Sampling Program Quality

Hongyu Zhang and Rongxin Wu
School of Software
Tsinghua University
Beijing 100084, China
{hongyu,wrx09}@tsinghua.edu.cn

Abstract— **Many modern software systems are large, consisting of hundreds or even thousands of programs (source files). Understanding the overall quality of these programs is a resource and time-consuming activity. It is desirable to have a quick yet accurate estimation of the overall program quality in a cost-effective manner. In this paper, we propose a sampling based approach - for a large software project, we only sample a small percentage of source files, and then estimate the quality of the entire programs in the project based on the characteristics of the sample. Through experiments on public defect datasets, we show that we can successfully estimate the total number of defects, proportions of defective programs, defect distributions, and defect-proneness - all from a small sample of programs. Our experiments also show that small samples can achieve similar prediction accuracies as larger samples do.**

Keywords- Sampling, program quality, software quality assurance, defect prediction, statistical quality control

I. INTRODUCTION

One challenge in today's software engineering is to deliver high-quality software on time within budget. Software quality is often measured in terms of defect [1]. Software quality assurance (SQA) is vital to the success of a software project. However, checking software quality is a resource and time-consuming activity, which may include manual code inspections, technical review meetings, static analysis, model checking and intensive software testing. Modern large software projects often consist of hundreds or even thousands of programs (source files). It could take much time and effort before we can achieve a complete understanding of a project's current quality status.

We believe it is significant to be able to obtain a quick yet accurate estimation of the overall quality of programs in a software project before full-scale SQA activities are completed. The ability to quickly estimate overall program quality can improve the cost-effectiveness of the SQA practices. For example, managers can allocate limited SQA resources more efficiently based on the estimation. For an outsourced project, the managers can quickly decide whether to reject the project due to its poor quality, without having to wait for all SQA activities to finish.

[1] Note that the wider definition of software quality includes many quality attributes such as reusability, maintainability, etc. In this study, we only measure software quality in terms of defects.

In this paper, we propose a sampling-based approach to program quality estimation. Sampling is a statistically sound and matured sampling technique [7, 20]. A small sample taken from the population can be used to draw inferences about the population. In this paper, we focus on *simple random sampling*, where each individual in a sample is chosen randomly and entirely by chance, such that each individual has the same probability of being chosen. That is, each member in a population has an equal chance of inclusion in the sample.

Although sampling has been widely used in many areas such as statistical quality control and market research, it has not been well explored in the area of software quality assurance. In our approach, we apply the simple random sampling method to SQA. We sample a small percentage of programs, examine the quality of the sampled programs, and then infer the quality of the entire population of programs in the software system based on the characteristics of the sample. In particular, we try to answer the following research questions:

- Can a sample be used to estimate the total number of defects as well as the total number of defective programs in a project?
- Can we understand the distribution of defects across programs based on a sample?
- Can we build cost-effective defect prediction models based on a sample?
- How different sample sizes affect the estimation accuracy?
- What is the optimal sampling plan for rejection/acceptance of software quality?

To answer the above questions, we perform extensive experiments on the public Eclipse defect datasets. Our findings are as follows:

- *Sample-based defect estimation*: We confirm that based on a small sample drawn from the entire population of programs, we can accurately estimate the proportion of defective programs (i.e., the number of programs having at least one defect), as well as defect totals (i.e., the total number of defects). For example, based on a 10% random sample drawn from the Eclipse 3.0, we can successfully estimate the total number of defects as well as defective programs, with the average relative prediction errors less than 5%.
- *Sample-based estimation of defect distribution*: We find that the distribution of defects in a sample is skewed, following the Weibull function. Furthermore, we can

26th IEEE International Conference on Software Maintenance in Timişoara, Romania
978-1-4244-8628-1/10/$26.00 ©2010 IEEE

estimate the distribution of defects across all the programs in a project via a small sample.

- *Sample-based defect prediction:* We can construct a classification model based on a small sample to predict the defect-proneness of un-sampled program. Sample-based defect prediction is particularly advantageous when historical defect data is not available. Furthermore, we find that a semi-supervised learning called *Co-Forest* is more effective in constructing sample-based defect prediction model.
- *The impact of different sample sizes*: We experiment quality sampling with different sample sizes. We find that larger sample sizes do not always improve estimation accuracy significantly. A small sample often achieves similar estimation accuracy as large samples do.
- *Sample-based quality control*: We propose to use sequential probability ratio test (SPRT) to help project managers make decision on rejection or acceptance of program quality based on a small sample.

