An Exploratory Study on Defect Prediction

Bo Dang
McGill University
Montréal, Quebec, Canada

ABSTRACT

Defect prediction is a highly active research topic in the software engineering. In this project, we do an exploratory study on this topic. First, for a certain interested evaluation metric, we want to identify a prediction model with a single set of features that can be dominant across all projects in Within-Project Defect Prediction (WPDP) setting. Second, we investigate under the same programming language context, whether Cross-Project Defect Prediction (CPDP) can yield comparable performance as WPDP. Third, we explore the feasibility of Cross-Version Defect Prediction (CVDP). In order to achieve our goal, we also implement a defect data collection system to collect defect data from GitHub. We use SMOTE to address data imbalance issue and Chi2 value to rank the features according to the label. For the model, we choose 6 widely used Machine Learning (ML) models and apply them to each project in three different settings (WPDP, CPDP, and CVDP). For each model, we use grid search to find the best feature combination. According to the results, we find that 1) there is no universally best model and best feature combination across all project in WPDP, 2) Even thought the projects are developed by the same programming language, simple CPDP model can still fail to yield as good performance as WPDP, 3) CVDP is not feasible, especially the time span between two versions is too long.

KEYWORDS

Software Engineering, Defect Prediction, Software Metrics, Supervised Learning

ACM Reference Format:
Bo Dang. 2020. An Exploratory Study on Defect Prediction.

1 INTRODUCTION

Software development projects require a critical and costly testing phase to test the functionality of the software project. However, as the size and complexity of project increases, testing can be very time-consuming and resource-consuming. This issue can be solved by automatic software defect prediction models.

Software defect prediction refers to predicting the defect-proneness of software components before the release of the software. The software components can be methods, classes, files, and modules. Most defect prediction models are based on machine learning. The model will take the extracted features from the component as the input and predict whether the component is defect-prone or not, which can be viewed as a binary classification process. During the reading of defect prediction literature, we identified several interesting topics we want to explore.

COMP762, Fall 2019, Montreal, QC
©

First, there are many studies aiming to locate software defects with the use of single model. In [14][24], the Naive Bayes models is used. In [1][4][5], the Logistic Regression model is used. In [10][7], the tree-based method is used. However, they all did not compare the model they used to others. We are very curious about how different models perform differently on defect prediction. Furthermore, we want to find that for a certain evaluation metric, whether there is a dominant model across all projects. Also, the software metrics used can vary a lot across different literature. We want to find whether there is a single set of metrics can perform the best in all projects.

Second, in CPDP setting, we find lots of work do a heavy feature engineering [16] [27]. Take [27] for example, they introduced project partition, similar projects clustering, and ranking function obtaining in the feature engineering task. Especially in obtaining ranking function, each feature distribution comparison need to be conducted between any two of the project, which is very costly. We think the main reason of feature distribution difference is caused by the different programming language adopted by different projects. Therefore, we want to explore that in CPDP, whether we can get the comparable performance as in WPDP when only considering programming language as a context factor.

Third, we find few of the research do cross-version defect prediction (CVDP). Cross-version defect prediction means that we train the predictor on a certain version of the project and apply it to another version of the same project. We want to investigate whether CVDP model can be as predictive as WPDP.

In order to achieve our goal, we also build a defect data collection system to collect training and testing data by ourselves. There are publicly available datasets [6][17][26] [18][21][25] which are widely used in defect prediction community. However, we think these datasets are outdated–most of them are collected ten years ago and some of them are collected even twenty years ago. We believe that as time goes by, the characteristics of current software project can be very different from those of ten or twenty years ago.

This paper is organized in a classical way. In section 2, we describe the related work. In section 3, we introduce the experimental methods we used in the project. In section 4, we proposed each of the research questions and illustrate corresponding experiment results and discussion. In Section 5, we summarize the work we have done. In Section 6, we present some thoughts and potential future work.

