

Predicting Defectiveness of Software Patches

Behjat Soltanifar
Data Science Lab, Ryerson
University
Canada
behjat.soltanifar@

Atakan Erdem
Data Science Lab, Ryerson
University
Canada
atakan.erdem@

Ayse Bener
Data Science Lab, Ryerson
University
Canada
ayse.bener@ryerson.ca

ABSTRACT

Context: Software code review, as an engineering best practice, refers to the inspection of the code change in order to find possible defects and ensure change quality. Code reviews, however, may not guarantee finding the defects. Thus, there is a risk for a defective code change in a given patch, to pass the review process and be submitted.

Goal: In this research, we aim to apply different machine learning algorithms in order to predict the defectiveness of a patch after being reviewed, at the time of its submission.

Method: We built three models using three different machine learning algorithms: Logistic Regression, Naïve Bayes, and Bayesian Network model. To build the models, we consider different factors involved in review process in terms of Product, Process and People (3P).

Results: Our empirical results show that, Bayesian Networks is able to better predict the defectiveness of the changed code with 76% accuracy.

Conclusions: Predicting defectiveness of change code is beneficial in making patch release decisions. The Bayesian Network model outperforms the others since it captures the relationship among the factors in the review process.

CCS Concepts

•Software and its engineering → Software management; •Software validation → *Software defect analysis*;

Keywords

Code review, Code Review Quality, Software Patch Defectiveness, Defect Prediction

1. INTRODUCTION

Software code review, as an engineering best practice [14], refers to the inspection of code change in order to find possible defects and ensures the quality of a software system [2]. Review process can be considered as part of bug life-cycle. Taking Mozilla bug life-cycle in Bugzilla as an exam-

ple, the description of bug life-cycle consists of the following: In Bugzilla, a bug report usually remains unconfirmed until a trigger confirms that a further investigation is needed. Then, the bug is assigned to, or picked by a developer to be fixed ¹. After fixing the bug, developer is required to assign the code change (patch) to a reviewer, as a review request. After reviewer conducts the review and approves the changes, the patch can be integrated to the source code. In software projects, code review process is used as an approval stage before integrating the code changes into the mainline in order to ensure the quality of code changes [11] [40].

Michael Fagan originally proposed software inspection model in 1976 [14]. The traditional approach of code review, known as code inspection, involves an in-person formal meeting of reviewers. This method and its different versions of it is used for several years as a way to ensure the overall software quality. However, it could not be adopted to the globally distributed software development systems [43] that now use modern code review (MCR) process. Formal inspections, such as code inspection, known as heavyweight processes, include a group of experts, distinguished group meetings, and several formal preparations and requirements in advance [14]. Modern Code Review (MCR), on the other hand, involves less formal requirements. In this model, review process time is shorter, and supporting tools, such as web based tools, are often used [3] [4]. According to this process, a developer submits a new review request to code review repository, and assigns an authorized developer (reviewer) to review and approve the changes. Today, MCR is dominantly used in practice [25], in both industrial and open source software (OSS) projects [4].

Although, according to the literature, finding defects is the primary goal of software code review, various research show that the output of current review process does not often meet this goal. The reviewers do not often find defects that are related to the functionality of the product due to inefficient use of review process in the workflows of developers [3] [11]. This calls for further research and investigation in methodologies offering improvement in this process [3] [11].

In this research, our purpose is to empirically investigate how process, product and people related attributes of code review process enable the prediction of patch defectiveness after code review task is conducted, and before patch is submitted to the mainline. Also, applying three commonly used machine learning techniques, we investigate which algorithm performs better to address this problem, and what would be

¹<http://www.bugzilla.org/docs/tip/html/lifecycle.html>. Accessed: 2014-05-18.

the possible reasons of having such results. The output of this study is beneficial for researchers in getting a better insight of review process and how to improve the overall quality in terms of providing less defect prone patches. This research is an initial step in building an algorithm to model the code review process by considering attributes of 3Ps as well as understanding the causalities among the attributes and process pieces. The ultimate goal is to provide reviewers with a predictive model which helps in making patch submission decision, checking if the patch meets the quality requirements for submission or if it requires more review. Therefore, our research question is stated as follow:

- RQ: How accurate can we predict defectiveness of patches?

The remainder of the paper is structured as follows. We discuss the related work in section 2. In section 3, we provide the methodology. The result of our analysis is reported in section 4. Section 5 is dedicated to the discussion of threats to validity of our results. We conclude our study in section 6 and also talk about future directions.

2. RELATED WORK

Several software projects, open source and proprietary organizations, utilize bug tracking systems (Bugzilla, MantisBT, Trac and etc.) in order to allow users, testers and developers submitting software issues they encounter [22]. These systems also aid developers to keep track of existing issues. A developer then either picks, or he/ she gets assigned to fix a bug. After the bug is fixed, corresponding developer should submit his/ her changed codes (patch) and assigns an authorized reviewer to check and approve the changed code. This is considered as an approval stage for integrating patches to the main repository. Integration is done by either reviewer or developer, depending on the company policy.

