussi connu que le temps de résolution ainsi que les ressources humaines impliquées dans l'assurance qualité sont onéreux pour une compagnie. Les bogues peuvent également entrainer la perte des clients du fait de leur insatisfaction.

Dans ce contexte, il est souhaitable d'avoir une procédure automatisée qui permet de détecter, le plus tôt possible, la venue ainsi que la sévérité des bogues logiciels. Le gain sera d'autant plus

important si cette procédure est indépendante du logiciel sur lequel elle est appliquée. Ainsi, dans ce mémoire, nous explorons et évaluons des métriques indépendantes du logiciel étudié dans la prédiction de la sévérité des bogues.

1.3 Organisation du mémoire

Le présent document est divisé en 05 parties. La première est une introduction sommaire à notre sujet de recherche.

La deuxième partie traite sommairement de l'état de l'art et de la problématique à laquelle nous nous sommes intéressés. Cette partie traite notamment de l'aperçu des jeux de données sur lesquels les travaux sont effectués, ainsi que la revue de littérature sur le sujet, objet de recherche. Enfin, l'étude traite des questions de recherche qui constituent le fondement de notre travail.

Ensuite, en troisième partie, la problématique de la méthodologie de collecte des données, des méthodes pour épurer et mettre en forme ces données, la définition des métriques et des méthodes d'évaluation de ces métriques sont également traitées. Les détails sur les méthodes d'apprentissage machine utilisées tout au long des différentes expérimentations constituent les points forts de cette partie.

Dans le chapitre 4, les différents résultats de nos expérimentations sont abordés. Le processus de comparaison de la performance des modèles, logiciel par logiciel, ainsi que la comparaison entre les métriques proposées et celles de la littérature est traitée. Enfin, les expérimentations continuent par les différentes comparaisons de la performance des modèles classiques à celle des réseaux de neurones profonds.

Une conclusion terminera le mémoire avec une ouverture sur des réflexions à explorer encore plus et les futurs travaux qui méritent d'être abordés.

Chapitre 2.

État de l'art

2.1 Introduction

Dans ce mémoire, la recherche aborde la classification multi-classes de la sévérité des bogues à partir de métriques indépendantes du source code du logiciel. L'étude porte sur 03 logiciels de la fondation Apache : ActiveMQ et Struts (1 et 2). Dans ce chapitre, il sera état des différentes littératures en lien avec le sujet étudié dans ce mémoire : la prédiction de la sévérité des bogues.

2.2 Revue de littérature

La revue de littérature s'articule autour des travaux liés à la prédiction de bogues utilisant les métriques en lien avec l'historique de versions des logiciels (git). Par la suite, elle fait le survol des grandes familles d'approches pour évaluer les métriques logicielles pour la prédiction de fautes.

L'usage de l'historique git dans la prédiction de bogues logiciels n'est pas tout à fait nouveau dans la littérature. Zhang et al. (2014) ont montré qu'avec certains mots-clés des "commits" on pouvait inférer les types de changements qui s'effectuent sur le code source. En effet, quand l'ingénieur logiciel effectue un changement dans le code, il y laisse une ou plusieurs phrases résumant l'acte de changement : c'est le "commit". Des études récentes ont montré qu'il existe une corrélation forte et significative entre des métadonnées comme la taille des mots des commit et l'apparition des fautes (Barnett et al., 2016).

Trois grandes familles d'approches sont fréquemment rencontrées dans la littérature de la prédiction de bogues logiciels.

La plus utilisée est sans doute l'évaluation des métriques étudiées logiciel par logiciel. Cette première approche consiste à voir une par une la performance des métriques selon différents dépôts logiciels choisis par les auteurs. L'étude récente de Toure, Badri et Lamontagne (2018) met en valeur cette façon de faire.

La deuxième famille d'approche appelée "cross defect prediction" vise à sonder la capacité d'un modèle et d'une métrique à généraliser d'un dépôt logiciel à l'autre. T. Zimmermann et al. (2009) est la référence en la matière. Les auteurs ont conclu qu'il était difficile de prédire la connaissance du bogue appris depuis les données d'un logiciel dans celles d'un autre logiciel.

Pour pallier à ce problème, il existe une troisième famille d'approches consistant à transformer les variables à prédire afin d'uniformiser leur distribution dans tous les logiciels. En effet, Zhang et al. (2014) ont voulu corriger la distribution statistique de la variable d'intérêt afin d'améliorer la prédiction d'un logiciel à un autre.

Notre travail est lié aux travaux de S. Kim et al. (2007) qui étaient les premiers à définir les caractéristiques d'une entité logicielle susceptible de contenir des bogues dans le futur. Pour la présente étude, la pertinence de leurs travaux est liée au fait qu'ils ont pu formuler des situations typiques qui amènent le développeur à commettre des erreurs. Ces caractéristiques sont les suivantes :

- « Changed entity locality » qui soutient que si une entité a connu une modification récente, alors elle sera susceptible de contenir des fautes bientôt.
- « New locality entity » : si une entité vient d'être créée récemment, alors elle est susceptible de contenir des fautes bientôt.
- «Temporal locality » : si une entité a introduit une faute récemment, alors elle est susceptible de contenir des fautes bientôt.
- « Spatial locality » : si une entité a connu des fautes récemment, alors les entités qui lui sont couplées seront susceptibles de contenir des fautes bientôt.

Ces caractéristiques sont utilisées pour formuler des métriques capables de prendre en compte simultanément tous ces aspects. Notre métrique est dérivée d'une fonction Cobb-Douglas (Paul Douglas (1928)) qui pondère l'apport en effort de test (que l'on définira par la suite) et le temps écoulé depuis le dernier changement ou création dans la probabilité de produire des bogues.

Les métriques sont construites à partir de 03 jeux de données logicielles : ActiveMQ et Apache Struts I et II. Le choix de ces logiciels a été motivé par la présence de données conséquentes sur le Web. Notre métrique se met à jour lorsqu'un changement ou création est réalisé au niveau d'une entité (ici, l'entité que nous nous sommes fixée est la classe).

L'approche ainsi adoptée est aussi motivée par celle de Barnet et al. (2016) qui ont établi des corrélations entre la taille des phrases des « commits » et la probabilité qu'un logiciel contienne des bogues.

Elle est aussi motivée par le papier précurseur de Zhang et al. (2014), qui sont les premiers à avoir émis l'hypothèse selon laquelle il y aurait un lien entre la description textuelle des « commits » et les raisons du changement : adaptive, corrective et autres.

Zhou et Leung (2006) ont établi la classification basée sur la sévérité des bogues en utilisant une dichotomie binaire : « Severe » - « Not Severe ».

G. Sharma, Sharma et Gujra (2015) ont élaboré un dictionnaire de termes critiques pour la prédiction de la sévérité des bogues et ont utilisé plusieurs modèles comme les *k*-plus proches voisins, la classification multi-classes de Bayes. Ils ont également utilisé des métriques d'information pour évaluer la qualité de leurs modèles.

Chapitre 3.

Méthodologie de la recherche

Dans ce chapitre, l'étude entame les détails des questions de recherche, objet de la problématique de recherche. Ensuite, une présentation globale de notre méthodologie suite à nos hypothèses est exposée. En troisième lieu, la construction du jeu de données est abordée suivie de la méthodologie d'évaluation des métriques. Dans la même ligne d'idées, une étude approfondie de chacun des modèles développés dans ce mémoire est effectuée.

3.1 Questions de recherche

Notre sujet de recherche aborde la classification de la sévérité des bogues logiciels. Les questions de recherche suivantes sont posées :

RQ1 : Peut-on construire des métriques indépendantes du code source (agnostique) du logiciel traité avec des performances (en termes de prédiction de fautes) comparables aux métriques traditionnelles ?

Rappelons qu'une métrique est agnostique au logiciel étudié si sa construction est indépendante du langage de programmation dans lequel le logiciel est écrit (voir, par exemple, Conejero et al. (2009)).

RQ2 : Est-ce qu'il existe une ou plusieurs métriques agnostiques aux logiciels étudiés et qui peuvent atteindre des niveaux de performance au moins égales aux métriques usuelles ?

Par métriques usuelles, nous entendons les métriques orientées objet, les métriques de complexité ainsi que les métriques de graphes qui ont été largement utilisées dans la littérature (McCabe, 1976).

Les métriques usuelles dépendent fortement des logiciels pour lesquels elles ont été construites puisqu'elles sont extraites du code source. A l'inverse, les métriques que nous proposons sur la base de l'historique de versions GIT ne dépendent pas du tout du langage logiciel étudié. Ces métriques ne touchent pas aux caractéristiques du langage de programmation du logiciel étudié (ex: nombre de classes publiques, nombre de "try..catch", etc.), mais concernent surtout les métadonnées et les explications textuelles du changement.

En effet, dans les applications qui suivent, des patrons bien établis quand le développeur logiciel réalise une modification du code source, il l'exprime par l'intermédiaire de phrases résumant ce qu'il a fait ou résolu à cet instant précis. Cette façon de faire permet de lier des classes logicielles à des descriptions textuelles qui servent de proxy à leurs états à tout moment des modifications.

Après avoir défini et validé des métriques agnostiques aux langages des logiciels, la question suivante est posée :

RQ3 : Est-ce que les métriques indépendantes du code source sont meilleures dans la prédiction de la sévérité des bogues logiciels comparativement aux métriques usuelles ?

Cette question semble tout à fait légitime car nous sommes tentés de penser que les métriques qui ne touchent pas directement au code source à partir duquel les bogues surviennent ne sont pas aussi efficaces que les métriques directement dépendantes du code source comme les métriques orientées objet, les métriques de graphes, etc.

Différentes façons d'évaluer les métriques se retrouvent dans la littérature. Celle de He et al. (2012) revoit l'entrainement d'un ensemble de différents modèles, logiciel par logiciel, puis les évaluent sur des logiciels complètement différents afin de voir si les modèles se généralisent correctement. D'autres études, comme celle de Zhang et al. (2014) sur les entrainements des modèles sur les données de dépôts logiciels, tentent d'ajuster statistiquement la variable indépendante pour avoir la même distribution dans tous les projets.

Une approche que nous essaierons dans cette étude est la combinaison de tous les logiciels en un seul jeu de données afin que les modèles puissent apprendre des différentes distributions de la sévérité des bogues. Cette approche permettrait aussi de prendre en compte la diversité au sein même des classes logicielles. Bien sûr, de prime abord, l'usage des modèles logiciel par logiciel est testé.

RQ4 : Est-ce qu'un modèle basé sur l'apprentissage profond est meilleur dans la prédiction de la sévérité des bogues ?

Les développements récents relatifs au sujet ont montré que les modèles de Deep Learning (apprentissage profond) sont très efficaces à la fois dans la classification d'images que celle des textes. Comme étudié par Kim et al. (2014), une simple architecture d'apprentissage profond permet d'atteindre l'état de l'art dans la classification de textes. Nous voulions tester l'architecture de Deep Learning utilisée par ces auteurs dans le contexte des mots des "commits" pour la classification de la sévérité des bogues logiciels.

Pour pouvoir utiliser des phrases dans les modèles d'apprentissage machine (ML), un prétraitement pour les transformer en une représentation matricielle s'avère nécessaire. Il consiste à utiliser des techniques de traitement des langues naturelles qui comprennent, entre autres, la standardisation des mots, le TF-IDF, l'encodage binaire. Des approches similaires sont trouvées dans Manning, Raghavan et Schutze (2008).

3.2 Présentation globale

Nous construisons, évaluons puis comparons principalement deux métriques candidates issues de l'historique de versions Git contre 78 métriques de code source largement étudiées dans la littérature du génie logiciel empirique. Ces 78 métriques utilisées en guise de comparaison sont retrouvées dans Vasa (2018), une thèse de doctorat sur l'évolution des métriques orientées objet.

Pour effectuer nos expérimentations, plusieurs sources de données sont jointes : Git, celles compilées par Vasa (2018), ainsi que les historiques de bogues sur JIRA.

Une fois les données compilées, deux grands ensembles de métriques issues de l'historique git sont définis, à savoir :

- Les métriques de changement de code liées aux « commits » : il s'agit du « Ranking Index » et du « Bug Index ».
- Les métriques extraites des phrases de « commits » qui ne sont pas des métriques, à vrai dire, mais un ensemble de mots qui résument au mieux un « commit ».

Enfin, un processus de comparaison des métriques non seulement entre elles mais aussi face aux différents logiciels utilisés est effectuée. Les comparaisons suivantes apparaissent :

- Comparaison des métriques en utilisant un logiciel à la fois
- Comparaison des métriques en utilisant la combinaison de tous les logiciels :
  Dans cette approche multi logiciels, nous effectuons une comparaison de nos deux métriques versus toutes les métriques combinées.
- Nos deux métriques versus deux métriques prises au hasard parmi les 78. Cette technique est utilisée pour comparer deux à deux la capacité de nos deux métriques à toute paire de combinaisons de métriques liées au code source.
- Comparaison selon différentes catégories de sortie de versions.

3.3 Présentation et construction du jeu de données

Trois logiciels de la fondation Apache ont fait l'objet de cette recherche. Il s'agit d'ActiveMQ et d'Apache Struts I et II. Comme la plupart des logiciels de la fondation Apache, ils sont hébergés dans GitHub, avec des lignes de commandes git de récupération, tout en récupérant les historiques des « commits » depuis leur début. Les prétraitements du jeu de données sont réalisés avec le langage de programmation Python. ActiveMQ contenait 591 classes Java pour la période allant du 12 décembre 2005 au 05 mai 2017, tandis que les classes d'Apache Struts I et II étaient au nombre de 378 sur la période du 23 mars 2006 au 04 novembre 2017. Ces classes représentes plus de 55000 observations dont 51000 proviennent d'ActiveMQ. Certains types de bogues sont également sous-représentés comme les bogues bloquants (au nombre de 768).

Par la suite, les 78 métriques citées ci-dessus sont obtenues (collectées) dans plusieurs jeux de données compilés par Vasa (2018) dans sa thèse sur l'évolution logicielle. Des métriques classiques de complexité, des métriques de graphes et bien entendu plusieurs métriques de

code source orientées objet s'y trouvent notamment. Ces jeux de données sont à notre connaissance les plus complets en ce qui concerne les métriques des logiciels de la fondation Apache. Ils peuvent être téléchargés à l'adresse.[1]

De plus, ces données sont choisies pour faciliter l'évaluation des métriques construites en comparaison avec celles existantes sans avoir besoin de les recréer à chaque fois. La Table 1 de l'annexe contient la liste complète de ces métriques compilées par Vasa (2018).

Enfin, le téléchargement des « issues » rapportés sur le site Web de gestion de tickets a été effectué : JIRA concernant nos 03 logiciels : Struts I et II, ActiveMQ. Ces données contiennent les informations sur la sévérité des bogues contenues dans la phase du prétraitement pour obtenir 05 classes (+1 pour toutes autres non liées aux bogues):

1. Bogues bloquants
2. Bogues majeurs et critiques (que nous avons regroupées)
3. Bogues mineurs
4. Bogues triviaux
5. Et Autres : tous ceux qui ne sont pas des bogues.