Our methods are simple yet effective. The ability to estimate the quality of the whole project based on a small sample is particularly useful when we want to quickly understand the overall software quality in a limited time frame and in a cost-effective manner. Such information could help project managers make better decisions in SQA practices. We believe our methods have potential to be applied to industrial practices as a cost-effective software quality assurance measure.

The organization of the paper is as follows. In Section II, we introduce the simple random sampling method. In Section III, we describe our experiments on the application of random sampling to software quality estimation. Section IV describes the estimation of defect distribution based on a sample. Section V describes how a small sample can be used to construct a classification model for predicting defective programs. In Section VI, we introduce optimal sampling plans for software quality control. We introduce related work in Section VII, discuss the threats to validity in Section VIII and conclude the paper in Section IX.

II. RANDOM SAMPLING

When a population is very large, it is often costly and impractical to investigate each member of the population to determine the characteristics of the population. For example, television operators want to know the proportion of television viewers who watched the Olympic Game. As a practical alternative, we take a small sample from the population and use the sample statistics to draw inferences about the population parameters.

Simple random sampling is a matured sampling technique that has been widely applied to many areas such as statistical quality control, market research and public opinion survey. In simple random sampling, each member in a population has an equal chance of inclusion in the sample. We assume that sampling is done without replacement so that each member of the population will appear in the sample at most once.

Some basic statistics about simple random sampling are as follows. For a population of size N and a sample of size n ($n \leq N$), we denote the values of the sample individuals by $X_1, X_2, \ldots, X_n$. The random variable X_i could be a numerical value or a boolean value (0 or 1). The sample mean $\overline{X}$ is the average value of the sample, which is defined as $\overline{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$. If the value of each individual is a boolean value (representing the presence or absence of certain characteristic), the $\overline{X}$ is also denoted as $\hat{p}$, which is the proportion of the sample that posses the characteristics. According to the central limit theorem [7], the sampling distribution of the mean of a random sample drawn from any population is approximately normal for a large sample size. The central limit theorem also reveals that $\overline{X}$ is approximately normally distributed, even the underlying distribution of population is non-normal.

There are two major methods - point estimation and interval estimation - for estimating the properties of the population based on sample statistics. The point estimation is to directly estimate the parameters of population using selected statistics of a single sample. For example, we can use the sample mean $\overline{X}$ and its standard error $S_{\bar{x}}$ to estimate the population total T and its standard error S_T as follows:

$$T = N\overline{X}\ ,\ S_T = NS_{\bar{x}}$$

The interval estimation is to estimate the range of a population parameter calculated from the sample, at certain confidence level *1-a* ($0 \leq a \leq 1$). The confidence intervals for population mean and total are $(\overline{X} - Z_{\alpha/2}S_{\bar{x}}, \overline{X} + Z_{\alpha/2}S_{\bar{x}})$ and $(T - Z_{\alpha/2}S_T, T + Z_{\alpha/2}S_T)$ respectively, where Z is the standard normal distribution. When a is 0.05, $Z_{a/2} = 1.96$.

Table I summarizes some important equations used in sample-based estimations. For detailed derivations of the equations, we refer the readers to statistics books such as [20].

To generate a random sample, we use the SPSS statistical package and specify the approximate percentage of the individuals to be selected. We should note that any tool that supports pseudo-random number generation can perform such sampling too.

