Investigation of Dataset Features for Just-in-Time Defect Prediction

Giuseppe Ng
De La Salle University
Malate, Manila
giuseppe_ng@dlsu.edu.ph

Charibeth Cheng
De La Salle University
Malate, Manila
charibeth.cheng@dlsu.edu.ph

ABSTRACT

Just-in-time (JIT) defect prediction refers to the technique of predicting whether a code change is defective. Many contributions have been made in this area through the excellent dataset by Kamei. In this paper, we revisit the dataset and highlight preprocessing difficulties with the dataset and the limitations of the dataset on unsupervised learning. Secondly, we propose certain features in the Kamei dataset that can be used for training models. Lastly, we discuss the limitations of the dataset's features.

CCS CONCEPTS

• **Software and its engineering → Software defect analysis**; • **Computing methodologies → Feature selection**;

KEYWORDS

empirical software engineering, software metrics, defect prediction, just-in-time prediction, software defect prediction

ACM Reference Format:
Giuseppe Ng and Charibeth Cheng. 2020. Investigation of Dataset Features for Just-in-Time Defect Prediction. In *Proceedings of Philippine Computing Science Congress (PCSC2020).* City of Manila, Philippines, 8 pages.

1 INTRODUCTION

Over time, the use and reliance to systems have increased and has lead to an exponential increase in complexity and size [11, 18]. This increase of complexity through new functionalities also introduce more defects [23]. As such, the impact of software defects have also increased [10, 15].

Automated tools can assist software engineers in identifying areas that contain flaws so that developers can focus optimization efforts and projects testing costs can be reduced. This can be critical especially for smaller software companies with a limited testing budget [11, 18, 27]. Software Defect Prediction (SDP) hence pertains to the detection of detect potential defects [25]. The goal of defect prediction is to identify the faults in the code and code modifications, estimate the number of defects and identify areas of optimization [13].

Traditional SDP approaches treat this challenge as a binary classification: defective or non-defective [25]. Simple prediction modeling techniques have been adopted in the field such as KNN, Naive Bayes, Support Vector Machine (SVM) and decision trees such as Random Forests [10, 18]. Supervised models are expected to learn from a corpus of data that encompasses code archive in classes or files form and source trees such as GIT, SVN, or CVS, and defect information in a bug ticketing system such as Bugzilla or Jira [13, 18]. Linking source commit information and bug tickets may also be necessary. Granularity of defect prediction can be by package, class, file, or function basis [13, 25].

Majority of the works on SDP focus on detecting defective modules or files, however, these are coarse grained approaches, leaving the work of discovering actual faulty lines to the developers [16]. Mockus and Weiss [20] were first in proposing to identifying faulty code changes or commits instead of files, classes or modules [14]. This technique is referred to as *Just-in-Time Defect Prediction*. Recently, this area of research has become prevalent due to its time-sensitive nature and its fine-grained potential in defect prediction [7]. This approach to defect prediction can be useful when checking in new code, reducing the number of lines inspected by a developer [14, 16]. Considering the lines of code to be inspected is referred to as Effort-aware Just-in-Time defect prediction [25].

Unsupervised learning has also been proposed, most notably by [31]. Unsupervised learning attempts to address the difficulty in labeling training data [14, 31]. In comparing the supervised and unsupervised approaches, Huang et al. [14] concluded that the simple unsupervised models proposed in [31] improved recall while sacrificing precision due to false positives. Yang's work [31] is further refuted by [9].

Several works in Effort-aware Just-in-Time defect prediction utilize the main dataset provided by Kamei and propose utilizing the same features for their experiments [9, 16, 25, 31]. As pointed in [7], more research work is needed to enhance the potential of Just-in-Time defect prediction.

In our feature analysis of the dataset, we report on some notable challenges with preprocessing that researchers may encounter in the dataset from [16] and share our findings on the most important features through Principal Component Analysis (PCA). We validate the features through the use of a simple neural network.

2 BACKGROUND

In this section, we discuss the details of Kamei's dataset and the feature extraction, the preprocessing techniques, and the basic process of building a classifier [15, 16, 25].

2.1 Kamei's Dataset

The dataset is created using six open source projects: (1) Bugzilla, (2) Columba, (3) Eclipse JDT, (4) Eclipse Platform, (5) Mozilla, and (6) Postgres SQL. The Bugzilla and Mozilla datasets were generated

PCSC2020, March 2020, Baguio City, Philippines

using data provided by Mining Software Repository 2007 Challenge[1]. The Eclipse JDT and Platform data were provided by the MSR 2008 Challenge[2]. Columba and Postgres were from their respective CVS repositories [16].

Labeling defective changes is done through the SZZ algorithm [26]. An open source implementation was created by [5]. With regards to the Columba and Postgres dataset, an approximated SZZ approach was taken due to the defect ids not referenced in the commit messages [16].

Table 1 illustrates the features of the datasets [16].