Michael Fagan originally proposed software inspection model in 1976 [14]. Unlike the formal code inspection which mandates strict criteria, Modern Code Review (MCR) requires fewer formal preparations. MCRs tends to have less number of people involved and eliminate in-person meetings. People involved in this process are source code change authors, and reviewers who assure the quality of the code change. A patch would be approved after the author implements reviewer's comments. The patch can then be committed and released to the main source code. The interaction between reviewers and authors occurs through supporting tools (e.g. Gerrit and Review Board) [3]. There are several research exploring MCR process. For example, after conducting a survey and interviews with several developers and managers in Microsoft, Bacchelli et al. [3] concluded that finding defects is the prominent reason of performing a review, but it is not addressed in practice as it was expected [3]. Similarly, in 2015, Czerwinka et al. also claimed that finding defects is the primary reason for code review [11]. They also argue that in most of the cases, reviewers do not really find functionality defects. Weigerber et al. [44] observed that smaller code changes have the highest chance of being accepted.

2.1 Code Review and Software Quality

There are many empirical studies that apply different software metrics on defect prediction models in order to predict software quality [35] [42]. In recent years researchers have

been using process, organization and people related metrics as well as the code related attributes [19]. Shin et al. [41], Pinzger et al. [37], and Alhassan et al. [1] tested the merits of the local collaboration metrics in the defect prediction problem. They found that the local collaboration metrics forms a strong alternative to the other well established metric sets. Nagappan et al. proposed organizational metrics to predict the defect proneness of software [34]. In their work, Nagappan et al. extracted metrics to understand the relations between organizational structure of a software company and the defect proneness of software modules. Calikli and Bener investigated the effect of cognitive biases on the developer/ tester activity [7]. They found that confirmation biases have strong effect on the defect proneness of the software. Kamei et al. [23] and Rahman and Devanbu [39] found the importance of process metrics compared to product metrics. Impact of code review on software quality can be investigated in the same manner. McIntosh et al. [30] investigated the impact of code review coverage, reviewer participations [29] and their expertise participations on the software quality using a linear regression model. This paper investigates the relationship of these three attributes with post release defects. In our work, however, we would like to investigate and model cause effect relationships in order to determine the release readiness of a patch. Unlike McIntosh et al. we build a defect prediction model to understand the defectiveness of a patch. Briand et al. [5] explored how management factors, such as the effort assigned and the inspection rate, affect inspection effectiveness. They performed a multivariate statistical analysis: multiple adaptive regression splines (MARS). Kononenko et al. [26] also built a multiple regression model to investigate the effect of various social and personal factors involved in MCR, on the quality of code review. In 2010 Klas et al. [24] proposed a hybrid defect content and effectiveness (HDCE) method which combines available historical project data and context-specific expert knowledge in a reusable quantitative causal model for factors influencing defect content and effectiveness. Their model was developed and evaluated for the software requirements analysis phase. All of the above mentioned methods and models were applied on MCR data. These are different from approaches used in traditional code inspection and they all aim to quantify the quality of software product. One of the well-known approaches used in this problem is capture-recapturing method [13]. In this technique, the overlap among defects found by reviewers is used as a basis for the estimation of number of faults remaining in an artefact. This approach is suitable for meeting based inspection model so that reviewers find defects individually and separately from each other. Thus, it is not applicable in modern code review process. Petersson et al. [36] provide a summary of capture-recapture techniques used in software inspections.

In this research, we compare the result of three machine learning algorithms in order to predict defectiveness of a patch considering the incident of pre or post release defects. In order to capture different aspects of review process and their effects on the outcome, we utilize metrics relevant to Process, Product and People within the code reviews. The underlying set of metrics are selected based on their potential relevance to the quality of patches.

3. METHODOLOGY

In order to address first research question, we extracted several product, process and people related metrics from the process, and used them in different learning algorithms to analyse how well these models can predict the defectiveness of an approved patch. Our goal is to utilize a well performed predictive model to check an approved patch, and estimate its quality in terms of predicting if the patch is a defective patch or not. Thus, if a patch is estimated as a defective patch, more review is required, otherwise, the patch is ready to be integrated to the source mainline.

In this study, we use review process dataset of an open source software project ², Mozilla Core. Mozilla Core is a module of Mozilla project. Mozilla projects encode the code reviews in Bugzilla system. Mozilla has recently started utilizing MozReview for conducting their review process, starting from October 2014. MozReview is a repository based code review platform based on Review Board ³. This tool is tightly integrated with Mozilla infrastructure. For the purpose of our research, we have extracted Mozilla Core review process data performed on MozReview. The starting date of selected data is the beginning of MozReview utilization by Mozilla, from October 2014, until February 2016. The dataset includes 4,750 unique review requests, out of which 2,660 review requests have been approved. Each review request corresponds to one patch. A patch consists of one or more source files that have been changed to address a bug report. Each bug report can have more than one review request associated with. Each patch has one developer associated with, who conducts the changes and submits the patch. It should then be assigned to one or more reviewers to check the changes and approve or recommend improvement of bug fixing in the changed code. After the developer addresses reviewers' recommendations, reviewer would approve the changes and submits the patch to the mainline.

3.1 Description of Measures

The purpose of this study is to predict patch defectiveness (Goal). We would like to understand how Product, Process and People affect the defectiveness of patches (Question). Thus, we extracted attributes of Product, Process, and People (Metric). The list of metrics are provided in table 1. Note that the metrics are measured at patch level.