TABLE I. SOME STATISTICS OF SIMPLE RANDOM SAMPLING

Para-meter	Point Estimate	Standard Error of Point Estimate	Interval Estimate (at confidence level *1 - a*)
Mean	$\overline{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$	$s_{\bar{x}} = \frac{s}{\sqrt{n}}\sqrt{1 - \frac{n}{N}}$	$(\overline{X} - z_{a/2}s_{\bar{x}}, \overline{X} + z_{a/2}s_{\bar{x}})$
Total	$T = N\overline{X}$	$S_T = NS_{\bar{x}}$	$(\overline{T} - z_{a/2}s_{\bar{T}}, \overline{T} + z_{a/2}s_{\bar{T}})$
Propor-tion	$\overline{X} = \hat{p}$	$S_{\bar{x}} = S_{\hat{p}} = \sqrt{\frac{\hat{p}(1-\hat{p})}{n-1}}\sqrt{1 - \frac{n}{N}}$	$(\overline{X} - z_{a/2}s_{\hat{p}}, \overline{X} + z_{a/2}s_{\hat{p}})$

III. SAMPLE-BASED DEFECT ESTIMATION

We apply the simple random sampling technique to estimate defect numbers. We randomly choose a small percentage of source files as a sample, examine the quality of the sampled programs, and then estimate the quality of entire

program populations in a software project based on the analysis of the sample. We can perform both point estimation and interval estimation, for estimating defect totals and proportions of defective programs. In this way, we can achieve a quick yet accurate understanding of software quality by only examining a small fraction of the programs. In this section, we describe our experiments on sample-based estimation of defects.

A. Datasets

We first describe the datasets that are used throughout this paper. We perform experiments on the public Eclipse defect data provided by the University of Saarland[2]. Eclipse is a widely used integrated development platform for creating Java, C++ and web applications. The Eclipse defect data was collected by mining Eclipse's bug databases and version achieves [28]. It has been used by many studies on software defect prediction [17, 24, 27, 28, 30]. Using public data enables replications and comparisons of the results.

We experiment with Eclipse 2.0 and 3.0, as well as the JDT.Core component in Eclipse 3.0 and the SWT component in Eclipse 2.0. In this study we only examine the pre-release defects, which are defects reported in the last six months before release. Table II summarizes the datasets used in this study. The total number of defects and the total number of defective programs (i.e., the number of programs that contain at least one defect) are also given. Eclipse 2.0 and 3.0 are large-scale systems, containing in average 1050 KLOC, 8660 programs (source files) and 7500 defects. The JDT.Core dataset contains 939 programs, 181 KLOC and 1759 defects. The SWT dataset contains 843 programs, 194 KLOC and 501 defects.

TABLE II. THE DATASETS

Project	LOC	#programs (source files)	#defects	#defective programs
Eclipse 3.0	1306K	10,593	7422	2913 (27.50%)
Eclipse 2.0	797K	6729	7635	2611 (38.80%)
JDT.Core	181K	939	1759	502 (53.46%)
SWT	194K	843	501	208 (24.67%)

B. Estimating Defect Totals and Proportions

We now describe our method using a 10% sample of Eclipse 3.0. We first draw 10% of the programs randomly from the population of 10,593 programs in Eclipse 3.0. The resulting sample contains 1035 programs, 9.7% of total LOC, and 752 defects obtained from 28.70% (or 297) sampled programs. The sample mean (the average number of defects in the sampled program) is 0.727, with standard deviation 0.066. Note that the size of actually selected random sample is only an approximation of the specified percentage (in this case, actually 1035 instead of 1059 programs are selected), as each individual in the sample is selected from independent random numbers.

Based on the 10% sample, we can estimate the total number of defective programs (programs contain at least one defect). By applying point estimation, we estimate the proportion of the defective programs in Eclipse 3.0 as: $p = \hat{p} = 28.70\%$. We can then estimate the number of defective programs in Eclipse 3.0 based on the sample:

$$T = N\overline{X} = N\hat{p} = 10593 \times 0.287 = 3040,$$

which is about 28.69% of all programs. Note that the actual number of defective programs in Eclipse 3.0 is 2913 (27.50% of all programs), which is very close to the estimated number. We evaluate the estimation accuracy using the MRE (Magnitude of Relative Error) measure, which is computed as (100% * |Estimate - Actual|/Actual). A commonly accepted criteria is MRE<=25% [3]. Here the estimation's MRE value is only 4.36%, showing that the point estimation of defect proportion is very accurate.

