Survey on Software Defect Prediction

Jaechang Nam

Abstract

Software defect prediction is one of the most active research areas in software engineering. Defect prediction results provide the list of defect-prone source code artifacts so that quality assurance teams can effectively allocate limited resources for validating software products by putting more effort on the defect-prone source code. As the size of software projects becomes larger, defect prediction techniques will play an important role to support developers as well as to speed up time to market with more reliable software products.

In this survey, we first introduce the common defect prediction process used in the literature and how to evaluate defect prediction performance. Second, we compare different defect prediction techniques such as metrics, models, and algorithms. Third, we discuss various approaches for cross-project defect prediction that is an actively studied topic in recent years. We then discuss applications on defect prediction and other emerging topics. Finally, based on this survey, we identify challenging issues for the next step of the software defect prediction.

I. Introduction

Software Defect (Bug) Prediction is one of the most active research areas in software engineering [9], [31], [40], [47], [59], [75].[1] Since defect prediction models provide the list of bug-prone software artifacts, quality assurance teams can effectively allocate limited resources for testing and investigating software products [40], [31], [75].

We survey more than 60 representative defect prediction papers published in about ten major software engineering venues such as Transaction on Software Engineering (TSE), International Conference on Software Engineering (ICSE), Foundations of Software Engineering (FSE) and so on in recent ten years. Through this survey, we investigated major approaches of software defect prediction and their trends.

We first introduce the common software defect prediction process and several research streams in defect prediction in Section II. Since different evaluation measures for defect prediction have

[1]Defect and bug will be used interchangeably across this survey paper.

been used across the literature, we present evaluation measures for defect prediction models in Section III. Section IV and V discusses defect prediction metrics and models used in the representative papers. In Section VI, we discuss about prediction granularity. Before building defect prediction models, some studies applied preprocessing techniques to improve prediction performance. We briefly investigate preprocessing techniques used in the literature in Section VII. As recent defect prediction studies have focused on cross-project defect prediction, we compare various cross-project defect prediction approaches in Section VIII. In the next two sections of Section VIII, we discuss about applications using defect prediction results and emerging topics. Finally, we conclude this survey by raising challenging issues for defect prediction in Section XI.

II. Overview of Software Defect Prediction

A. *Software Defect Prediction Process*

Figure 1 shows the common process of software defect prediction based on machine learning models. Most software defect prediction studies have utilized machine learning techniques [3], [6], [10], [20], [31], [40], [45].

The first step to build a prediction model is to generate instances from software archives such as version control systems, issue tracking systems, e-mail archives, and so on. Each instance can represent a system, a software component (or package), a source code file, a class, a function (or method), and/or a code change according to prediction granularity. An instance has several metrics (or features) extracted from the software archives and is labeled with buggy/clean or the number of bugs. For example, in Figure 1, instances generated from software archives are labeled with 'B' (buggy), 'C' (clean), or the number of bugs.

After generating instances with metrics and labels, we can apply preprocessing techniques, which are common in machine learning. Preprocessing techniques used in defect prediction studies include feature selection, data normalization, and noise reduction [27], [40], [47], [63], [71]. Preprocessing is an optional step so that preprocessing techniques were not applied on all defect prediction studies, e.g., [10], [31].

With the final set of training instances, we can train a prediction model as shown in Figure 1. The prediction model can predict whether a new instance has a bug or not. The prediction for bug-proneness (buggy or clean) of an instance stands for binary classification, while that for the number of bugs in an instance stands for regression.

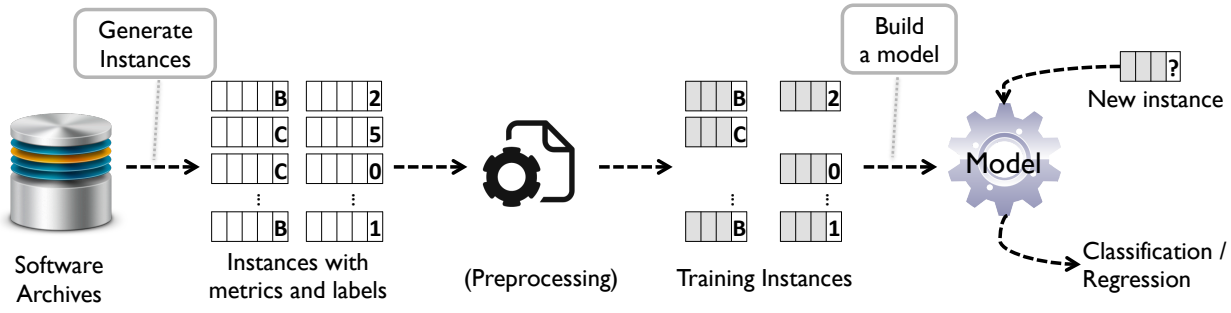

Fig. 1: Common process of software defect prediction

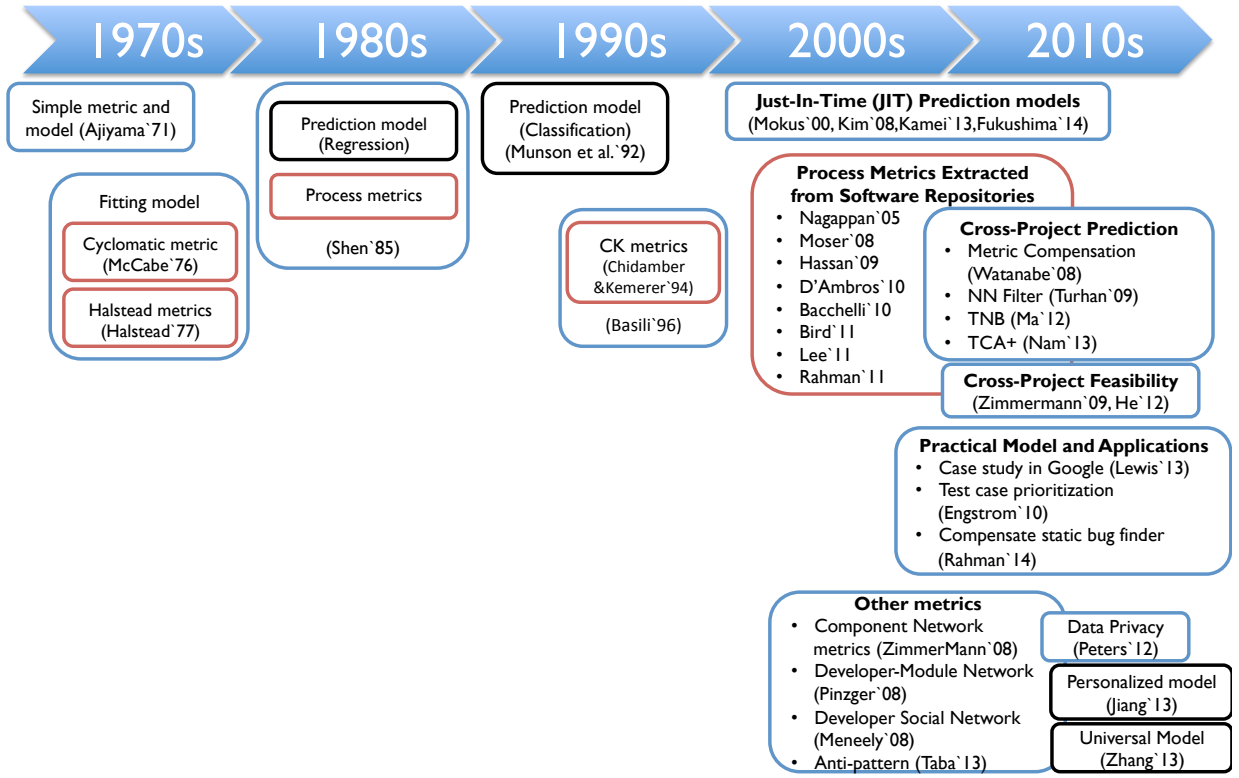

Fig. 2: History of software defect prediction studies

B. Brief History of Software Defect Prediction Studies

Figure 2 briefly shows the history of defect prediction studies in about last 50 years. The first study estimating the number of defects was conducted by Akiyama in 1971 [1]. Based on the

assumption that complex source code could cause defects, Akiyama built a simple model using lines of code (LOC) since LOC might represent the complexity of software systems. However, LOC is too simple metric to show the complexity of systems. In this reason, MaCabe and Halstead proposed the cyclomatic complexity metric and Halstead complexity metrics in 1976 and 1977 respectively [38], [18]. These metrics were very popular to build models for estimating defects in 1970s and the early of 1980s [13].