Table 1: Features of the Kamei dataset [9, 16]

Metric	*Description*
NS	Number of modified subsystems
ND	Number of modified directories
NF	Number of modified files
Entropy	Distribution of the modified code across each file
LA	Lines added
LD	Lines deleted
LT	Lines of code in a file before the current change
Fix	Whether the change is a bug fix
NDEV	Number of developers that changed the modified file
AGE	The average time interval between the last and the current change
NUC	The number of unique changes to the modified file
EXP	Developer experience in terms of number of changes
REXP	Recent developer experience
SEXP	Developer experience on the subsystem

As discussed by Kamei [16], the diffusion aspect is represented by NS, ND, NF and Entropy. Highly distributed changes are more complex and are more likely to be defective. Number of modified subsystems, modified directories, and modified files using the root directory name, directory names, and file names respectively [16].

Entropy is calculated using a similar measure to Hassan [12, 16]. It is the measurement over time how distributed the changes are in the code base. Changes in one file is simpler than one that impacts many different files. Time period used for the dataset is 2 weeks [16]. Entropy is defined as [16]:

$$H(P) = -\sum_{k=1}^{n}(p_k * log_2 p_k) \quad (1)$$

where probabilities $p_k \geq 0, \forall k \in 1, 2, ..., n$, n is the number of files in the change, P is a set of p_k, where p_k is the proportion that $file_k$ is modified in a change and $(\sum_{k=1}^{n} p_k) = 1$ [16]. Figure 1 illustrates entropy in a sample scenario [8].

For time interval $t1$, there are four changes and the probabilities are $P_A = \frac{2}{4}, P_B = \frac{1}{4}, P_C = \frac{1}{4}$. The entropy in $t1$ is calculated as $H = -(0.5 * log_2 0.5 + 0.25 * log_2 0.25 + 0.25 * log_2 0.25) = 1$. For time interval $t2$, entropy is 1.378 [8].

[1] http://2007.msrconf.org/challenge/
[2] http://2008.msrconf.org/challenge/

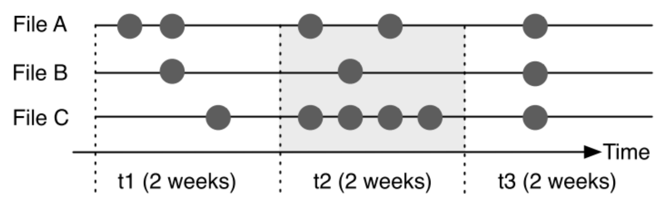

Figure 1: Example of Entropy [8].

Lines added (LA), lines deleted (LD), and lines of code before the current change (LT) were calculated directly from the source code. LA and LD were normalized by dividing by LT while LT is normalized by dividing by NF since these two metrics have high correlation [16]. NUC is calculated by counting the number of commits that caused changes to specific files. This metric was also normalized by dividing by NF due to their correlation with the NF feature [16]. This technique follows findings made by Nagappan and Ball [22].

Age is calculated as the average of time interval between the last and current change of files [9, 16]. Should Files A, B and C were modified 3 days ago, 5 days ago, and 4 days ago respectively, Age is calculated as 4 (i.e., $\frac{3+5+4}{3}$). More detailed explanation of the features of the dataset can be found in [16].

The table 2 shows the brief summary of the datasets that were generated in [16].

Table 2: Summary of the datasets [16, 31]

Project	Period	#change	%Defect induced change
BUG	08/1998-12/2006	4620	36%
COL	11/2002-07/2006	4455	14%
JDT	05/2001-12/2007	35386	14%
PLA	20/2001-12/2007	64250	5%
MOZ	01/2000-12/2006	98275	25%
POS	07/1996-05/2010	20431	20%

2.2 Features Used in Existing Studies

NS, NM, NF, NDEV, PD, EXP, REXP and SEXP are raw data. Entropy, LA, LD, LT and NUC are normalized. Fix is a boolean value [25]. In the downloaded dataset from Kamei [16], NM refers to ND, PD refers to AGE, and NPT refers to NUC.

In [14], ND and REXP are removed because NF and ND are correlated. Correlated features will decrease the accuracy of the classifier. LA and LD were also found to be highly correlated [16]. In addition, LA and LD are not included as learning features because these are used to sort the inspection effort [14].

The feature recommendations when using Just-in-Time defect prediction from these studies are: NS, NF, Entropy, LT, FIX, NDEV, PD, NPT, EXP, SEXP. Given the Just-in-Time defect prediction $Y(x)$, effort-aware assessment is calculated as $\frac{Y(x)}{Effort(x)}$ where $Effort(x) = LA_x + LD_x$ [16, 31].These insights were echoed in [25].

2.3 Dataset Preprocessing

Two specific issues with the dataset were highlighted in [16]. First, the dataset is highly skewed [14]. This is addressed through the use of logarithmic transformation [16, 25].

Data imbalance is another issue where in the defect-inducing changes only make up a small percentage of all changes in a project. This can be addressed by performing random undersampling. Undersampling, however, is not done on the validation set [16].

2.4 Building the Classifier

The process of building the classifier begins with resampling the training set through undersampling [16]. While oversampling can be used, [15] found that undersampling was more effective [3, 13]. After, ND, REXP, LA, and LD are removed from the features of the training set. Then, a standard logarithm transformation is applied on each of the metrics. The classifier can then be trained [15].