We can also calculate the standard error S_T and the 95% confidence interval for the total number of defective programs as follows:

$$S_T = NS_{\hat{p}} = N\sqrt{\frac{\hat{p}(1-\hat{p})}{n-1}}\sqrt{1-\frac{n}{N}}$$

$$= 10593 \times \sqrt{\frac{0.287\,(1-0.287\,)}{1035\ -1}} \times \sqrt{1-\frac{1035}{10593}} = 142$$

$$(T - 1.96s_T, T + 1.96s_T) =$$

$$(3040 - 1.96 \times 142, 3040 + 1.96 \times 142) = (2762, 3317)$$

In similar way, we can estimate the total number of defects (defect total). We calculate the average number of defects in Eclipse 3.0 based on the 10% sample. We then estimate the defect total as follows:

$$T = N\overline{X} = 10593 \times 0.727 = 7697$$

Note that the actual number of defects in Eclipse 3.0 is 7422. The MRE value is only 3.70%, showing that the result of point estimation of defect total is very accurate. The 95% confidence interval of T is (6337, 9057).

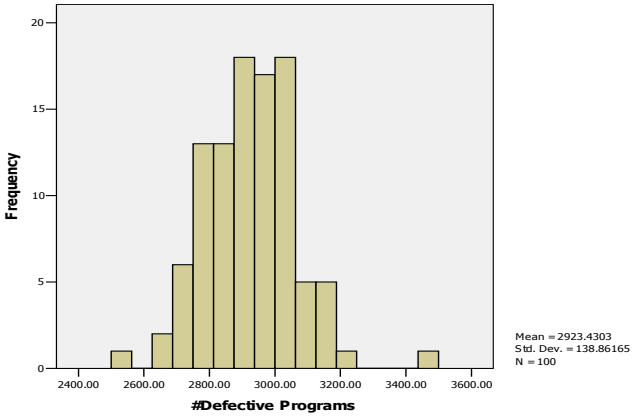

Figure 1. The estimations based on 100 * 10% samples of Eclipse 3.0

The experiment described above is based on one single random sample. To confirm the reliability of our method, we perform the above experiment 100 times. We draw 100*10% samples from the population of Eclipse 3.0 programs, estimate the total number of defective programs based on each sample, and depict a histogram of the estimations (Figure 1). We then examine the shape of sampling distribution and determine the confidence level. In Figure 1, the estimated numbers are centered on the value 2913, which is the actual number of defective programs in Eclipse 3.0. The average of the 100

[2] http://www.st.cs.uni-saarland.de/softevo/bug-data/eclipse/

estimations is 2923. The sampling distribution is approximately normal, which is consistent with the central limit theorem. It can also be computed that a vast majority (95%) of sample means differ from the population mean by less than two standard deviations. That is, if we sample the Eclipse 3.0 programs 100 times, we can obtain reasonably accurate estimation 95 times. Therefore, we are confident of using one single sample of programs to estimate the quality of the entire programs in the project.

Table III summaries the average results of 100 experiments based on 10% samples, for estimating both defective programs and defect totals for all studied projects. The averaged estimations of 100 experiments are close to the actual values. For all estimations except the estimation of defect totals for SWT, the averaged MRE values are below 25% (ranging from 1.72% to 19.95%). For the estimation of defect totals for SWT, its MRE is 26.98%, which is just slightly higher than the 25% criteria. Therefore, all estimations can be considered satisfactory. Among 100 times of sampling, 95-98 times lead to sampling errors less than two standard deviations (as shown in the Confidence column). These results show that we can obtain a good estimation of defect totals and proportions in a large project based on a small sample with high confidence.

TABLE III. THE AVERAGE DEFECT ESTIMATION RESULTS BASED ON 100*10% RANDOM SAMPLES

	Project	Actual	Estimate (T)	Std. Err. (S_T)	Confi-dence	MRE
#defective programs	Eclipse 3.0	2913 (27.50%)	2923 (27.59%)	139	95%	3.84%
	Eclipse 2.0	2611 (38.80%)	2605 (38.71%)	132	96%	4.94%
	JDT. Core	502 (53.46%)	501 (53.35%)	44	96%	7.02%
	SWT	208 (24.67%)	208 (24.67%)	38	95%	14.34%
#defect totals	Eclipse 3.0	7422	7550	591	97%	1.72%
	Eclipse 2.0	7635	7652	703	98%	7.31%
	JDT. Core	1759	1739	433	95%	19.95%
	SWT	501	510	176	95%	26.98%

C. *The Impact of Sample Size*

So far, we only described the estimation results based on the 10% samples drawn from the program population. We also experiment with different sample sizes (5%, 15%, 20% and 25% of the program population). All sample sizes achieve good estimation results. For example, for Eclipse 3.0, when estimating the defect totals, the MRE values range from 1.62% to 8.56%. When estimating the defect proportions, the MRE values range from 0.98% to 4.35%. All MRE values fall within the acceptable criteria (<=25%).