2 RELATED WORK

2.1 Software Metrics

Software metrics are used to evaluate software quality with proper thresholds and ranges of metric values [27]. Numerous software metrics have been investigated, including complexity metric (e.g. lines of code and McCabe's cyclomatic complexity[15]), structured metrics[9], process metrics(e.g. recent activities, number of changes, and the complexity of changes[16]). In this project, we use complexity metrics and process metrics as features of our model. These metrics can be extracted by a tool called *scitool understand.* [1]

2.2 Defect Prediction

Defect prediction, also called software fault prediction, which have been studied since 1990s until now. A defect prediction model use previous software metrics and defect data to predict defect-prone components for the next release of software. According to [2], the benefits of software defect prediction are listed as follows:

- Reaching a highly dependable system,
- Improving test process by focusing on defect-prone components
- Selection of best design from design alternatives using object-oriented metrics,
- Identifying refactoring candidates that are predicted as defect-prone,
- Improving quality by improving test process.

The defect prediction can be classified into Within-Project Defect Prediction (WPDP) and Cross-Project Defect Prediction (CPDP). In the WPDP setting, a predictor is trained by using labeled instances in the project and predict unlabeled instances in the same project. The labeled instances can be extracted from historical data of software repositories such as version control and issue tracking system [20]. However, the historical data is not always available in small or new projects, in which situation we need to introduce CPDP model.

A CPDP model is trained on a single or a group of projects and then can be applied to other projects. One difficulty for building CPDP models may be related to the variations in the distribution of features of software components. To overcome this challenge, there are typically two approaches: 1) use the data from projects with similar distribution to the target project [25][19][13]; and 2) transform features in both training and target projects to make the more similar in their distribution [17][11].

3 EXPERIMENTAL METHODS

3.1 Data Collection

In this project, we collect defect dataset from GitHub projects. GitHub is a very large and popular repositories for open source projects. 4 Java projects and 4 C projects are studied in this paper. Table 1 list a detailed information about these projects. Notice that for *maven* project, we collect 2 versions of it to conduct CVDP experiment.

For each project, we first decide a version as a release time. Then we clone the repository snapshot at that time. After that, we use *understand* to generate the code metrics for each file in the project. One thing should be noticed is that, for java and c projects, we only collect metric values for files with .java and .c suffix respectively. Then we parse all commits after the release time to identify the bug files. Finally we parse the commits before the release time to extract process metrics for each file. The overall process can be seen in figure 1.

[1]https://scitools.com/feature-category/metrics-reports/

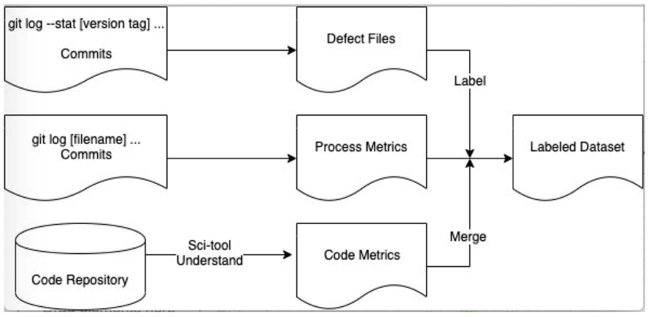

Figure 1: Dataset Collection Process

3.1.1 Code Metrics. The Code Metrics are collected by *understood.* For C projects, 39 kinds of metrics are collected. For Java projects, besides the same 39 kinds of metrics as C projects, we also collect 10 kinds of CK metrics [22]. CK metrics is class specific and each .java file may contain multiple classes. So for each CK metric we decide to calculate minimum, maximum, and average value of each kind. Therefore, for Java projects, we can have 39+10*3=69 Code metrics.

3.1.2 Process Metrics. Besides collecting code metrics, there are also literature[27][12]using process metrics as features in defect prediction. In this project, we collect 4 kinds of process metrics, which are listed below:

- *NumRevision:* The total number of revision of the file before the release time
- *LineAdded:* The total lines of code added to the file before the release time
- *LineDeleted:* The total liens of code of the file deleted before the release time
- *NumAuthor:* The total number of distinct programmers who have revised the file before the release time

The process metrics are collected by parsing the commits before the release time.