2.5 Principal Component Analysis

Principal Component Analysis (PCA) is a technique used to reduce the complex features into simple principal components that are seen as the summary of the features. It is an unsupervised learning technique used to identify patterns and clusters from a particular dataset [17]. It can be used to discover hidden features that are not readily seen in analytical studies [4]. To the best of our knowledge, we have not seen PCA applied into the Kamei Dataset. We use this technique to uncover hidden attributes with this dataset.

3 RESEARCH METHODOLOGY

The dataset are examined to identify any visible issues. PCA is done on the raw dataset to gain a visual analysis of the dataset and the identify potential recommended features. Initially, no preprocessing is done to ensure that the test is easily reproduceable. PCA is performed on all the datasets and the distinct features found will be selected for experimentation.

Following the process of building the classifier, we perform preprocessing on the dataset and report on the decisions we made. For validation, we conduct experiments using a simple neural network on the feature selection and other feature combinations. The evaluation of the features selected is based on the Recall. These metrics are typically used in evaluating classifiers in SDP [1, 28]. The equation is shown below [29]:

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

Where TP is the True Positive, and FN is the False Negative. Final recall reported is the average of all recall tests of the model.

4 RESEARCH FINDINGS

In this section, we discuss the findings of our dataset study, PCA and experiments.

4.1 Non-normalized Instances Found

In analyzing the features of the dataset in [16], we found that usually, when LT is 0, LA and LD are also 0. However, there are also instances where LA and LD are not 0. This most likely means that the submitted changes consisted of completely new files. In such a scenario, LA and LD are not normalized and instead retain their raw data. LA and LD are usually only factored into Effort-Aware sorting and we wonder what impact these characteristics could have. Though these instances are few, in new software projects, there are more instances of new files being added into a source repository.

4.2 Pre-processing Challenges

Logarithm transformation is done to address skewness of data [16]. However, some of the data in the features lead to $log(0)$ which can cause infinity values in the dataset. To the best of our knowledge, we have not found a paper that raises this concern. The work by [2] discusses in general this issue though it is not specific to the field of defect prediction.

In [25], normalization of data is suggested using the equation below:

$$Norm(x) = \frac{x - min(x)}{max(x) - min(x)} \tag{3}$$

However, in light that some instances of LA and LD are raw values, we wonder what the impact of this is.

4.3 Overlapping of Non-defective and Defective Dataset

Visualizing the dataset as shown in earlier figures indicate that the challenge in training for Just-in-Time defect prediction is that the labeled data for defective and non-defective changes overlap tightly. As unsupervised learning involves clustering according to [17], the overlapping nature of the PCA shows the limitation of Kamei's dataset with regards to unsupervised learning. This is consistent with the findings of [14] and [30] where unsupervised learning methods do not perform well compared to supervised learning methods.

4.4 Analyzing Kamei's Recommended Features

Using the principal component analysis (PCA) technique, we compare the PCA of the Bugzilla dataset with all the features and the recommended features. Figure 2 illustrates this. Blue refers to non-defective while red indicates defective change.

Figure 3 illustrates the PCA with the recommended features from [16], namely, NS, NF, Entropy, LT, FIX, NDEV, PD, NPT, EXP, SEXP. Observations show that there is minimal change on the dataset.

This suggests that using the recommended features do not alter the behaviour of the dataset for training purposes. Surprisingly, however, this PCA can be achieved with close proximity by utilizing only two features in the Bugzilla dataset: LT, and PD.

Interestingly, these features were highlighted in the resulting unsupervised model by [31].

In the Mozilla dataset, Figures 5 and 6 illustrate the PCA visualizations of recommended features and EXP, PD respectively.

The PCAs for Columba, Eclipse JDT and Platform datasets also point to EXP and PD. Interestingly, the Postgres dataset showed that PD, EXP, and SEXP. Figure 7 shows the PCA for Postgres.

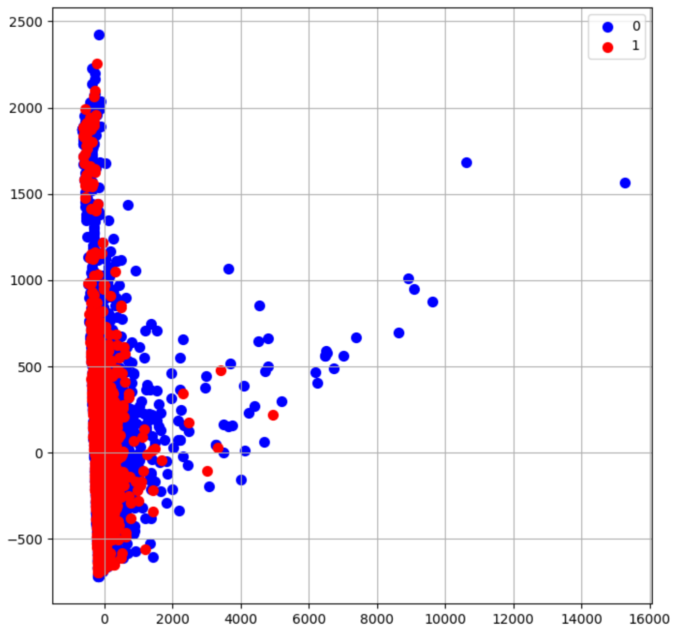

Figure 2: PCA of Bugzilla with all the Features of the Dataset.