Having said that though, the models studied in that period were not actually prediction model but just fitting model that investigated the correlation between metrics and the number of defects [13]. These models were not validated on new software modules. To resolve this limitation of previous studies, Shen et al. built a linear regression model and test the model on the new program modules [60]. However, Munson et al. claimed that the state-of-the art regression techniques at that time were not precise and proposed a classification model that classify modules into two groups, high risk and low risk [43]. The classification model actually achieved 92% of accuracy on their subject system [43]. However, Munson et al.'s study still have several limitations such as no metrics for object-oriented (OO) systems and few resources to extract development process data. As Shen et al. pointed out at that time [60], it was not possible to collect error fix information informally conducted by individual developers in unit testing phases.

In terms of OO systems, Chidamber and Kemerer proposed several object-oriented metrics in 1994 [7] and was used by Basili et al. to predict defects in object-oriented system [4]. In 1990s, version control systems were getting popular, development history was accumulated into software repositories so that various process metrics were proposed from the middle of 2000s [3], [6], [9], [20], [31], [42], [45], [59].

In 2000s, there had been existed several limitations for defect prediction. The first limitation was the prediction model could be usable before the product release for the purpose of quality assurance. However, it would be more helpful if we can predict defects whenever we change the source code. To make this possible, Mockus et al. proposed a defect prediction model for changes [41]. Recently, this kind of models is called as just-in-time (JIT) defect prediction models. JIT prediction models have been studied by other researchers in recent years [15], [25], [26].

The second limitation is that it was not possible or difficult to build a prediction model for new

projects or projects having less historical data. As use of process metrics was getting popular, this limitation became the one of the most difficult problems in software defect prediction studies [76]. To resolve this issue, researchers proposed various cross-project defect prediction models [36], [47], [67], [69]. In cross-project defect prediction, identifying cross-prediction was another issue so that Zimmermann et al. and He et al. conducted the study on cross-prediction feasibility [76], [22].

The third limitation was from the question, "are the defect prediction models really helpful in industry?". In this direction, several studies such as case study and proposing practical applications have been conducted [12], [33], [57].

There have been several studies following the trends of information technology as well. By using social network analysis and/or network measures, new metrics were proposed by Zimmermann et al. [75], Pinzger et al [54], and Taba et al. [66]. Privacy issue of defect datasets was also addressed by Peters et al. [53]. New concepts of prediction models have been proposed such as personalized defect prediction model [23] and universal model [73] recently.

In the next subsection, we categorized these studies by related subtopics.

C. Categories of Software Defect Prediction Studies

Table I lists the representative studies in software defect prediction. Many research studies in a decade have focused on proposing new metrics to build prediction models. Widely studied metrics are source code and process metrics [56]. Source code metrics measure how source code is complex and the main rationale of the source code metrics is that source code with higher complexity can be more bug-prone. Process metrics are extracted from software archives such as version control systems and issue tracking systems that manage all development histories. Process metrics quantify many aspects of software development process such as changes of source code, ownership of source code files, developer interactions, etc. Usefulness of process metrics for defect prediction has been proved in many studies [20], [31], [45], [56].

As introduced in Section II-A, most defect prediction studies are conducted based on statistical approach, i.e. machine learning. Prediction models learned by machine learning algorithms can predict either bug-proneness of source code (classification) or the number of defects in source code (regression). Some research studies adopted recent machine learning techniques such as active/semi-supervised learning to improve prediction performance [34], [68]. Kim et al. proposed

TABLE I: Representative studies in software defect prediction

Type	Categories	Representatives
Within/Cross	Metrics	Source code [40], Process (Churn [45], [10], Change [42], Entropy [20], Popularity [3], Authorship [55], Ownership [6], MIM [31]), Network measure [39], [54], [75], Antipattern [66]
	Algorithm/Model	Classification, Regression, Active/Semi-supervised learning [34], [68], BugCache [29]
	Finer prediction granularity	Change classification [26], Method-level prediction [21]
	Preprocessing	Feature selection/extraction [63], [45], Normalization [40], [47], Noise handling [27], [71]
Cross	Transfer learning	Metric compensation [69], NN Filter [67], TNB [36], TCA+ [47]
	Feasibility	Decision Tree [76], [22]

BugCache algorithm, which utilizes locality information of previous defects and keeps a list of most bug-prone source code files or methods [29]. BugCache algorithm is a non-statistical model and different from the existing defect prediction approaches using machine learning techniques.

Researchers also focused on finer prediction granularity. Defect prediction models tried to identify defects in system, component/package, or file/class levels. Recent studies showed the possibility to identify defects even in module/method and change levels [21], [26]. Finer granularity can help developers by narrowing the scope of source code review for quality assurance.

Proposing preprocessing techniques for prediction models is also an important research branch in defect prediction studies. Before building a prediction model, we may apply the following techniques: feature selection [63], normalization [40], [47], and noise handling [27], [71]. With the preprocessing techniques proposed, prediction performance could be improved in the related studies [27], [63], [71].

Researchers also have proposed approaches for cross-project defect prediction. Most representative studies described above have been conducted and verified under the within-prediction setting, i.e. prediction models were built and tested in the same project. However, it is difficult for

new projects, which do not have enough development historical information, to build prediction models. Representative approaches for cross defect prediction are metric compensation [69], Nearest Neighbour (NN) Filter [67], Transfer Naive Bayes (TNB) [36], and TCA+ [47]. These approaches adapt a prediction model by selecting similar instances, transforming data values, or developing a new model [67], [47], [36], [69].

Another interesting topic in cross-project defect prediction is to investigate feasibility of cross-prediction. Many studies confirmed that cross-prediction is hard to achieve; only few cross-prediction combinations work [76]. Identifying cross-prediction feasibility will play a vital role for cross-project defect prediction. There is a couple of studies regarding cross prediction feasibility based on decision trees [76], [22]. However their decision trees were verified only in specific software datasets and were not deeply investigated [76], [22].

III. Evaluation Measures

For defect prediction performance, various measures have been used in the literature.

A. Measures for Classification

To measure defect prediction results by classification models, we should consider the following prediction outcomes first:

- True positive (TP): buggy instances predicted as buggy.
- False positives (FP): clean instances predicted as buggy.
- True negative (TN): clean instances predicted as clean.
- False negative (FN): buggy instances predicted as clean.

With these outcomes, we can define the following measures, which are mostly used in the defect prediction literature.

1) False positive rate (FPR): False positive rate is also know as probability of false alarm (PF) [40]. PF measures how many clean instances are predicted as buggy among all clean instances.

$$\frac{FP}{TN + FP} \tag{1}$$

2) Accuracy:

$$\frac{TP + TN}{TP + FP + TN + FN} \tag{2}$$

Accuracy considers both true positives and true negatives over all instances. In other words, accuracy shows the ratio of all correctly classified instances. However, accuracy is not proper measure particularly in defect prediction because of class imbalance of defect prediction datasets. For example, average buggy rate of PROMISE datasets used by Peters et al. [53] is 18%. If we assume a prediction model that predicts all instances as clean, the accuracy will be 0.82 although no buggy instances are correctly predicted. This does not make sense in terms of defect prediction performance. Thus, accuracy has not been not recommended for defect prediction [58]

3) Precision:

$$\frac{TP}{TP + FP} \tag{3}$$

4) Recall: Recall is also know as probability of detection (PD) or true positive rate (TPR) [40]. Recall measures correctly predicted buggy instances among all buggy instances.

$$\frac{TP}{TP + FN} \tag{4}$$

5) F-measure: F-measure is a harmonic mean of precision and recall [31].

$$\frac{2 \times (Precision \times Recall)}{Precision + Recall} \tag{5}$$

Since precision and recall have trade-off, f-measure has been used in many papers [27], [31], [47], [56], [71].

6) AUC: AUC measures the area under the receiver operating characteristic (ROC) curve. The ROC curve is plotted by PF and PD together. Figure 3 explains about a typical ROC curve. PF and PD vary based on threshold for prediction probability of each classified instance. By changing the threshold, we can draw a curve as shown in Figure 3. When the model gets better, the curve tends to be close to the point of PD=1 and PF=0. Thus, AUC of the perfect model will have "1". For a random model, the curve will be close to the straight line from (0,0) to (1,1) [40], [58]. AUC with 0.5 is regarded as the random prediction [58]. Other measures such as precision and recall can vary according to prediction threshold values. However, AUC considers prediction performance in all possible threshold values. In this reason, AUC is a stable measure to compare different prediction models [58].