En résumé, 03 types de données concernant 03 logiciels ont été combinés pour pouvoir mener à terme nos expérimentations :

1. L'historique git de ces 03 logiciels extraits et mis en forme à l'aide de programmation informatique.
2. Les métriques traditionnelles : orientées objet, complexité, graphes compilées par Vasa (2018) et qui concernent les 03 logiciels.
3. Enfin, les données de bogues et sévérité qui ont été téléchargées manuellement depuis le site Web JIRA de la fondation Apache.

La combinaison de ces 03 types de données est la résultante d'une jointure utilisant plusieurs clés comme : le nom du logiciel, le nom de la classe, la date de changement, l'identifiant du ticket sur JIRA et bien d'autres. Cette jointure est aussi faite par programmation (en utilisant Python) après la mise en forme des jeux de données : apurement des noms de classes logiciels, alignement des dates de "commits", des dates d'ouverture de ticket sur JIRA ainsi que les dates où les métriques traditionnelles ont été calculées.

3.4 Méthode d'évaluation des métriques

Toutes les métriques utilisées dans la comparaison sont classées par ordre d'importance dans la prédiction selon différents algorithmes comme le « Mean Decrease Gini ». La performance des modèles utilisant les métriques a été aussi observée. Ces performances sont mesurées à partir du taux de bonne classification (« Accuracy »), mais aussi l'AUC (Area Under Curve) ou courbe ROC dans certaines comparaisons. Le premier mesure le pourcentage des observations

[1] http://www.ict.swin.edu.au/research/projects/helix/download.html

correctement classifiées par un modèle donné, tandis que le second résume la performance d'un modèle à classification binaire (« one vs all classification »).

3.5 Présentation des métriques

Dans cette partie, nous présentons les métriques orientées objet fréquemment utilisées dans la littérature, mais aussi les métriques construites et avec lesquelles les résultats de nos différentes expérimentations ont fait l'objet de comparaison.

La Table I (disponible en annexe) dresse une liste des métriques que Vasa (2018) a compilées dans sa thèse de doctorat sur l'évolution logicielle. Il s'agit de plusieurs métriques liées aux langages orientés objet (exemple : "Depth in Inheritance Tree"), des métriques de graphes (exemple : "Clustering Coeff Graph measure") et même des métriques de processus plus proches des métadonnées (exemple : "Age in days").

Ce dictionnaire des données est retrouvé au lien[2]. Les métriques définies peuvent être subdivisées en 02 catégories :

- M1 : Les métriques basées sur le changement d'état du code. Il s'agit de classer les classes logicielles par ordre de risque à chaque fois que celles–ci sont modifiées par le développeur.
- M2 : Les métriques textuelles basées sur les mots des « commits ».

Dans M1 : métriques basées sur le changement d'état du code, deux (02) métriques ainsi définies qui, à elles seules, résument les caractéristiques d'une entité à risque de bogues selon S. Kim et al. (2007) :

- P1 : « Changed entity locality » qui soutient que si une entité a connu une modification récente, alors elle sera susceptible de contenir des bogues bientôt.
- P2 : « New locality entity » : si une entité vient d'être créée récemment, alors elle est susceptible de contenir des bogues bientôt.
- P3 : « Spatial locality » : si une entité a connu des bogues récemment, alors les entités qui lui sont couplées seront susceptibles de contenir des bogues bientôt.

Pour se faire et après avoir effectué différentes tentatives (essais empiriques), l'option d'utiliser une fonction de type Cobb-Douglas s'est avérée pertinente:

$$R_{ct} = \delta T_{ct}^{\alpha} \delta L_{c't}^{1-\alpha}$$

δT_{ct} est le temps écoulé depuis le dernier changement ou création (si nouvellement créée) de la classe *c* à l'instant *t*. En effet, si δT_{ct} de classe *c* augmente au moment *t*, alors elle a moins de risque de contenir des fautes dans le futur. Ce qui se traduit par l'augmentation de la quantité R_{ct} qui est son rang. $\delta L_{c't}$ est la variation en lignes de code de la classe de test *c'* associée à *c*

[2] http://www.ict.swin.edu.au/research/projects/helix/metric-data-format.html .

(principalement des tests unitaires). C'est une variable qui pourrait potentiellement mesurer l'effort de test et que nous introduisons dans le rang car les tests logiciels jouent un rôle primordial dans la détection des erreurs de développement logiciel. Ainsi, on fait l'hypothèse que plus cette quantité est élevée, plus la classe à moins de risque de contenir des bogues dans le futur du fait des efforts de test qui sont déployés sur elle.

En résumé, un rang R_{ct} plus élevé se traduit par une classe qui n'a pas été modifiée ou touchée depuis longtemps, donc un risque moins élevé de contenir des bogues et inversement. Ainsi, R_{ct} tient compte des propriétés P1 et P2 mentionnées par S. Kim et al. (2007).

Avant de parler du deuxième membre de R_{ct}, une deuxième métrique prise à part est introduite afin de tenir compte de P3. C'est l'indice de bogues ou « Bug Index » qui n'est autre qu'une variable indicatrice (1 ou 0) selon que la classe a connu des bogues antérieurs à l'instant t.

Le rang R_{ct} est une de nos métriques qui sera comparée aux autres métriques de la littérature. Il est très facile à interpréter et permet de savoir la contribution du changement de code et de l'effort de test dans l'apparition de fautes. Il est capté par le coefficient α où α et est un paramètre à déterminer empiriquement. Il vérifie, par ailleurs, les propriétés P1 et P2 comme l'a constaté par S. Kim et al. (2007). Il est associé à une variable « Bug Index », qui sera également notre deuxième métrique dans l'objectif de compléter la propriété *P3*. Enfin, il contient un proxy de l'effort de test qui est aussi un élément primordial qui prévient les éventuelles fautes dans le développement logiciel.

Pour M2, un ensemble de métriques formées par les mots des "commits" est considéré pour la prédiction des bogues logiciels :

1. Bogues bloquants
2. Bogues majeurs et critiques (qu'on a regroupées)
3. Bogues mineurs
4. Bogues triviaux
5. Et Autres : tous ceux qui ne sont pas des bogues.

Ces variables sont les mots des "commits", c'est-à-dire, des mots jugés pertinents par les techniques de sélection automatique à l'instar de l'ACP[3]. En effet, au cours de nos expériences, ces mots, une fois prétraités et mis dans un modèle adéquat, peuvent être des variables précieuses dans la prédiction de la sévérité des bogues. La Figure 1 qui suit est le nuage de mots des mots des "commits", c'est-à-dire, un résumé de l'ensemble des mots les plus populaires.

[3] Analyse en Composante Principale

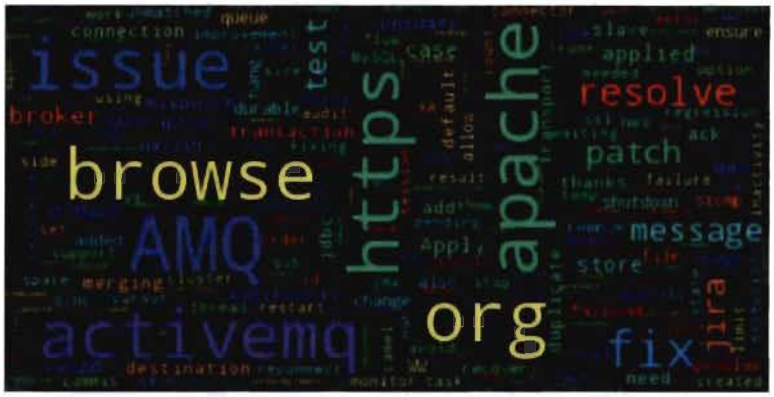


Figure 1. Nuages des mots des « commits » réalisés par l'auteur sur le langage Python[4].

3.5 Techniques de traitement des langues naturelles

Comme nous utilisons beaucoup les techniques de traitement des langues naturelles, les exposer au lecteur s'avère pertinent. En effet, les modèles ne savent pas composer avec les phrases des « commits » directement, encore moins avec les mots. Ainsi, l'idée est de subdiviser la phrase en mots puis par la suite transformer les mots en variables numériques qui puissent être données aux modèles.

Une des manières la plus simple pour représenter un mot en variable est l'usage d'une indicatrice 1 ou 0 si la ligne qui contient le commit correspondant contient le mot en question. On verra plus tard qu'il existe une variante bien plus intéressante de cette méthode qui permettra de pondérer les mots.

Néanmoins, tous les mots n'apportent pas le même degré d'information au modèle d'apprentissage. En effet, il existe des mots très communs du langage étudié (ici, les commits en anglais) et qui ne nous renseignent en rien sur l'information contenue dans le commit. Exemple : « the », « is », « are » ne distinguent pas les phrases et sont des groupes de mots très communs de la langue anglaise. En général, ce ne sont que des articles, des participes passés, présents, etc. C'est la raison pour laquelle, en premier lieu, les différentes techniques d'extraction de mots clés à partir de phrases à l'instar des lemmatisations, du « stemming » ou racinisation des mots vont être étudiées.

3.5.1 Lemmatisation et "Stemming" (ou racinisation)

La lemmatisation est une technique d'analyse lexicale des mots dont le but est d'extraire le mot commun (lemme) à tous les mots provenant d'une même famille. En effet, un mot (lemme) peut être en masculin, singulier, pluriel, en mode impératif, indicatif, etc. Exemple : fille, fillette, filleule ont pour forme canonique le mot (lemme) : fille.

[4] https://www.python.org/

L'algorithme de lemmatisation implémenté dans la librairie NLTK[5] de python a permis d'extraire chaque lemme correspondant à chaque mot des commits et ainsi de réduire le nombre de mots de départ.

Par la suite, la technique du « stemming » ou racinisation est appliquée aux mots. Comme son nom l'indique, cette technique consiste à extraire la racine des mots. Contrairement à la lemmatisation, cette racine peut n'avoir aucun lien avec la grammaire de la langue étudiée. Par exemple, la racine du mot « chercher » après application du "stemming" sera « cherch ». Cette technique connait plusieurs avantages dans l'indexation des mots pour la recherche de mots dans les moteurs de recherche.

Cette deuxième technique permet également de réduire le corpus de mots qui est au départ extrait des phrases des "commits", et par conséquent permet d'éliminer des signaux non pertinents.

3.5.2 TF-IDF: Term Frequency, Inverse Document Frequency

L'encodage binaire des mots, exposé ci-dessus, est un moyen simple et rapide pour transformer les mots d'une phrase d'une ligne d'un jeu de données en une variable prête pour la prédiction. Elle n'est, cependant, pas sans limites. En effet, admettons deux articles où le mot « indépendance » revient 05 fois sur le premier et une fois sur le deuxième. Le premier est sans doute un article portant sur le thème de l'indépendance d'un pays, d'une société tandis que le deuxième ne peut pas vraiment être déterminé sans information supplémentaire.

Dans une représentation avec encodage binaire, les deux articles auraient eu la même pondération pour la variable « indépendance ». Il est donc nécessaire de prendre en considération l'importance « locale » d'un mot dans la phrase. Dans la pondération TF-IDF, elle est prise en compte par son numérateur qui représente la fréquence du terme dans la phrase « Term-Frequency » relativement à tous les mots du corpus et qui s'exprime comme suit :

$$t_i = \frac{m_i}{\sum_k^{|D|} m_k}$$

Où m_i est l'occurrence du mot *i* dans la phrase considérée, et le dénominateur est l'occurrence totale du mot *i* dans tous les documents avec $|D|$ le cardinal du corpus.

Un autre aspect à tenir en compte est la fréquence des mots dans tous les corpus. Si le mot *i* est fréquent dans tous les documents i.e. toutes les phrases, alors, il ne caractérise pas beaucoup un document en particulier. Ainsi, pour minimiser la pondération de ces mots, le TF-IDF introduit l'« Inverse Document Frequency » (IDF). Elle s'écrit comme suit :

$$\log \frac{(|D|)}{(|\{d_k, i \in d_k\}|)}$$

[5] https://www.nltk.org/

Où le dénominateur représente le cardinal de documents (ou phrases) contenant le mot *i*. Ainsi, plus le mot i est contenu dans plusieurs documents, plus cette quantité sera faible (car on lui accorde moins d'importance).

Ainsi, la formule de pondération du TF-IDF revient à :

$$\mathrm{TFIDF_i} = \log \frac{(|D|)}{(|\{d_k, m_i \in d_k\}|)} t_i$$

Cette formule va permettre de pondérer individuellement les mots présélectionnés selon qu'ils se trouvent dans telle ou telle phrase des « commits ».

3.6 Modèles et techniques utilisés lors des expérimentations

Dans cette partie, les modèles et techniques statistiques utilisés lors de nos différentes expérimentations feront l'objet de présentation.

Dans l'approche utilisée, différentes combinaisons de modèles et de métriques sont employées en vue de comparer les métriques entre elles, mais aussi, pour déterminer un modèle qui, potentiellement, s'associe bien avec les métriques considérées. Rappelons que deux grands ensembles de métriques sont à comparer :

- Les métriques définies à partir de l'historique de versions GIT
- Les métriques sélectionnées à partir de l'étude de Vasa (2018) sur l'évolution des métriques logicielles

Le premier ensemble est tiré des métriques définies tout le long de cette recherche à travers l'historique git des logiciels étudiés. Il s'agit des métriques de rang et de bogues ("Ranking Index" et "Bug Index") qui permettent de classer les classes logicielles par ordre de priorité de l'apparition des bogues, mais aussi un ensemble de mots des « commits » traités à partir d'algorithmes de traitement des langues naturelles.

Le second, quant à lui, est constitué d'une sélection de métriques fréquemment utilisées dans la littérature du génie logiciel empirique. Il s'agit des familles de métriques orientées objet, de complexité et celles de graphes.

A cet égard, des algorithmes classiques de Machine Learning sont utilisés : le "Random Forest" l'algorithme Naïf de Bayes, un Perceptron Multicouches, une machine à vecteur de support ainsi que des modèles plus récents comme le réseau de neurones de convolution.

Mais, avant de présenter les résultats de ces différents modèles, une présentation successive de chaque modèle est nécessaire. C'est l'objet de l'étude des sections qui vont suivre.

3.6.1 Les arbres de décision, Random Forest

Le Random Forest est un algorithme d'apprentissage machine basé sur les arbres de décision. Un arbre de décision est un algorithme basé sur des nœuds de décision à partir desquels, la réponse à une tache d'apprentissage s'affine de plus en plus lorsqu'on avance dans la construction des nœuds enfants.

Par exemple, dans une tache où l'on doit classifier si une personne sera atteinte du cancer du poumon par un « Oui » ou un « Non ». Dans ce cas, les différents nœuds seront les différents antécédents de l'individu :

- Nœud 1 : Si l'individu fume
- Nœud 2 : Si l'individu a fumé par le passé
- Nœud 3 : Si l'entourage de la personne fume aussi.
- ...

Différents nœuds peuvent être imbriqués ou disjoints selon les réponses, et à chaque réponse une probabilité se dessine de plus en plus en fonction de nos connaissances de l'historique de patients similaires.

