Software Fault Severity Prediction Using Git History Metrics and Commits

Herimanitra Ranaivoson, Mourad Badri*

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

* Corresponding author. Tel.: +1 (819) 692-8958; email: Mourad.Badri@uqtr.ca
Manuscript submitted June 15, 2021; accepted August 15, 2021.
doi: 10.17706/jsw.17.2.36-47

Abstract: In this paper, we propose new software agnostic metrics extracted from Git history. We compared the proposed metrics to many traditional code-based metrics in terms of fault severity prediction. We used three Machine Learning Algorithms (Random Forest, SVM and Multilayer Perceptron) to build the prediction models. We used data (source code, source code metrics, fault severity information) collected from three different data sources. Results show that the proposed software agnostic metrics perform better in terms of fault severity prediction compared to traditional code-based metrics. They were able to achieve 84% of accuracy in fault severity prediction. We also introduced some terms extracted from commits and showed their effectiveness for fault severity classification.

Key words: Bug tracking system, commit messages, fault severity classification, git metrics, machine learning.

1. Introduction

Software faults are, in most cases, inevitable when developing software systems, particularly complex ones. Having a good software fault prediction model is highly desirable to effectively perform tests prioritization on high-risk components, even more, when the model is independent of the programming language used. In this paper, we propose new software agnostic metrics derived from Git history. We compared the proposed metrics to many traditional code-based metrics (Object-Oriented metrics, Complexity metrics, Graph metrics, etc.) proposed in the literature in terms of fault severity prediction. Fault severity was categorized in five different levels: Blocker, Critical/Major, Minor, Trivial and non-bug issues. We used three Machine Learning Algorithms (Random Forest, SVM and Multilayer Perceptron) to build the prediction models. We used data (source code, source code metrics, fault severity information) collected from three different data sources. We investigated the following main research question (RQ):

Can we build software agnostic *metrics, extracted from Git histories, that can predict fault severity better than traditional source code-based metrics?*

Using metrics that are independent of the source code promotes, in fact, reusability across different software projects. A software metric is considered as software agnostic when its (definition and) calculation does not depend on the programming language used [1]. Radjenovic *et al*. [2] presented a comprehensive literature review on metrics used in software defect prediction. Metrics were often compared in terms of

fault-proneness prediction using different Machine Learning algorithms. However, few papers have paid attention to the reusability of these metrics across different software projects. To the best of our knowledge, only few works have been done on the benefits of having a metric that is independent of the used programming languages [3].

Git commit data histories have been used to analyze software defects [4], and metrics derived from Git histories (also known as change metrics) have already shown to be effective when predicting software faults [5]. Object-Oriented (OO) and complexity metrics heavily depend on the software project as they are calculated from the source code characteristics. Metrics extracted from Git commit histories are unrelated to the source code characteristics, and seem to have a good potential power for fault proneness prediction according to Eyolfson *et al*. [6]. Zhang *et al.* [7] have, for example, shown that commit terms were able to infer change types in a set of software projects that have Git histories. This will support more effective tests prioritization on high-risk components, for different projects, thus allowing to fix faults having higher severity as a priority.

2. Related Work

In this section, we present a brief summary of a few related studies. First of all, our study is related to the work done by Kim *et al*. [8]. The proposed agnostic metrics verify the principles that they defined. Actually, Kim et al. [8] tried to categorize faults that software developers often make when operating changes on their code. Several situations were discussed: (1) Changed-entity locality ("If an entity was changed recently, it will tend to introduce faults soon"), (2) New-entity locality ("If an entity has been added recently, it will tend to introduce faults soon"), (3) Temporal locality ("If an entity introduced a fault recently, it will tend to introduce other faults soon"), and (4) Spatial locality ("If an entity introduced faults recently, "nearby" entities (in the sense of logical coupling) will also tend to introduce faults soon").

We were also motivated by the work done by Barnett *et al*. [9]. They found a significant correlation between the size of commits and software defects. Moreover, in their seminal paper, Zhang *et al.* [7] hypothesized a relationship between textual description of commits and the reason of changes.

3. Research Methodology

Our methodology is organized around the following stages: data collection, data construction, metrics construction and finally metrics evaluation and comparison. The last part of our study includes the construction and comparison of the different prediction models.