We also compare the sampling errors introduced by different sample sizes. The results for all studied projects are shown in Figures 2 and 3. The sampling error (standard error of the sample mean) decrease when the sample size n increases. However, the decreasing rate is getting slower. For example, for estimating the number of defects in Eclipse 3.0, the increase of sample size from 5% to 15% decreases the sampling error by 0.033 (from 0.080 to 0.047). While further increase of sample size to 25% only decreases the sample mean by 0.012 (from 0.047 to 0.035). For estimating defect proportions for Eclipse 3.0, the increase of sample size from 5% to 15% decreases the sample mean by 0.009 (from 0.019 to 0.010). While further increase of sample size to 25% only decreases the sample mean by 0.002 (from 0.010 to 0.008). Approximately the sampling error is reversely proportional to the square root of n: $\Delta_{\bar{x}} \propto 1/\sqrt{n}$. Therefore, when $\sqrt{n}$ is large enough, further increasing sample size will not significantly improve the estimation accuracy.

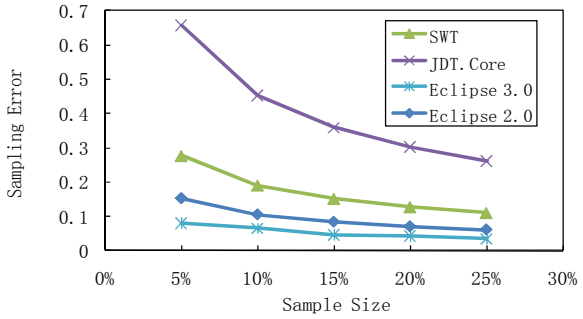

Figure 2. Sampling size vs. error (for estimating defect totals)

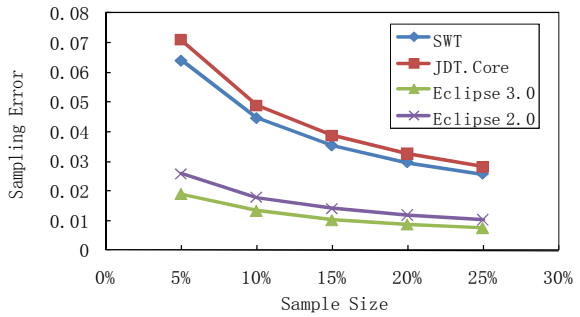

Figure 3. Sampling size vs. error (for estimating defective programs)

As software quality assurance is a time and resource consuming task, a cost-effective sampling should keep the sample size small yet representative. We explore the minimum sample size for quality estimations with respect to certain confidence level and error rate. It is known that at the confidence level *1-a*, the sampling error for defect proportion $\hat{p}$ is computed as:

$$\Delta_{\hat{p}} = Z_{a/2}\sqrt{\frac{\hat{p}(1-\hat{p})}{n-1}}\sqrt{1-\frac{n}{N}}$$

Therefore, when $N>>n$, the sample size that satisfies the requirements of confidence level and error rate is:

$$n = \left(\frac{Z_{a/2}}{\Delta_{\hat{p}}}\right)^2 \hat{p}(1-\hat{p})$$

Clearly, when $\hat{p}$ =0.5, the above equation has the maximum value. Therefore, a consecutive value for the minimum sample size n_0 is obtained as follows:

$$n_0 = \left(\frac{Z_{a/2}}{2\Delta_{\hat{p}}} \right)^2$$

As an example, if the required sampling error is less than 10% at the confidence level 90%, the above equation suggests the minimum sampling size to be 68:

$$n_0 = \left(\frac{Z_{a/2}}{2\Delta_{\hat{p}}} \right)^2 = \left(\frac{1.65}{2 \times 0.1} \right)^2 = 68$$

To test the effectiveness of using the minimum sample size for estimating defect proportions, we randomly choose 68 programs from all studied projects and then perform defect estimation based on the sample. This process is repeated 100 times and the averaged results are reported in Table IV. We can see that the estimated defect proportions are close to the actual values. All MRE values are within the acceptable range (<=25%). The results confirm that using the sample size as small as 68 programs, we can still achieve good estimation accuracy. This makes the sampling method more cost-effective in practice.