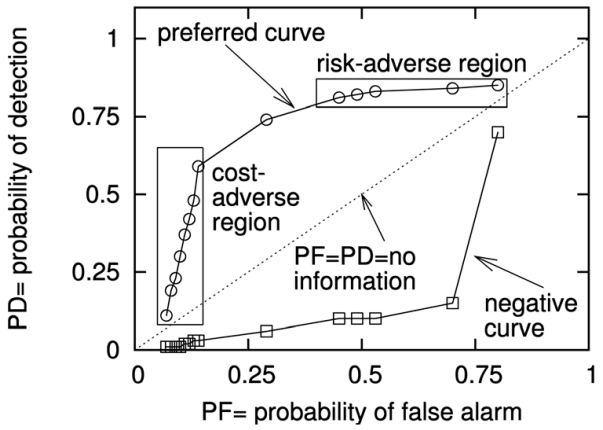

Fig. 3: A typical ROC curve [40]

7) AUCEC: Area under cost-effectiveness curve (AUCEC) is a defect prediction measure considering lines of code (LOC) to be inspected or tested by quality assurance teams or developers. The idea of cost-effectiveness for defect prediction models is proposed by Arisholm et al. [2]. Cost-effectiveness represents how many defects can be found among top $n\%$ LOC inspected or tested. In other words, if a certain predection model can find more defects with less inspecting and testing effort comparing to other models, we could say the cost-effectiveness of the model is higher.

Figure 4 shows cost-effectiveness curves as examples [59]. The x-axis represents the percentage of LOC while y-axis represents the percentage of defects found [59]. In the left side of the figure, three example curves are plotted. Let assume the curves, O, P, and R, represent the optimal, practical, and random models, respectively [59]. If we consider the area under the curve, the optimal model will have the highest AUCEC comparing to other models [59]. In case of the random model, AUCEC will be 0.5. The higher AUCEC of the optimal model means that we can find more defects by inspecting or testing less LOC than other models [59].

However, considering AUCEC from the whole LOC may not make sense [59]. In the right side of Figure 4, the cost-effectiveness curves of the models, P1 and P2, are identical so that considering the whole LOC for AUCEC does not give any meaningful insight [59]. Having said that, if we set a threshold as top 20% of LOC, the model P2 has higher AUCEC than R and

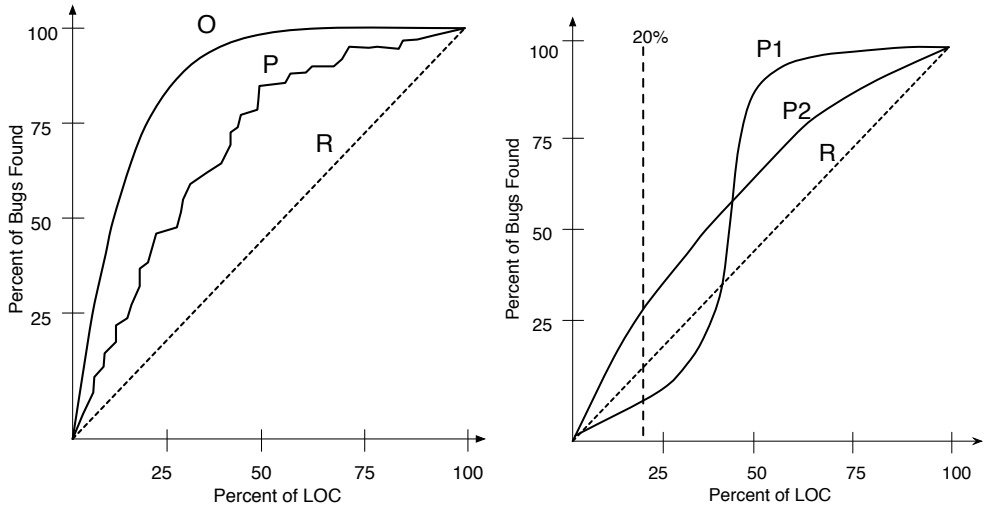

Fig. 4: Cost-effectiveness curve [59]

P2 [59]. In this reason, we need to consider a particular threshold for the percentage of LOC to use AUCEC as a prediction measure [59].

B. Measures for Regression

To measure defect prediction results from regression models, measures based on correlation calculation between the number of actual bugs and predicted bugs of instances have been used in many defect prediction papers [3], [46], [62], [75]. The representative measures are Spearman's correlation, Pearson correlation, R^2 and their variations [3], [46], [62], [75]. These measures also has been used for correlation analysis between metric values and the number of bugs [75].

C. Discussion on Measures

Figure 5 shows the count of evaluation measures used in the representative defect prediction papers for classification. As shown in the figure, f-measure is the most frequently used measure for defect classification. Since there are trade-off between precision and recall, comparing different models are not easy as some models have high precision but low recall and vice versa for other models. Since f-measure is a harmonic mean of precision and recall and provides one

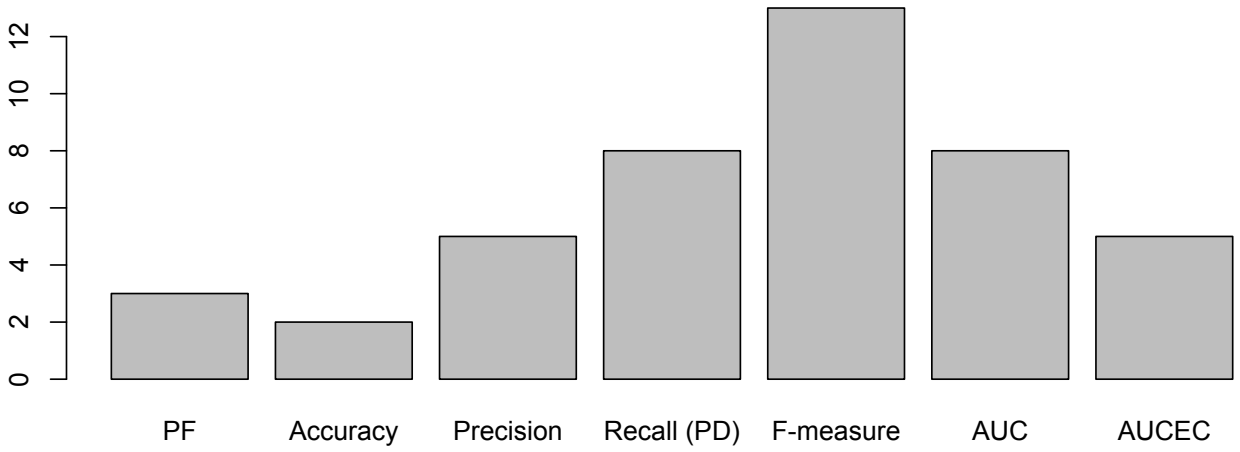

Fig. 5: Count of evaluation measures used in the representative defect prediction papers for classification.

single score as prediction performance, f-measure has been used to compare different prediction models in many defect prediction papers [27], [31], [47], [53], [58], [61].

However, f-measure varies by different thresholds (cutoffs) for prediction probability of an instance. When a model predicts an instance as buggy, it provides prediction probability that represents if the instance is buggy. Lessmann et al. pointed out that what thresholds were used in evaluation usually overlooked in the defect prediction literature so that it led to inconsistent prediction results across defect prediction papers [32].

To overcome the limitation of f-measures, researchers also used other measures such as AUC and AUCEC that are independent from the thresholds [16], [32], [58], [65]. Particularly, AUCEC recently used a lot by Rahmann et al. since AUCEC could be a good measure to evaluate prediction models in the view of practical use of models [56], [58], [59].

IV. Defect Prediction Metrics

Defect prediction metrics play the most important role to build a statistical prediction model. Most defect prediction metrics can be categorized into two kinds: code metrics and process metrics. Code metrics are directly collected existing source code while process metrics are collected from historical information archived in various software repositories such as version control and issue tracking systems.

A. *Code metrics*

Code metrics also known as product metrics measure complexity of source code. Its ground assumption is that complexity source is more bug-prone. To measure code complexity, researchers proposed various metrics.

TABLE II: Representative code metrics

Code Metrics	Size [1]	Halstead [18]	McCabe [38], [48]	CK [7], [4]	OO [11]
Venue	IFIPSC'71	(Book)'77	TSE'76,'96	TSE'94,'96	JSS'94

The size metrics measure "volume, length, quantity, and overall magnitude of software products" [8]. The representative of size metrics is lines of code (LOC). To our knowledge, Akiyama's model was the earliest study to predict defects using LOC [1]. Afterwards, LOC was used in most defect prediction papers to build a model [10], [21], [31], [40], [61], [65].

Halstead proposed several size metrics based on the number of operators and operands [18]. The proposed metrics are program vocabulary, length, volume, difficult, effort, and time [18]. Most metrics are related to size or quantity. Halstead metrics has been used popularly in many studies [40], [65], [67].