3.1. Data Collection

The goal of data collection is to gather all data needed to facilitate the comparison, in terms of fault severity prediction, between the proposed agnostic metrics and the traditional source code-based metrics we have selected. In order to perform all our experiments, we collected and prepared three different data sources:

The first data source is the source code of the projects themselves (Apache Struts I and II: 378 java classes, and ActiveMQ: 591 java classes). They have been collected from their Github repository. From these projects, historical commits were extracted using some Git commands. The second data source is from Vasa [10], who compiled a comprehensive set of metrics (78 in total) related to Apache software projects. They are essentially composed of metrics used in the literature. These metrics were used to benchmark our proposed metrics. The third data source comes from the JIRA platform. It is composed of issues and faults with their severity. This data source is the source of our target variable.

3.2. Data Construction

First, we have the selected software projects coming from the Helix software evolution datasets (recalled

D1 for brevity) compiled by Vasa [10], which are in our knowledge the most comprehensive set of metrics available on Apache software projects.

Second, we have data on annotated issues collected from the JIRA website for the projects we analyzed (recalled D2 for brevity). What we did was simply to recode non-bug issues as an entire category. From that, we ended up with five distinct levels of severity as mentioned earlier (Blocker, Critical/Major, Minor, Trivial and non- bug issues).

Third, we extracted commit messages (recalled D3 for brevity) from Git histories of the Apache software projects (ActiveMQ, Struts I and II) we used. Git histories are unstructured datasets that we cleaned into standard datasets using some text mining techniques such as lemmatization, stemming, etc. Git histories include deletions and additions for any commits made on any files (classes) of each repository project. They allowed us to define our metrics considering: (1) Changed-entity locality ("If an entity was changed recently, it will tend to introduce faults soon"), (2) New-entity locality ("If an entity has been added recently, it will tend to introduce faults soon"), (3) Spatial locality ("If an entity introduced faults recently, "nearby" entities (in the sense of logical coupling) will also tend to introduce faults soon"), and (4) Temporal locality ("If an entity introduced faults recently, it will tend to introduce other faults soon").

	raw	software
254775	58\t0\twebapps/starter/src/webapp/WEB-INF/web.xml	struts
254776	1\t0\twebapps/starter/src/webapp/index.jsp	struts
254777	7\t0\twww/index.html	struts
254778	4\t0\twww/project_tools.html	struts
254779	--f0e3ff3--Wed Feb 22 05:16:10 2006 +0000--Gar...	struts

Fig. 1. Raw commits.

By default, this last dataset (Git messages) is similar to what is given in Fig. 1, where all the information on commits are compressed into a single column called "raw." A text transformation (data cleaning) was applied in order to make the commits exploitable. After transformation, we got a tabular dataset similar to what is given in Fig 2., where we can find the Java classes and their corresponding tests. With this dataset, we computed days elapsed since the last commit for any given Java class and all intermediary calculus needed to compute our final software agnostic metrics. The final dataset was obtained by merging D1, D2 and D3 (as defined above) using the following keys: software name, class name, date of commits, date of release and date of issues.

test_filename	days_sinceLastModif	message
truts2/views/ut...	145.000000	WW-1709 fixed nullpointer when scheme is null
truts2/views/ut...	145.000000	WW-1709 fixed nullpointer when scheme is null
ruts2/dispatch...	25.375000	Reformatting default action mapper WW-1349
ruts2/dispatch...	233.666667	WW-1410 - Provide a hook for FilterDispatche...
/struts2/dispat...	201.208333	XW-399 - renamed xwork packaging due to xwor...

Fig. 2. Commit data set.

3.3. Metrics Construction

The metrics we defined are:

Ranking Index: which weights the contribution of the testing effort undertaken on a class and also time elapsed since last changes to the likelihood of producing bugs in future. It is defined as follows:

$$\delta T_{ct}^{1/2}\, \delta L_{c\prime t}^{1/2}$$

We have two members in the formula of *Ranking Index*: δT_{ct} is the time elapsed since the latest modification for a class *c*. $\delta L_{c\prime t}$ is the code length difference in the test class *c'* corresponding to the class *c*. If a class does not have a test, this will be set to one by default.