Par exemple, dans un modèle d'arbre décisionnel de prédiction de température, les résultats de prédictions peuvent être illustrés à la manière de la Figure 2 suivante :

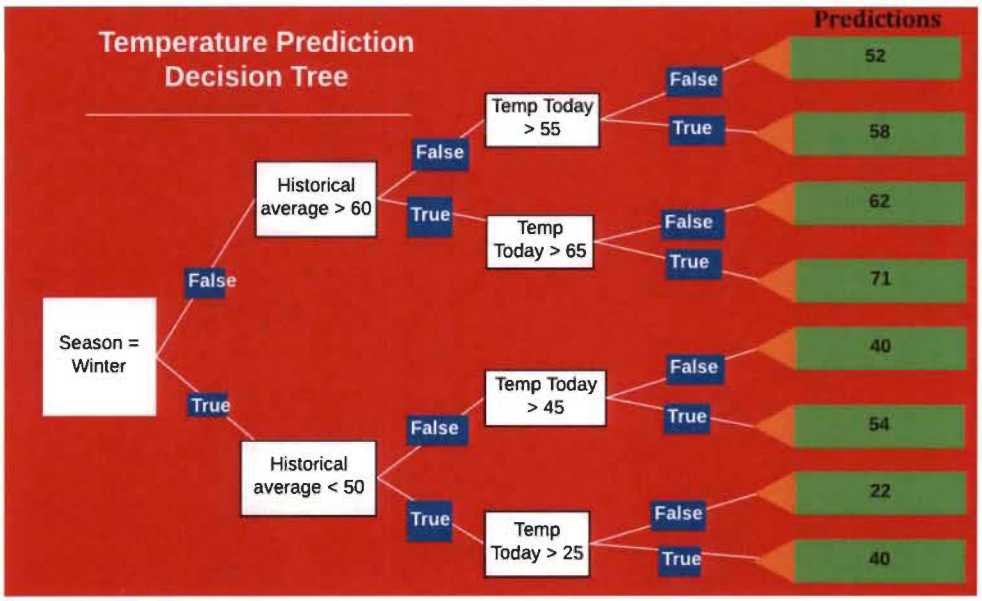


Figure 2 Illustration d'un arbre de décision, William Koerhsen, Medium[6].

Le processus de construction de l'arbre décisionnel peut découler de plusieurs questions posées. Les différentes réponses sont sauvegardées pour constituer un historique de base de données conséquentes qui permettent de faire ressortir des "pattern" invisibles au sens commun.

[6] https://medium.com/@williamkoehrsen/random-forest-simple-explanation-377895a60d2d

Ces différentes questions peuvent être :

- Est-ce qu'on est en hiver ?
- Quelle est la moyenne historique ?
- Quelle est la température du jour ?

C'est ainsi que des arbres de décision à partir des données se forment.

L'entrainement du modèle consiste à apprendre les relations entre les variables (les nœuds, ici) et les températures effectivement enregistrées et qu'on souhaite prédire pour une prochaine fois. L'entrainement du modèle conduit a la détermination de la meilleure question (séparation) à soumettre pour obtenir la prédiction la plus précise.

Le Random Forest comme une agrégation de plusieurs arbres décisions:

Le Random Forest utilise la combinaison de plusieurs arbres de décision (paramètre fixé par le Scientifique des données) sur différents échantillons aléatoires de son jeu de données. En effet, un arbre de décision peut ne pas être assez précis dans sa prédiction du fait qu'il est trop spécialisé dans la particularité de l'exemple pris par la personne (le Scientifique des données), ou trop "extrême" comparé à ce qui est d'habitude observé. Cependant, combinés ensemble, les arbres de décision deviennent de plus en plus précis car ils vont tendre vers la vraie valeur.

3.6.2 Le réseaux de neurones perceptron

Un réseau de neurones est une collection de neurones (par analogie au cerveau humain) connectés par un graphe acyclique.

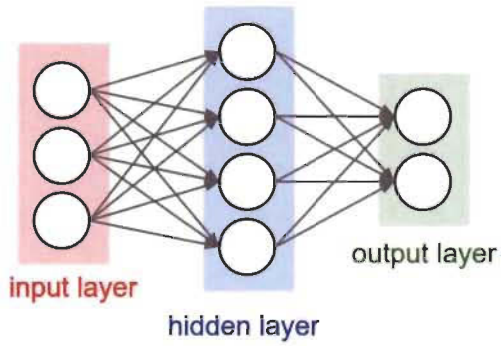


Figure 3 Réseau de neurones à une couche de 04 neurones

Un réseau de neurones est organisé en une ou plusieurs couches cachées. Le type de couche, le plus communément rencontré, est la couche pleinement connectée (« fully-connected layer »). Cela signifie tout simplement que tous les neurones de deux couches adjacentes sont connectés entre eux.

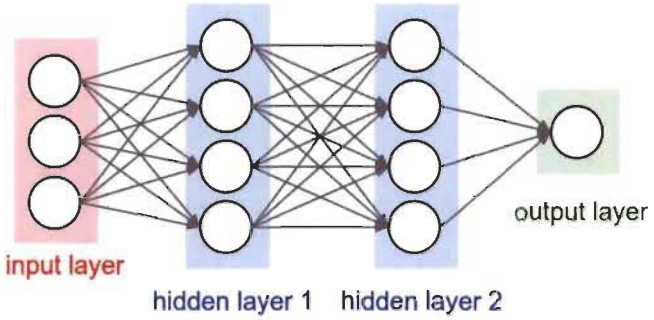


Figure 4 cs231, Stanford, Réseau de neurones à deux couches de 04 neurones chacune, pleinement connectée.

Dans ce modèle, une matrice de paramètres W est apprise par entraînement à partir des données. Chaque élément de cette matrice correspond à la "force" de connexion entre 02 neurones (matérialisé par les liens de connexion du graphique ci-haut).

Les neurones sont responsables d'activer leur sortie $w^T x + b$ (où x représente le vecteur d'entrée et b, d'activation le vecteur des constantes d'initialisation) par une fonction appelée fonction d'activation. Une des plus connues est la fonction sigmoïde définie de la manière suivante :

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Cette fonction est définie sur R et à valeur dans l'intervalle entre [0, 1].

Ainsi, à chaque passage d'une couche à l'autre, un vecteur de fonction d'activation est calculé ceci dans le but de normaliser les valeurs des neurones des couches intermédiaires et ainsi d'éviter les problèmes de convergence.

Le choix du nombre de neurones et de couches dépendra de la complexité du phénomène étudié. Par exemple, comme montrée par la figure ci-après, un réseau de neurones multicouches (aussi connu sous le nom de perceptron multicouches) peut très vite sur-ajuster les données lorsque l'exemple est relativement facile et que le nombre de neurones est élevé.

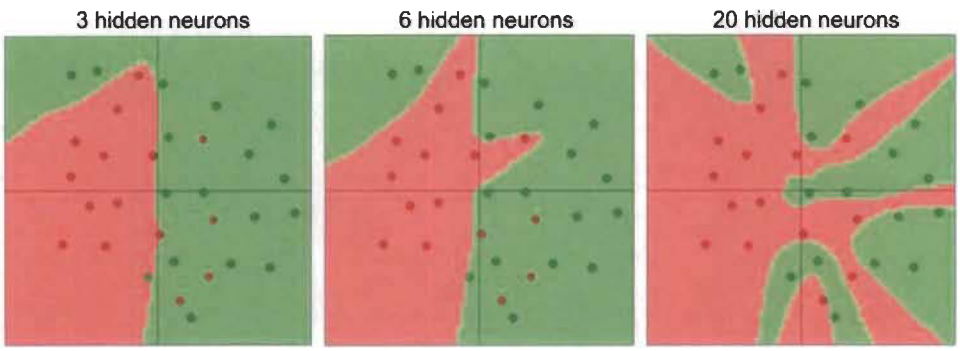


Figure 5 Surface de classification selon le nombre de neurones, cs231n, Stanford.

En effet, le sur-ajustement ou 'overfitting' confère au modèle une très forte précision sur l'échantillon de données sur lequel il apprend, mais une faible capacité de généralisation (prédiction) sur de nouveaux exemples qu'il n'a jamais vus.

Dans le modèle des réseaux de neurones, le "dropout" pour prévenir ce sur-ajustement est le plus souvent utilisé.

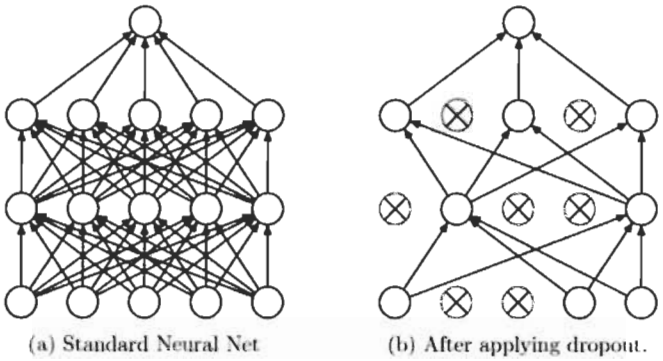


Figure 6 Illustration du dropout en réseau de neurones, cs231n, Stanford.

Comme le montre la Figure 6 ci-dessus, le "dropout" consiste à supprimer de façon aléatoire les connections entre les neurones du réseau de sorte qu'il y ait moins de paramètres appris en même temps et donc évite le sur-ajustement. Il a aussi pour effet de diminuer le temps d'apprentissage du modèle.

3.6.3 L'analyse en Composantes Principales pour la réduction de dimension

L'analyse en Composante Principale (ACP) est une technique qui permet de sélectionner les variables les plus pertinentes dans l'explication de la variabilité des jeux de données. Elle repose sur des techniques mathématiques d'algèbre linéaire (décomposition en valeur singulière (SVD)).

L'ACP fait partie de la famille des analyses statistiques appelées analyses factorielles. Les statistiques descriptives (moyenne, médiane, variance, etc.) s'avèrent être limitées lorsque l'on cherche à décrire ou à résumer un volume de données important. Dans ce contexte, il est préférable d'avoir une méthode qui permet de résumer les principaux facteurs qui régissent un jeu de données. C'est l'analyse factorielle.

L'analyse factorielle permet à un individu (représenté par une ligne) et/ou à une variable (modalité) d'avoir des coordonnées sur le plan (x, y) ==> plan factoriel. Cependant, il ne suffit pas juste de représenter les individus et/ou les variables sur le plan factoriel. Il faudrait que ces individus et/ou variables soient les plus dispersées possible de telle sorte à faciliter les interprétations après. En termes mathématiques, cela se traduit par la maximisation de la variance du nuage projeté :

$$\max((M^t v)^t M^t v) = \max\,(v^t M M^t v)$$

$$s/c : v^t v = 1$$

Où *M*, est la matrice (jeu de données) de départ.

L'idée est de partir d'un jeu de données initiales avec *N* variables pour réduire à *(N-k)* variables où k>0 tout en gardant le maximum d'information possible dans les données.

Par ailleurs, on sait utiliser cette optimisation par des techniques d'algèbre linéaire notamment la décomposition en valeur singulière (SVD). Cette SVD souligne, entre autres, que toute matrice *M* dans R^{nk} peut être décomposée en un produit de 03 facteurs :

$$M = USV^t$$

Où *U* est une matrice de dimension m*xn* avec certaines propriétés algébriques, *S* est une matrice diagonale de R^{nk} et V^t , une matrice carrée de dimension *k*.

Schématiquement, cette décomposition en valeurs singulières s'illustre de la manière suivante :

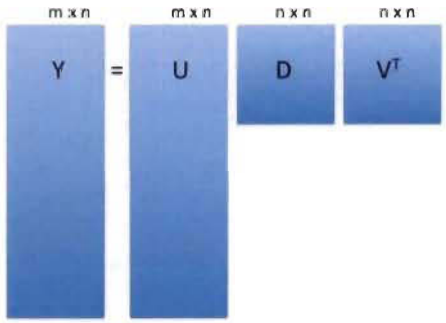


Figure 7 Formule de décomposition en valeur singulière, tirée du cours de Rafael Irizarry, High-Dimensional Data Analysis sur edX.

Et après réduction, cela ressemble à l'image suivante :

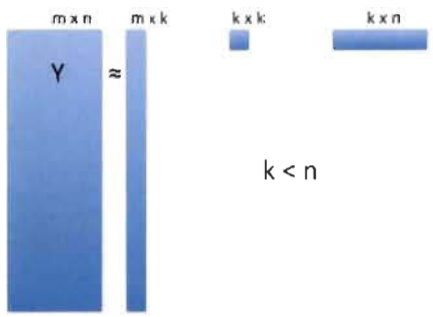


Figure 8 Illustration de la décomposition en valeur singulière tirée du cours de Rafael Irizarry, High-Dimensional Data Analysis sur edX.

En résumé, l'ACP peut donc être utilisée pour réduire le bruit dans les données et potentiellement améliorer le niveau de la prédiction.

3.6.4 Machine à vecteur support : SVM

Inventée par Lyunov Vapnik, cette technique, auparavant l'une des plus reconnues pour la classification, consiste à séparer le plan de sorte à maximiser la séparation des différentes classes d'appartenance des observations.

L'algorithme du SVM introduit la notion de marge maximale, qui n'est autre que la distance qui sépare les frontières de séparation des N-classes d'observations du jeu de données.

Dans le graphique ci-dessous, la frontière correspond à la droite d'équation $y = x$

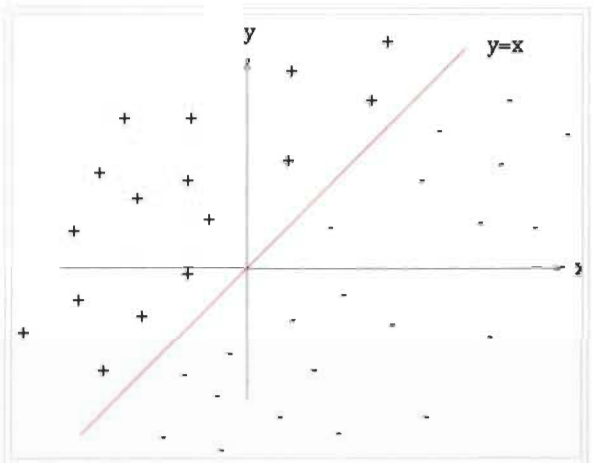


Figure 9 Séparation parfaite du plan, image prise de Wikipédia sur les SVM.

Et L'échantillon de points proches de cette droite est formé par l'ensemble des observations aux alentours. On dit souvent que le SVM est une généralisation de la régression linéaire pour des problèmes non linéairement séparables, comme le montre l'exemple de la Figure 10 ci-dessous :

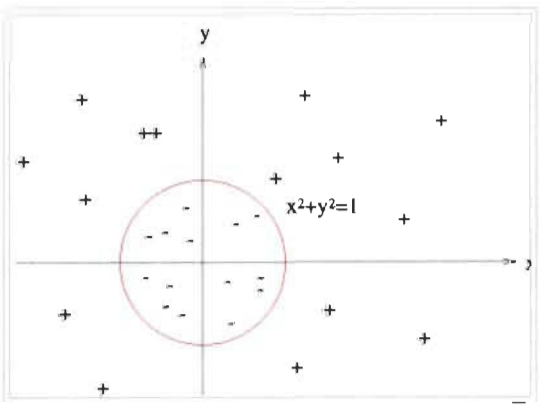


Figure 10 Illustration de la séparation du plan dans le cas non linéaire, image prise de Wikipédia sur les SVM.

Par ailleurs, les SVM introduisent la notion de noyau qui consiste à transformer l'espace de départ des données en un espace de plus grande dimension, et dans lequel il est probable de trouver une séparation beaucoup plus linéaire.