3.1.3 Label Dataset. After collecting the file data, we need to identify the defective files in the dataset. We decide to follow the method of [27]. They use keyword matching method to identify bug files from commits. The regular expression used is:

(?!de)bug|(?!pre)fix|issue|error|crash|problem|fail|defect|patch|incorrect

In the original paper, they roughly use 6 months ago timestamp as the release time. However, the projects in GitHub may contain multiple release times and an identified bug commits after the predefined release time may not refer to that specific release time. Therefore after finding the bug commits, we will examine whether the files involved in the bug commits have been modified before. If the files have been modified between the release time and the commit time, we will not use this commit to label the files. The process can be illustrated in figure 2.

3.2 Data Imbalance

After collecting the dataset and tentatively testing them on the ML models, we find that the data imbalance issue in our collected dataset can significantly reduce the performance of the classifiers.

	Projects	Version	Stars	Files	Defects	Ratio
Java	maven	v3.0	1.9k	826	82	9.927%
	maven	v3.3.0	1.9k	944	172	18.22%
	spring-boot	v1.4.0	3.3k	2848	110	3.86%
	spring-framework	v5.1.0	33.8k	6473	310	4.79%
	guava	v20.0	34.8k	1632	807	49.45%
C	git	v2.11.0	30.4k	385	172	44.68%
	neo-vim	v0.2.0	34.2k	115	45	39.13%
	mpv	v1.4.0	10.9k	289	72	24.91%
	openssl	v1_1_0	11.8k	844	150	17.77%

Table 1: Collected Dataset

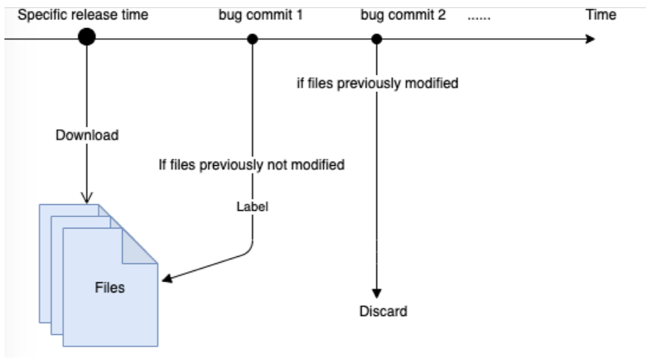

Figure 2: Dataset Labelling Process

Therefore, before applying these dataset to models, we decide to use SMOTE[3] to make our dataset more balanced.

SMOTE stands for Synthetic Minority Oversampling Technique. This is a statistical technique for increasing the number of instances in your dataset in a balanced way. The module works by generating new instances from existing minority instances. SMOTE does not change the number of majority class.

3.3 Models

After reading literature in defect prediction, we dicide to incorporate 5 widely used ML models in our study.

- 3-Layer Neural Network
- Naive Bayes
- XGBoost
- Random Forest
- Logistic Rgression

We also notice that Naive Bayes is reported to be preferable than others[15] [8]. Few of the literature conduct research based on more powerful models such as XGBoost. We are very curious about whether more powerful classifiers can yield better performance.

3.4 Feature Ranking

Despite comparing different classifiers, we also want to select the best feature combination for each classification model. Naively we can enumerate all possible combination and select the best one. However, this method can be computational costly. Instead, we compute a chi-square score between each feature and class. Then we rank the features based on the descending order of the chi2 score. The chi2 score can capture the relevance between the feature and class. The bigger the score, the more relevant between feature and class.

When finding the best feature combination, we select the top-k features after ranking as the input features of the model. We vary the value k and use grid search method to select the best k value for each classifier and project.

3.5 Evaluation Metrics

The following evaluation metrics are used in this project.

- **Precision.** Precision measures the proportion of actual defective files that are predicted as defects against all predicted defective files. It is defined as: $prec = TP/TP + FP$
- **Recall.** Recall evaluates the proportion of actual defective files that are predicted as defects against all actual defects. It is defined as $rec = TP/(TP + FN)$
- **AUC.** AUC is the area under the receiver operating characteristics (ROC) curve. ROC is independent of the cut-off value that is used to compute the confusion matrix.
  The AUC is known as a useful measure for comapring different models and is widely used because AUC is unaffected by class imbalance as well as being independent from cutoff probability (prediction threshold) that is used to decide whether an instance should be classified as positive or negative[16].