Bug Index: a binary indicator variable to point out whether a class has experienced bug(s) from the past.

Interpretation of the *Ranking Index*: The smaller the value of the *Ranking Index* of a class *c* at a given time *t* is, the more the class is prone to faults. Conversely, the higher the value of the *Ranking Index* of a class *c* at a given time *t* is, the lower the risk of the class to be fault-prone. In fact, we capture: 1) Changed-entity locality because whenever a commit is made on a class c, δT_{ct} will become smaller thus lowering the Ranking Index. 2) One or more classes may be concerned by a single commit, which should capture spatial locality as well. 3) We capture new-entity locality as well because whenever a class is created, the *Ranking Index* will take the smallest possible value of 1. And 4) We capture temporal locality by using a *Bug Index* (as a separate variable) taking 1 if a class c has experienced a bug since its creation and 0 if not. We introduce testing effort locality as well by adding in the formula of the *Ranking Index*, a term: $\delta L_{c\prime t}$. This term increases the *Ranking Index* if a change has been made simultaneously on the test class *c'* of a class *c*. This new addition should capture the testing effort given the change made on its class.

3.4. Parameters in the Formula

The Cobb-Douglas [11] function is a generic formula defined as follow:

$$\delta T_{ct}^{\alpha}\, \delta L_{c\prime t}^{1-\alpha}$$

The parameter α = 1/2 of the *Ranking Index* has been carefully obtained after minimizing intra-class variance and further maximizing inter-class variance among our five distinct severity groups: Blocker, Critical and Major, Minor, Trivial and non-bug issues. The goal of choosing parameters by an algorithm is that we wanted a metric that best separates the different severity groups. Thus, we choose the *k-nearest neighbors* classification algorithm (*knn*) to select the best formula metric among the following candidates:

$$\textit{Formula 0: } \delta T_{ct}/\delta L_{c\prime t}$$
$$\textit{Formula 1: } \delta T_{ct}^{1/2}\, \delta L_{c\prime t}^{1/2}$$
$$\textit{Formula 2: } \delta T_{ct}^{2/3}\, \delta L_{c\prime t}^{1/3}$$
$$\textit{Formula 3: } \delta T_{ct}^{3/4}\, \delta L_{c\prime t}^{1/4}$$
$$\textit{Formula 4: } \delta T_{ct}^{1/3}\, \delta L_{c\prime t}^{2/3}$$
$$\textit{Formula 5: } \delta T_{ct}^{1/4}\, \delta L_{c\prime t}^{3/4}$$

These formulas vehicle the principles defined by [8] on the characteristics of a faulty component. These formulas are the same Cobb-Douglas function with different values of α.

Formula 0: we wanted to penalize classes where modification happened a very long time ago by the amount of modifications in its test class counterpart. A riskier class may have an imbalanced code delta compared to time elapsed since the last commit.

Formula 1 to Formula 5 are the same with different values of the parameter α. We varied α and selected the one that is optimal in terms of accuracy in classification. α and 1 − α are weights which can be interpreted as

contribution of time elapsed since last modification and testing effort undertaken on a given class c during the appearance of faults.

We ran a *knn* algorithm for each metric formula, and evaluated its accuracy at predicting our target variable on a randomly chosen test dataset. According to the results (Table 1), Formula 1 is the best one. It separates all different categories of bugs with an accuracy of 88% (Table 1, Line-4, col-6 and Line-8, col-3). It means that modification time and testing effort contribute equally to the appearance of the bug severity. *Formula 0* was competitive too, but gave relatively poor results on K=5 nearest neighbors votes (82% accuracy, Table I, Line-1).