3.6.5 Classification Naïve de Bayes

La classification naïve de Bayes tient son origine du théorème de Bayes. Le théorème est un résultat de probabilité conditionnelle :

$$P(C\ |A) = \frac{p(A)p(A\ |C)}{p(A)p(A\ |C) + (1 - p(A))p(C\ |non\ A)}$$

Exemple, si dans un jeu de données représentatif, le pourcentage de fumeur ($p(A)$) est connu. Que de plus, le ministère de la santé publique indique la proportion de fumeurs ayant contracté le cancer ($p(A\ |C)$), (dans le langage d'apprentissage machine : prédire) le risque de cancer compte tenu du passé de fumeur de l'individu pourrait être estimé.

C'est un classifieur probabiliste qui prend comme hypothèse que les variables qui caractérisent le « target» sont indépendantes entre elles. Dans l'exemple ci-dessous, un seul facteur qui est la variable « Fumer ou pas » est mis en exergue. Le modèle probabiliste pour la probabilité d'appartenance d'une observation à une classe C est généralisable si le « target » contient plusieurs classes (comme le problème étudié dans le présent mémoire)

$$P(C_i\ |A) = \frac{P(C_i)P(A|C_i)}{\sum_{i=1}^{n} P(A|C_i)P(C_i)}$$

Où C_i constitue différentes classes d'appartenance des observations (Bogues bloquants, majeures, etc.).

3.6.6 L'apprentissage profond : réseaux de neurones de convolution

Les réseaux de neurones de convolution « convnet » sont similaires aux perceptrons multicouches sur plusieurs aspects :

- Composés de neurones.
- Des pondérations dont les valeurs sont apprises durant l'entrainement du modèle.
- Les fonctions d'activation sont calculées (exemple : ReLu : $\max(0, x)$) neurone par neurone, ce qui laisse le volume de l'image inchangée pour cette opération.
- La dernière couche est pleinement connectée (« fully connected layer ») contenant les classes d'appartenance prédites de l'image.

Les différences fondamentales :

- Primo dans l'input. Pour les réseaux de convolution, les entrées sont des images (de 03 dimensions gérant les 03 couleurs de l'image R, G, B).

- Secundo, les « convnet » ne calculent pas de pondérations sur chaque neurone pris individuellement mais sur un ensemble connectés de neurones. Ce qui s'avère efficient pour le traitement des images comparées aux perceptrons multicouches.
- Tertio, les « convnet » ont une opération appelée « Pool » qui consiste à sous-échantillonner l'image en longueur et en largeur (comme le montre le dernier volume tout à fait à droite de l'image de la Figure 11 ci-dessous).

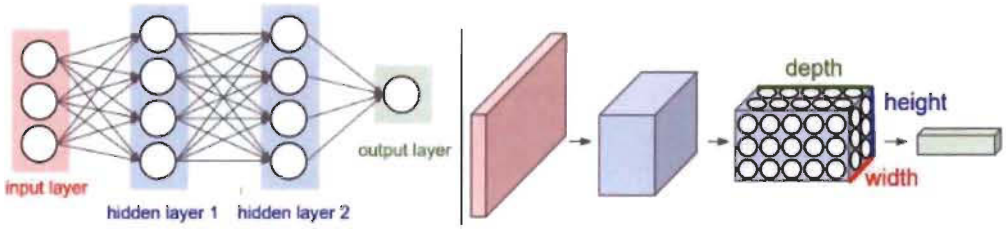


Figure 11 Un « convnet » traite ses neurones selon 03 dimensions : la profondeur, la largeur et la longueur, cs231n, Stanford.

La profondeur correspond au nombre de filtres de convolution appliqués à l'image. La largeur et la hauteur peuvent être variées en utilisant l'opération de « Pool » mentionnée précédemment.

Les opérations conventionnelles sur les « convnet » sont résumées par la Figure 12 suivante :

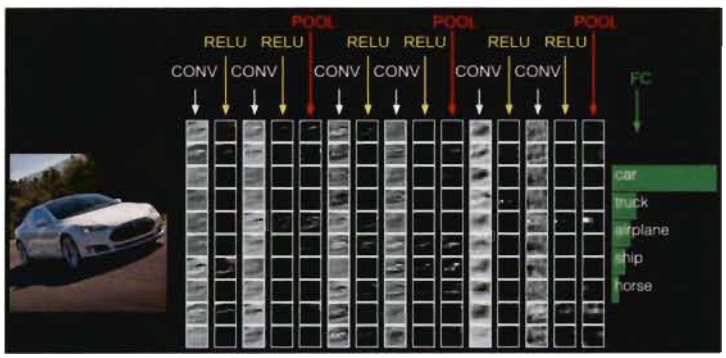


Figure 12 Les opérations sur un « convnet », cs231n, Stanford.

3.6.7 k plus proches voisins (k-nn)

Cette méthode d'apprentissage figure dans la famille des modèles d'apprentissage supervisé. Pour une observation donnée (exemple : une ligne d'un jeu de données), dont le « target » est a priori inconnu, les *k*-points les proches de lui et pour lesquels les valeurs du « target » sont connues sont à déterminer. Ainsi, la prédiction de sa valeur est possible selon une certaine métrique de distance pour la régression et par système de vote pour la classification.

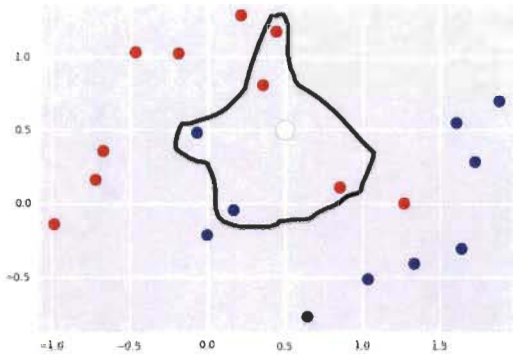


Figure 13 Illustration du k-NN, Openclassroom.

Dans l'image ci-dessus, le point blanc correspond à l'observation dont la valeur d'intérêt est inconnue et que sa prédiction est à chercher.

Elle est projetée sur le plan selon une certaine distance et pour un k=5, les 05 points qui lui sont les plus proches sont pris. Selon le type de tâche, une décision est donnée quant à la classe d'appartenance du point blanc.

Pour une classification, par exemple, la classe majoritaire parmi les 5 points voisins est déterminée. Tandis que pour une régression, une moyenne de la valeur de la variable d'intérêt pour ces 5 points fera l'objet d'un calcul.

Chapitre 4.

Expérimentations et résultats

Dans ce chapitre, les différents résultats de nos expérimentations seront abordés. Le processus adopté pour avoir d'abord le paramètre de la formule de métrique de rang ('Ranking Index'), ensuite la comparaison de la performance des modèles logiciel par logiciel, et enfin la comparaison entre les métriques proposées et celles de la littérature ainsi que les différentes comparaisons de la performance des modèles classiques à celles des réseaux de neurones profonds seront exposées minutieusement.

4.1 Détermination empirique du coefficient α de la métrique de rang

Dans un premier temps, il s'agissait de trouver la meilleure formule possible pour la métrique de rang définie :

$$R_{ct} = \delta T_{ct}^{\alpha} \delta L_{c't}^{1-\alpha}$$

Pour rappel, cette métrique prend une valeur d'autant plus faible que la classe logicielle concernée s'expose à un risque de bogues plus élevé et inversement. Ce risque de bogue est divisé en deux :

- Les risques associés aux modifications de code source sont capturés par le terme δT_{ct}^{α}.
- Les risques associés à l'effort de test, matérialisés par le second terme $\delta L_{c't}^{1-\alpha}$.

La détermination du coefficient α est faite sur la base de la maximisation de la prédiction de la sévérité des bogues. Ainsi, le coefficient α matérialise le pourcentage d'implication de chaque phénomène dans l'apparition des bogues.

Le paramètre α est varié et l'algorithme de regroupement *k* plus proches voisins (*k-nn*) pour voir la pureté des classes obtenues par la classification est utilisé.

En effet, le *k-nn* minimise la distance euclidienne à l'intérieur d'un groupe d'une même instance (exemple : les bogues sévères).

Nous choisissons 6 variantes du rang, dont 5 qui dépendent de la valeur que prend α :

$$F_0 = \frac{\delta T_{ct}}{\delta L_{c't}}$$

La formule F_1 stipule que la modification récente de code source ainsi que l'effort de test faible contribuent à proportion égale (un et demi chacun) à l'apparition de bogues.

$$F_1 = \delta T_{ct}^{\frac{1}{2}} \delta L_{c't}^{\frac{1}{2}}$$

La formule F_2 stipule que la modification récente de code source contribue à 66% dans l'apparition de bogues.

$$F_2 = \delta T_{ct}^{\frac{2}{3}} \delta L_{c\prime t}^{\frac{1}{3}}$$

La formule F_3 stipule que la modification récente de code source contribue à 75% dans l'apparition de bogues.

$$F_3 = \delta T_{ct}^{\frac{3}{4}} \delta L_{c\prime t}^{\frac{1}{4}}$$

La formule F_4 stipule que le faible effort de test contribue à 66% dans l'apparition de bogues.

$$F_4 = \delta T_{ct}^{\frac{1}{3}} \delta L_{c\prime t}^{\frac{2}{3}}$$

La formule F_5 stipule que le faible effort de test contribue à 75% dans l'apparition de bogues.

$$F_5 = \delta T_{ct}^{\frac{1}{4}} \delta L_{c\prime t}^{\frac{3}{4}}$$

La formule F_0 est l'une des premières formes que l'étude avait empiriquement évaluée. En effet, nous voulions pénaliser les classes logicielles dont la modification est récente mais que cette pénalisation soit plus faible si la classe a fait l'objet d'effort de test.

F_0 est différente en interprétation contrairement aux autres car plus sa valeur est faible, plus la classe logicielle a moins de risque de connaitre des bogues dans le futur.

F_1 a F_5 sont des variantes de la fonction Cobb-Douglas pour différentes valeurs du paramètre α.

Dans notre expérimentation, ces formules sont soumises au *k-nn* pour déterminer celle qui discrimine le plus les classes de bogues :

1. Bogues bloquants
2. Bogues majeurs
3. Bogues critiques
4. Bogues mineurs
5. Bogues triviaux
6. Et Autres : tous ceux qui ne sont pas des bogues.

Comparativement aux expérimentations qui vont suivre, ici, les 02 classes de bogues majeures puis critiques ne sont pas regroupées.

4.2 Comparaison de la performance des modèles, logiciel par logiciel

Notre série d'expérimentations commence par la comparaison de 03 modèles : Random Forest, SVM et un perceptron multicouches.

En premier lieu, toutes les variables sont incluses dans chaque modèle. Ces variables constituent les 78 métriques fréquemment utilisées dans la littérature du génie logiciel, plus les 02 métriques définies précédemment :

- Métriques de rang pour le classement des classes à risque de bogues.
- Indice de bogue (« bug Index ») pour les classes qui avaient expérimentés des bogues par le passé (relatif à la date de « commit » courante).

Dans cette série, le modèle capable de gérer ces variables sera ressorti ainsi que l'obtention d'un classement global de toutes les variables confondues.

Dans un deuxième temps, les mêmes modèles seront ré-entrainés mais en séparant les groupes de variables (métriques) : nos 02 métriques définies versus les 78 métriques fréquemment utilisées dans la littérature.

4.3 Comparaison entre nos métriques et celles de la littérature

Dans la précédente expérimentation, un classement global des variables (métriques) par ordre d'importance dans la prédiction est acquis. Ensuite, une comparaison par groupe de variables (nos 02 métriques définies versus les 78 métriques fréquemment utilisées dans la littérature) est effectuée. Cependant, cette comparaison ne peut pas affirmer la supériorité de nos métriques par rapport aux autres. En effet, il se peut que la performance des modèles utilisant les 78 métriques soit due uniquement à quelques métriques. Si tel est le cas, alors nos 02 métriques ne diffèrent pas tant des autres déjà étudiées dans la littérature.

Ainsi, la comparaison des performances des modèles utilisant 02 métriques prises au hasard parmi les 78 versus les modèles utilisant nos 02 métriques d'intérêt devra être effectuée. La comparaison utilise une classification binaire.

Pour chaque type de bogues (bloquants, majeurs, critiques, mineurs, triviaux), une variable indicatrice est créée et ainsi on obtient des variables binaires pour chacune des catégories. Chacune des variables est par la suite utilisée comme variable "target" dans des modèles d'apprentissage machine utilisant les métriques à comparer. A l'issu de cette expérimentation, la performance de chaque couple de métriques classiques contre les 02 définies plus haut sera étudiée.

4.4 Comparaison des modèles utilisant les mots des "commits"

Dans les expérimentations précédentes, les 02 métriques définies contre les 78 métriques présentées dans la Table 5 de l'annexe sont comparées.

Rappelons que nos 02 métriques sont :

$$R_{ct} = \delta T_{ct}^{\frac{1}{2}} \delta L_{c't}^{\frac{1}{2}} \qquad (1)$$

Qui est une métrique de rang pour classer les classes logicielles en terme de risque de bogues (plus sa valeur est faible, plus la classe c au moment t risque de contenir des bogues à un instant ultérieur).

- Indice de bogue (« Bug Index ») pour les classes qui avaient expérimenté des bogues par le passé (relatif à la date de « commit » courante).

$$B_{ct} = 1_{\{if\ c\ a\ une\ faute\ à\ t_0<t\}}, 0\ sinon \qquad (2)$$

L'historique de versions contient d'autres informations dont la pertinence pour la prédiction de bogues n'est pas pleinement explorée. En effet, l'historique de git contient, entre autres, des données textuelles intéressantes comme les « commits ».

L'hypothèse admise implicitement dans cette étude est que les « commits » sont l'expression du changement d'état du code par les développeurs. Ainsi, ils sont susceptibles de contenir des indices des bogues dans le futur.

D'ailleurs, plusieurs chercheurs ont étayé cette hypothèse. En effet, Barnet et al. (2016) ont trouvé une corrélation significative entre la taille des « commits » et les bogues logiciels. Zhang et al. (2014) ont émis l'hypothèse qu'il existe une relation entre la description textuelle des « commits » et les raisons du changement.

Notre approche consiste à prendre les phrases des « commits », à les prétraiter par les techniques de traitement de langues naturelles (voir Chap.3).

Enfin, une fois transformés en mots clés individuels, ils sont intégrés dans des modèles d'apprentissage machine. Entre autres, l'utilisation de 02 modèles de Machine Learning :

- Random Forest : qui est apparu comme notre meilleur modèle des expérimentations précédentes. Les variables données à ce modèle sont au nombre de 1200. Ce sont les mots clés retenus par les techniques de traitement de langues naturelles. Ces mots clés sont, par la suite, pondérée par TF-IDF.
- Ensuite, un réseau de neurones de convolution (CNN) qui est reconnu pour sa performance dans la classification de documents: Kim et al.(2014). Pour ce modèle de réseau de neurones de convolution, l'architecture implémentée dans Kim et al. (2014)

est adoptée avec une adaptation du code[7] pour prendre en considération la classification multiclasses.