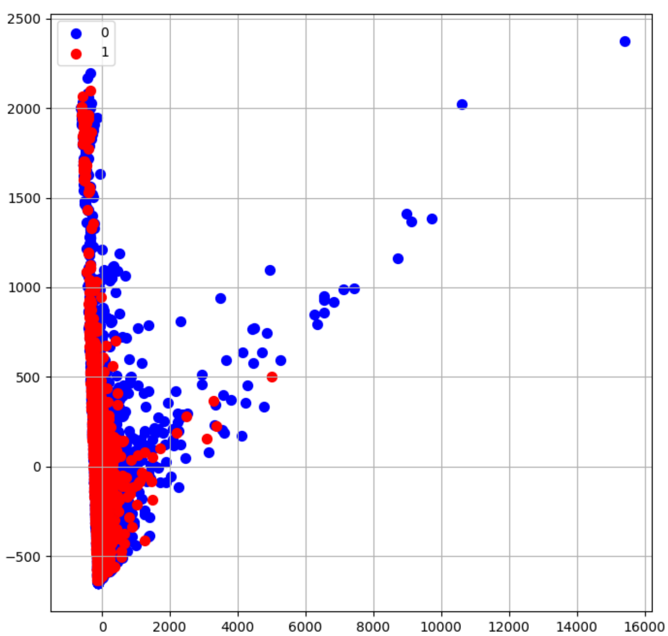

Figure 3: PCA of Bugzilla with Kamei's Recommended Features.

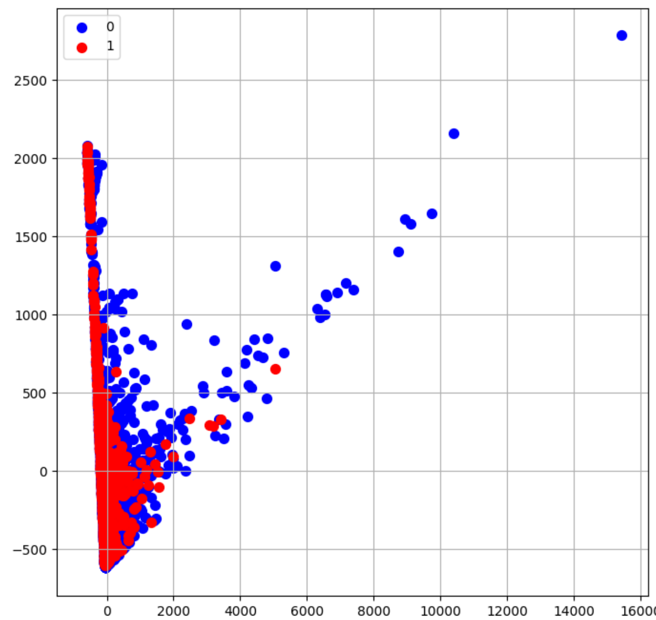

Figure 4: PCA of Bugzilla using LT and PD.

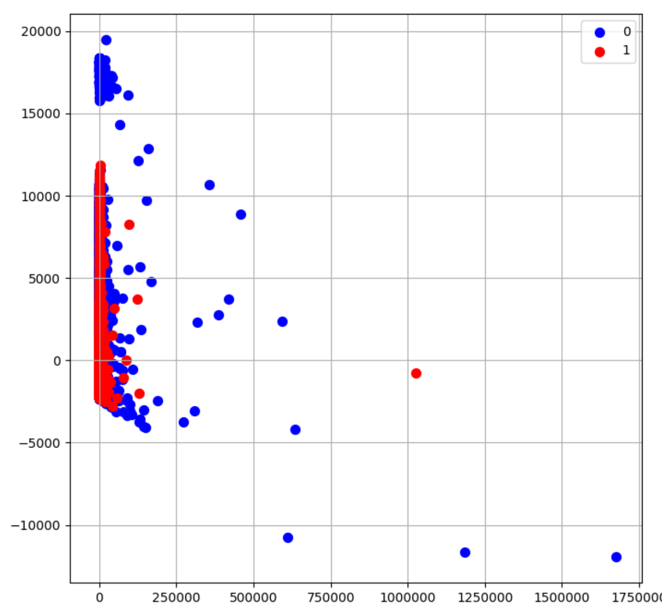

Figure 5: PCA of Mozilla using Kamei's Recommended Features.

With these results from PCA, we focus on using the following features: (1) LT, (2) PD, (3) EXP, (4) SEXP. LT as an important feature corroborates with the findings of [21] and [22]. The selection of PD is consistent with the work of [16] and [19]. EXP and SEXP being important features is consistent with the findings in [19] and [20] where more experienced developers are less likely to produce defective code. Experience-related features can be used in defect prediction according to [24].

Given that the Bugzilla, Mozilla and Postgres datasets yielded the unique set of features. We use these datasets in training our model with the features identified in the PCA. Our experiments focus on these identified features to determine which are most important.