Table 1. K-NN One Variable to Classify Fault Severity

Neighbours	Metrics	Accuracy	Neighbours	Metrics	Accuracy
5	Formula0	82.22%	15	Formula3	65.97%
5	Formula1	86.89%	15	Formula4	65.97%
5	Formula2	65.97%	15	Formula5	65.97%
5	Formula3	65.97%	25	Formula0	88.14%
5	Formula4	65.97%	25	Formula1	88.13%
5	Formula5	65.97%	25	Formula2	65.97%
10	Formula0	88.13%	25	Formula3	65.97%
10	Formula1	88.18%	25	Formula4	65.97%
10	Formula2	65.97%	25	Formula5	65.97%
10	Formula3	65.97%	50	Formula0	88.17%
10	Formula4	65.97%	50	Formula1	87.95%
10	Formula5	65.97%	50	Formula2	65.97%
10	Formula0	88.16%	50	Formula3	65.97%
15	Formula1	87.98%	50	Formula4	65.97%
15	Formula2	65.97%	50	Formula5	65.97%

3.5. Performance Evaluation

In order to assess our metrics, we needed a target variable and some evaluation metrics. The target variable is the class of severity we downloaded from the JIRA website. The severity of faults has been categorized into five groups by the JIRA website. We labeled them as follows: Blocker bugs labeled as 1. Critical and major bugs labeled as 2. Minor bugs labeled as 3. Trivial bugs labeled as 4. Any other issues different from bugs are labeled as 9 in order to feed the data to the Machine Learning models.

4. Experimental Results and Discussion

In this section, the results of the different experiments we performed, in order to investigate our main research question from different perspectives, are presented and discussed. We trained our prediction models using the metrics in different conditions, and pre-releases as well as releases. We also used a two by two comparison of the metrics and lastly a variance explanation of each metric we compared.

4.1. Cross-project and Multi-project Comparison

A cross-project comparison consists of evaluating metrics on each software repository, while multi-project comparison consists of training models on a combination of all repositories. All of our experiments were conducted with a multi-project setup because we wanted as much data as possible and also to show that the *Ranking Index* and *Bug Index* metrics are really software agnostic. The results of our comparisons are summarized in Table 2.

In Table 2, 03 kinds of models (Random Forest, Support Vectors Machine and Multilayer Perceptron) were used to compare in isolation selected traditional metrics and our proposed metrics (Git features versus selected metrics from [10]). Each model and a set of feature combinations were used to assess the overall

performance (third column of Table 2) of the proposed metrics: *Ranking Index* and *Bug Index*. In multi-projects setting, the *Ranking Index* and *Bug Index* metrics, and the source code-based metrics we used both depicted similar performance for fault severity classification with an accuracy ranging from 70.9% to 71% (Table 2, lines 1-6).

Table 2. All OO and Traditional Metrics vs (Ranking Metric+ Bug Index)

Models	Features	Accuracy	Datasets
RandomForest	Traditional metrics	71.01%	all
RandomForest	Ranking metric + bug index	71.28%	all
SVM (linear kernel)	Traditional metrics	71.50%	all
SVM (linear kernel)	Ranking metric + bug index	71.29%	all
MLP	Traditional metrics	71.13%	all
MLP	Ranking metric + bug index	70.92%	all
RandomForest	Traditional metrics	66.19%	activeMQ
RandomForest	Ranking metric + bug index	52.56%	Struts
RandomForest	Ranking metric + bug index	72.08%	activeMQ

However, in a cross-project setting, Git features gave the best accuracy with 72%, while 66% for traditional metrics. Table 2 leads us to conclude that our proposed metrics can perform as good as the selected traditional metrics when predicting fault severity. They are significantly better in the case of the ActiveMQ project (bringing 5% more accuracy than traditional source code-based metrics).

4.2. Two by Two Comparison of Metrics

Table 3. 02 Random Metrics vs Ranking Metric + Bug Index

pair	AUC	type	pair	AUC	type
AGE-LIC	0.5	Blocker	YMC-MFR	0.50	Blocker
AGE-LIC	0.5	critical/major	YMC-MFR	0.50	critical/major
AGE-LIC	0.5	minor	YMC-MFR	0.50	minor
AGE-LIC	0.5	trivial	YMC-MFR	0.50	trivial
AGE-LIC	0.5	Non bug issues	YMC-MFR	0.50	Non bug issues
AMC-IIC	0.5	Blocker	ZFC-LMCI	0.50	Blocker
AMC-IIC	0.5	critical/major	ZFC-LMCI	0.50	critical/major
AMC-IIC	0.5	minor	ZFC-LMCI	0.50	minor
AMC-IIC	0.5	trivial	ZFC-LMCI	0.50	trivial
AMC-IIC	0.5	Non bug issues	ZFC-LMCI	0.50	Non bug issues
BRS-NOF	0.5	Blocker	ZOC-IOC	0.50	Blocker
BRS-NOF	0.5	critical/major	ZOC-IOC	0.50	critical/major
BRS-NOF	0.5	minor	ZOC-IOC	0.50	minor
BRS-NOF	0.5	trivial	ZOC-IOC	0.50	trivial
BRS-NOF	0.5	Non bug issues	ZOC-IOC	0.50	Non bug issues
CAC-THC	0.5	Blocker	ranking0-bugIndex	0.97	Blocker
CAC-THC	0.5	critical/major	ranking0-bugIndex	0.46	critical/major
CAC-THC	0.5	minor	ranking0-bugIndex	0.52	minor
CAC-THC	0.5	trivial	ranking0-bugIndex	0.58	trivial
CAC-THC	0.5	Non bug issues	ranking0-bugIndex	0.50	Non bug issues