McCabe proposed the cyclomatic metric to represent complexity of software products [38]. Cyclomatic metric is computed by the number of nodes, arcs and connected components in control flow graphs of source code. This metric represents how much control paths are complex. Since McCabe's cyclomatic metric measure the complexity of source code structure, its characteristic is inherently different from size and Halstead metrics that measure volume and quantity of source code. Ohlsson and Alberg adopted McCabe's cyclomatic metric to predict fault-prone modules in telephone switched [48] and other defect prediction studies [32], [40], [42], [47], [65], [67] also used McCabe's cyclomatic metric to build a prediction model.

Since object-oriented programming is getting popular, code metrics for object-oriented languages have proposed to improve development process. The representative metrics for object-oriented programs are Chidamber and Kemerer (CK) metrics [7]. Table III lists CK metrics. The CK metrics were designed from the characteristics from object-oriented languages such as

inheritance, coupling, and cohesion [7]. Basili et al. [4] validated if it is possible to build a defect prediction model using the CK metrics. After that, many studies include CK metrics to build prediction models [3], [10], [24], [27], [31], [50], [71].

TABLE III: CK metrics [7]

Name	Description
WMC	Wighted methods per class
DIT	Depth of inheritance tree
NOC	Number of children
CBO	Coupling between object classes
RFC	Response for a class
LCOM	Lack of cohesion in methods

Besides the CK metrics, other object-oriented (OO) metrics based on volume and quantity of source code, have been proposed as well [11]. As size metrics, the OO metrics simply counts the number of instance variables, methods, and etc. as shown in Table IV. Many defect prediction studies for object-oriented programs have used the OO metrics to build prediction models [10], [27], [31], [50], [71], [74], [75].

B. Process metrics

Table V lists seven representative process metrics. In this section, we briefly introduce the metrics and their design concepts.

1) Relative code change churn: Nagappan and ball [45] proposed 8 relative code churn metrics (M1-M8) measuring the amount of code changes [45]. For example, M1 metric is computed by churned LOC (the accumulative number of deleted and added lines between a base version and a new version of a source file) divided by Total LOC [45]. Other metrics (M2-M8) consider various normalized changes such as deleted LOC divided by total LOC, file churned (the number of changed files in a component) divided by file count, and so on [45]. In case study by Nagappan and ball, the relative churn metrics is proved as a good predictor to explain the defect density of a binary and bug-proneness [45].

TABLE IV: Example class-level OO metrics used by D'Amabros et al [10]

Name	Description
FanIn	Number of other classes that reference the class
FanOut	Number of other classes referenced by the class
NOA	Number of attributes
NOPA	Number of public attributes
NOPRA	Number of private attributes
NOAI	Number of attributes inherited
LOC	Number of lines of code in a class
NOM	Number of methods
NOPM	Number of public methods
NOPRM	Number of private methods
NOMI	Number of methods inherited

TABLE V: Representative process metrics

Process Metrics	# of Metrics	Metric Source	Venue
Relative code change churn [45]	8	Version control	ICSE'05
Change [42]	17	Version control	ICSE'08
Change Entropy [20]	1	Version control	ICSE'09
Code metric churn, Code Entropy [9], [10]	2	Version control	MSR'09
Popularity [3]	5	E-mail archive	FASE'10
Ownership [6]	4	Version control	FSE'11
Micro interaction metrics [31]	56	Mylyn	FSE'11

2) Change metrics: Change metrics are to measure the extent of changes in the history recorded in version control systems. For example, we can count the number of revisions/bug-fix changes/refactorings of a file and the number of authors editing a file. Moser et al. [42] extracted 18 change metrics from the Eclipse repositories to conduct a comparative analysis

between code and change metrics. Moser et al.'s change metrics [42] also include added and deleted LOC similar to relative code change churn [45]. However, Moser et al.'s change metrics did not consider any relativeness by the total LOC and the file count but consider average and maximum values of change churn metrics. Moser et al.'s metrics also include maximum and average of change sets (the number of files committed together) and age metrics (age of a file in weeks and the weighted age normalized by added LOC) [42]. Moser et al. concluded that change metrics are better predictors than code metrics [42].

3) Change Entropy: Hassan applied Shannon's entropy to capture how changes are complex and proposed history complexity metric (HCM) [20]. To validate the HCM, Hassan built statistical linear regression models based on HCM or two change metrics, the number of previous modifications and previous faults on six open-source projects [20]. Their evaluation on six open-source projects showed that prediction models build using HCM outperform those using the two change metrics [20]. The idea adopting the Entropy concept to measure change complexity is novel but comparing models by HCM to those by only two change metrics reveals the weakness of the evaluation of HCM. In addition, evaluation was conducted in the subsystem-level rather than the file-level.

4) Code metric churn, Code Entropy: D'Ambros et al. conducted extensive comparisons study of defect prediction metrics [9]. In their metric comparison, there is no study about code metric churn and code Entropy while code churn and change Entropy metrics have studied as introduced in previous subsections [45], [20]. Thus, D'Ambros et al. proposed code metric churn (CHU) and code Entropy (HH) metrics [9].

In contrast to code change metrics based on the amount of lines, CHU measures the change in biweekly basis of code metrics such as CK metrics and OO metrics. Since code metric churn computes the amount of changes in biweekly basis [9], CHU captures the extent of changes more precisely than code change churn that computes the amount of changes between a base revision and a new revision [45]. Four variants of CHU by applying decay functions (WCHU, LDCHU, EDCHU, and LGDCHU) also were proposed [9].

While change Entropy is computed based on the count of file changes, code Entropy (HH) is computed based on the count of involved files when a certain code metric is changed. As in CHU, D'Ambros et al. also defined four variants of HH by applying decay functions (HWH, LDHH, EDHH, and LGDHH) [9].

In the comparison evaluation, D'Ambros et al. concluded that WCHU and LDHH metrics led to good prediction results on all subjects used in their experiments [9]. However, limitations of these novel metrics is heavy computation resources and data because it tracks biweekly changes from version control systems [9].

5) Popularity: Bacchelli et al. proposed popularity metrics by analyzing e-mail archives by developers in a group mailing list [3]. The main idea of popularity metrics is more discussed software artifacts in e-mail archives are more bug-prone [3]. Table VI lists the popularity metrics. Most metrics quantify how many times a certain class are discussed in mails [3]. The extracting metrics from e-mail archives is novel but their evaluation of the metrics shows that popularity metrics themselves did not outperform other code and process metrics [3].

TABLE VI: Popularity metrics [3]

Name	Description
POP-NOM	The number of mails discussing a class.
POP-NOCM	The number of characters in all mail discussing a class.
POP-NOT	The number of e-mail threads discussing different topics for a class
POP-NOMT	The number of e-mails in a thread discussing a class in at least one of mails in a thread.
POP-NOA	The number of authors motioning about the same class.

6) Ownership and Authorship: Bird et al. proposed four ownership metrics based on authorship of a component [6]. This study is started from the question, "How much does ownership affect quality?" [6]. The ownership of a component is defined by the portion of commits of the component and minor and major contributors are defined by less than and more than 5% portions of the ownership respectively [6]. Four ownership metrics are defined a follows; MINOR (the number of minor contributors), MAJOR (the number of major contributors), TOTAL (the total number of contributors), and OWNERSHIP (portion of ownership of the contributor with the highest portion of ownership). They concluded that higher ownership leads to less bug-prone [6].

Rahman et al. conducted the fine-grained investigation on relationship between defects and human factors such as ownership and developer experience [55]. The interesting finding of this study is that quality assurance should be focused on source code files touched by less experienced

developers [55].

7) Micro interaction metrics: Lee at al. proposed micro interaction metrics (MIM) extracted from Mylyn that captures developer interactions to Eclipse [31]. The main idea of MIM is from the fact that defects could be introduced by mistakes of developers [31]. For example, more editing time of a certain source code file may cause more bug-proneness. Since Mylyn data contains developer's interactions to Eclipse, Lee et al. extracted 56 metrics from Mylyn data and compared their performance with code and process metrics [31].

In their experiments, MIM outperformed code and process metrics in both classification and regression [31]. However, MIM is highly depended on Mylyn, a plug-in of Eclipse so that MIM might be hard to apply for other development environments that does not support the tool like Mylyn [31].

C. Other metrics

Apart from code and process metrics, researchers proposed new kinds of metrics based on existing knowledge such as network measure [39], [54], [75] and anti-pattern [66].

Meneely et al. extracted developer metrics from a developer social network that represents collaboration structure extracted from source code repositories [39]. Based on this developer social network, this study found that software failure is highly correlated with developer network metrics [39].

Pinzger et al. also constructed developer network but also with software modules, i.e. developer-module network [54]. This network represents how each developer contributes to each module so that the network is called 'contribution network' as well [54]. Pinzger et al. found that the centrality measures for the contribution network can prediction post-release defects significantly [54].