For each submitted review request, there is a corresponding patch consisting of one or more source files. Regarding product related metrics, "Avg LOC of patch" is the total lines of code in a patch divided by number of files of the patch (equation 1). LOC is a well known metrics used in different software problems such as defect prediction, measuring maintainability and productivity, and effort estimation problems.

$$AvgLOCofPatch = \frac{TotalLOCofPatch}{NumberofFilesinPatch} \quad (1)$$

"Avg code complexity of patch" is measured by adding up average complexity of all files in a patch. Average complexity of a file equals the total number of changed lines of code divided by total lines of code of associated source file, multiply the result by the complexity of the file. Cyclomatic complexity is a well known software metrics used to indicate the complexity of a program by measuring the number

of linearly independent paths in the target source code [28]. Equation 2 shows the formula for calculating this metric:

$$AvgPatchComplexity = \sum_{i=1}^n \frac{ChL_{f_i}}{TL_{f_i}} * Comp_{f_i} \quad (2)$$

Where n is the total number of files in a patch, f_i refers to i^{th} file in a patch, ChL_{f_i} is the number of changed lines of code in i^{th} file, TL_{f_i} is the total number of lines of code in i^{th} file, and $Comp_{f_i}$ is the cyclomatic complexity of file f_i .

"Size of patch" indicates patch size in terms of the number of distinct files in a patch. These metrics have been used previously, e.g. to predict the acceptance of a patch or patch reviewers in a study by G. Jeong et. al. [21]. In bug life cycle, each patch is associated with a bug-report. Bug reports are mostly associated with different levels of severity and priority [21]. These indicators are assigned to bug reports by triage or by the development team. To extract "severity of associated bug report" for each patch, we mined Bugzilla repository, as bug tracking system of Mozilla. In Bugzilla, bug severity can be set as blocker, critical, major, normal, minor, trivial and enhancement ⁴. Description of each level of severity is provided in table 3. Similarly, we extracted bug priority associated with each patch. Priorities ⁵ are categorized as P1, P2, P3, P4, P5 . description is provided in table 2.

In terms of process metrics, we extracted "Avg LOC deleted", "Avg LOC added". These metrics measure sum of lines of code added to or deleted from each file of a patch divided by number of files in that patch. Added and deleted lines of code (aka churn metrics) are common software metrics used in different domains such as defect prediction, effort estimation and etc. "patch review time duration" is used as a metric indicating the time from patch review assignment to the time that reviewer approves the patch. The next metric relates to the number of comments reviewers put on the patch and named as "number of reviewers' comments on patch". In fact, when a reviewer reviews the set of changes in a patch, s/he comments on the patch regarding any possible problems they observe. In MozReview, an issue indicates a problem that reviewer pointed out and submitter needs to address. Issue list can be considered as a to do list for submitter. After developer receives the comments, s/he will address them and then related open issues will change their status to resolved issues. To capture this process, we extract "number of resolved issues" from a patch. These metrics are similar to the code review quality metrics that are used to measure the quality of inspection processes in traditional review methods [10].

We also use metrics to measure people related attributes. In our dataset, number of developers working on a patch is always one, but, number of reviewers working on a specific patch can be more than one. Therefore, we extract "number of reviewers" as an attribute for each patch. "Avg number of interaction of developer with components in patch" and "Ave number of interaction of reviewer with components in patch" are the two metrics that measure the number of interactions of reviewer/developer with the files under the change within the target patch on a monthly basis. That means, if a reviewer interacted with a file in the current month, their interaction rate would be 1, if it is in the last 2 months, then

²<https://github.com/RyersonU-DataScienceLab/esem2016>

³<https://www.reviewboard.org/>

⁴wiki.documentfoundation.org/QA/Bugzilla/Fields/Severity

⁵<https://wiki.mozilla.org>

Table 1: Attributes Related to Different Aspects of Review Process

Product	Process	People
Avg LOC of patch	Ave LOC added of patch	number of reviewers
Ave code complexity of patch	Ave LOC deleted of patch	Ave interaction of developer with components in patch
Size of patch (number of files)	Patch review time duration	Ave interaction of reviewer with components in patch
Severity of associated bug-report	number of reviewers' comments on patch	Ave recency of interaction of developer with components in patch
Priority of associated bug-report	number of resolved issues	Ave recency of interaction of reviewer with components in patch

Table 2: Categories of Bug Priority

P1	We definitely want this. It's a major feature, and it's obvious that it would be useful to everybody
P2	We want this, but it's not totally clear or extremely important
P3	This isn't a bad idea, and maybe we'll want to implement it at some point in the future, but it's not near-term roadmap material. Some core Bugzilla developer may work on it
P4	This isn't a terrible idea, but it's not important to our long-term plans for Bugzilla. We would review a patch if somebody posted it, but a core developer is unlikely to work on it. If the patch proves to be too complex, there's a chance that the feature will be marked WONTFIX or delayed until some unknown future release.
P5	We basically never want this. If somebody implements it and asks for review, we might look at it. If a posted patch involves any significant complexity, it will likely be rejected.

their interaction rate would be $1/2$, and so on. We can then take an average based on number of files in the patch, and formulate these metrics as: Average interaction of reviewer r with patch p , which consists of n files ($f_1, \dots, f_n$), with m reviewers is:

$$AvgInteraction_{rp} = \frac{1}{m} \frac{\sum_{i=1}^n interaction_{rf_i}}{n} \quad (3)$$

where

$$interaction_{rf_i} = \frac{1}{k}$$

is the number of interaction of reviewer r with file i in the past.

We measure "Avg number of interaction of developer with components in patch" in the exact same way.