4 EXPERIMENTAL RESULT AND DISCUSSION

4.1 For a standalone evaluation metric, is there a dominant model with a single set of metrics that has the best performance in WPDP setting across all projects?

In this experiment. We apply all models to each project and 5-fold cross-validation is used. For each model, grid search is introduced to find the best top k features based on a specific evaluation metric. The evaluation metrics we are interested in are AUC and Recall. Notice that we do not include precision as a metric because of the class imbalance in defect prediction dataset. We can achieve a high precision by predicting most of the instances as non-defective.

For each metric, we report the metric value of each model on each project. For each model, we only report best metric value across all k values. For AUC, the results are shown in table 2. For recall, the results are shown in table 3. For clarity, an example interpretation can be illustrated by the block with a footnote marker 2 in table 2.

From the tables we can see for any of the evaluation metrics, the dominant model and the best feature combination can both vary a lot. So the strict answer to the research question in this section is No. However, we observed that there is a tendency that 3-layer neural network is more preferable when considering only AUC since it dominant 5 projects out of 9, especially in Java projects (4 out of 5). SVM is more preferable when considering solely recall. It dominant 7 out of 9 projects. One interesting finding is that though the overall performance in all proejcts of more powerful models such as XGBoost and Random Forest may not as good as we expect, they can sometimes perform extremely well when evaluating AUC. The AUC value can be achieved up to 0.7 by these two model, which we never observe on other models.

4.2 For CPDP, whether we can obtain comparable performance as WPDP by considering only programming language as context factor?

In order to answer this question, we divide the dataset into two sets. The first set consists all Java projects and the second one consists all C projects. Then we do CPDP for each project within the same set. Specifically, for each project, we will train the predictor on the remaining projects within the same set, and apply the predictor to the project.

After this, we find that the performance is not as good as we thought. All AUC values are around 0.50, which means that the model has no class separation capacity. Therefore, we try to improve the performance by normalize the features. For each project, we scale each feature value into 0-1 before feeding them into the predictor. The motivation here is that after examining the dataset, we find that even thought within the same programming language set, the feature scale can be quite different across projects.

However, even though we normalize each feature, there is no obvious performance improvement. Most of AUC values still remain around 0.5. For each programming language set, we select one project with the best AUC values and report their performance in the table 4.

From the table we can see that even for the best AUC values, we can only slightly exceed 0.6, which is obtained by SVM model on git project. We conclude that even though for the projects with the same programming language, the feature distribution difference is still very big. When building CPDP models, well-designed feature engineering is absolutely necessary to yield good performance.

4.3 For a single project, can cross-version defect prediction(CVDP) model be as predictive as WPDP?

When collecting dataset, we find that for a single project, it may have multiple release version. For example, in the datset, we have two version of maven project, one is version3.0 and the other is version 3.3.0. One interesting question arose here is that whether we can use the predictor built on one version to predict defects in the other version. Here we train each model on one version and test on the other and the grid search method is also introduced to find the best k value. We report the performance of each model with the best k value in Table 5 and Table 6

From tables we can see that the performance of cross-version defect is not so promissing. The best AUC value is achieved by Random Forest. However, the AUC value is still below 0.6, which is far worse than WPDP.

We think this is casued by the huge time span between the two versions. Maven3.0 was released 9 years ago, whereas Maven3.3.0 was released 4 years ago. During these five years, changes such as programmers and new features added to the project are all potential reasons that lead to the significant change of feature distribution. Therefore, when building the CVDP models, we should carefully check release time of the two version to make sure they are not too far away.

5 CONCLUSION

In this project, we initially modify a widely used defects data collection technique to collect up-to-date defect data by ourselves. Then based on the dataset we collect, an extensive study on defect prediction is conducted. First, we apply different machine learning models to all projects in WPDP setting. For AUC and recall, we respectively report the best model with the best top k features. We find that for a single evaluation metric, there is not a strictly universal model that can dominant across all projects. But for AUC the 3-Layer Neural Network is preferable whereas the SVM is preferable for Recall. Second, we do CPDP under the same programming language context. We make a bold assumption that the feature distribution difference is mainly caused by adoption of different programming language. However, the result show that even though we normalize the features, the CPDP performance is still not promising–almost every model fail to achieve AUC value of 0.6, which is far worse than WPDP. Finally, a CVDP experiment is conducted to show the feasibility of applying a model trained on one version of the project to the other version of the same project. According to AUC metric, the CVDP model is not comparable to WPDP model. We guess this is caused by the long time span between these two version.