Zimmermann et al. constructed dependency (such as data and call dependencies) graphs of binaries and conducted network analysis on those dependency graphs [75]. From various network analysis measures such as centralness, closeness, betweenness, and so on, Zimmermann et al. built prediction models and compared them to models constructed by code and process metrics [75]. In their evaluation, network measure could predict more bug-prone binaries than code and process metrics.

Taba et al. [66] proposed four antipattern metrics. Antipatterns are poor design of software so that there might be higher chance to introduce defects in the source code files [66]. In

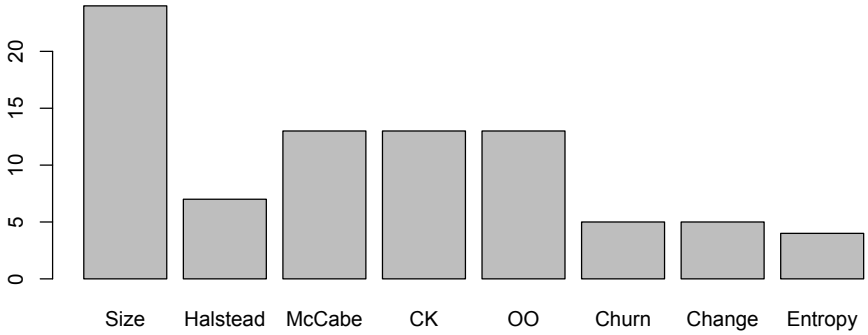

Fig. 6: Use frequency of defect prediction metrics in the representative defect prediction papers.

their evaluation with two open source projects, antipattern metrics could improve prediction performance in terms of f-measure [66].

D. Discussion on code metrics vs. process metrics

Figure 6 shows the use frequency of metrics in the representative defect prediction papers. Since code metrics such as size, Hastead, McCabe, CK and OO metrics have used from 1970s or 1990s, absolute use frequency of code metrics is higher than process metrics. In addition, code metrics used a lot for comparison study whenever new kinds of metrics are proposed. Most process metrics have been proposed in 2000s from when software repositories such as version control and issue tracking systems get popular.

There are lots of debates if code metrics are good defect predictors and process metrics are better than code metrics. Menzies et al. confirmed that code metrics are still useful to build a defect prediction model [40]. However, according to Rahman et al.'s recent study comparing code and process metrics, code metrics is less useful than process metrics because of stagnation of code metrics [56].

V. Defect Prediction Models

Most defect prediction models are based on machine learning. Depending on what to predict (bug-proneness of the number of bugs), models based on machine learning are divided into two types, classification and regression. Since new machine learning techniques are being developed, active or semi-supervised learning techniques have been applied to build better defect prediction

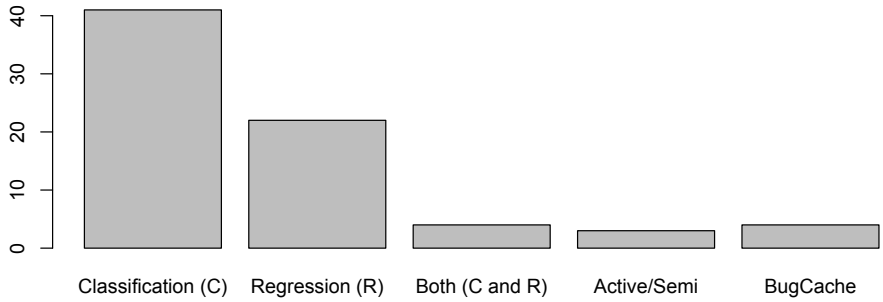

Fig. 7: Use frequency of defect prediction models in the representative defect prediction papers.

models as well [34], [35]. Apart from machine learning models, non-statistical model such as BugCache [29] has been proposed.

Figure 7 shows the use frequency of defect prediction models in the representative defect prediction papers. Since statistical models based on machine learning studied for a long time, classification and regression models are dominant models. As Kim et al. proposed BugCache [29], there were a couple of studies investigating BugCache models as well as case studies [12], [33], [59].

Classification and regression have the similar prediction process since they are based on machine learning. The difference between classification and regression models is what to predict. Classification models usually identify bug-proneness [40], [42], [76] while regression models predict the number of bugs [45], [44], [75]. The answer for the question, 'which model should be used by quality assurance teams?' is depended on the intended purpose of the model users.

Figure 8 shows the use frequency of representative machine learners in the literature. Logistic regression is the most frequent machine learners in the representative papers. Naive Bayes and Decision Tree are also frequently used in the literature.

In terms of machine learners for regression models, Linear Regression and Negative Binomial Regression have been mostly used in the literature [6], [10], [24], [31], [49], [62], [70].

As new machine learning approaches such as active or semi-supervised learning have been proposed, software engineering community has tried to adopt those approaches for defect prediction. Lu and Cukic [35] proposed defect prediction models based on active learning where a sample set of instances is selected and the instances are asked to an oracle (human professionals)

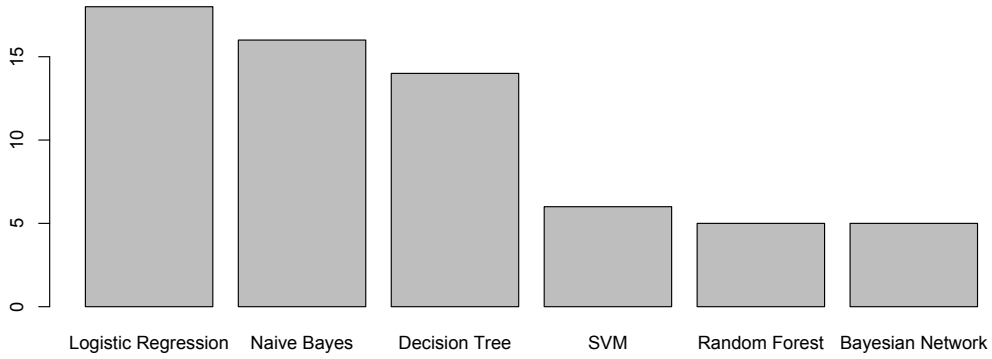

Fig. 8: Use frequency of classification machine learners in the representative defect prediction papers.

if the instances could be a good training set. Li et al. proposed CoFest and ACoFest [35]. CoFest is a sampling approach based on semi-supervised learning by repeatedly evaluating prediction performance with a random sample through Random Forest to find the best sample [35]. ACoFest is an extended version of CoFest by applying active learning [35].

Kim et al. proposed BugCache, which maintains the priority of bug-prone entities in a cache [29]. In their evaluation on seven open source projects, 10% of files have 73% - 95% of whole defects [29]. The BugCache facilitates locality information of bugs such as temporal and spacial locality (if a bug of an entity is introduced recently or the entity is changed with other entities, those entities might have bugs with higher change.) [29].

VI. Prediction Granularity

In the literature, defect prediction models were constructed in various levels of granularity such as sub-system [14], [20], [30], component/package [32], [45], [75], file/class [40], [37], [47], [49], [76], method [16], [21], and change (hunk) [26]. Since the resource allocation for software quality assurance can be conducted by quality assurance teams' own focus, studies on defect prediction models seem to be conducted on various granularity levels.

A recent study by Hata et al. [21] proposed method-level defect prediction and concluded that method-level defect prediction is more cost-effective than other higher granularity levels such as package- and file-levels.

Kim et al. proposed a novel defect prediction model called change classification [26]. Different from common defect prediction models, change classification can be directly helpful to developers since a change classification model can provide an instant prediction result whenever a developer makes any change on source code files and commit it to a version control system [26]. However, change classification models are too heavy to use in practice since the models are built by more than ten thousand features [26].

To use defect prediction models in practice, we should consider cost-effectiveness [58], [59]. One of the ways to improve cost-effectiveness of prediction models is to predict defects in finer-grained levels [21]. In this sense, researchers need to more focus on defect prediction on finer-grained levels such as line-level defect prediction and change classification.

VII. Preprocessing for Defect Datasets

Preprocessing is a widely used step in machine learning. Since most defect prediction studies are based on machine learning, there are several studies using preprocessing techniques [27], [40], [47], [63], [71]. Depending on each study, preprocessing techniques are selectively used or not used since many studies are conducted by different metrics, models, and subjects.

A. Normalization

Normalization is a common technique to give a same weight for metric values to improve performance of classification models [19], [17].

Menzies et al. recommended to use the logarithmic filter (log-filter) to normalize metric values [40] for metrics that have exponential distribution. Other studies using the same experimental subjects used by Menzies also applied the log-filter [40], [67], [68].

Nam et al. observed that cross-prediction performance varies on different normalization techniques [47]. Nam et al. defined rules for selecting proper normalization techniques such as min-max normalization, z-score and variations of z-score to improve the performance of cross-predictions [47].