Next metrics measures the recency of last interaction of reviewer r with the components of patch p , consists of n files ($f_1, \dots, f_n$):

$$AvgRecencyofInteraction_{rp} = \frac{1}{m} \frac{\sum_{i=1}^n \frac{1}{t_{rf_i}}}{n} \quad (4)$$

where, t_{rf_i} is the last interaction of reviewer r with file i , in month. That means if reviewer r has interacted with file i in the last 30 days, $t_{rf_i} = 1$. If the last interaction happened in the last month, $t_{rf_i} = 2$, and, so on. We use the same formula for recency of developer interaction.

3.2 Description of Model

In order to train and evaluate predictive models, we need to feed the models with labelled data, as defective and non-defective instances. To label data points in our dataset, we apply a heuristic. We assume that the patch was defective at the time of submission if any file in a submitted patch emerges in review repository later again (within a specific time threshold), and we label it as a defective patch. According to this approach, for each patch in our dataset, we investigate review repository in two weeks after its submission time and look for any files that emerge again in the target patch. If at least one of the files in the target patch appeared in a new patch, first we check Mozilla bug-tracking system (Bugzilla) to find associated bug-report to the new patch. Bug report information shows that if the new patch is related to a bug or it is enhancement related to a change. If the new patch is linked to a bug, we assume that target patch was defective at the time of submission, and label that patch as a defective patch. Time threshold is selected based on average release time interval. This value can be varied depending on which dataset and project is used. There was no information about the release schedule of Mozilla Core. According to Mozilla Wiki, release timeline of Firefox products varies from daily, weekly to around five weeks⁶. We were advised by the local Mozilla office in Toronto that we could take the Firefox release schedule and assume Mozilla Core would follow the same schedule, and consider two weeks as the average of these release schedules. To build the model, we select three different models: Logistic regression algorithm, Naive Bayes and Bayesian Network model. In prior work we have explored a range of data mining methods for defect prediction and found that classifiers based on Bayes theorem yields better performance than rule based methods (i.e. decision trees, oneR) [31] [32]. An extensive study by Lessmann et.al also shows that Naive Bayes and Logistic-

⁶<https://wiki.mozilla.org/ReleaseManagement/ReleaseProcess>

Table 3: Categories of Bug Severity

blocker	Prevents function from being used, no work-around, blocking progress on multiple fronts
critical	Prevents function from being used, no work-around
major	Prevents function from being used, but a work-around is possible
normal	A problem making a function difficult to use but no special work-around is required
minor	A problem not affecting the actual function, but the behavior is not natural
trivial	A problem not affecting the actual function, a typo would be an example
enhancement	Request for enhancement

regression perform equivalently well with 15 other methods [27]. Hall et al. also show that Naive Bayes and Logistic Regression perform better than list of commonly used machine learning algorithms in defect prediction problems [18]. We also select Bayesian Network, as a model that considers relationships and dependencies among independent variables, a property that does not exist in Naive Bayes or Logistic Regression. We think this is an important factor since it enables the model to better capture actual relationships among review process data. As the correlation analysis demonstrates that high correlation between some of the independent variables indicates the possibility of the existence of such dependencies. To build each model, we used Weka⁷ as an automated tool. Weka is a popular software that contains lots of visualization tools, pre-processing and Machine Learning algorithms used for data mining tasks.

3.3 Performance Measures

Confusion Matrix is a table which visually represents the performance of a prediction algorithm in classification problems. Each row of the table represents the actual class of instances and each column shows the predicted class (or vice versa). The cells on the positive diagonal of the table represent correct predictions and the rest of the cells represent incorrect predictions. In a binary classification (our case), the table of confusion matrix has two rows and two columns. The cells of the table report True Positives (TP), False Positives (FP), False Negatives (FN) and True Negatives (TN) [38]. Using these information, several performance measurements can be calculated. In this study, we used five well-known performance measures. Probability of detection (pd), and probability of false alarms (pf) rates in order to assess and compare the performance of different models. [6]. Pd, which is also known as recall, measures how good a prediction model performs in order to find actual defective patches. While, Pf measures the false alarms of predictor, for the cases that the model classifies non-defective patches as defective ones. Ideally, the expectation is that a good predictor is able to catch all defective patches (pd = 1). Also, it should not provide any false alarm, determining as misclassification of actual non-defective patches as defective (pf

= 0). It is very rare to have an ideal case since the predictors are mostly activated in order to get higher probability of recall (or detection rates) [6]. Consecutively, this would lead to higher false alarm rates. Thus, in general, a desirable prediction performance is the one that is (1,0) in terms of (pd, pf) rates. The formula of pd and pf is provided in equation 5 and 6:

$$pd = \frac{TP}{TP + FN} \quad (5)$$

$$pf = \frac{FP}{FP + TN} \quad (6)$$

In our problem, TP is the number of true defect estimations, known as true positive. FP is the number of false defect estimations, known as false positive. TN, true negative, is the number of true safe patch estimations, and FN is the number of false safe patch estimations known as false negative. The optimum pf, pd rate combinations varies in different projects, depending on their natures. For example, in a safety-critical project, pd rate is probably more critical, whereas, on a budget-constrained project, having low pf rate might be more desirable, for the purpose of resource allocation.