6 FUTURE WORK

6.1 Obtain More Accurate Dataset

The data collection method in this project is borrowed from [27]. We think this method has extraordinary meaning in defect prediction because it allow researchers to collect defect data by themselves. Previously, the defect prediction community heavily relied on these datasets [6] [17] [26] [18] [21] [25]. These datasets are outdated. As time goes by, the evolution of software development such as new features introduced in programming language and the emergence of new design patterns will significantly change the feature distributions of defect data. Therefore a solid method of collecting defect data is vital for researcher to propose more applicable models.

However, during defect data collection process, we find the keyword matching method is not as reliable as we thought. There will

[2]Explanation: When using top 67 features, 3-Layer Neural Network can yield best AUC value on maven3.0 in WPDP setting. The AUC value is 0.635.

AUC							
	Project	**3-Layer NN**	**Naive Bayes**	**SVM**	**XGBoost**	**Random Forest**	**Logistic Regression**
Java	*maven3.0*	0.635 (k = 67) [2]	0.481 (k = 61)	0.641 (k = 63)	**0.770 (k = 60)**	0.716 (k = 64)	0.594 (k = 66)
	maven3.3	**0.671 (k = 61)**	0.563 (k = 66)	0.597 (k = 61)	0.532 (k = 69)	0.617 (k = 62)	0.639 (k = 62)
	spring-framework	**0.672 (k = 65)**	0.611 (k = 68)	0.671 (k = 60)	0.619 (k = 70)	0.655(k = 60)	0.661 (k = 68)
	spring-boot	**0.651 (k = 65)**	0.560 (k = 65)	0.627 (k = 63)	0.574 (k = 60)	0.615 (k = 67)	0.615 (k = 60)
	guava	**0.642 (k = 64)**	0.583 (k = 64)	0.599 (k = 62)	0.562 (k = 63)	0.611 (k = 60)	0.621 (k = 60)
C	*git*	0.54 (k = 30)	0.634 (k = 35)	**0.678 (k = 30)**	0.664 (k = 36)	0.637 (k = 39)	0.675 (k = 39)
	neo-vim	0.53 (k = 34)	0.43 (k = 32)	0.585 (k = 36)	0.669 (k = 31)	**0.772 (k = 30)**	0.66 (k = 30)
	mpv	**0.672 (k = 37)**	0.542 (k = 30)	0.621 (k = 35)	0.661 (k = 38)	0.648 (k = 39)	0.55 (k = 36)
	openssl	0.628 (k = 39)	0.581 (k = 38)	0.623 (k = 39)	0.683 (k = 36)	**0.709 (k = 39)**	0.564 (k = 39)

Table 2: WPDP AUC

Recall							
	Project	**3-Layer NN**	**Naive Bayes**	**SVM**	**XGBoost**	**Random Forest**	**Logistic Regression**
Java	maven3.0	0.428 (k = 63)	0.428 (k = 63)	0.809 (k = 60)	**0.857 (k = 60)**	0.714 (k = 61)	0.380 (k = 61)
	maven3.3	0.648 (k = 62)	0.594 (k = 61)	**1.000 (k = 61)**	0.703 (k = 64)	0.675 (k = 64)	0.594 (k = 62)
	spring-framework	0.462 (k = 63)	0.444 (k = 65)	**0.962 (k = 64)**	0.351 (k = 69)	0.518 (k = 60)	0.537 (k = 68)
	spring-boot	0.884 (k = 65)	0.798 (k = 63)	**1.000 (k = 62)**	0.939 (k = 60)	0.750 (k = 67)	0.585 (k = 69)
	guava	0.865 (k = 64)	0.798 (k = 64)	**1.000 (k = 62)**	0.939 (k = 60)	0.737 (k = 64)	0.567 (k = 60)
C	git	0.631 (k = 30)	0.473 (k = 38)	**0.921 (k = 34)**	0.815 (k = 36)	0.736 (k = 36)	0.657 (k = 37)
	neo-vim	0.571 (k = 34)	0.571 (k = 31)	**0.857 (k = 30)**	0.857 (k = 30)	0.857 (k = 30)	0.571 (k = 30)
	mpv	0.727 (k = 37)	0.727 (k = 39)	**0.909 (k = 30)**	0.727 (k = 33)	0.636 (k = 39)	0.636 (k = 36)
	openssl	0.580 (k = 37)	0.516 (k = 39)	0.774 (k = 30)	**0.935 (k = 35)**	0.903 (k = 30)	0.548 (k = 38)