Dans ce qui suit, les résultats des différentes expérimentations présentées dans le chapitre précédent seront détaillés. Le but de toutes ces expérimentations est de montrer l'efficacité des métriques dans la prédiction de la sévérité des bogues logiciels. La prédiction des bogues logiciels à partir des mots des "commits" sera également abordée avec deux grandes familles d'apprentissage machine: Random Forest et Deep Learning.

4.5 Résultats de la sélection de la formule optimale pour la métrique de rang

Avant de comparer les métriques, le choix de la formule optimale pour la fonction Cobb-Douglass (1) qui matérialise le rang des classes logicielles à risque est primordial.

Les résultats de cette partie concernent la détermination de α par l'algorithme des k-plus proches voisins. Le paramètre *k est varié* de 5 à 50 voisins, et les formules (F_0, F_1) qui séparent au mieux les différentes classes de sévérité est surlignée dans la Table 1, ci-dessous :

Table 1 Application du k-nn selon différents paramètres de la formule de Rang.

Voisins (k)	Formules	Accuracy	Voisins (k)	Formules	Accuracy
5	F_0	82.22%	15	F_5	65.97%
5	F_1	86.89%	25	F_0	88.14%
5	F_2	65.97%	25	F_1	88.13%
5	F_3	65.97%	25	F_3	65.97%
5	F_4	65.97%	25	F_4	65.97%
5	F_5	65.97%	25	F_5	65.97%
10	F_0	88.13%	50	F_0	88.17%
10	F_1	88.18%	50	F_1	87.95%
10	F_2	65.97%	50	F_2	65.97%
10	F_3	65.97%	50	F_3	65.97%
10	F_4	65.97%	50	F_4	65.97%
10	F_5	65.97%	50	F_5	65.97%
15	F_0	88.16%			
15	F_1	87.98%			
15	F_2	65.97%			
15	F_3	65.97%			
15	F_4	65.97%			

Source: Calculs à partir de l'entraînement du modèle *k-nn*

La Table 1 illustre les résultats de la sélection de la formule qui constituera la proposition de l'étude. Ces résultats sont obtenus en entrainant le modèle k-nn dans la prédiction de la sévérité des bogues et cela tout en variant le paramètre k (de 5 à 50).

[7] http://www.wildml.com/2015/12/implementing-a-cnn-for-text-classification-in-tensorflow/

On remarque que les modèles de k-nn utilisant les formules F_0 et F_1 donnent d'excellents taux de bonne classification (entre 87% à 88%). Néanmoins, la formule F_0 se comporte un peu moins bien pour le paramètre *k=5*, car son taux de bonne classification devient relativement plus faible (82%), mais reste néanmoins très bon.

A la lumière de ces résultats, nous optons pour la formule F_1 , qu'on appellera "Ranking Index". Elle est non seulement excellente à séparer les classes de sévérité (Table 1), mais aussi facile d'interprétation :

$$F_1 = \delta T_{ct}^{\frac{1}{2}} \delta L_{c't}^{\frac{1}{2}}$$

4.6 Résultats de la comparaison logiciel par logiciel

Dans cette partie, la construction des modèles de prédiction de sévérité de bogues, logiciel par logiciel, pour se conformer à ce qui se fait généralement dans la littérature sera étudiée. La focalisation se situe principalement sur un logiciel (ActiveMQ) bien que les résultats pour Apache Struts I et II soient aussi exposés.

Dans la prochaine étape, les 78 métriques collectées par Vasa (2018) contre celles que l'étude a définies plus haut à savoir « Rank Index » et « Bug Index » seront comparées. Le choix est de ne présenter qu'un seul modèle de référence, sachant que dans nos expérimentations, le Random Forest semble être le modèle qui donne les résultats les plus satisfaisants.

Table 2 Performance comparative des métriques par logiciel

Modèle	Métriques	Accuracy	Logiciel utilisé
Random Forest	L'ensemble 78 métriques	66.19%	Apache ActiveMQ
Random Forest	« Ranking Index » et « Bug Index »	72.08%	Apache ActiveMQ
Random Forest	« Rank Index » et « Bug Index »	52.56%	Apache Struts I et II

Source: Calculs à partir de l'entraînement de modèle.

Dans la Table 2, un résumé des résultats les plus saillants de l'entrainement des modèles classiques de Machine Learning (ML) est exposé. En effet, plusieurs modèles de ML ont été évalués mais le Random Forest apparait comme le plus régulier en termes de performances (taux de bonne classification et AUC). Pour le logiciel ActiveMQ, il est constaté que le modèle entrainé avec les deux métriques proposées par l'étude a une performance nettement supérieure à celui utilisant les 78 métriques issues de la littérature (+6% de taux de bonne classification, Table 2, ci-dessus). Néanmoins, les performances du premier modèle ont été très inférieures lorsqu'il a été entrainé sur les données du logiciel Apache Struts I et II.

Pour se conformer à la littérature (Toure, Badri et Lamontagne (2018)), la présente section a évalué les performances des 02 familles de métriques pour chaque logiciel. Cependant, aucune conclusion probante ne peut être établie bien que pour ActiveMQ, les performances des 02 métriques proposées soient supérieures aux métriques traditionnelles sélectionnées par l'étude. Dans la section qui suit, au lieu de construire un modèle pour chaque logiciel, une modélisation prédictive combinant l'ensemble des logiciels étudiés est effectuée.

4.7 Résultats de la comparaison pour l'ensemble du jeu de données

Les disparités des performances des métriques selon les logiciels nous conduit à les évaluer en utilisant une combinaison de tous les logiciels. Trois types de modèles d'apprentissage sont utilisés : Random Forest (RF), Support Vector Machine (SVM) et Perceptron multicouches (MLP). Chaque modèle est utilisé trois (03) fois :

1. Avec toutes les variables (métriques traditionnelles, Ranking Index et Bug Index)
2. Utilisant seulement les 78 métriques traditionnelles
3. Utilisant uniquement les 02 métriques : Rank Index et Bug Index.

En entrainant les modèles avec toutes les données, le constat est que les métriques indépendantes du code (celles que nous proposons) rivalisent avec la sélection de métriques choisies dans la *Table 3 (ci-dessous)*. En effet, en utilisant Rank Index et Bug Index comme variables explicatives, un modèle de Random Forest peut atteindre 71% de bonne classification (tous logiciels confondus). A la lumière de ces résultats, nous pouvons raisonnablement affirmer que les métriques issues des dépôts GIT sont tout à fait capables de prédire la sévérité des bogues logiciels au même titre que les métriques usuelles de la littérature (RQ1 et RQ2[8]).

- [8] RQ1: La possibilité de construire des métriques indépendantes du code telles que "Ranking Index", "Bug Index" et les mots des "commits" est possible.
- RQ2: Les métriques indépendantes du code rivalisent de paire avec les métriques de la littérature (orientée objet, graphes, complexité, etc.) et parfois même plus dans certaines catégories de sévérité.

Table 3 Performance comparative des métriques par logiciel

Modèle	Variables (métriques)	Taux de bonne classification	Logiciels utilisés
Random Forest	78 métriques	71.01%	ActiveMQ , Struts I et II
Random Forest	« Rank Index » et « Bug Index »	71.28%	ActiveMQ , Struts I et II
SVM (noyau linéaire)	78 métriques	71.5%	ActiveMQ , Struts I et II
SVM (noyau linéaire)	« Rank Index » et « Bug Index »	71.29%	ActiveMQ , Struts I et II
Perceptron Multicouches	78 métriques	71.13%	ActiveMQ , Struts I et II
Perceptron Multicouches	« Rank Index » et « Bug Index »	70.92%	ActiveMQ , Struts I et II

Source: Calculs à partir de l'entraînement de modèles sur 03 logiciels combinés.

Dans cette section, il est constaté que les métriques proposées par l'étude peuvent atteindre des performances similaires aux métriques de code bien établies de la littérature et qui ont été choisies en guise de "benchmark". Cependant, cette performance cache l'apport individuel de chaque métrique dans la prédiction. En effet, si parmi les métriques prises en "benchmark", il existe deux qui peuvent atteindre les performances des 02 proposées par l'étude, alors, la question de la supériorité de notre proposition ne tient plus.

Dans ce qui suit, une évaluation 02 par 02 des métriques faites et ce, pour chaque catégorie de sévérité. On effectue alors une classification binaire pour chaque catégorie de bogues et non plus une classification multi-classes comme auparavant.

4.8 Comparaison 02 par 02 des métriques

Dans la section précédente, il est constaté qu'aucun ensemble de métriques ne se démarque de manière significative dans la prédiction de la sévérité des bogues logiciels. Cela ne permet pas de conclure sur l'efficacité de Rank Index et Bug Index ainsi que de répondre directement à la question de recherche RQ3 qui consiste à affirmer si les métriques issues du git sont supérieures dans la prédiction de la sévérité des bogues.

D'ailleurs, il se peut que les performances des modèles utilisant les 78 métriques puissent être influencées par quelques métriques seulement (2, 3, etc.). Ainsi, il est beaucoup plus judicieux d'effectuer une comparaison 02 par 02 des métriques.

Dans les résultats résumés à la Table 4 (ci-dessous), un tirage aléatoire de 02 métriques parmi les 78 fréquemment utilisées dans la littérature a été effectué, puis une comparaison des résultats de leur prédiction aux modèles qui utilisent uniquement Ranking Index et Bug Index est réalisée.

Table 4 Extrait Performance comparative 02 métriques choisies aléatoirement

Paire de métriques	AUC	Type
TIC-THC	0,501	Blocker
TIC-THC	0,500	critical/major
TIC-THC	0,499	minor
TIC-THC	0,499	trivial
TIC-THC	0,500	Non bug issues
YMC-MFR	0,503	Blocker
YMC-MFR	0,500	critical/major
YMC-MFR	0,501	minor
YMC-MFR	0,503	trivial
YMC-MFR	0,500	Non bug issues
ZFC-LMCI	0,499	Blocker
ZFC-LMCI	0,500	critical/major
ZFC-LMCI	0,500	minor
ZFC-LMCI	0,499	trivial
ZFC-LMCI	0,500	Non bug issues
ZOC-IOC	0,501	Blocker
ZOC-IOC	0,500	critical/major
ZOC-IOC	0,500	Minor
ZOC-IOC	0,499	Trivial
ZOC-IOC	0,500	Non bug issues
Ranking Index-bug Index	0,967	Blocker
Ranking Index-bug Index	0,459	critical/major
Ranking Index-bug Index	0,525	minor
Ranking Index-bug Index	0,583	trivial
Ranking Index-bug Index	0,495	Non bug issues

Source: Extraits, Calculs de l'auteur à partir de l'entraînement de modèles (Voir Annexe pour le reste des paires).

Le couple de métriques proposées par l'étude (Ranking Index et Bug Index) atteignent des AUC de 0.96, 0.45, 0.52, 0.58 et 0.49 respectivement sur les catégories de bogues: bloquants, majeurs, mineurs, triviaux et Autres. A l'inverse, pour tous les autres couples de métriques traditionnelles, les AUC tournent autour de 0.5 sans exception.

A la lumière de ces expériences, la conclusion est que nos deux métriques ont la capacité de bien prédire les bogues bloquants (AUC : 0.967) et une performance plus ou moins égale à celle des autres métriques dans la prédiction des autres classes de bogues logiciels. Rappelons qu'un AUC de l'ordre de 0.5 est l'équivalent d'un classifieur qui réalise une prédiction complètement aléatoire et donc, de très faible performance.

Le couple de métriques évaluées par l'étude rivalise globalement avec les métriques traditionnelles. Elles sont, par ailleurs, excellentes dans la prédiction de certains types de bogues notamment les plus importants que les études de la littérature cherchent à prédire : les bogues bloquants.

Les résultats expérimentaux confirment ainsi, les deux questions de recherches RQ1 et RQ2, mais ne permettent pas d'établir la supériorité absolue des 02 métriques proposées comparativement au reste.

4.9 Usage des mots des « commits » dans des modèles prédictifs

Dans toutes les parties précédentes, l'étude s'est intéressée à la construction et à la validation empirique de deux métriques définies à partir du Git (historique de versions) de projets logiciels. Il s'agit de Rank Index et Bug Index. A la section 4.8, le résultat montre que Rank Index et Bug Index surpassent les autres métriques dans la prédiction des bogues bloquants avec un AUC de ~0.96. Néanmoins, on ne peut rien dire sur leur supériorité absolue, car pour mieux trancher sur RQ3, une autre facette de l'historique de versions GIT fera l'objet d'étude. Il s'agit de l'aspect textuel.

Ainsi, l'exploration de l'historique de git notamment l'aspect textuel de celui-ci continue. En effet, plusieurs auteurs tels que Barnett et al. (2016) ont trouvé des liens de corrélation entre la taille d'une phrase de commit et le risque de bogues à venir. Mockus et al. (2014) vont même jusqu'à inférer les raisons du changement de code source à partir du message des commits. Tout cela corrobore notre hypothèse selon laquelle les phrases des commits contiendraient des informations sur les bogues futurs liées aux classes logicielles.

Ces informations sont judicieusement exploitées, comme dans toutes nos expériences, à l'aide de la modélisation prédictive. Cependant, les modèles ne prennent pas les phrases comme des variables. Leur exploitation utilise les techniques de traitement des langues naturelles pour transformer les phrases en variables pour nos modèles. Les détails de cette méthodologie peuvent être consultés dans la partie méthodologie de ce mémoire (Chapitre 3).

4.9.1 Avec un Random Forest

La pondération TF-IDF (« Term Frequency Inverse Document Frequency ») dans le modèle Random Forest sera utilisée après avoir extrait les mots clés des phrases des commits.

Après épurement et prétraitement, nous atteignons presque 1200 mots clés caractérisant l'ensemble des commits des 03 logiciels, objets de l'étude : Apache ActiveMQ, Struts I et II.

Les résultats du modèle sont consignés dans la Table 6 suivante :

TABLE 6 : RF AVEC LES MOTS DES COMMITS

Modèle	Taux de bonne classification	Variables utilisées
Random Forest +	84.56%	Toutes les variables issues du git
Random Forest ++	85.40%	Toutes les variables issues du git avec réduction de la dimension par ACP
Random Forest +++	80.35%	Commits uniquement
Essai de réplication (« SpamBayes ») de Barnett et al sur Scikit-learn (Python)	65.94%	Commits uniquement

Random Forest + : utilise les mots des commits et les deux métriques : « Bug Index » et « Ranking Index ».

Random Forest +++ : est le fruit de l'apport net des mots des « commits » sans réduction de dimension ni ajout d'autres variables. Ce résultat est 6% plus précis que le meilleur modèle utilisant l'ensemble des 78 métriques avec « Ranking Index » et « Bug Index ». Enfin, Il est 8% plus précis que le meilleur modèle utilisant « Ranking Index » et « Bug Index » seulement.

A la lumière de ces résultats, il ressort que l'ajout des mots des commits entraine une amélioration nette de la qualité de la prédiction de la sévérité des bogues logiciels. En effet, le taux de bonne classification d'un modèle de Random Forest utilisant les métriques de "benchmark" est de 71% (Table 3, ligne 1) tandis qu'ici, il est de 80.35% (Table 6, ligne 3). Ce qui renforce notre hypothèse selon laquelle les phrases des commits contiennent des « leaks » ou indices sur la survenance prochaine de bogues dans les classes logicielles.