In addition to pd and pf, we have reported their geometric mean as balance, and the F-measure. Area under ROC curve (AUC) is also calculated and presented in the analysis of model comparison.

$$balance = 1 - \frac{\sqrt{(1 - pd)^2 + (0 - pf)^2}}{\sqrt{2}} \quad (7)$$

$$F-Measure = \frac{2 * pr * pd}{pr + pd} \quad (8)$$

Where pr is precision, which is measured as:

$$pr = \frac{TP}{TP + FP}$$

3.4 Model Comparison

In order to compare the performance of our models, we applied the 10-fold cross-validation and ran each model for 100 repetitions. Therefore, we derived five performance measurements and replicated the result 10 * 100 = 1000 times for each model. In order to statistically verify which prediction model outperform others, Mann-Whitney U test is applied.

4. RESULTS

After labelling the dataset, we found 724 defective data points comparing to 1,936 non-defective patches. In data preprocessing, using Likert scaling, we converted ordered categorical variables such as severity and priority to ordinal numeric values. For example, we assigned 1 to priority levels of P5, 2 to P4 and so on. Checking the values of the variables, we noticed that the range of some variables are very large. However, since the frequency of upper and lower bound of the range is significant, we could not consider them as outliers and remove them. Thus, to eliminate the impact of extreme values, we normalized the variability of the dataset and scaled them down into a range of 0 to 1 using Weka. Variables such as severity and priority with low ranges remained the same. We also replaced missing

⁷<https://weka.wikispaces.com/>

data instances that occurred in some factors such as priority, severity, and number of reviewers with mode value of each attribute. We chose mode value rather than mean to capture the frequency rather than the average. Since our dataset is imbalanced we used SMOTE: Synthetic Minority Over-sampling Technique to re-sampled the data [9]. Synthetically generated data to over sample (random or by replacement) simply adds copies of the minority examples so that it makes some sub-regions of the minority class very specific leading to over-fitting. However, SMOTE avoids over-fitting because the synthetic samples are not exact copies of the original ones. Instead synthetic samples are interpolated along the segment forcing the decision region of the minority class to become more general. SMOTE operates in "feature space" rather than "data space". New instance values are derived from interpolation rather than extrapolation, so they still carry relevance to the underlying data set. For each minority class instance SMOTE interpolates values using a k- nearest neighbor technique and creates attribute values for new data instances. [8][9].

4.1 Correlation Analysis

Table 4 (provided at the end of the paper) illustrates the pairwise Spearman rank correlation between metric measures. In this table the "-" values represents that two attributes are negatively correlated, and "+" values show that two attributes are positively correlated. Rank correlation is chosen over other correlations, such as Pearson, since rank correlation is resilient to the data that is not normally distributed. It is suitable for variables measured at the ordinal level [20]. Correlation analysis gives us an insight to understand the relationship between attributes. As it can be seen, some of the metrics have strong relationship and are highly correlated. For example, Average Lines of Code Added to the patch is correlated with number of files in the patch, as well as average deleted lines of code. Also, Number of Resolved Issues of a patch and Number of Reviewers' Comments on a patch have strong relationship. The result of correlation analysis also shows a relationship between the Average of Reviewer Interaction with patch components with the Recency of Reviewer Interaction with patch components. This also shows that the Average Interaction of developer with patch components is related with the patch Average Complexity.

4.2 Comparison of Prediction Models

Table 5 indicates different performance measurements for each of mentioned models: Logistic Regression, Naive Bayes and Bayesian Network. The values presented in this table are the mean of the result of 1000 repetition for each model, and each measurement. As it can be seen from the table, Bayesian Network performs far better in terms of each of performance measurements. According to this result, Logistic Regression outperform Naive Bayes, however, Mann-Whitney U test does not confirm that this is a statistically significant difference. In this table, "*" indicates that the significance of outperforming outcomes are verified with Mann-Whitney U test ($p < 0.05$).

There are many factors affecting the success of an algorithm. Data quality is known to be one of the factors that can impact performance of algorithms. For example, redundant and irrelevant data make it difficult to discover knowledge while training the algorithm. Attribute selection is one

way to improve the quality of dataset, and remove possible irrelevant or redundant features [17].

Table 5: small Comparison of Prediction Models in Terms of Different Performance Measurements

Measurements	LR	Naive Bayes	BN
pd	0.58	0.47	0.76*
pf	0.53*	0.42*	0.28*
balance	0.52	0.52	0.74*
F-Measure	0.48	0.4	0.75*
ROC Area	0.59	0.55	0.83*

Thus, in this study, in order to have a fair comparison, we conducted feature selection method with the hope of improving the performance of algorithms. This might end up in different results for comparison purpose.