TABLE IV. THE AVERAGE RESULTS FOR ESTIMATING DEFECTIVE PROGRAMS BASED ON 100*68 SAMPLES

Project	**Actual**	**Estimate (*T*)**	**Std. Err (S_T)**	**Conf-idence**	**MRE**
Eclipse 3.0	2913 (27.50%)	2893 (27.30%)	519.92	96%	14.19%
Eclipse 2.0	2611 (38.80%)	2597 (38.60%)	381.40	95%	11.68%
JDT.Core	502 (53.46%)	492 (52.40%)	50	94%	8.07%
SWT	208 (24.67%)	206 (24.44%)	47	94%	17.78%

IV. SAMPLE-BASED ESTIMATION OF DEFECT DISTRIBUTION

It is often desirable to be able to answer questions such as "how many defects the top 10% most defective programs contain?" The answers can help project managers achieve a better understanding of the overall software quality and a better allocation of SQA resources. Previous works [8, 19] shows that the distribution of defects in a large software system is skewed. In this section, we show that by analyzing the characteristics of a small sample, we can estimate the defect distribution in a large software system.

We use the Eclipse 3.0 defect dataset as an example to illustrate our method. For the 10% Eclipse 3.0 sample that is described in Section III.B, we rank all the programs (source files) by the number of defects they are responsible for, and calculate the cumulative percentage of defects over programs. We find that in the sample, the defect distribution is highly skewed --- that a small number of programs accounts for a large proportion of the defects. For example, the top 10% "most defective" programs in the sample contain 71.41% defects. We also find that the defect distribution in the sample is very similar to the distribution in the population. For example, in the actual population of Eclipse 3.0 programs, the top 10% "most-defective" programs contain 71.34% defects, which is very close to the percentage in the 10% sample. Figure 4 illustrates the cumulative percentages of defects over programs in samples and in population. Both 10% and 68 samples can model the actual distribution of defects well, with the larger sample (10%) having the better fit. We obtained similar results for other projects. These results suggest that we can achieve a good understanding of the distribution of defects in a large software system by only analyzing a small sample.

In our prior work [24], we discovered that when programs in a large software system are ranked according to the number of defects, the distribution of defects follows the Weibull distribution, which is one of the most widely used probability distributions in the reliability engineering discipline. In this study, we find that the same regularity exists in a random sample of program defect data. The curves shown in Figure 4 can be also modeled by Weibull functions. The CDF (cumulative density function) of the Weibull distribution can be formally defined as:

$$P(x) = 1 - \exp\left(-\left(\frac{x}{\gamma} \right)^{\beta} \right)$$

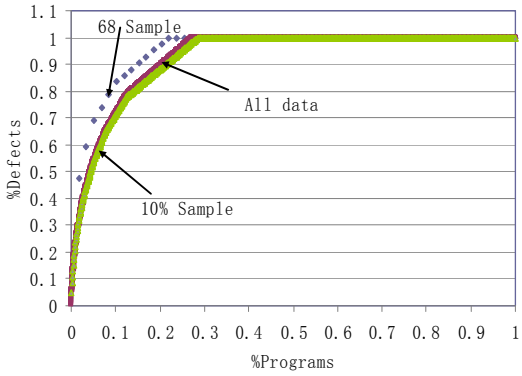

Figure 4. The distribution of defects in samples (Eclipse 3.0)

Using statistical packages such as SPSS, we are able to perform non-linear regression analysis and derive the parameters for each distribution. We compute the coefficient of determination (R^2) and the Standard Error of Estimate (S_e) to statistically compare the goodness-of-fit of the Weibull distribution. Table V summarizes the Weibull parameters and the accuracy measures for the 10% samples. These results confirm that defects in a sample follow the Weibull distribution, with R^2 values ranging from 0.983 to 0.994 and S_e values ranging from 0.004 to 0.028.