B. Feature selection and extraction

Shivaji et al. pointed out that the poor performance of defect prediction models is due to a number of metrics to build the models [63]. In this reason, Shivaji et al. proposed a feature

selection technique that can improve change classification [63]. Turhan et al. also applied feature subset selection using information gain before conducting cross-company defect prediction [67].

Since defect prediction datasets may have the multicollinearity issue, researchers applied principal component analysis (PCA) to extract new features for prediction models [9], [45], [75].

C. Noise reduction

Since defect data are usually collected from version control and issue tracking systems automatically by using tools and algorithms [28], [64], defect data may be bias as in Bird et al.'s study [5]. To reduce noise, Wu et al. proposed ReLink that automatically recovers the correct links between commit logs and issue IDs [71]. Kim et al. proposed a noise detection and elimination algorithm called Closest List Noise Identification (CLNI) [27].

VIII. Cross-project Defect Prediction

Since new software projects do not have enough training data, building a good prediction model for the new projects is a challenging issue. For example, Zimmermann et al. conducted 622 cross-predictions and found only 3.4% actually worked [76]. To improve cross-prediction performance, researchers focused on studies based on transfer learning and cross-prediction feasibility.

A. Transfer Learning

Transfer learning is one of very active research areas in machine learning [51]. Figure 9 explains difference between traditional machine learning and transfer learning. Traditional machine learning assumes that distribution of training and test data is same. Thus, in case that the distribution changed, it is required to rebuild the model with newly collected data. However, collecting new data and labeling them is cost-expensive [51]. In this reason, researchers in machine learning community have focused on transfer learning where we can transfer knowledge from a domain with enough training data to another domain with few training data to build a leaning model as shown in Figure 9 [51].

Software engineering community has been adopted transfer learning concepts and techniques for cross-project defect prediction. Table VII summarizes the representative works for cross-project defect prediction using transfer learning.

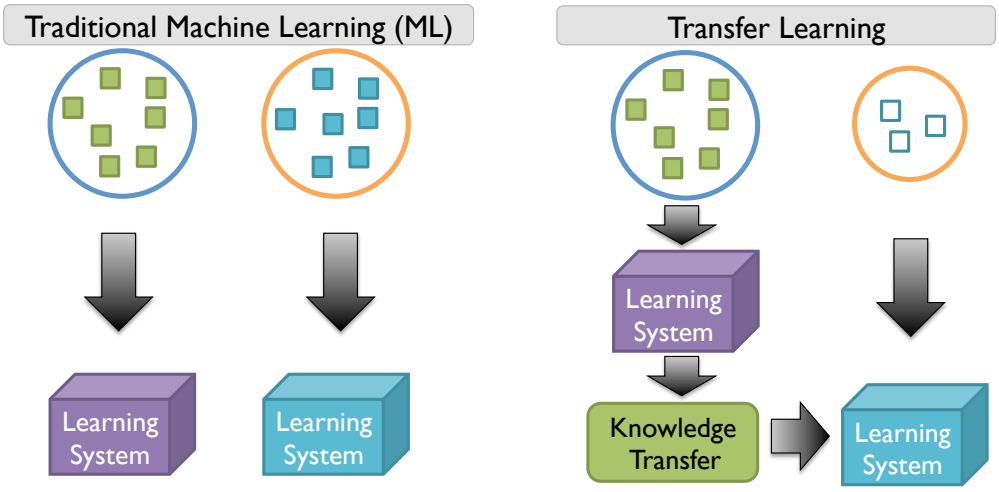

Fig. 9: Traditional machine learning vs. transfer learning [52].

TABLE VII: Cross-project defect prediction based on transfer learning techniques.

Transfer learning	Metric compensation [69]	NN Filter [67]	TNB [36]	TCA+ [47]
Preprocessing	N/A	Feature selection, Log-filter	Log-filter	Normalization
Machine learner	C4.5	Naive Bayes	TNB	Logistic Regression
#Subjects	2	10	10	8
#Predictions	2	10	10	26
Avg. F-measure	0.67 (W:0.79,C:0.58)	0.35 (W:0.37,C:0.26)	0.39 (NN:0.35,C:0.33)	0.46 (W:0.46,C:0.36)
Venue	PROMISE'08	ESEJ'09	IST'12	ICSE'13

1) Metric compensation: Watanabe et al. applied metric compensation to transform target data by using source data [69]. The main idea of metric compensation is to normalize each metric (feature) value of target data by using the average metric value of corresponding metric of source data [69]. In detail metric compensation is designed as follows [69]:

$$new\_v_{T,i,j} = \frac{v_{T,i,j} \times avg\_v_{S,j}}{avg\_v_{T,j}} \quad (6)$$

, where $new\_v_{T,i,j}$ is each compensated metric value of i-th instance of j-th metric of target, $v_{T,i,j}$ is each metric value of i-th instance of j-th metric of target, and $avg\_v_{S,j}$ and $avg\_v_{T,j}$ are average values of j-th metric of source and target respectively.

With two project datasets, Watanabe et al. conducted cross-project predictions and reported precision and recall in the paper [69]. We computed average f-measure for cross-predictions with/without metric compensation and within-predictions (W) as shown in Table VII. Average f-measure (0.67) of cross-predictions with metric compensation outperforms that (0.58) of cross-predictions without metric compensation but still is worse than that (0.79) of within-predictions.

The major limitation of the work conducted by Watanabe et al. [69] is the weak evaluation of their approach; only two cross-predictions were conducted and there is no statistical test to validate their research questions [69].

2) NN filter: Turhan et al. applied nearest neighbour filter (NN filter) to improve performance of cross-company defect prediction [67]. The basic idea of the NN filter is to collect similar source instances to target instances to train a prediction model. In other words, if we can build a prediction model using selected source instances that have similar data characteristics to target instances, the model may perform better on predicting target instances than the model trained by using all source instances [67]. The NN filter chooses 10 source instances as nearest neighbours for each target instance.

To evaluate performance of cross-company defect prediction using the NN filter, Turhan et al. conducted experiments with ten proprietary datasets from NASA and SOFTLAB [67]. In addition, they conducted Mann-Whitney U test to validate their experimental results. As we computed average f-measure from their PD and PF results, the average f-measure (0.35) in cross-predictions with the NN filter is better than that (0.26) without the NN filter. However, within-predictions were still the best comparing to cross-predictions.

3) Transfer Naive Bayes: Ma et al. proposed Transfer Naive Bayes (TNB) for cross-company defect prediction [36]. The basic idea of TNB is to compute new prior probability and conditional probability of Naive Bayes model by using the weight value of a source instance. The weight of the source instance is computed according to the similarity between the source instance and

target instances.

To compute the instance similarity between a source instance and target instances, min and max values of each feature of a test dataset is used. In other words, the similarity is computed by the number of features of a source instance whose feature values are in between min and max values of a corresponding target feature. Then, the weight value is computed by using Newton's Universal Gravitation law [36]. The more weight of a training instance means the more similarity to test instances. Finally, TNB model is computed with new prior and conditional probabilities by using these weights for source instances.

As shown in Table VII, TNB (0.39) led to better prediction performance than NN filter (0.35) in terms of average f-measure. Please note different cross-prediction (CC) results without NN filter by Turhan et al. [67] and Ma et al. [36]. The reason is that Ma et al. did not apply feature selection in their experimental setting [36].

The limitation of this study is that TNB is not applicable for other machine leaning algorithms that do not use prior and conditional probabilities. In addition, Ma et al. did not report within-prediction results so that we could not conclude cross-prediction using TNB is comparable with within-prediction [36].

4) TCA+: Nam et al. proposed TCA+ for cross-project defect prediction [47]. TCA+ is an extended version of transfer component analysis (TCA) that is a state-of-the-art transfer learning algorithm proposed by Pan et al. [52]. TCA tries to find a common latent feature space where the distribution of source and target datasets are similar by using projection. Projection is a feature extraction technique to reduce feature space in machine learning. Principal component analysis (PCA) is a representative feature extraction approach by projecting instances in lower dimensional space [51]. While PCA tries to keep original data characteristics on lower dimensional feature space, TCA tries to find a lower dimensional feature space where source and target data have similar distribution as well as to keep original data characteristics as PCA does [51]. Figure 10 clearly shows how PCA and TCA results are different. PCA result shown in the center of the figure explains that distributions between source (red) and target (blue) are still different in the latent feature space. However, TCA result in the right side of the figure shows that distribution between source and target is similar in the new latent feature space [51].