Algorithms used for selecting features belong to two main categories: Wrapper methods and filter methods. In wrapper methods, features are selected based on learning algorithms, using cross validation technique to measure the benefit of adding or removing a feature from set of selected ones. Filter methods, on the other hand, do not rely on any learning algorithm and are known as general preprocessing algorithms. That means, wrapper method produce a subset of feature that are specified to a specific learning algorithms. Filter methods, on the other hand, produce a list of ordered feature that can be used in any learning algorithms. The main advantage of filter methods is speed and ability to scale to large dataset. While, wrapper methods are computationally expensive, which make them unsuitable to large datasets. The primary advantage of wrapper methods is the use of learning algorithms itself to decide on which features to select. This is important since different learning algorithms might have better performance with different features, even if the training set is same [12]. In this study, since we do not have a scalability problem, we use a wrapper method to conduct feature selection and investigate if it can change our initial results. Mark A. Hall and Geoffrey Holmes [17] show that Wrapper Subset Evaluation method provides best accuracy for Naive Bayes algorithm comparing to several other methods. Inspired by this study, we select Wrapper Subset Evaluation in Weka and come up with a subset of attributes for each algorithm: NB, LR and BN, Table 6. Then using each subset for the respective algorithm, we repeat the experiment and report the results. Table 7 shows the final results. As it can be seen, although by applying attribute selection method, the performance of models slightly increases, Bayesian Networks still statistically outperform the two other algorithms, and the final conclusion has not changed. Applying selected attributes in Logistic Regression algorithm, reduces the possibility of multicollinearity in Logistic Regression since the highest correlated metrics are not in the list of selected metrics. This ensure more reliable results for this algorithm. Checking the selected attributes by wrapper method for each predictive model, we can conclude that mostly all the categories of metrics play role in finding defective patches.

Table 6: Features Selected for Each Algorithm

Logistic Regression
number of reviewers, Ave recency of reviewer interaction, Ave LOC deleted, Size of patch, Ave LOC added, Avg interaction of reviewer, bug severity, bug priority, Patch review time duration, Avg LOC of patch, Avg code complexity
Naive Bayes
Ave recency of reviewer interaction, number of resolved issues, bug severity, bug priority, Avg interaction of reviewer, Avg code complexity
Bayesian Network
number of resolved issues, bug priority, bug severity, Size of patch, number of reviewers' comments, Ave LOC added, Patch review time duration, Avg LOC of patch, Avg code complexity

However, using Bayesian Network, people related metrics have no effect of extracting a pattern from the dataset. This indicates that, using this model, product and process related metrics are capable of extracting pattern from data points, and probably the effort for extracting people related metrics is not required. This, however, can not be a general conclusion. In order to reach a more reliable claim, we need to conduct a Bayesian Network that fits perfectly our dataset, rather than using Weka, which is suitable for more general purposes. According to the result of experiments, Bayesian Network is more capable of predicting patch defectiveness in the review process. This result is driven from a comparison between Bayesian Network with Logistic Regression and Naive Bayes, as two commonly used algorithms in software defect prediction domain, with generally promising results [18]. Bayesian Network models have been previously used in defect perdition problems [33] [16]. For example, in [16] Fenton et al. developed a Bayesian Network model for predicting the number of residual defects which might be found during independent testing or operational usage. Fenton and his colleagues argue that the prediction models should be build based on models that take into account the major causal influences [15]. In our case, the out performance of Bayesian Network might be due to its ability to consider relationships and dependencies between independent variables (i.e. cause-effect). While Naive Bayes and Logistic Regression algorithms do not take this into account, and that might cause loosing some important information about data. Considering the fact that in review process different attributes might have effect on each other, makes this claim stronger. For example, the correlation between some of the independent variables indicates the possibility of existence of such relationships and dependencies. These arguments, however, need to be investigated in depth in order to confirm these relationships and then model these relationships. Such investigation may result in providing reasoning and finding the main causes for the quality of review process outcome, in terms of the defect proneness of the patch. Bayesian Network, as a model that can capture causality relationships and provide reasoning may be a good option for construct-

ing such a model.

Table 7: Comparison of Prediction Models with Selected Attributes, in Terms of Different Performance Measurements

Measurements	LR	Naive Bayes	BN
pd	0.61	0.60	0.76*
pf	0.45	0.49	0.29*
balance	0.58	0.55	0.75*
F-Measure	0.59	0.55	0.74*
ROC Area	0.63	0.61	0.83*

5. THREATS TO VALIDITY

In order to mitigate the threat caused by error measurement, we use four common used performance measurements in software defect prediction domain: pd, pf, F-measure and balance. An internal validity threat might be sampling bias. To avoid this threat, we applied 10-fold cross validation with 100 repetitions for each experiment. Using 10-fold cross validation can also help to avoid over fitting in the model. Using Mann-Whitney U test, as a non-parametric significance test, helps to avoid the restriction of having normally distributed data, which is required in a parametric statistical test. We also used SMOTE to treat data imbalance problem. We applied SMOTE method on the whole dataset. Then used the balanced dataset for both training and testing of the model.

In regards to external validity, we are aware that we can not make generalizations based on one project. However, in this research our aim was to explore the effect of product, people and process metrics in building a defect prediction model to make the patch release decision. As it is described in the paper by Wieringa and Daneva [45] software engineering studies suffer from the variability of the real world and the generalization problem can not be solved completely. As they indicate to build a theory we need to generalize to a theoretical population and have adequate knowledge of the architectural similarity relation that defines the theoretical population. In this study we do not aim to build a theory, rather we would like to have a deeper understating of factors in the performance of a defect prediction model within a case study. We believe that other researchers can repeat our experiments in different settings to confirm or refute our results and build theories.

Another threat to validity might occur in labelling data points. In this study, we assume if any file consists in a given patch emerges in the review repository later in a specific time threshold, as a bug related issue, that file was defective at the time of integration of its associated patch. We understand, this might not be a totally reliable assumption, since there is a possibility that different part of the file is effected in the new emerging in review repository, comparing to last time the file was changed. Thus, to make sure, the defectiveness labelling is more accurate, we need to check for the same part of the file being defective again, as previous change. However, in this study, the log data did not provide us with information about which lines of code has been changed in each files, to enable us to label data accordingly. Going forward, we plan to mine source code in order to extract

such information, and getting more accurate results.