Ainsi, on vient de voir que l'usage des mots clés des "commits" à eux seuls permet de surpasser grandement les métriques traditionnelles dans la prédiction de la sévérité des bogues (+9% de bonne classification, Table 3 et Table 6, ligne 1 et 3 resp.). Alors, l'étude montre qu'il existe des métriques indépendantes du code source qui sont meilleures que les métriques traditionnelles dans la prédiction de la sévérité des bogues logiciels (RQ1 à RQ3[9]).

[9] Est-ce que ces métriques agnostiques au code source sont plus performantes que les métriques traditionnelles?

4.9.2 Réduction de la dimension avec l'Analyse en Composantes Principales (ACP)

Random Forest ++ , de *Table 6*, utilise une réduction de dimension sur les mots (par l'ACP) et combinée avec les deux métriques : Bug Index et Ranking Index. En effet, parmi les 1200 mots de commits extraits, affirmer qu'il existe encore du bruit semble possible. Pour réduire ces bruits, l'ACP est utilisée tout en faisant en sorte de retenir le maximum d'information (inertie) dans le jeu de données initiales.

Dans notre implémentation algorithmique, l'intervention d'un module du logiciel R appelé « Factoshiny »[10] et une fonction définie manuellement pour extraire les 500 variables (mots) contribuant le plus à la formation des 300 premiers axes factoriels est utilisée.

Comme le montrent les résultats, cette manière est efficiente car le temps destiné à l'étude est réduit en temps d'entrainement du modèle (500 au lieu de 1200 mots), mais aussi en taux de bonne classification (+1%).

Il est à rappeler que dans cette partie, les pondérations appliquées aux mots sont du TF-IDF et le modèle utilisé est un modèle Random Forest. Dans cette section, les "commits" sont introduits dans la prédiction de la sévérité des fautes. Les résultats sont très prometteurs dans la mesure où à eux seuls, ils peuvent classifier 80% des bogues des logiciels étudiés. L'étude veut évaluer une autre famille de modèles qui est réputée très efficace avec les données textuelles. Il s'agit des modèles d'apprentissage profond : Deep Learning.

4.9.3 Avec un modèle d'apprentissage profond

Dans cette section, les commits et un modèle d'apprentissage profond sont combinés pour la classification des bogues logiciels. Les réseaux de neurones de convolution sont notamment connus pour être très performants dans le traitement automatique des langues naturelles (traduction de langues, suggestion automatique de suites de phrases, compréhension du langage, etc.).

La modélisation prédictive se fait sur la base de l'étude faite par Kim et al. (2014). L'architecture du modèle comme décrite par les auteurs est illustrée à la figure 14 suivante :

[10] http://factominer.free.fr/graphs/index_fr.html , François Husson, Université Agrocampus Ouest, France

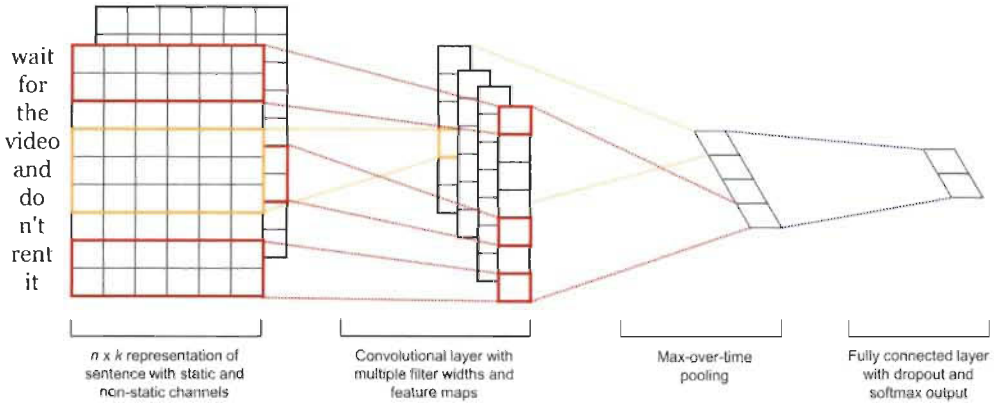


Figure 14 Kim Y. Convolutional Neural Networks for Sentence Classification,2014.

Avec cette architecture, le modèle est capable de scanner les groupes de mots successifs sur une fenêtre allant de 3 à 5 mots et ainsi capter toute la sémantique du message véhiculé par les commits.

Sur cette image d'architecture (Figure 14), la matrice de mots en entrée est de [*nxkx1*], la profondeur de la première couche est de 04, car 04 filtres de convolution ont été appliqués. La couche finale pleinement connectée est de [*1x1x4* classes].

L'étude a adapté un code écrit sur tensorflow par Denny Britz (sur son blogue WildML)[11] pour pouvoir faire de la classification multi-classes. Dans l'adaptation de code, la dernière couche a été modifiée pour prendre 05 classes de sorties (correspondant aux 05 catégories de bogues, mentionnées précédemment).

Il est à rappeler que les réseaux de neurones (à l'instar des "convnet") sont reconnus, supérieurs aux modèles classiques d'apprentissage machine (exemple: les arbres de décisions) dans le domaine de l'imagerie et de l'analyse textuelle. En effet, selon la théorie de l'approximation universelle, un perceptron multicouche est capable d'approximer n'importe quelle fonction continue dans l'ensemble R. Néanmoins, dans les domaines de la prédiction classique (classification multi-classe, classification binaire, régression, etc.), les arbres de décisions rivalisent voire dépassent les réseaux de neurones.

Avec ce « convnet », une performance de 89% sur un jeu de validation est atteinte facilement (du jamais vu par le modèle). Les graphiques qui suivent montrent la progression de

[11] http://www.wildml.com/2015/12/implementing-a-cnn-for-text-classification-in-tensorflow/

l'entrainement du modèle jusqu'au point optimal (point à partir duquel il commence à sur-ajuster).

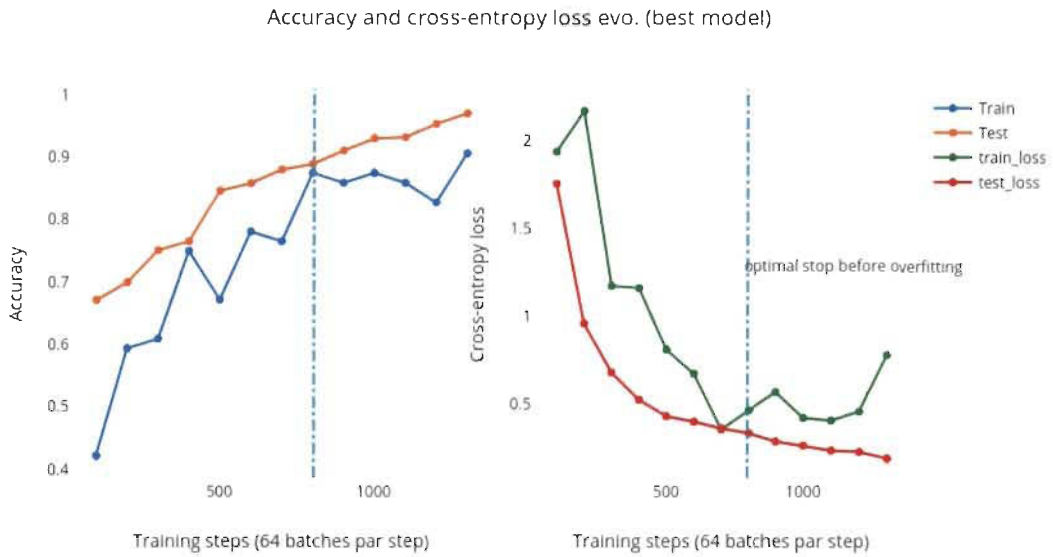


Figure 15 Évolution du taux de bonne classification et de l'erreur d'apprentissage.

Dans la Figure 15, 02 métriques sont suivies : la métrique de perte et le taux de bonne classification. La métrique de perte pour ce type de classification est le "cross-entropy". Elle est suivie sur l'ensemble d'apprentissage ainsi que les données de test (à droite de la Figure 15). Le taux d'apprentissage est également suivi pour l'ensemble d'apprentissage et de test (à gauche de la Figure 15). L'apprentissage du modèle est arrêté lorsque l'erreur de l'ensemble test commence à augmenter (sur-ajustement). L'arrêt de l'entrainement s'est effectué à l'itération 800 où le taux de bonne classification est de 89%.

4.10 Interprétations et impacts dans la littérature

Tout au long de cette recherche, le modèle et les variables « idéales » pour construire un système automatisé de classification de sévérité des bogues logiciels sont trouvés progressivement.

L'étude a débuté par définir des métriques basées sur les postulats de Kim S. et al. (2014) à propos des classes potentiellement à risque de bogues. Ensuite, elle est comparée aux métriques usuellement rencontrées dans la littérature du génie logiciel empirique. Ces comparaisons ont donné des résultats rassurants en ce qui concerne la performance de ces métriques notamment dans la prédiction des bogues bloquants.

Ensuite, l'étude a procédé à l'exploitation de l'historique des « commits » du GIT des dépôts logiciels en se penchant sur l'aspect traitement des langues naturelles des phrases des « commits ». A l'issu de la recherche, il est clair que les « commits » à eux seuls donnent des

résultats supérieurs à toutes les autres expérimentations effectuées jusqu'à ce jour. Au final, couplés à un modèle de « convnet », les mots relatifs aux « commits » approchent les 90% de taux de bonne classification.

Par la même occasion, l'étude a procédé à la vérification (investigation) des différentes questions de recherches 1 à 4 suivantes :

- RQ1: La possibilité de construire des métriques indépendantes du code source telles que "Ranking Index", "Bug Index" et les mots des "commits" est possible.
- RQ2: Les métriques indépendantes rivalisent de paire avec les métriques de la littérature (orientée objet, graphes, complexité, etc.) et parfois même plus dans certaines catégories de sévérité.
- RQ3: Est-ce que ces métriques agnostiques au code source sont plus performantes que les métriques traditionnelles? L'étude a permis de voir que les mots des "commits" sont significativement plus performants dans la prédiction de la sévérité des bogues.
- RQ4: Est-ce qu'un modèle basé sur l'apprentissage profond accroit davantage le taux de bonne classification? Oui, car joint avec les mots des "commits", un modèle "convnet" peut facilement atteindre les 89% de taux de bonne classification.

4.11 Visualisation et interprétation des résultats de la classification

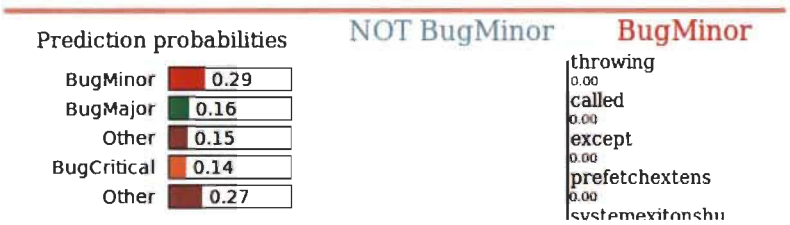
Dans cette section, donner un sens aux décisions prises par nos modèles lorsque ces derniers attribuent une classe de sévérité à une phrase de « commit » est envisageable. En effet, il est fort intéressant d'avoir un modèle qui donne un excellent taux de classification. Mais, si nous ne sommes pas capables de comprendre ce qui se passe derrière, parfois, il semble plus difficile de convaincre une audience pour son adoption en mode production.

Pour faire la prédiction (classification ou résultats) des modèles, il est intéressant d'utiliser un outil d'interprétation par la visualisation appelé LIME (« Local Interpretable Model agnostic Explanations).

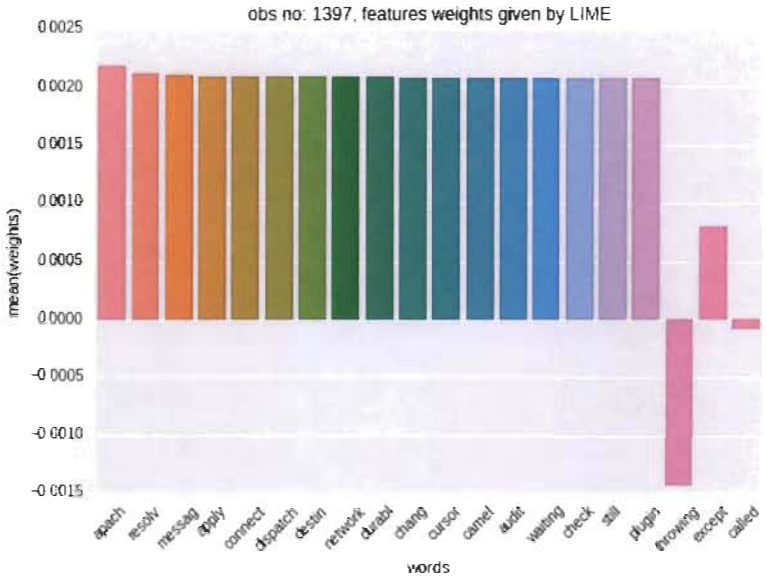
Pour les 05 classes de sévérité, des exemples de phrases de « commits » correctement classifiées par un modèle d'arbre de décision sont pris au hasard, puis les pondérations associées à chaque mot clés de la phrase sont montrées.

Exemple de classification de bogue mineur

Phrase: "throwing except called"



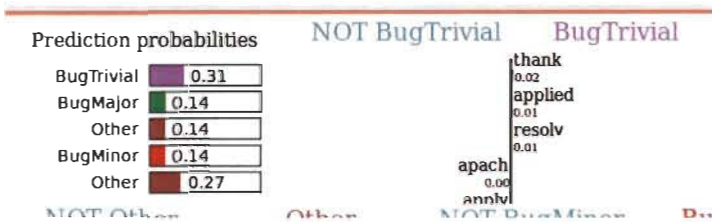
Source : Résultats d'une classification de faute mineure, utilisant LIME



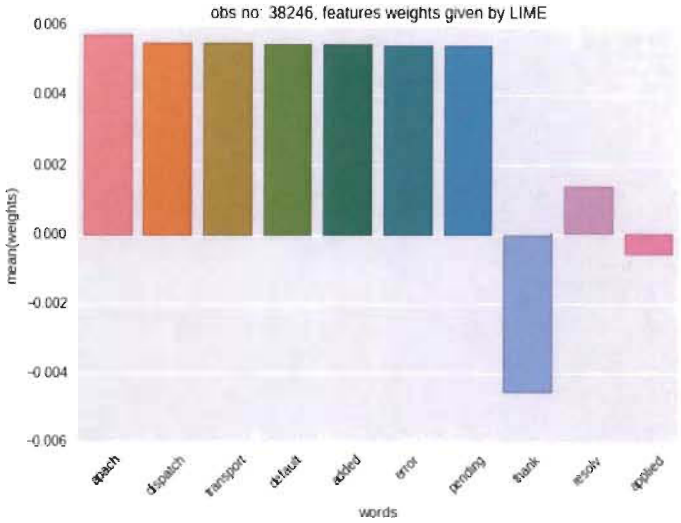
Source : Résultats d'une classification de faute mineure, utilisant LIME

Exemple de classification de bogue trivial

Phrase: « resolv apach applied thank »



Source : Résultats d'une classification de faute triviale, utilisant LIME

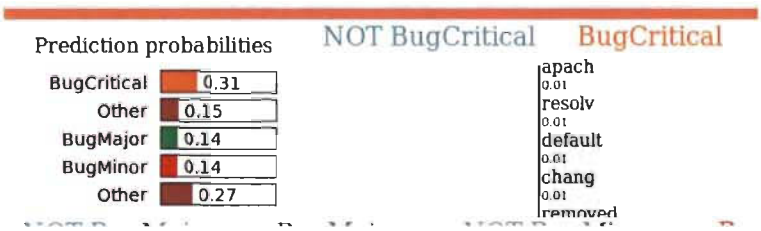


Source : Résultats d'une classification de faute triviale, utilisant LIME

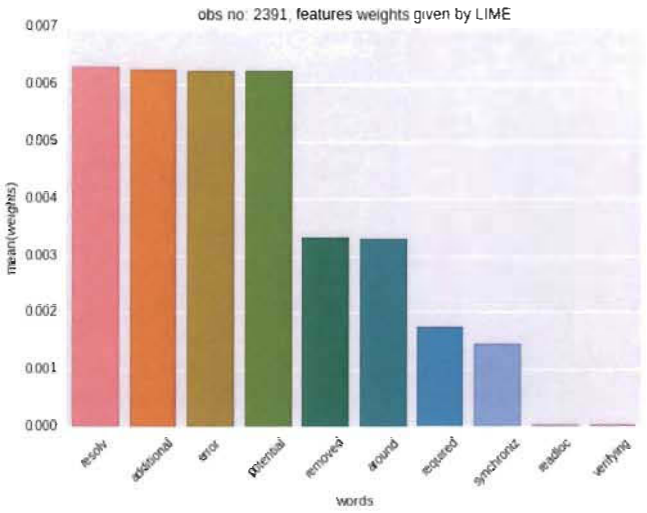
Ici, la présence du mot « thank », entre autres, entraine le classement de la phrase dans la classe des fautes triviales. Le mot en question possède la pondération moyenne parmi les plus élevées en valeur absolue.