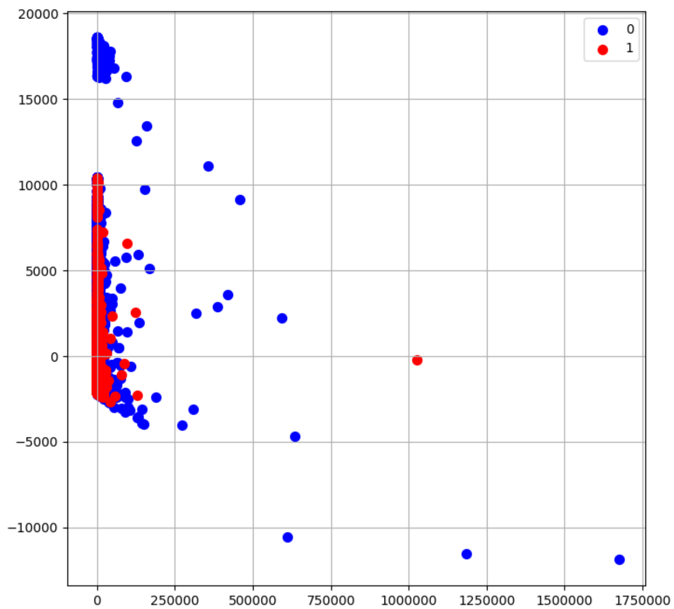

Figure 6: PCA of Mozilla using EXP and PD.

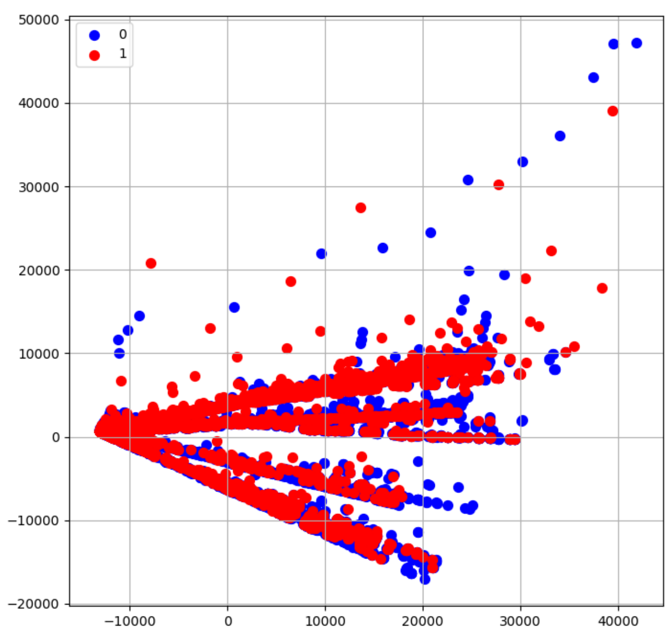

Figure 7: PCA of Postgres using EXP, SEXP and PD.

In terms of validation, the neural network design used is inspired by [25] with some slight deviations. Our implementation is on PyTorch and uses 2 hidden layers (20 tanh activation nodes, 10 ReLU nodes) and a Linear output layer. Epoch used was 150, with an Adam Optimizer and a learning rate of 0.001 with a drop out layer ($p = 0.2$). 10-fold cross validation on our training. The Bugzilla dataset was selected with a training set, testing set split of 90:10. Effort aware computations were not considered as our focus was on the regression model itself and that the features useful for effort aware prediction were not part of the features considered during PCA.

We used the features recommended by previous works earlier and also the LT, PD combination to see whether the results were comparable. The following experiment was done using the Bugzilla dataset whose training data has been randomly undersampled, natural logarithm applied on NS, NF, NDEV, NPT, EXP, REXP, and SEXP. Normalization performed on all features. These are the preprocessing steps advised by [25] though we limited logarithm operations to these columns to avoid the infinity values from entering the dataset. For evaluation, we used recall as suggested in [25]. Table 3 shows the results on the Bugzilla dataset.

Table 3: Experiment on Bugzilla Dataset

Features	Recall
LT,PD	56.80%
LT,PD,SEXP	55.17%
PD,SEXP	54.20%
LT,PD,Entropy,NS,NF,Fix	54.04%
LT,PD,Entropy,NS,NF,Fix,NDEV	50.32%
LT,PD,Entropy,NS,NF,Fix,NDEV,NPT	48.59%
LT,PD,Entropy	46.35%
LT,PD,Entropy,NS	44.98%
LT,PD,Entropy,NS,NF	41.69%
LT,PD,Entropy,NS,NF,Fix,NDEV,NPT,EXP	40.46%
LT,PD,Entropy,SEXP	38.30%
PD,EXP,SEXP	34.95%
NS,NF,Entropy,LT,FIX,NDEV,PD,NPT,EXP,SEXP	33.68%
LT,PD,EXP	17.28%
PD,EXP	6.89%

The LT,PD model and LT,PD,SEXP model performed notably better than LT,PD,EXP and LT,PD,SEXP,Entropy. This suggests that as far as the Bugzilla dataset is concerned, LT,PD is the minimal most influential features for consideration.