From previous results summarized in Table 2, it can be seen that our *Ranking Index* and *Bug Index* metrics can achieve similar performance as the 78 traditional metrics compiled by Vasa [10], which we used in our study. However, it does not answer whether they are better than traditional metrics at predicting the different severities of a fault. In order to answer this question, we have to compare metrics individually. Actually, when comparing 02 metrics versus our selection of metrics (78), we don't know which of them drive the results of the latter. That's the reason why we performed a 02 by 02 comparison of *Ranking Index* and *Bug Index* metrics instead of a raw comparison as we did previously. By conducting our experiments in two by two comparison (Table 3), we noticed that for every kind of severity, except for bug blocker, both 02 random metrics versus our proposed metrics behave like a flipping coin (AUC of 0.5). For these categories, the Random Forest (RF) model does not bring any value added at all. However, for bug blocker classification, the *Ranking Index* and

Bug Index metrics achieve an excellent performance with an AUC of 0.97 (bottom of Table 3). From these experiments, we can conclude that the proposed metrics, *Ranking Index* and *Bug Index*, outperform selected metrics compiled by Vasa [10] for bug blocker classification.

4.3. Another Performance Metric with Random Forest

Last experiment showed that *Ranking Index* and *Bug Index* metrics are excellent at predicting bug blocker with an AUC of 0.97. However, it does not mean that they are superior metrics for bug severity classification tasks. Thus, we used RF to assess Mean Decrease Gini Score of all variables in order to see top variables that contribute most in accuracy of predictions. This process should help us to decide which metrics are the most important for bug severity classification. Mean Decrease Gini Score is an index of impurity of a node in tree-based machine learning models [12]. It indicates how important a predictor is in classifying the output response (target). It measures the importance of a variable or a set of features in terms of node impurity. A node is considered pure if its output prediction is correct at 100%. The more a node is discriminative (its output tends to 100%), the more it is pure, which indicates an accurate prediction. In order to obtain the Mean Decrease Gini Score, we picked up the RF model built previously, see Table III (with all the metrics, using a multi-project dataset, line-2). With this model, we plotted the graphical representation of the Mean Decrease Gini Score (as can be seen in Fig. 3).

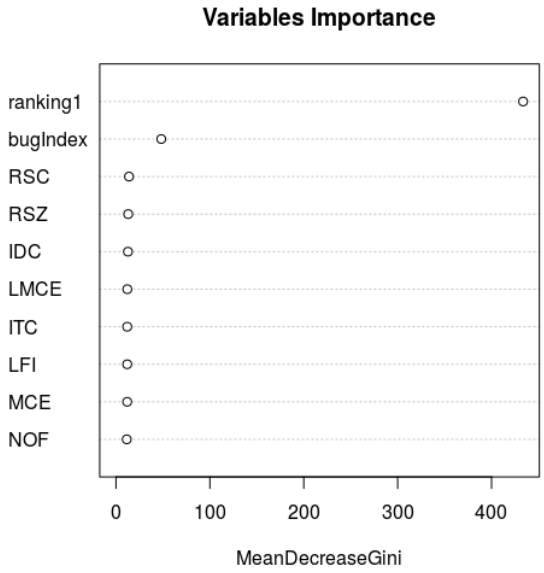


Fig. 3. Variable Importance.