Table 3: WPDP Recall

Model	**Spring-boot**	**git**
3-Layer Neural Network	0.552	0.506
Naive Bayes	**0.598**	0.576
SVM	0.589	**0.611**
XGBoost	0.561	0.565
Random Forest	0.556	0.572
Logistic Regression	0.580	0.600

Table 4: Selected CPDP AUC Result

maven3.3.0->maven3.0			
Model	Precision	Recall	AUC
3-Layer Neural Network	0.096	0.975	0.490
Naive Bayes	0.108	0.580	0.531
SVM	0.114	0.630	0.550
XGBoost	0.120	0.716	**0.572**
Random Forest	0.113	0.531	0.539
Logistic Regression	0.114	0.580	0.544

Table 6: CVDP from maven3.3.0 to maven3.0

maven3.0->maven3.3.0			
Model	Precision	Recall	AUC
3-Layer Neural Network	0.221	0.444	0.549
Naive Bayes	0.211	0.485	0.541
SVM	0.216	0.421	0.541
XGBoost	0.236	0.596	**0.585**
Random Forest	0.220	0.573	0.562
Logistic Regression	0.243	0.480	0.575

Table 5: CVDP from maven3.0 to maven3.3.0

be a little false positive instances and false negative instances in our collected dataset. Due to the time constrain, we do not clean our dataset. We identify a possible approach which can potentially remove noisy data [8]. On the other hand, we can also adopt other methods to identify bugs in commit messages beyond simple keyword matching technique. This is also a binary classification problem and we believe this can be achieved by some Natural Language Processing (NLP) techniques.

6.2 Beyond Software Metrics

A software metric is a measure of software characteristics which are quantifiable or countable. But it is just a surface-level proxy of a characteristic of the code. When collecting the dataset, we find there are cases where only one line code changed maybe because of the wrong usage of a function or a variable. We believe this kind of minor change of the code is hard to be captured by any of existing software metrics. Therefore, no matter how powerful our model is, it can not tell the differences between the defective instance and the non-defective one.

Why not think outside of the box? Instead of dealing with metrics features, why can not we directly dive into the code itself? We have seen many Deep Learning + NLP techniques applied to code generation tasks and some of them showed us promising results. We think this End-to-End approach can also be used in defect prediction task. This will be a very interesting research direction we can follow.

6.3 Beyond File Defect Prediction

The usage of defect prediction is to help to allocate limited resources in testing and maintenance [15] [23]. In real production setting, module testing is always involved and a module typically consists of multiple files. However, in this project, we only focus on file-level defect prediction. Besides, from the dataset we also find that the defective files always occur in the same directory, which means there is a high probability that they belong to the same module. Therefore, we think shifting the focus to module-level defect prediction is more practical and may yield better performance compared to file-level.

ARTIFACTS RELEASE

All codes and collected dataset are available here.[3]

ACKNOWLEDGEMENT

First I want to thank my teammates, Sitao Lu and Yuchen Wang for their continuous contribution to the projects and their constant encouragement when I suffer depression. Then I want to thank our professor, Jin Guo for her careful and patient guidance, especially when we get stuck during the project. Finally, I want to thank all members of the class for their valuable feedback and active discussion. Thank you all for giving me a wonderful and extraordinary experience this semester!