Table 4 shows the results when using a Mozilla dataset. Tests using the Postgres dataset are shown on Table 5.

With the test done, despite the fact that EXP,PD was the deduced features influencing the PCA for Mozilla, LT,PD performed far better. Interestingly, these features performed better than the features recommended that was reported in previous works.

LT,SEXP,PD,Entropy only performed well in the Mozilla dataset while doing significantly worse in the Bugzilla dataset. LT,PD,Entropy,NS,NF was surprisingly good in the Mozilla dataset though it did not perform nearly as well on the Bugzilla dataset. LT,PD, PD,SEXP, and LT,PD,SEXP performed fairly closely in the datasets that we ran our experiments on. PD,EXP,SEXP performed well in the Mozilla dataset.

We decided to select notable features based on these results and run a different test on a combined dataset of Bugzilla and Mozilla to see a comparison of their performance on a modified dataset. The combined dataset utilized the preprocessed Bugzilla and Mozilla

Table 4: Experiment on Mozilla Dataset

Features	Recall
LT,PD,Entropy,NS,NF	71.36%
LT,PD,SEXP	69.50%
LT,PD	66.69%
PD,EXP,SEXP	63.49%
LT,PD,Entropy,NS,NF,Fix,NDEV,NPT,EXP	63.25%
LT,PD,Entropy	60.76%
LT,PD,Entropy,NS	60.36%
LT,PD,Entropy,NS,NF,Fix,NDEV	57.50%
LT,PD,Entropy,SEXP	57.48%
PD,SEXP	56.77%
LT,PD,Entropy,NS,NF,Fix	56.83%
LT,PD,Entropy,NS,NF,Fix,NDEV,NPT	55.04%
NS,NF,Entropy,LT,FIX,NDEV,PD,NPT,EXP,SEXP	51.14%
PD,EXP	14.72%
LT,PD,EXP	12.24%

Table 5: Experiment on Postgres Dataset

Features	Recall
LT,PD	65.85%
LT,PD,Entropy,NS	62.20%
LT,PD,Entropy	61.58%
LT,PD,Entropy,SEXP	61.51%
LT,PD,Entropy,NS,NF	58.85%
LT,PD,SEXP	58.18%
PD,SEXP	51.78%
LT,PD,EXP	41.82%
LT,PD,Entropy,NS,NF,Fix,NDEV,NPT	38.63%
PD,EXP	32.34%
LT,PD,Entropy,NS,NF,Fix,NDEV	32.01%
PD,EXP,SEXP	30.82%
LT,PD,Entropy,NS,NF,Fix,NDEV,NPT,EXP	28.20%
LT,PD,Entropy,NS,NF,Fix	28.15%
NS,NF,Entropy,LT,FIX,NDEV,PD,NPT,EXP,SEXP	23.98%

datasets with undersampling for equal distribution of classes between the two target datasets. Table 6 shows the results.

LT,PD performs well with LT,PD,SEXP close to this feature. LT,PD seemed to be a reliable feature set to use in the dataset. With the results shown of our experiments, SEXP is a possible complement to LT and PD. The other features did not stand out in all of the experiments.

5 ANALYSIS

Based on our own findings, LT and PD are important features. This is different from findings from [16] where the features recommended were NS, NF, Entropy, LT, FIX, NDEV, PD, NPT, EXP, SEXP. Our selection of LT as a notable feature is in disagreement with the results by [24]. However, LT and PD were the features recommended by [31] in their work on unsupervised models. The result from [19] and [20] indicate experience as an important dimension in determining defective changes. Experience-based metrics was also mentioned as good features in [24]. In our experiment, feature combinations with EXP did not perform as expected while SEXP is a good complement to the features selected. In selecting LT, PD and SEXP, our findings reinforce the assertion of [20] and [22] while SEXP in the case of the work from [19] and [24].

Table 6: Selected Feature Combinations on Combined Bugzilla and Mozilla Dataset

Features	Recall
LT,PD	69.03%
LT,PD,SEXP	66.55%
PD,SEXP	59.09%
LT,PD,Entropy,NS,NF	58.34%
LT,PD,Entropy,SEXP	58.26%
LT,PD,Entropy,NS,NF,Fix,NDEV,NPT,EXP	55.88%
NS,NF,Entropy,LT,FIX,NDEV,PD,NPT,EXP,SEXP	47.11%
PD,EXP,SEXP	35.63%

In assessing LT and PD individually, would be zero valued if the change introduced new files. Considering that in new projects for software companies, introducing completely new files into the code repository is not uncommon. Since [16] used open source projects, the capture of the data may not represent newly created projects.

PD pertains to how old the last change was, and that the older the change means it is less likely to have a defect introduced [16]. As pointed out by Nagappan et al., looking at short burst of changes indicate a higher probability of defectiveness [22]. However, this feature may not accurately describe new files introduced in the dataset and hence, this feature may have difficulty determining a new file's defectiveness.