Applying TCA for cross-predictions, Nam et al. [47] observed that prediction performance varies by what normalization approach is applied for preprocessing. In this reason, Nam et

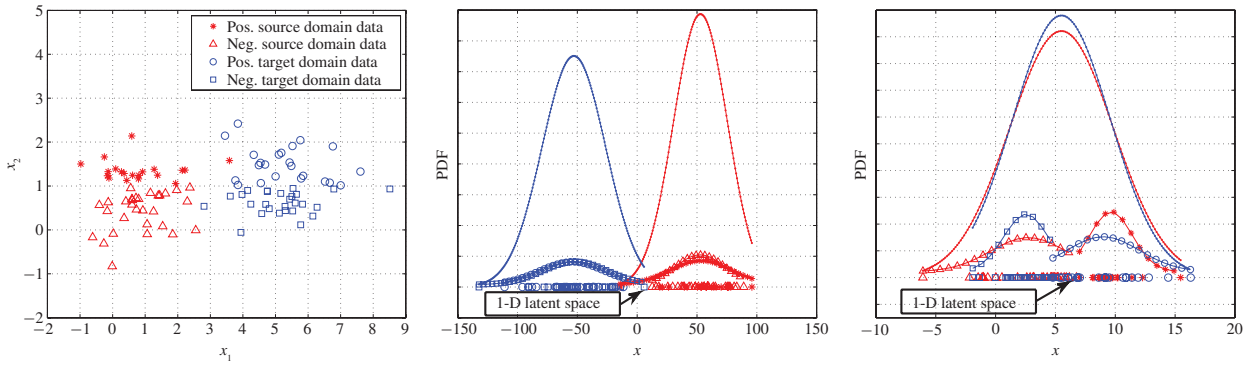

Fig. 10: Projection result on one dimensional space by PCA (center) and TCA (right) [52]. Figure in the left side shows instances in the original feature space (two dimensional).

al. proposed TCA+ by adding decision rules to select proper normalization options into TCA. For normalization options, min-max, z-score, and variations of z-score are used in their experiments [47]. As shown in Table VII, cross-prediction result (0.46) in TCA+ show comparable to within-prediction result (0.46) in terms of average f-measure.

5) Discussion on cross-predictions based on transfer learning: Until now, we compare various approaches for cross-project defect prediction. The main goal of cross-project defect prediction is to reuse existing defect datasets to build a prediction model for a new project or a project lacking in the historical data. However, all approaches discussed above could conduct cross-predictions across datasets with the same feature space. As shown in Table VII, the number of subjects used in TCA+ is 8 but the number of predictions are 26. If we consider all possible cross-prediction combinations, it should be 56 ($= 8 \times (8-1)$). However, it could not be done because of the different feature space of datasets. In TCA+ experiments, the size of feature space of three datasets is 26 while that of five datasets is 61. Thus, cross-predictions could be possible within the datasets with the same feature space, ie. 6 cross-predictions ($= 3 \times (3-1)$) and 20 cross-predictions ($= 5 \times (5-1)$). Achieving cross-predictions on datasets with different feature spaces is an open question to be resolved.

B. Cross-prediction Feasibility

There are few studies on cross-project feasibility [76], [22] Zimmermann et al. [76] built a decision tree to validate cross-project predictability by using project characteristics such as languages used and number of developers. However, the decision tree was constructed and validated within the subjects used in their empirical study so that the decision tree could not be used general purpose [76].

He et al. [22] also constructed the decision tree based on cross prediction results to validate cross-project feasibility. Their decision tree is built by difference of distributional characteristics of source and target datasets such as mean, median, variance, skweness and so on [22]. However, validation of the decision tree is conducted on the best prediction results on different samples of training sets so that the validation results do not fully support its validity.

IX. Applications on Defect Prediction

One of major goals of defect prediction models is effective resource allocation for inspecting and testing software products [40], [59]. However, the case studies using defect prediction models in industry is few [12], [33]. In this reason, many studies by Rahman et al. [58], [59], [56] considers cost-effectiveness. A recent case study conducted in Google by Lewis et al. [33] comparing BugCache and Rahman's algorithm based on the number of closed bugs [59] found that developers preferred Rahman's algorithm. However, developers still did not get benefits from using defect prediction models [33].

One of recent studies conducted by Rahman [57] showed that defect prediction could be helpful to prioritize warnings reported by static bug finders such as FindBug.

Anther possible application is that we can apply defect prediction results to prioritize or select test cases. In regression testing, executing all test suites for regression testing is very costly so that many prioritization and selection approaches for test cases have been proposed [72]. Since defect prediction results provide bug-prone software artifacts and their ranks [29], [59], [75], it might be possible to use the results for test case prioritization and selection.

X. Other Emerging Topics

Apart from the representative papers discussed in previous sections, there are interesting and emerging topics in defect prediction study. One topic is about defect data privacy [53] and the

other topic is the comparative study between defect prediction models and static bug finders [57].

A. Defect Data Privacy

Peters et al. proposed MORPH that mutates defect datasets to resolve privacy issue in defect datasets [53]. To accelerate cross-project defect prediction study, publicly available defect datasets are necessary. However, software companies are reluctant to share their defect datasets because of "sensitive attribute value disclosure" [53]. Thus, cross-project defect prediction studies usually conducted on open source software products or very limited proprietary systems [36], [47], [58], [67]. Experiments conducted by Zimmermann et al. for cross-project defect prediction are not reproducible since Microsoft defect datasets are not publicly available [76].

To address this issue, MORPH moves instances in a random distance by still keeping class decision boundary [53]. In this way, MORPH could privatize original datasets and still achieve good prediction performance as in models trained by original defect datasets [53].

B. Comparing Defect Prediction Models to Static Bug Finders

In contrast to defect prediction models (DP), static bug finders (SBF) detect bugs by using "semantic abstractions of source code" [57]. Rahman et al. compared defect prediction techniques and static bug finders in terms of cost-effectiveness [57]. Rahman et al. found that DP and SBF could compensate each other since they may find different defects [57]. In addition, SBF warnings prioritized by DP could lead to better performance than SBF's native priorities of warnings [57]. This comparative study provided meaningful insights that explains how different research streams having the same goal can be converged together to achieve the better prediction/detection of defects.

XI. Challenging Issues

Defect prediction studies are still have many challenging issues. Even though there are many outstanding studies, it is not easy to apply those approaches in practice because of following reasons:

- Most studies were verified in open source software projects so that current prediction models may not work for any other software products including commercial software. However, proprietary datasets are not publicly available because of privacy issue [53].

Although Peters et al. proposed MORPH algorithm to increase data privacy, MORPH was not validated in cross-project defect prediction [53]. Investigating privacy issue in cross-project defect prediction is required since if we have more available proprietary datasets, evaluation of prediction models will be more sound.

- Cross prediction is still a very difficult problem in defect prediction in terms of two aspects. **Different feature space:** There are many publicly available defect datasets. However, we cannot use many of datasets for cross prediction since datasets from different domains have different number of metrics (features). Prediction models based on machine learning cannot be built on the datasets, which have different feature spaces. **Feasibility:** Studies on cross prediction feasibility are not mature yet. Finding general approaches to check the feasibility in advance will be very helpful for practical use of cross prediction models.
- Since software projects are getting larger, file-level defect prediction may not be enough in terms of cost-effectiveness. There are still few studies for finer prediction granularity. Studies on finer-grained defect prediction such as line-level defect prediction and change classification are required.
- Defect prediction metrics and models proposed until now may not always guarantee generally good prediction performance. As software repositories evolve, we can extract new types of development process information, which never used for defect prediction metrics/models. New metrics and models need to be kept investigating.

REFERENCES

[1] F. Akiyama. An Example of Software System Debugging. In *Proceedings of the International Federation of Information Processing Societies Congress*, pages 353–359, 1971.

[2] E. Arisholm, L. C. Briand, and M. Fuglerud. Data mining techniques for building fault-proneness models in telecom java software. In *Proceedings of the The 18th IEEE International Symposium on Software Reliability*, ISSRE '07, pages 215–224, Washington, DC, USA, 2007. IEEE Computer Society.

[3] A. Bacchelli, M. D'Ambros, and M. Lanza. Are popular classes more defect prone? In *Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering*, FASE'10, pages 59–73, Berlin, Heidelberg, 2010. Springer-Verlag.

[4] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22:751–761, October 1996.

[5] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and Balanced? Bias in Bug-Fix Datasets. In *ESEC/FSE'09*, 2009.

[6] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't touch my code!: Examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 4–14, New York, NY, USA, 2011. ACM.

[7] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20:476–493, June 1994.

[8] S. D. Conte, H. E. Dunsmore, and V. Y. Shen. *Software Engineering Metrics and Models*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1986.

[9] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 31 –41, May 2010.

[10] M. D'Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empirical Softw. Engg.*, 17(4-5):531–577, Aug. 2012.