REFERENCES

[1] Lionel C Briand, VR Brasili, and Christopher J Hetmanski. 1993. Developing interpretable models with optimized set reduction for identifying high-risk software components. *IEEE Transactions on Software Engineering* 19, 11 (1993), 1028–1044.
[2] Cagatay Catal. 2011. Software fault prediction: A literature review and current trends. *Expert systems with applications* 38, 4 (2011), 4626–4636.
[3] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* 16 (2002), 321–357.
[4] Giovanni Denaro. 2000. Estimating software fault-proneness for tuning testing activities. In *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium*. IEEE, 704–706.
[5] Giovanni Denaro, Luigi Lavazza, and Mauro Pezze. 2003. An empirical evaluation of object oriented metrics in industrial setting. In *The 5th CaberNet Plenary Workshop, Porto Santo, Madeira Archipelago, Portugal*.
[6] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2012. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering* 17, 4-5 (2012), 531–577.
[7] Taghi M Khoshgoftaar, Naeem Seliya, and Kehan Gao. 2005. Assessment of a new three-group software quality classification technique: An empirical case study. *Empirical Software Engineering* 10, 2 (2005), 183–218.
[8] Sunghun Kim, Hongyu Zhang, Rongxin Wu, and Liang Gong. 2011. Dealing with noise in defect prediction. In *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 481–490.
[9] A Güneş Koru and Hongfang Liu. 2007. Identifying and characterizing change-prone classes in two large-scale open-source products. *Journal of Systems and Software* 80, 1 (2007), 63–73.
[10] A Güneş Koru and Jeff Tian. 2003. An empirical comparison and characterization of high defect and high complexity modules. *Journal of Systems and Software* 67, 3 (2003), 153–163.
[11] Ying Ma, Guangchun Luo, Xue Zeng, and Aiguo Chen. 2012. Transfer learning for cross-company software defect prediction. *Information and Software Technology* 54, 3 (2012), 248–256.
[12] Suvodeep Majumder, Rahul Krishna, and Tim Menzies. 2019. Learning GENERAL Principles from Hundreds of Software Projects. *arXiv preprint arXiv:1911.04250* (2019).
[13] Tim Menzies, Andrew Butcher, Andrian Marcus, Thomas Zimmermann, and David Cok. 2011. Local vs. global models for effort estimation and defect prediction. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 343–351.
[14] Tim Menzies, Justin DiStefano, Andres Orrego, and R Chapman. 2004. Assessing predictors of software defects. In *Proc. Workshop Predictive Software Models*.
[15] Tim Menzies, Jeremy Greenwald, and Art Frank. 2006. Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering* 33, 1 (2006), 2–13.
[16] Jaechang Nam and Sunghun Kim. 2015. Heterogeneous defect prediction. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*. ACM, 508–519.
[17] Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. 2013. Transfer defect learning. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 382–391.
[18] Fayola Peters and Tim Menzies. 2012. Privacy and utility for defect prediction: Experiments with morph. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 189–199.
[19] Fayola Peters, Tim Menzies, and Andrian Marcus. 2013. Better cross company defect prediction. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 409–418.
[20] Foyzur Rahman and Premkumar Devanbu. 2013. How, and why, process metrics are better. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 432–441.
[21] Daniel Rodriguez, Israel Herraiz, and Rachel Harrison. 2012. On software engineering repositories and their open problems. In *2012 First International Workshop on Realizing AI Synergies in Software Engineering (RAISE)*. IEEE, 52–56.
[22] Ramanath Subramanyam and Mayuram S. Krishnan. 2003. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on software engineering* 29, 4 (2003), 297–310.
[23] Ayse Tosun, Ayse Bener, and Resat Kale. 2010. Ai-based software defect predictors: Applications and benefits in a case study. In *Twenty-Second IAAI Conference*.
[24] Burak Turhan and Ayse Bener. 2009. Analysis of Naive Bayes' assumptions on software fault data: An empirical study. *Data & Knowledge Engineering* 68, 2 (2009), 278–290.
[25] Burak Turhan, Tim Menzies, Ayşe B Bener, and Justin Di Stefano. 2009. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering* 14, 5 (2009), 540–578.
[26] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. 2011. Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 15–25.
[27] Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. 2014. Towards building a universal defect prediction model. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 182–191.

[3]https://github.com/abnerdang/762-project-public