Using metrics to determine defectiveness can often overlook fine grained details. This was pointed out by [11]. The placement of a single line would have very similar code metrics. Figure 8 illustrates such an issue.

```
try {
  l.lock()
  readFile(f);
  l.unlock();
}
catch (Exception e) {
  // Do something
}
finally {
  closeFile(f);
}
```

Listing 1: File1.java

```
l.lock()
try {
  readFile(f);
}
catch (Exception e) {
  // Do something
}
finally {
  l.unlock();
  closeFile(f);
}
```

Listing 2: File2.java

Figure 8: Code that could have similar metrics from [11].

Although the above shows a snippet of code, NF, NS, and ND would be the same where as other features could yield exactly the same calculated values.

The definition of software defects not only include problematic code but also code that does not conform to user specification [6, 10]. Using that definition, context of code becomes important in determining defectiveness. We find that none of the features in the dataset establishes code context.

6 THREATS TO VALIDITY

The manner in which the dataset was preprocessed ensured that the infinity values were avoided in the dataset. However, that also means that issues regarding the skewness of data would not be addressed in the features that were ignored. This may adversely affect the performances. However, we observed that the LT, and PD selection of features performed relatively stable across the different datasets. We also checked how the features affect the precision of defect prediction. Table 7 shows the precision results of the features LT, PD, EXP and SEXP combinations with the Bugzilla dataset. The values shown are sorted in descending order.

Table 7: Precision on Bugzilla Dataset on LT, PD, EXP, and SEXP

Features	Precision
LT,PD	70.08%
LT,PD,SEXP	52.38%
PD,SEXP	51.85%
LT,PD,EXP	45.58 %
PD,EXP	43.62%

As can be seen, LT,PD yielded a precision 70.08% compared to LT,PD,SEXP at 52.38%, PD,SEXP at 51.85%, LT,PD,EXP at 45.58%, and PD,EXP at 43.62%. Hence, we believe the features will perform well on other datasets.

The considered datasets in the experiments were from Kamei's dataset and are Open-Source Systems (OSS). It is possible that commercial systems have different characteristics to open source ones as pointed out in [16].

7 CONCLUSION

We found LT, and PD were identified as important features in training neural networks that predict defective code change submissions. These results were corroborated by the experiments that were conducted. However, in studying the features, we have come to identify that there needs to be features that can improve the determination of defective code. We also note that on newly contributed files into the source code repository, that the features present may not provide ample insight in determining defective code. We suggest that future works focus on doing more work in uncovering new features and adding more change-level datasets.

ACKNOWLEDGMENT

We would like to thank the Software Technology department of De La Salle University, Dr. Charibeth Cheng, and Mr. Neil Patrick Del Gallego for their support and guidance in this research.

REFERENCES

[1] Abdullah Alsaeedi and Mohammad Zubair Khan. 2019. Software Defect Prediction Using Supervised Machine Learning and Ensemble Techniques: A Comparative Study. *Journal of Software Engineering and Applications* 12, 05 (2019), 85–100. https://doi.org/10.4236/jsea.2019.125007

[2] Christophe Bellego and Louis-Daniel Pape. 2019. Dealing with the log of zero in regression models. *Série des Documents de Travail* (2019), 16. https://doi.org/10.2139/ssrn.3444996

[3] Kwabena Ebo Bennin, Jacky Keung, Passakorn Phannachitta, Akito Monden, and Solomon Mensah. 2018. MAHAKIL: Diversity Based Oversampling Approach to Alleviate the Class Imbalance Issue in Software Defect Prediction. *IEEE Transactions on Software Engineering* 44, 6 (2018), 534–550. https://doi.org/10.1109/TSE.2017.2731766

[4] Rui Bo and Fangxing Li. 2008. Power flow studies using Principal Component Analysis. *40th North American Power Symposium, NAPS2008* 1 (2008), 1–6. https://doi.org/10.1109/NAPS.2008.5307323

[5] Markus Borg, Oscar Svensson, Kristian Berg, and Daniel Hansson. 2019. SZZ unleashed: an open implementation of the SZZ algorithm - featuring example usage in a study of just-in-time bug prediction for the Jenkins project. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation - MaLTeSQuE 2019*. ACM Press, New York, New York, USA, 7–12. https://doi.org/10.1145/3340482.3342742 arXiv:1903.01742

[6] Pierre Bourque and Richard E. Fairley. 2014. *SWEBOK v.3 - Guide to the Software Engineering - Body of Knowledge*. 346 pages. https://doi.org/10.1234/12345678

[7] Liang Cai, Yuan Rui Fan, Meng Yan, and Xin Xia. 2019. Just-in-time software defect prediction: literature review. *Ruan Jian Xue Bao/Journal of Software* 30, 5 (2019), 1288–1307. https://doi.org/10.13328/j.cnki.jos.005713

[8] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2010. An extensive comparison of bug prediction approaches. *Proceedings - International Conference on Software Engineering* May 2010 (2010), 31–41. https://doi.org/10.1109/MSR.2010.5463279

[9] Wei Fu and Tim Menzies. 2017. Revisiting unsupervised learning for defect prediction. (2017), 72–83. https://doi.org/10.1145/3106237.3106257 arXiv:1703.00132