The *Ranking Index* and *Bug Index* metrics appear to be the most important variables among all of our metrics combined in the prediction of bugs severity. They are followed by traditional metrics such as Ref Store Op Count (RSC), Raw Size (RSZ) and In Degree Count (IDC). We also evaluated the proposed agnostic metrics to predict bugs at release date using RF (Table 4). The reported accuracy is between 61% and 75% depending on the release date. These results remain consistent with those reported in Table 3.

Summary

We evaluated the proposed metrics (*Ranking Index* and *Bug Index*) in comparison to the 78 traditional metrics as compiled by Vasa [10] in a: 1) Software project comparison using Machine Learning models. 2) 02 vs all comparison which consists of running models using all traditional metrics as features and another set of models using *Ranking Index* and *Bug Index* metrics solely. 3) 02 by 02 random comparison which consists of selecting any 02 random metrics among traditional ones and using them in a model which will be compared

to a model using the *Ranking Index* and *Bug Index* metrics solely. And finally 4) A Mean Decrease Gini Score was used to assess top predictors for fault severity classification among all metrics we analyzed. Results (Fig. 3) show that the *Ranking Index* and *Bug Index* metrics are among the most important features for fault severity prediction.

These results suggest that the proposed agnostic metrics are interesting alternatives to the usual source code-based metrics found in the literature of empirical software engineering. Our results show that they are excellent at predicting bug blocker.

4.4. Comparison of Metrics Using Pre-release vs Release Periods

Several studies such as the systematic literature review of Radjenovic *et al*. [2] reported that process metrics are superior in post release. Thus, we decided to investigate further the *Ranking Index* and *Bug Index* metrics using our best model (Random Forest) by evaluating them in pre-release and release periods. In fact, our dataset contains time series, so we can split it by date. Moreover, we already have different dates of release which made our split easier. In this section, we tried to validate fault severity prediction using the last release of each project as a validation set.

Table 4. Accuracy of EF for each Release of ActiveMQ Software

Models	PReleaseSize	ReleaseDate	ReleaseSize	Accuracy
RF(2000 trees)	3240	2008-05-01	8730	69.68%
RF(2000 trees)	11970	2008-11-06	14040	65.23%
RF(2000 trees)	26010	2009-10-08	16200	61.5%
RF(2000 trees)	42210	2010-03-18	3780	74.47%
RF(2000 trees)	45990	2010-04-26	2790	61.16%

Our results (Table 4) suggest that the performance of the proposed metrics remains consistent in pre-release and release periods. Actually, they can reach an AUC of 74.47% which is 3% better than the same model using normal cross-validation (CV) split (Table III). We also used a different setup (binary classification task) to distinguish among severe bugs or others to see if predictions are improved. Our results suggest that we gain about 2% in accuracy by doing a binary classification task. In Table 5, we can see noticeable gain in accuracy (78.67%) compared to the results of Sharma et al. (75.83%) [13] in the best case of their experimentations.

Table 5. Accuracy of Different Classifiers Using all Metrics: 78 Metrics + Ranking and Bug Index

Models	SetUp	BinaryTask
RandomForest+	7000 trees	78.67%
Multilayer Perceptron++	200 neurons, 30 hidden layers	-
NaiveBayes+++	5 folds Cross-Validation	60%
SVM+*	5 folds Cross-Validation with linear kernel	-

Summary

Combining datasets from different projects does not alter results as one could expect. In contrast, we gained in accuracy compared to [13] among many others. Again, here our proposed Git features (*Ranking Index* and *Bug Index* metrics) are consistent in their accuracy for fault severity classification.

4.5. Commit Terms as Features in Fault Severity Classification

In previous experimentations, we mainly focused on building our Git features (*Ranking Index* and *Bug Index*) and comparing them to the traditional source code-based metrics we selected. In this section, we will introduce Git commit keywords for fault severity classification. Git commit messages are another feature of Git that can produce software agnostic metrics for fault severity classification.

We assume that the comments written by developers on the changes applied on the code portion they are

working on contains relevant elements that can be used to predict future bugs. We opened up the possibility that commit messages contain hidden information that can help to predict the severity of future bugs. We are not the first who investigated this issue as can be seen in [4] and [6] who showed a significant correlation between commit message length and fault-proneness.