6. CONCLUSION

Software code review process is an important activity within the software development life cycle to ensure the quality of code changes. The review is usually performed by different developers than the submitter of the changes. Patches can be integrated to the main repository after reviewers approved all the code changes. According to several studies, reviewer approval does not guarantee the high quality of code changes, and there is still a risk that the approved patch is defective in the time of integration. This is an important issue since it potentially affects the overall quality of software product. In this study, our aim is to investigate that, using 3P related factors of review process, how a learning algorithm can predict the defectiveness of a reviewed patch. Comparing three well known machine learning algorithms, the results show that Bayesian Networks outperforms Naive Bayes and Logistic Regression.

In order to improve the performance of the algorithms, we conducted attribute selection method, and run the models using the selected features. Selected metrics for each model mostly lies in all three categories of metrics, indicating the importance of all groups of attributes in prediction models. Except in Bayesian Network model, process and product metrics seem to be more important than people attributes. The results also show that although the performance of each algorithm increases using selected attributes, comparison outcome remains the same. Our interpretation of the final results is that the outperform of BN is potentially because they are able to capture and model relationship between independent variables. In this respect, Bayesian Network models the problem similar to its real word interactions. However, to reach a strong conclusion, we need to have an in-depth investigation on how Bayesian Network can model the review process, and if there is a way for improvement in its performance. This study is a primary study of our current research program on review process improvement and quality modelling. It gives us an insight on which algorithm is a better choice to be used for further investigation and research on review process, and prediction of patch defectiveness. Going forward, we would like to build a Bayesian Network, that fits the data better, rather than relying on tools like Weka, which is suitable for more general purposes. We also plan to improve our labelling approach, in order to have a better confidence on the reliability and quality of our dataset.

7. ACKNOWLEDGEMENT

This research is supported in part by NSERC Discovery Grant no. 402003- 740 2012.

8. REFERENCES

- [1] S. Alhassan, B. Caglayan, and A. B. Bener. Do more people make the code more defect prone?: Social network analysis in oss projects. In *SEKE*, pages 93–98, 2010.
- [2] A. Aurum, H. Petersson, and C. Wohlin. State-of-the-art: software inspections after 25 years. *Software Testing, Verification and Reliability*, 12(3):133–154, 2002.
- [3] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 712–721. IEEE Press, 2013.
- [4] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens. Modern code reviews in open-source projects: which problems do they fix? In *Proceedings of the 11th working conference on mining software repositories*, pages 202–211. ACM, 2014.
- [5] L. C. Briand, B. Freimut, and F. Vollei. Using multiple adaptive regression splines to support decision making in code inspections. *Journal of Systems and Software*, 73(2):205–217, 2004.
- [6] B. Caglayan, A. T. Misirli, A. B. Bener, and A. Miranskyy. Predicting defective modules in different test phases. *Software Quality Journal*, 23(2):205–227, 2015.
- [7] G. Çalıkli and A. B. Bener. Influence of confirmation biases of developers on software quality: an empirical study. *Software Quality Journal*, 21(2):377–416, 2013.
- [8] N. V. Chawla. Data mining for imbalanced datasets: An overview. In *Data mining and knowledge discovery handbook*, pages 853–867. Springer, 2005.
- [9] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, pages 321–357, 2002.
- [10] J. Cohen, E. Brown, B. DuRette, and S. Teleki. *Best kept secrets of peer code review*. Smart Bear, 2006.
- [11] J. Czerwonka, M. Greiler, and J. Tilford. Code reviews do not find bugs. how the current code review best practice slows us down. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 2, pages 27–28. IEEE, 2015.
- [12] S. Das. Filters, wrappers and a boosting-based hybrid for feature selection. In *ICML*, volume 1, pages 74–81. Citeseer, 2001.
- [13] S. G. Eick, C. R. Loader, M. D. Long, L. G. Votta, and S. Vander Wiel. Estimating software fault content before coding. In *Proceedings of the 14th international conference on Software engineering*, pages 59–65. ACM, 1992.
- [14] M. E. Fagan. Design and code inspections to reduce errors in program development. In *Pioneers and Their Contributions to Software Engineering*, pages 301–334. Springer, 2001.
- [15] N. Fenton, P. Krause, and M. Neil. Software measurement: uncertainty and causal modeling. *IEEE software*, 19(4):116, 2002.
- [16] N. Fenton, M. Neil, W. Marsh, P. Hearty, L. Radliński, and P. Krause. On the effectiveness of early life cycle defect prediction with bayesian nets. *Empirical Software Engineering*, 13(5):499–537, 2008.
- [17] M. A. Hall and G. Holmes. Benchmarking attribute selection techniques for discrete class data mining. *Knowledge and Data Engineering, IEEE Transactions on*, 15(6):1437–1447, 2003.
- [18] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *Software Engineering, IEEE Transactions on*, 38(6):1276–1304, 2012.