[10] Jianxin Ge, Jiaomin Liu, and Wenyuan Liu. 2018. Comparative study on defect prediction algorithms of supervised learning software based on imbalanced classification data sets. *Proceedings - 2018 IEEE/ACIS 19th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, SNPD 2018* (2018), 399–406. https://doi.org/10.1109/SNPD.2018.8441143

[11] Aditya Ghose, Shien Wee Ng, Truyen Tran, Trang Thi Minh Pham, John Grundy, and Hoa Khanh Dam. 2018. Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering* 14, November (2018), 1–1. https://doi.org/10.1109/tse.2018.2881961

[12] Ahmed E. Hassan. 2009. Predicting faults using the complexity of code changes. *Proceedings - International Conference on Software Engineering* (2009), 78–88. https://doi.org/10.1109/ICSE.2009.5070510

[13] Seyedrebvar Hosseini, Burak Turhan, and Dimuthu Gunarathna. 2019. A systematic literature review and meta-analysis on cross project defect prediction. *IEEE Transactions on Software Engineering* 45, 2 (2019), 111–147. https://doi.org/10.1109/TSE.2017.2770124

[14] Qiao Huang, Xin Xia, and David Lo. 2017. Supervised vs Unsupervised Models: A Holistic Look at Effort-Aware Just-in-Time Defect Prediction. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 159–170. https://doi.org/10.1109/ICSME.2017.51

[15] Qiao Huang, Xin Xia, and David Lo. 2018. *Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction.* Empirical Software Engineering. https://doi.org/10.1007/s10664-018-9661-2

[16] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (2013), 757–773. https://doi.org/10.1109/TSE.2012.70

[17] Jake Lever, Martin Krzywinski, and Naomi Altman. 2017. Points of Significance: Principal component analysis. *Nature Methods* 14, 7 (2017), 641–642. https://doi.org/10.1038/nmeth.4346

[18] Zhiqiang Li, Xiao-Yuan Jing, and Xiaoke Zhu. 2018. Progress on approaches to software defect prediction. *IET Software* 12, 3 (2018), 161–175. https://doi.org/10.1049/iet-sen.2017.0148

[19] Shinsuke Matsumoto, Yasutaka Kamei, Akito Monden, Ken Ichi Matsumoto, and Masahide Nakamura. 2010. An analysis of developer metrics for fault prediction. *ACM International Conference Proceeding Series* January (2010). https://doi.org/10.1145/1868328.1868356

[20] Audris Mockus and David M. Weiss. 2000. Predicting risk of software changes. *Bell Labs Technical Journal* 5, 2 (2000), 169–180. https://doi.org/10.1002/bltj.2229

[21] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. 2008. A Comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. *Proceedings - International Conference on Software Engineering* January 2008 (2008), 181–190. https://doi.org/10.1145/1368088.1368114

[22] Nachiappan Nagappan and Thomas Ball. 2005. Use of relative code churn measures to predict system defect density. *Proceedings - 27th International Conference on Software Engineering, ICSE05* (2005), 284–292. https://doi.org/10.1145/1062455.1062514

[23] Jaechang Nam, Wei Fu, Sunghun Kim, Tim Menzies, and Lin Tan. 2018. Heterogeneous Defect Prediction. *IEEE Transactions on Software Engineering* 44, 9 (2018), 874–896. https://doi.org/10.1109/TSE.2017.2720603

[24] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. 2019. Fine-grained just-in-time defect prediction. *Journal of Systems and Software* 150 (2019), 22–36. https://doi.org/10.1016/j.jss.2018.12.001

[25] Lei Qiao and Yan Wang. 2019. Effort-aware and just-in-time defect prediction with neural network. *PLoS ONE* 14, 2 (2019), 1–19. https://doi.org/10.1371/journal.pone.0211359

[26] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005* (2005). https://doi.org/10.1145/1082983.1083147

[27] Le Hoang Son, Nakul Pritam, Manju Khari, Raghvendra Kumar, Pham Thi Minh Phuong, and Pham Huy Thong. 2019. Empirical study of software defect prediction: A systematic mapping. *Symmetry* 11, 2 (2019). https://doi.org/10.3390/sym11020212

[28] Tao Wang and Wei Hua Li. 2010. Naïve bayes software defect prediction model. *2010 International Conference on Computational Intelligence and Software Engineering, CiSE 2010* 2006 (2010), 1–4. https://doi.org/10.1109/CISE.2010.5677057

[29] Xu-Ying Liu, Jianxin Wu, and Zhi-Hua Zhou. 2009. Exploratory Undersampling for Class-Imbalance Learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 39, 2 (apr 2009), 539–550. https://doi.org/10.1109/TSMCB.2008.2007853 arXiv:arXiv:1011.1669v3

[30] Meng Yan, Yicheng Fang, David Lo, Xin Xia, and Xiaohong Zhang. 2017. File-Level Defect Prediction: Unsupervised vs. Supervised Models. *International Symposium on Empirical Software Engineering and Measurement* 2017-Novem (2017), 344–353. https://doi.org/10.1109/ESEM.2017.48

[31] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. 2016. Effort-Aware just-in-Time defect prediction: Simple unsupervised models could be better than supervised models. *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering* 13-18-Nove (2016), 157–168. https://doi.org/10.1145/2950290.295035