Our approach consists of transforming commit messages into features that models understand. After this transformation, we will then use those features to build Machine Learning models and evaluate their predictive performance for fault severity classification. Transformation process involves standard text pre-processing including stop words removal from English dictionary, lemmatization, etc. Before using them to build Machine Learning models, we transformed words into vectors using one hot encoding and TF-IDF weighting techniques which are commonly used in text preparation for predictive analytics [14].

TF-IDF Weighting

In order to capture the importance of each word in the commit corpus data set, we used the TF-IDF weighting technique. In fact, rare words that belong only to a few documents and characterize them should not be treated like common words that appear within all documents.

Table 6. Illustrations of an One Hot Encoded Matrix with Phrase: "Adding Component Reduce Time Loading"

commit	adding	component	reduce	loading	time	fix
commit1	1	1	1	1	1	0
commit2	0	0	0	0	1	0
commit3	0	1	0	0	0	1

For each keyword, we applied one hot encoding as seen in Table 5, then assigned weights to each individual cleaned words (terms) of commit messages using Term Frequency-Inverse Document Frequency formula [14] defined as follows:

$$TFIDF_{id} = f_{id} log(\frac{|D|}{d_k, m_i \subset d_k})$$

This weighting technique allows to give more importance to term m_i, which is both frequent in the entire document or corpus (by its frequency $f_{id} = \frac{|m_i|}{\sum_{j=1}^{|d|} m_{jd}}$) and rare in other commit messages (this is captured by the logarithmic part of the equation defined above) relative to the total number of commit messages (Corpus) |D|.

Table 7. Accuracy of Different Classifiers with Commits and Other Combinations

Models	SetUp	Accuracy
RandomForest+	Commits + Ranking & bug index	84.56%
RandomForest++	Commits reduced by half using PCA + Ranking & bug index	85.40%
RandomForest+++	Commits only	80.35%
Implementation of Barnet et al. in scikit-learn+*	Commits only using Bayesian model (similar to Barnett et al.)	65.94%

After the transforming process, we ended up with more than 1200 cleaned words, which we used to build our RF model. We trained the model with all Git metrics we collected (Table 5, line-1), then we used a dimension reduction technique to lower the number of features (Table7, line-2) and finally we used solely words from commits (Table 5, line-3). According to our results, text features appear to have more predictive power than any traditional metric we used in this study with at least 80% accuracy. The best accuracy (84%) was obtained using a combination of agnostic metrics: key word commits, *Bug Index* and *Ranking Index* (Table VII, line-1). Furthermore, the *Ranking Index* and *Bug Index* metrics are among the top 5 most important features for the severity prediction according to Fig. 4.

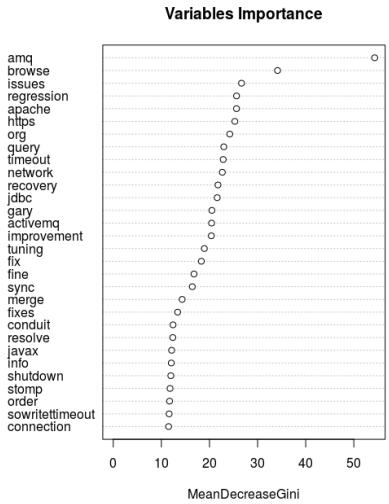


Fig. 4. Variable Importance of RF models in Table 7.

5. Threats to Validity

First, the way we gathered datasets may introduce issues. In fact, our metrics have different frequencies of calculation. Metrics compiled by Vasa [10], used as a benchmark, are calculated for every major version while our *Ranking Index* and *Bug Index* metrics are computed for every change made in the source code. Thus, this difference in frequency of calculation may introduce bias in the results. Second, unbalanced classes of severity may prevent models we have used to learn the maximum amount of information on each different class. Thus, our results may not be generalized for every dataset.

6. Conclusion and Future Work

In this study, we introduced and evaluated two software agnostic metrics derived from Git histories in order to predict fault severity. Our results showed that the metrics we proposed, *Ranking Index* and *Bug Index*, are better at predicting Bug Blocker (AUC of 0.97, Table 3, col-2) compared to our selection of source code-based metrics extracted from Vasa [10]. However, like these traditional metrics, *Ranking Index* and *Bug Index* metrics gave poor performance for the other categories of severity. Our study showed clearly that it is possible to extract agnostic features from Git history for fault severity classification. In our future work, we plan to extend our study by considering other software systems, and investigating the use of Deep Learning.