[11] F. B. e Abreu and R. Carapua. Candidate metrics for object-oriented software within a taxonomy framework. *Journal of Systems and Software*, 26(1):87 – 96, 1994. Achieving quality in software.

[12] E. Engström, P. Runeson, and G. Wikstrand. An empirical evaluation of regression testing based on fix-cache recommendations. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 75–78, April 2010.

[13] N. Fenton and M. Neil. A critique of software defect prediction models. *Software Engineering, IEEE Transactions on*, 25(5):675 –689, sep/oct 1999.

[14] N. Fenton, M. Neil, W. Marsh, P. Hearty, . Radliski, and P. Krause. On the effectiveness of early life cycle defect prediction with bayesian nets. *Empirical Software Engineering*, 13(5):499–537, 2008.

[15] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Ubayashi. An empirical study of just-in-time defect prediction using cross-project models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 172–181, New York, NY, USA, 2014. ACM.

[16] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall. Method-level bug prediction. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '12, pages 171–180, New York, NY, USA, 2012. ACM.

[17] A. B. A. Graf and S. Borer. Normalization in support vector machines. In *in Proc. DAGM 2001 Pattern Recognition*, pages 277–282. SpringerVerlag, 2001.

[18] M. H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[19] J. Han, M. Kamber, and J. Pei. *Data mining : concepts and techniques*. Elsevier/Morgan Kaufmann, Waltham, Mass., 3rd ed. edition, 2012.

[20] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 78–88, 2009.

[21] H. Hata, O. Mizuno, and T. Kikuno. Bug prediction based on fine-grained module histories. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 200–210, 2012.

[22] Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang. An investigation on the feasibility of cross-project defect prediction. *Automated Software Engineering*, 19(2):167–199, 2012.

[23] T. Jiang, L. Tan, and S. Kim. Personalized defect prediction. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 279–289, Nov 2013.

[24] 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 *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[25] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Trans. Softw. Eng.*, 39(6):757–773, June 2013.

[26] S. Kim, E. J. Whitehead, Jr., and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Softw. Eng.*, 34:181–196, March 2008.

[27] S. Kim, H. Zhang, R. Wu, and L. Gong. Dealing with noise in defect prediction. In *Proceeding of the 33rd international conference on Software engineering*, ICSE '11, pages 481–490, New York, NY, USA, 2011. ACM.

[28] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead. Automatic identification of bug-introducing changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 81–90, Washington, DC, USA, 2006. IEEE Computer Society.

[29] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 489–498, 2007.

[30] M. Kläs, F. Elberzhager, J. Münch, K. Hartjes, and O. von Graevemeyer. Transparent combination of expert and measurement data for defect prediction: an industrial case study. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 119–128, New York, NY, USA, 2010. ACM.

[31] T. Lee, J. Nam, D. Han, S. Kim, and I. P. Hoh. Micro interaction metrics for defect prediction. In *SIGSOFT '11/FSE-19: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, 2011.

[32] 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, July 2008.

[33] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. W. Jr. Does bug prediction support human developers? findings from a google case study. In *International Conference on Software Engineering (ICSE)*, 2013.

[34] M. Li, H. Zhang, R. Wu, and Z.-H. Zhou. Sample-based software defect prediction with active and semi-supervised learning. *Automated Software Engineering*, 19(2):201–230, 2012.

[35] H. Lu and B. Cukic. An adaptive approach with active learning in software fault prediction. In *Proceedings of the 8th International Conference on Predictive Models in Software Engineering*, PROMISE '12, pages 79–88, New York, NY, USA, 2012. ACM.

[36] Y. Ma, G. Luo, X. Zeng, and A. Chen. Transfer learning for cross-company software defect prediction. *Inf. Softw. Technol.*, 54(3):248–256, Mar. 2012.

[37] A. Marcus, D. Poshyvanyk, and R. Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Trans. Softw.*, 34(2):287 –300, 2008.

[38] T. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308–320, Dec 1976.

[39] A. Meneely, L. Williams, W. Snipes, and J. Osborne. Predicting failures with developer networks and social network analysis. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 13–23, 2008.

[40] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Trans. Softw. Eng.*, 33:2–13, January 2007.

[41] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the International Conference on Software Maintenance*, 2000.

[42] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 181–190, 2008.

[43] J. C. Munson and T. M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Trans. Softw. Eng.*, 18(5):423–433, May 1992.

[44] N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 580–586, 2005.

[45] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 284–292, 2005.

[46] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 452–461, 2006.

[47] J. Nam, S. J. Pan, and S. Kim. Transfer defect learning. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 382–391, Piscataway, NJ, USA, 2013. IEEE Press.

[48] N. Ohlsson and H. Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Trans. Softw.*, 22(12):886 –894, Dec. 1996.

[49] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE Trans. Softw. Eng.*, 31:340–355, April 2005.

[50] G. Pai and J. Bechta Dugan. Empirical analysis of software fault content and fault proneness using bayesian methods. *Software Engineering, IEEE Transactions on*, 33(10):675–686, Oct 2007.

[51] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, October 2010.

[52] S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Trans. on Knowl. and Data Eng.*, 22:1345–1359, October 2010.

[53] F. Peters and T. Menzies. Privacy and utility for defect prediction: Experiments with morph. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 189–199, Piscataway, NJ, USA, 2012. IEEE Press.

[54] 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*, SIGSOFT '08/FSE-16, pages 2–12, New York, NY, USA, 2008. ACM.

[55] F. Rahman and P. Devanbu. Ownership, experience and defects: A fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 491–500, New York, NY, USA, 2011. ACM.

[56] F. Rahman and P. Devanbu. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 432–441, Piscataway, NJ, USA, 2013. IEEE Press.

[57] F. Rahman and P. Devanbu. Comparing static bug finders and statistical prediction. In *Proceedings of the 2014 International Conference on Software Engineering*, ICSE '14, 2014.

[58] F. Rahman, D. Posnett, and P. Devanbu. Recalling the ”imprecision” of cross-project defect prediction. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE ’12, pages 61:1–61:11, New York, NY, USA, 2012. ACM.

[59] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu. Bugcache for inspections: Hit or miss? In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE ’11, pages 322–331, New York, NY, USA, 2011. ACM.

[60] V. Y. Shen, T.-J. Yu, S. M. Thebaut, and L. R. Paulsen. Identifying error-prone software an empirical study. *IEEE Trans. Softw. Eng.*, 11(4):317–324, Apr. 1985.

[61] E. Shihab, A. Mockus, Y. Kamei, B. Adams, and A. E. Hassan. High-impact defects: a study of breakage and surprise defects. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE ’11, pages 300–310, New York, NY, USA, 2011. ACM.

[62] Y. Shin, A. Meneely, L. Williams, and J. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *Software Engineering, IEEE Transactions on*, 37(6):772 –787, nov.-dec. 2011.

[63] S. Shivaji, E. Whitehead, R. Akella, and S. Kim. Reducing features to improve code change-based bug prediction. *Software Engineering, IEEE Transactions on*, 39(4):552–569, 2013.

[64] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of the 2005 international workshop on Mining software repositories*, MSR ’05, pages 1–5, 2005.

[65] Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu. A general software defect-proneness prediction framework. *Software Engineering, IEEE Transactions on*, 37(3):356–370, May 2011.

[66] S. E. S. Taba, F. Khomh, Y. Zou, A. E. Hassan, and M. Nagappan. Predicting bugs using antipatterns. In *ICSM*, pages 270–279, 2013.

[67] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Softw. Eng.*, 14:540–578, October 2009.

[68] B. Turhan, A. T. Msrl, and A. Bener. Empirical evaluation of the effects of mixed project data on learning defect predictors. *Information and Software Technology*, 55(6):1101 – 1118, 2013.

[69] S. Watanabe, H. Kaiya, and K. Kaijiri. Adapting a fault prediction model to allow inter languagereuse. In *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering*, PROMISE ’08, pages 19–24, New York, NY, USA, 2008. ACM.

[70] E. Weyuker, T. Ostrand, and R. Bell. Comparing the effectiveness of several modeling methods for fault prediction. *Empirical Software Engineering*, 15(3):277–295, 2010.

[71] R. Wu, H. Zhang, S. Kim, and S. Cheung. Relink: Recovering links between bugs and changes. In *SIGSOFT ’11/FSE-19: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, 2011.

[72] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, Mar. 2012.

[73] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou. Towards building a universal defect prediction model. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 182–191, New York, NY, USA, 2014. ACM.

[74] H. Zhang and R. Wu. Sampling program quality. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10, Sept 2010.

[75] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 531–540, 2008.

[76] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 91–100, New York, NY, USA, 2009. ACM.