Exemple de classification de bogue critique

Phrase: « removed synchroniz around readloc verifying required apach »



Source :Résultats d'une classification de faute critique, utilisant LIME

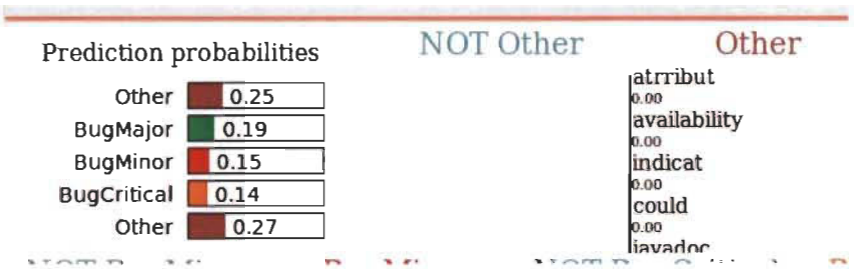


Source : Résultats d'une classification de faute critique, utilisant LIME

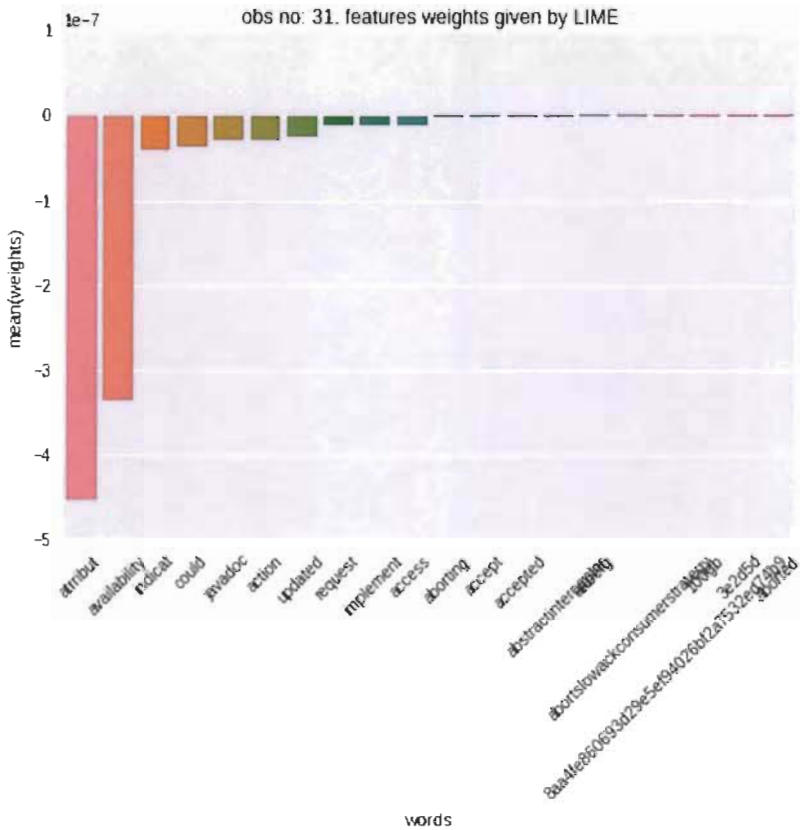
Dans cette classification, la présence des mots « required » et « synchroniz » a permis de classifier la phrase comme étant une faute critique.

Exemple de classification Autres que bogues

Phrase : « updated javadoc indicat availability action could implement access reques attribut »



Source : Résultats d'une classification "Autres", utilisant LIME



Dans cette phrase, il est clair que ce n'est qu'une mise à jour de la documentation qui n'a rien à voir avec des bogues. Ce que le modèle a bien classifié. Ses meilleurs mots sont : « attribut », « availability », alors que l'intuition aurait donné plus de pondération à « javadoc ».

Chapitre 5.

Discussions et Conclusions

Tout au long de cette étude, la possibilité d'utiliser des modèles et des métriques indépendantes du logiciel, pour la classification de la sévérité des bogues, a été explorée. Le Ranking Index est défini en utilisant les hypothèses de Kim S. et al. (2014), des classes à risque de bogues. Cette métrique est facile d'interprétation et est complémentée par le Bug Index qui est, quant à lui, une variable indicatrice de bogues par le passé. Le paramètre α du Ranking Index a été déterminé empiriquement en faisant en sorte que la métrique sépare au mieux les différentes classes de sévérité, une fois entrainée dans un modèle. La valeur $\alpha = 0.5$ de ce paramètre indique que le faible effort de test et les modifications récentes de code source contribuent de façon égale à l'apparition de bogues.

Par ailleurs, les phrases des « commits » sont largement exploitées par le traitement des langues naturelles qui s'est avéré encore plus efficace dans la classification des bogues logiciels en surpassant de 9% le taux de bonne classification du « Ranking Index ».

Notre recherche s'est également attardée sur un modèle d'apprentissage profond « convnet » qui prend comme entrée les phrases des « commits ». Ce modèle, développé par Kim. Y (2014), nous a permis d'atteindre 89% de taux de bonne classification.

Notre étude vient démontrer le potentiel des données issues du Git pour la classification de la sévérité des bogues. Qu'elles soient numériques ou textuelles, ces données rivalisent de pair avec les métriques traditionnelles fréquemment utilisées dans la littérature du génie logiciel empirique et dans certains cas, les dépassent en terme de performance.

Cependant, les applications de notre étude sont limitées si l'on ne dispose pas d'outils pour les mettre à disposition des développeurs. Pour ce faire, il faut, au préalable, standardiser l'acquisition et la mise en forme des données issues des dépôts Git. Ainsi, toutes nos réflexions et futurs travaux s'articulent autour de l'encapsulation de nos algorithmes et métriques pour les servir en tant qu'API dans les principaux fournisseurs de dépôts Git infonuagiques.

Bibliographie

Barnett, J. G., C. K. Gathuru, L. S. Soldano, and S. McIntosh. 2016. "The Relationship Between Commit Message Detail and Defect Proneness in Java Projects on Github." In 2016 Ieee/Acm 13th Working Conference on Mining Software Repositories (Msr), 496–99.

Boucher, Alexandre, and Mourad Badri. 2018. "Software Metrics Thresholds Calculation Techniques to Predict Fault-Proneness: An Empirical Comparison." Information and Software Technology 96: 38–67.

Breiman, Leo. 2001. "Random Forests." Machine Learning 45 (1): 5–32.

Conejero, Jos M., Eduardo Figueiredo, Alessandro Garcia, Juan Hernndez, and Elena Jurado. 2009. "Early Crosscutting Metrics as Predictors of Software Instability." In Objects, Components, Models and Patterns, edited by Manuel Oriol and Bertrand Meyer, 136–56. Berlin, Heidelberg: Springer Berlin Heidelberg.

Cortes, Corinna, and Vladimir Vapnik. 1995. "Support-Vector Networks." In Machine Learning,

273–97.

Eyolfson, Jon, Lin Tan, and Patrick Lam. 2014. "Correlations Between Bugginess and Time-Based Commit Characteristics." Empirical Software Engineering 19 (4):1009–39.

He, Zhimin, Fengdi Shu, Ye Yang, Mingshu Li, and Qing Wang. 2012. "An Investigation on the Feasibility of Cross-Project Defect Prediction." Automated Software Engineering 19 (2): 167–99.

Kim, S., T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. 2007. "Predicting Faults from Cached History." In 29th International Conference on Software Engineering (Icse'07), 489–98.

Kim, Yoon. 2014. "Convolutional Neural Networks for Sentence Classification." In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (Emnlp), 1746–51. Doha, Qatar: Association for Computational Linguistics.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton. 2012. "ImageNet Classification with Deep Convolutional Neural Networks." In Advances in Neural Information Processing Systems 25, edited by F. Pereira,

C. J. C. Burges, L. Bottou, and K. Q. Weinberger, 1097–1105. Curran Associates, Inc. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

Liaw, Andy, and Matthew Wiener. 2001. "Classification and Regression by Randomforest." Forest 23 (November).

Luhn, H. P. 1957. "A Statistical Approach to Mechanized Encoding and Searching of Literary Information." IBM Journal of Research and Development 1 (4): 309–17.

T. J. McCabe. 1976. A Complexity Measure. IEEE Trans. Softw. Eng. 2, 4 (July 1976), 308-320.

Manning, C.D., P Raghavan, and H Schutze. 2008. "Scoring, Term Weighting, and the Vector Space Model," January, 100–123.

Paul Douglas, Charles Cobb. 1928. "A Theory of Production." American Economic Review 18.

Radjenovi, Danijel, Marjan Heriko, Richard Torkar, and Aleivkovi. 2013. "Software Fault Prediction Metrics: A Systematic Literature Review." Information and Software Technology 55 (8): 1397–1418.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams. 1986. "Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1." In, edited by David E. Rumelhart, James L. McClelland, and CORPORATE PDP Research Group, 318–62. Cambridge, MA, USA: MIT Press. http://dl.acm.org/citation.cfm?id=104279.104293.

Sharma, Gitika, Sumit Sharma, and Shruti Gujral. 2015. "A Novel Way of Assessing Software Bug Severity Using Dictionary of Critical Terms." Procedia Computer Science 70: 632–39.

Singh, Yogesh, Arvinder Kaur, and Ruchika Malhotra. 2010. "Empirical Validation of Object-Oriented Metrics for Predicting Fault Proneness Models." Software Quality Journal 18 (1). Hingham, MA, USA: Kluwer Academic Publishers: 3–35.

Toure, Fadel, Mourad Badri, and Luc Lamontagne. 2018. "Predicting Different Levels of the Unit Testing Effort of Classes Using Source Code Metrics: A Multiple Case Study on Open-Source Software." Innovations in Systems and Software Engineering 14 (1): 15–46.

Vasa, Rajesh. 2018. "Growth and Change Dynamics in Open Source Software Systems," March.

Zhang, Feng, Audris Mockus, Iman Keivanloo, and Ying Zou. 2014. "Towards Building a Universal Defect Prediction Model." In Proceedings of the 11th Working Conference on Mining Software Repositories, 182–91. MSR 2014. New York, NY, USA: ACM.

Zhou, Yuming, and Hareton Leung. 2006. "Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults." IEEE Transactions on Software Engineering 32 (10): 771–89.

Zimmermann, Thomas, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. 2009. "Cross-Project Defect Prediction: A Large Scale Experiment on Data Vs. Domain Vs. Process." In Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the Acm Sigsoft Symposium on the Foundations of Software Engineering, 91–100. ESEC/Fse '09. New York, NY, USA: ACM.

Annexes

Table 5 Liste des métriques utilisées en benchmark

Abréviation	Description	Abréviation	Description
NOM	Method Count	MCE	External Method Call Count (outside core)
AMC	Abstract Method Count	EC	Exception Count
CCC	Class Constructor Count	CLC	Constant Load Count
RMC	Protected Method Count	CC	Branch Count
PMC	Public Method Count	IAC	InstanceOf Count
IMC	Private Method Count	CAC	Check Cast Count
SMC	Static Method Count	ODC	Out Degree Count
FMC	Final Method Count	IDC	In Degree Count
YMC	Synchronized Method Count	EODC	External out Degree Count
NOF	Field Count	IODC	Internal out Degree Count
ZFC	Initialized Field Count	TCC	Type Construction Count
UFC	Uninitialized Field Count	NAC	New Array Count
PFC	Public Field Count	NOC	Number of Children
IFC	Private Field Count	NOD	Number of Descendants
RFC	Protected Field Count	DIT	Depth in Inheritance Tree
FFC	Final Field Count	THC	Throw Count
SFC	Static Field Count	LVC	Local Variable Count
ICC	Inner Class Count	MCC	Method Call Count
INC	Interface Count	MCI	Internal Method Call Count (inside core)
CBC	Try Catch Block Count	RLC	Ref Load Op Count
SCC	Super Class Count	RSC	Ref Store Op Count
LIC	Total Load Operation Count	IOC	Increment Op Count
SIC	Total Store Operation Count	ITC	Number of Bytecode Instructions
ILC	Primitive Load Count	IAS	Is Abstract (0 or 1)
ISC	Primitive Store Count	INF	Is Interface (0 or 1)
LFI	Load Field Count	IE	Is Exception (0 or 1)
SFI	Store Field Count	II	Is Private (0 or 1)
IR	Is Protected (0 or 1)	AGE	Age (in days)
IP	Is Public (0 or 1)	MSB	Modification Status Since Birth
IIN	Is Inner Class (0 or 1)	NBC	Normalized Branch Count (size

			is bytecode instructions)
IK	Is Package Accessible (0 or 1)	ZOC	Zero Op Insn Count (byte code ops without operands)
ID	Is Deleted in Next Version (0 or 1)	NCN	New Count (new operation count)
IMN	Is Modified in Next Version (0 or 1)	MMC	Modified Metric Count
EVS	Evolution Status Compared to Previous Release (0 - Unchanged, 1 - Modified, 3 - New)	MMB	Modified Metric Count Since Birth
JUO	Java Util Out Degree Count (dependency measure)	IIC	Is Io Class (i.e. depends on IO code directly)
NOP	Number of Parameters Count	IGC	Is GUI Class (i.e. depends on GUI code directly)
RSZ	Raw Size (bytes)	GUD	Gui Distance (Distance from nearest GUI class)
NVS	Next Version Status (0 - Unchanged, 1 - Modified, 2 - Deleted)	CCE	Clustering Coeff (Graph measure)
USC	Usage Count	LAY	Layer (0, 1, 2 or 3)
BRS	RSN class is Born	MFR	Modification Frequency (Number of times modified)