Conflict of Interest

The authors declare no conflict of interest.

Authors Contributions

Mourad Badri: Project definition, Conceptualization, Methodology, Writing - Review & Editing, Supervision, Project administration

Ranaivoson Herimanitra : Methodology, Software, Data Collection, Formal data analysis, Investigation, Writing - Original Draft, Writing - Review & Editing, Visualization

Acknowledgment

This work was partially supported by NSERC (Natural Sciences and Engineering Research Council of

Canada).

References

[1] Conejero, J. M. F., Alessandro, G., Juan, H., & Elena, J. (2009). Early crosscutting metrics as predictors of software instability. *Objects, Components, Models and Patterns*. Berlin, Heidelberg: Springer Berlin Heidelberg.

[2] Radjenovic, D., Marjan, H., Richard, T., & Ales, Z. (2013). Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, *55(8)*, 1397–1418.

[3] Christoffer, R., Ben, G., & Emad, S. (2015). Commit guru: Analytics and risk prediction of software commits. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA.

[4] Xu, J., Yan, L., Wang, F., & Ai, J. (2019). A github-based data collection method for software defect prediction. (2019). *Proceedings of the 6th International Conference on Dependable Systems and Their Applications* (pp. 100-108).

[5] Choudhary, G. R., Kumar, S., Kumar, K., Mishra, A., & Catal, C. (2018). Empirical analysis of change metrics for software fault prediction. *Computers and Electrical Engineering*, *67*, 15-24.

[6] Eyolfson, J., Lin, T., & Patrick, L. (2014). Correlations between bugginess and time-based commit characteristics. *Empirical Software Engineering*, *19(4)*, 1009–1039.

[7] Zhang, F., Audris, M., Iman, K., & Ying, Z. (2014). Towards building a universal defect prediction model. *Proceedings of the 11th Working Conference on Mining Software Repositories* (pp. 182–191). New York, NY, USA: ACM.

[8] Kim, S., Zimmermann, T., Whitehead, E. J. J., & Zeller, A. (2007). Predicting faults from cached history. *Proceedings of the 29th International Conference on Software Engineering* (pp. 489–498).

[9] Barnett, J. G., Gathuru, C. K., Soldano, L. S. & McIntosh, S. (2016). The relationship between commit message detail and defect proneness in java projects on github. *Proceedings of the* 2016 IEEE/ACM *13th Working Conference on Mining Software Repositories* (MSR) (pp. 496–499).

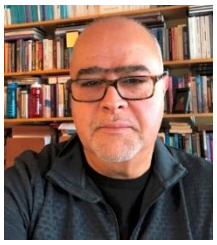
[10] Vasa, R. (2010). Growth and change dynamics in open source software systems. *Faculty of Information and Communication Technologies*, 254.

[11] Cobb, C. W., & Paul, H. D. (2021). A theory of production. *The American Economic Review*, *18*.

[12] Liaw, A., & Matthew, W. (2001). Classification and regression by random forest. *Forest 23* (November).

[13] Sharma, G., Sumit, S., & Shruti, G. (2015). A novel way of assessing software bug severity using dictionary of critical terms. *Procedia Computer Science*, *70*, 632–639.

[14] Luhn, H. P. (1957). A statistical approach to mechanized encoding and searching of literary Information. *IBM Journal of Research and Development, 1(4)*, 309–217.



Mourad Badri is a full professor of computer science at the Department of Mathematics and Computer Science of the University of Quebec at Trois-Rivières (Canada). He holds a PhD in computer science (software engineering) from the National Institute of Applied Sciences in Lyon, France. His main areas of interest include object and aspect-oriented software engineering, software quality attributes, software testing, refactoring, software evolution and application of machine learning in software engineering.

Ranaivoson Herimanitra is a graduate student (master - applied mathematics and computer science) from the Department of Mathematics and Computer Science of the University of Québec at Trois-Rivières (Canada). His main research fields are software engineering, big data, data science, and application of machine learning in software engineering.