- [19] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88. IEEE Computer Society, 2009.
- [20] J. Hauke and T. Kossowski. Comparison of values of pearson’s and spearman’s correlation coefficients on the same sets of data. *Quaestiones geographicae*, 30(2):87–93, 2011.
- [21] G. Jeong, S. Kim, and T. Zimmermann. Improving bug triage with bug tossing graphs. 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*, pages 111–120. ACM, 2009.
- [22] S. Just, R. Premraj, and T. Zimmermann. Towards the next generation of bug tracking systems. In *Visual languages and Human-Centric computing, 2008. VL/HCC 2008. IEEE symposium on*, pages 82–85. IEEE, 2008.
- [23] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. E. Hassan. Revisiting common bug prediction findings using effort-aware models. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.
- [24] M. Kläs, H. Nakao, F. Elberzhager, and J. Münch. Support planning and controlling of early quality assurance by combining expert judgment and defect data—a case study. *Empirical Software Engineering*, 15(4):423–454, 2010.
- [25] S. Kollanus and J. Koskinen. Survey of software inspection research. *The Open Software Engineering Journal*, 3(1):15–34, 2009.
- [26] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey. Investigating code review quality: Do people and participation matter? In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 111–120. IEEE, 2015.
- [27] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *Software Engineering, IEEE Transactions on*, 34(4):485–496, 2008.
- [28] T. J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.
- [29] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 192–201. ACM, 2014.
- [30] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, pages 1–44, 2015.
- [31] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *Software Engineering, IEEE Transactions on*, 33(1):2–13, 2007.
- [32] T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic, and Y. Jiang. Implications of ceiling effects in defect predictors. In *Proceedings of the 4th international workshop on Predictor models in software engineering*, pages 47–54. ACM, 2008.
- [33] A. T. Misirli and A. B. Bener. Bayesian networks for evidence-based decision-making in software engineering. *Software Engineering, IEEE Transactions on*, 40(6):533–554, 2014.
- [34] N. Nagappan, B. Murphy, and V. Basili. The influence of organizational structure on software quality: an empirical case study. In *Proceedings of the 30th international conference on Software engineering*, pages 521–530. ACM, 2008.
- [35] A. D. Oral and A. B. Bener. Defect prediction for embedded software. In *Computer and information sciences, 2007. isis 2007. 22nd international symposium on*, pages 1–6. IEEE, 2007.
- [36] H. Petersson, T. Thelin, P. Runeson, and C. Wohlin. Capture–recapture in software inspections after 10 years research—theory, evaluation and application. *Journal of Systems and Software*, 72(2):249–264, 2004.
- [37] M. Pinzger, N. Nagappan, and B. Murphy. Can developer-module networks predict failures? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 2–12. ACM, 2008.
- [38] D. M. Powers. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. 2011.
- [39] F. Rahman and P. Devanbu. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 432–441. IEEE Press, 2013.
- [40] P. C. Rigby, D. M. German, and M.-A. Storey. Open source software peer review practices: a case study of the apache server. In *Proceedings of the 30th international conference on Software engineering*, pages 541–550. ACM, 2008.
- [41] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *Software Engineering, IEEE Transactions on*, 37(6):772–787, 2011.
- [42] B. Turhan and A. Bener. A multivariate analysis of static code attributes for defect prediction. In *Quality Software, 2007. QSIC’07. Seventh International Conference on*, pages 231–237. IEEE, 2007.
- [43] L. G. Votta Jr. Does every inspection need a meeting? *ACM SIGSOFT Software Engineering Notes*, 18(5):107–114, 1993.
- [44] P. Weißgerber, D. Neu, and S. Diehl. Small patches get in! In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 67–76. ACM, 2008.
- [45] R. Wieringa and M. Daneva. Six strategies for generalizing software engineering theories. *Science of computer programming*, 101:136–152, 2015.

Table 4: Correlation Analysis (correlation above 0.5 is bold)

	N-rev	Rec-rev	Rec-dev	LOC-del	N-res-iss	Size	Avg-inte-dev	N-rev-com	Avg-inte-rev	Ave-LOC-add	severity	priority	review time	Avg-LOC	complex
number of reviewers	1														
Ave re-cency of reviewer interaction	0.08	1													
Ave re-cency of developer interaction	-0.02	0.23	1												
Ave LOC deleted	0.11	0.047	0.022	1											
number of resolved issues	0.15	-0.04	0.01	0.07	1										
Size of patch	0.21	0	-0.02	0.56	0.17	1									
Avg inter-action of developer	-0.05	0	0.32	-0.01	0.02	0	1								
number of reviewers' comments	0.13	-0.04	0.01	0.08	0.78	0.18	-0.01	1							
Avg inter-action of reviewer	-0.04	0.55	0.16	0.07	-0.1	0.01	0.35	-0.09	1						
Ave LOC added	0.22	-0.06	-0.01	0.5	0.27	0.64	-0.02	0.31	-0.07	1					
bug sever-ity	0.01	-0.03	-0.01	-0.09	-0.02	-0.05	-0.02	-0.02	-0.03	-0.06	1				
bug prior-ity	0.1	0.07	0.09	0.02	0.04	0.02	-0.07	0.06	-0.08	0.039	0.07	1			
Patch re-view time duration	0.1	-0.08	0.04	0.038	0.31	0.12	0.02	0.32	-0.11	0.2	-0.07	0.2	1		
Avg LOC of patch	-0.05	0.04	0.02	-0.03	0.05	-0.11	0.04	0.1	0.1	-0.07	-0.01	-0.05	0.1	1	
Avg code complexity	-0.07	-0.18	-0.05	0.048	-0.03	0.047	0.47	0.002	0.4	0.04	-0.05	-0.15	0.01	0.14	1