TABLE IV FULL : PERFORMANCE COMPARATIVE 02 MÉTRIQUES CHOISIES ALÉATOIREMENT

Paire de métriques	AUC	Type
AGE-LIC	0,502	Blocker
AGE-LIC	0,500	critical/major
AGE-LIC	0,500	minor
AGE-LIC	0,503	trivial
AGE-LIC	0,500	Non bug issues
AMC-IIC	0,500	Blocker
AMC-IIC	0,500	critical/major
AMC-IIC	0,500	minor
AMC-IIC	0,500	trivial
AMC-IIC	0,500	Non bug issues
BRS-NOF	0,498	Blocker
BRS-NOF	0,501	critical/major
BRS-NOF	0,500	minor
BRS-NOF	0,498	trivial
BRS-NOF	0,499	Non bug issues
CAC-THC	0,499	Blocker

CAC-THC	0,499	critical/major
CAC-THC	0,500	minor
CAC-THC	0,502	trivial
CAC-THC	0,501	Non bug issues
CBC-LMCI	0,499	Blocker
CBC-LMCI	0,500	critical/major
CBC-LMCI	0,500	minor
CBC-LMCI	0,502	trivial
CBC-LMCI	0,500	Non bug issues
CC-RFC	0,501	Blocker
CC-RFC	0,500	critical/major
CC-RFC	0,501	minor
CC-RFC	0,499	trivial
CC-RFC	0,500	Non bug issues
CCC-IFC	0,499	Blocker
CCC-IFC	0,499	critical/major
CCC-IFC	0,502	minor
CCC-IFC	0,502	trivial
CCC-IFC	0,500	Non bug issues
CCE-IOC	0,500	Blocker
CCE-IOC	0,500	critical/major
CCE-IOC	0,500	minor
CCE-IOC	0,500	trivial
CCE-IOC	0,500	Non bug issues
CLC-FFC	0,499	Blocker
CLC-FFC	0,500	critical/major
CLC-FFC	0,501	minor
CLC-FFC	0,499	trivial
CLC-FFC	0,500	Non bug issues
DIT-LAY	0,500	Blocker
DIT-LAY	0,500	critical/major
DIT-LAY	0,500	minor
DIT-LAY	0,500	trivial
DIT-LAY	0,500	Non bug issues
DMB-ZFC	0,500	Blocker
DMB-ZFC	0,500	critical/major
DMB-ZFC	0,500	minor
DMB-ZFC	0,500	trivial
DMB-ZFC	0,500	Non bug issues
DMV-RSC	0,499	Blocker

DMV-RSC	0,500	critical/major
DMV-RSC	0,500	minor
DMV-RSC	0,499	trivial
DMV-RSC	0,500	Non bug issues
EC-LFI	0,503	Blocker
EC-LFI	0,500	critical/major
EC-LFI	0,500	minor
EC-LFI	0,499	trivial
EC-LFI	0,499	Non bug issues
EODC-EVD	0,499	Blocker
EODC-EVD	0,500	critical/major
EODC-EVD	0,500	minor
EODC-EVD	0,499	trivial
EODC-EVD	0,500	Non bug issues
EVD-NOP	0,503	Blocker
EVD-NOP	0,499	critical/major
EVD-NOP	0,501	minor
EVD-NOP	0,499	trivial
EVD-NOP	0,500	Non bug issues
EVS-AGE	0,500	Blocker
EVS-AGE	0,500	critical/major
EVS-AGE	0,500	minor
EVS-AGE	0,500	trivial
EVS-AGE	0,500	Non bug issues
FFC-NOM	0,499	Blocker
FFC-NOM	0,500	critical/major
FFC-NOM	0,501	minor
FFC-NOM	0,499	trivial
FFC-NOM	0,500	Non bug issues
FMC-LMCE	0,499	Blocker
FMC-LMCE	0,500	critical/major
FMC-LMCE	0,500	minor
FMC-LMCE	0,502	trivial
FMC-LMCE	0,499	Non bug issues
IAC-AGE	0,500	Blocker
IAC-AGE	0,500	critical/major
IAC-AGE	0,500	minor
IAC-AGE	0,500	trivial
IAC-AGE	0,500	Non bug issues
IAS-SCC	0,500	Blocker

IAS-SCC	0,500	critical/major
IAS-SCC	0,500	minor
IAS-SCC	0,500	trivial
IAS-SCC	0,500	Non bug issues
ICC-SIC	0,498	Blocker
ICC-SIC	0,500	critical/major
ICC-SIC	0,500	minor
ICC-SIC	0,504	trivial
ICC-SIC	0,500	Non bug issues
IDC-LIC	0,502	Blocker
IDC-LIC	0,500	critical/major
IDC-LIC	0,500	minor
IDC-LIC	0,503	trivial
IDC-LIC	0,500	Non bug issues
IFC-IM	0,500	Blocker
IFC-IM	0,500	critical/major
IFC-IM	0,500	minor
IFC-IM	0,500	trivial
IFC-IM	0,500	Non bug issues
IIC-LRT	0,500	Blocker
IIC-LRT	0,500	critical/major
IIC-LRT	0,500	minor
IIC-LRT	0,500	trivial
IIC-LRT	0,500	Non bug issues
ILC-AMC	0,500	Blocker
ILC-AMC	0,500	critical/major
ILC-AMC	0,500	minor
ILC-AMC	0,498	trivial
ILC-AMC	0,500	Non bug issues
IM-LMCE	0,499	Blocker
IM-LMCE	0,501	critical/major
IM-LMCE	0,500	minor
IM-LMCE	0,502	trivial
IM-LMCE	0,499	Non bug issues
IMC-IAS	0,500	Blocker
IMC-IAS	0,500	critical/major
IMC-IAS	0,500	minor
IMC-IAS	0,500	trivial
IMC-IAS	0,500	Non bug issues
IMN-MCI	0,500	Blocker

IMN-MCI	0,500	critical/major
IMN-MCI	0,500	minor
IMN-MCI	0,500	trivial
IMN-MCI	0,500	Non bug issues
INC-RSZ	0,501	Blocker
INC-RSZ	0,500	critical/major
INC-RSZ	0,500	minor
INC-RSZ	0,502	trivial
INC-RSZ	0,500	Non bug issues
INS-EC	0,501	Blocker
INS-EC	0,500	critical/major
INS-EC	0,499	minor
INS-EC	0,502	trivial
INS-EC	0,499	Non bug issues
IOC-MCI	0,500	Blocker
IOC-MCI	0,500	critical/major
IOC-MCI	0,500	minor
IOC-MCI	0,500	trivial
IOC-MCI	0,500	Non bug issues
IODC-EC	0,499	Blocker
IODC-EC	0,501	critical/major
IODC-EC	0,499	minor
IODC-EC	0,499	trivial
IODC-EC	0,500	Non bug issues
IP-YMC	0,504	Blocker
IP-YMC	0,500	critical/major
IP-YMC	0,501	minor
IP-YMC	0,503	trivial
IP-YMC	0,500	Non bug issues
ISC-LFI	0,504	Blocker
ISC-LFI	0,500	critical/major
ISC-LFI	0,500	minor
ISC-LFI	0,503	trivial
ISC-LFI	0,500	Non bug issues
ITC-IAC	0,501	Blocker
ITC-IAC	0,500	critical/major
ITC-IAC	0,499	minor
ITC-IAC	0,499	trivial
ITC-IAC	0,500	Non bug issues
LAY-RLC	0,499	Blocker

LAY-RLC	0,500	critical/major
LAY-RLC	0,500	minor
LAY-RLC	0,499	trivial
LAY-RLC	0,500	Non bug issues
LFI-IDC	0,503	Blocker
LFI-IDC	0,500	critical/major
LFI-IDC	0,500	minor
LFI-IDC	0,502	trivial
LFI-IDC	0,500	Non bug issues
LIC-IMN	0,500	Blocker
LIC-IMN	0,500	critical/major
LIC-IMN	0,501	minor
LIC-IMN	0,500	trivial
LIC-IMN	0,499	Non bug issues
LMCE-RLC	0,502	Blocker
LMCE-RLC	0,500	critical/major
LMCE-RLC	0,500	minor
LMCE-RLC	0,501	trivial
LMCE-RLC	0,500	Non bug issues
LMCI-EODC	0,499	Blocker
LMCI-EODC	0,500	critical/major
LMCI-EODC	0,500	minor
LMCI-EODC	0,502	trivial
LMCI-EODC	0,500	Non bug issues
LRT-IMN	0,500	Blocker
LRT-IMN	0,500	critical/major
LRT-IMN	0,500	minor
LRT-IMN	0,500	trivial
LRT-IMN	0,500	Non bug issues
LVC-PFC	0,502	Blocker
LVC-PFC	0,500	critical/major
LVC-PFC	0,499	minor
LVC-PFC	0,501	trivial
LVC-PFC	0,500	Non bug issues
MCC-SFI	0,504	Blocker
MCC-SFI	0,500	critical/major
MCC-SFI	0,499	minor
MCC-SFI	0,501	trivial
MCC-SFI	0,500	Non bug issues
MCE-LFI	0,501	Blocker

MCE-LFI	0,500	critical/major
MCE-LFI	0,500	minor
MCE-LFI	0,499	trivial
MCE-LFI	0,500	Non bug issues
MCI-MCE	0,499	Blocker
MCI-MCE	0,500	critical/major
MCI-MCE	0,500	minor
MCI-MCE	0,502	trivial
MCI-MCE	0,501	Non bug issues
MFR-NCN	0,498	Blocker
MFR-NCN	0,500	critical/major
MFR-NCN	0,500	minor
MFR-NCN	0,501	trivial
MFR-NCN	0,500	Non bug issues
MMB-MFR	0,503	Blocker
MMB-MFR	0,501	critical/major
MMB-MFR	0,499	minor
MMB-MFR	0,499	trivial
MMB-MFR	0,499	Non bug issues
MMC-ILC	0,497	Blocker
MMC-ILC	0,500	critical/major
MMC-ILC	0,499	minor
MMC-ILC	0,497	trivial
MMC-ILC	0,500	Non bug issues
MSB-NOM	0,499	Blocker
MSB-NOM	0,500	critical/major
MSB-NOM	0,499	minor
MSB-NOM	0,499	trivial
MSB-NOM	0,500	Non bug issues
NAC-SMC	0,500	Blocker
NAC-SMC	0,500	critical/major
NAC-SMC	0,500	minor
NAC-SMC	0,500	trivial
NAC-SMC	0,500	Non bug issues
NCN-MSB	0,502	Blocker
NCN-MSB	0,500	critical/major
NCN-MSB	0,500	minor
NCN-MSB	0,500	trivial
NCN-MSB	0,500	Non bug issues
NOC-SMC	0,500	Blocker

NOC-SMC	0,500	critical/major
NOC-SMC	0,500	minor
NOC-SMC	0,500	trivial
NOC-SMC	0,500	Non bug issues
NOD-SFC	0,500	Blocker
NOD-SFC	0,500	critical/major
NOD-SFC	0,500	minor
NOD-SFC	0,500	trivial
NOD-SFC	0,500	Non bug issues
NOF-IMN	0,499	Blocker
NOF-IMN	0,500	critical/major
NOF-IMN	0,501	minor
NOF-IMN	0,499	trivial
NOF-IMN	0,500	Non bug issues
NOM-MCE	0,499	Blocker
NOM-MCE	0,499	critical/major
NOM-MCE	0,500	minor
NOM-MCE	0,502	trivial
NOM-MCE	0,501	Non bug issues
NOP-EC	0,502	Blocker
NOP-EC	0,501	critical/major
NOP-EC	0,499	minor
NOP-EC	0,501	trivial
NOP-EC	0,500	Non bug issues
NVS-FFC	0,500	Blocker
NVS-FFC	0,500	critical/major
NVS-FFC	0,500	minor
NVS-FFC	0,500	trivial
NVS-FFC	0,500	Non bug issues
ODC-SIC	0,506	Blocker
ODC-SIC	0,500	critical/major
ODC-SIC	0,501	minor
ODC-SIC	0,498	trivial
ODC-SIC	0,499	Non bug issues
PFC-MMC	0,499	Blocker
PFC-MMC	0,500	critical/major
PFC-MMC	0,499	minor
PFC-MMC	0,499	trivial
PFC-MMC	0,500	Non bug issues
PMC-TCC	0,501	Blocker

PMC-TCC	0,500	critical/major
PMC-TCC	0,501	minor
PMC-TCC	0,499	trivial
PMC-TCC	0,500	Non bug issues
RFC-LVC	0,501	Blocker
RFC-LVC	0,501	critical/major
RFC-LVC	0,499	minor
RFC-LVC	0,499	trivial
RFC-LVC	0,499	Non bug issues
RLC-YMC	0,503	Blocker
RLC-YMC	0,500	critical/major
RLC-YMC	0,500	minor
RLC-YMC	0,499	trivial
RLC-YMC	0,500	Non bug issues
RMC-TCC	0,500	Blocker
RMC-TCC	0,500	critical/major
RMC-TCC	0,500	minor
RMC-TCC	0,500	trivial
RMC-TCC	0,500	Non bug issues
RSC-AGE	0,499	Blocker
RSC-AGE	0,500	critical/major
RSC-AGE	0,500	minor
RSC-AGE	0,499	trivial
RSC-AGE	0,500	Non bug issues
RSZ-FMC	0,499	Blocker
RSZ-FMC	0,500	critical/major
RSZ-FMC	0,501	minor
RSZ-FMC	0,499	trivial
RSZ-FMC	0,500	Non bug issues
SCC-DMV	0,500	Blocker
SCC-DMV	0,500	critical/major
SCC-DMV	0,500	minor
SCC-DMV	0,500	trivial
SCC-DMV	0,500	Non bug issues
SFC-MCE	0,499	Blocker
SFC-MCE	0,500	critical/major
SFC-MCE	0,500	minor
SFC-MCE	0,499	trivial
SFC-MCE	0,500	Non bug issues
SFI-EVD	0,499	Blocker

SFI-EVD	0,500	critical/major
SFI-EVD	0,500	minor
SFI-EVD	0,499	trivial
SFI-EVD	0,500	Non bug issues
SIC-TIC	0,500	Blocker
SIC-TIC	0,501	critical/major
SIC-TIC	0,500	minor
SIC-TIC	0,498	trivial
SIC-TIC	0,498	Non bug issues
SMC-LRT	0,500	Blocker
SMC-LRT	0,500	critical/major
SMC-LRT	0,500	minor
SMC-LRT	0,500	trivial
SMC-LRT	0,500	Non bug issues
TCC-INS	0,501	Blocker
TCC-INS	0,500	critical/major
TCC-INS	0,500	minor
TCC-INS	0,502	trivial
TCC-INS	0,499	Non bug issues
THC-FMC	0,499	Blocker
THC-FMC	0,500	critical/major
THC-FMC	0,500	minor
THC-FMC	0,499	trivial
THC-FMC	0,500	Non bug issues
TIC-THC	0,501	Blocker
TIC-THC	0,500	critical/major
TIC-THC	0,499	minor
TIC-THC	0,499	trivial
TIC-THC	0,500	Non bug issues