Having understood the nature of defect distribution, we can estimate the number of defects the top x% "most defective" programs contain based on a sample as follows:

$$N_x = P_s(x) \times T = \left(1 - \exp\left(-\left(\frac{x}{\gamma} \right)^{\beta} \right) \right) \times T,$$

where $P_s(x)$ is the distribution of defects in a sample, modeled by the Weibull function, T is the estimated total number of defects. As an example, we evaluate the above equation using the 10% sample of Eclipse 3.0 given in Section III.B. The $P_s(x)$ of the 10% sample is calculated based on Table V and the point estimate T is given in Section III.B (T = 7697). We can then

estimate the number of defects the top 5%, 10%, 15% and 20% "most-defective" programs are responsible for. The results (Table VI) show that all estimates are accurate. For example, we estimate that the top 10% "most defective" programs in Eclipse 3.0 contain 5634 defects, which is close to the actual number 5295. For all estimations, the MRE values range from 1.08% to 6.40%, with an average of 4.10%.

TABLE V. THE WEIBULL DISTRIBUTION OF DEFECTS IN 10% SAMPLES

Project	γ	β	R^2	S_e
Eclipse 3.0	0.07	0.80	0.989	0.017
Eclipse 2.0	0.07	0.76	0.992	0.015
JDT.Core	0.16	0.84	0.983	0.028
SWT	0.08	0.98	0.994	0.004

TABLE VI. ESTIMATING THE NUMBER OF DEFECTS IN TOP *x*% MOST DEFECTIVE PROGRAMS IN ECLIPSE 3.0

Top *x*	$P_s(x)$	#Estimate in top *x*	#Actual in top *x*	MRE
5%	0.53	4080	3969	2.80%
10%	0.73	5634	5295	6.40%
15%	0.84	6452	6098	5.80%
20%	0.90	6921	6628	4.42%
25%	0.94	7235	7158	1.08%

V. SAMPLE-BASED DEFECT PREDICTION

A. Training Classification Models

Software defect prediction (predicting if a particular module is defective) can be seen as a classification problem in machine learning. A classification model can be learnt from the training data with labels Defective (having one or more defects) and Non-defective (no defects), the model is then used to classify unknown modules. In recent years, many defect prediction models have been proposed (e.g., [11, 13, 14, 16, 18, 19, 25, 27, 29, 30]).

In this section, we explore if an effective defect prediction model can be built based on a small sample. For a project, we first select a small percentage of programs. We then use data collected from this percentage of programs as a sample to train a defect prediction model. The model can be used to predict defect-proneness of an un-selected program in the project. Unlike the existing work on defect prediction, our method does not require historical data from past projects, which may not be available for some organizations.

We choose three commonly used classification techniques, namely Decision Tree (a C4.5 decision tree learner), Naive Bayes (a standard probabilistic Naive Bayes model), and Logistic Regression (a linear logistic regression based classification). All classifiers are supported by WEKA [22], a public domain data mining tool.

We also apply a new machine learning algorithm, called Co-Forest [15, 31], to defect prediction. Co-Forest is a semi-supervised learning [4], which constructs a model by learning from a small number of labeled data and then refining with the information derived from the unlabeled ones. The Co-Forest method has been successfully applied to the domain of computer-aided medical diagnosis [15]. In computer-aided medical diagnosis, conducting a large amount of routine examinations places heavy burden on medical experts. The Co-Forest method was used there to help learn hypothesis from diagnosed and undiagnosed samples in order to assist the medical experts in making diagnosis. The detailed algorithm of Co-Forest is presented in [15]. Basically, it works as follows. *N* random trees are firstly initiated from the training set bootstrap sampled from the labeled set for creating a Random Forest. Then, in each learning iteration, each random tree is refined with the newly labeled examples selected by its concomitant ensemble. The final prediction is made by ensembling all the classifiers. This approach exploits the advantage of both semi-supervised learning and ensemble learning [31].

We apply these four machine learning techniques to build prediction models from a sample of programs in the studied projects, and then use the models to predict the defect-proneness of un-sampled programs. The prediction models are learned from 198 static code attributes, including complexity metrics (such as LOC, cyclomatic complexity, number of classes, etc.) and metrics about abstract syntax trees (such as number of blocks, method references, etc.).

We perform experiments on all the datasets. For each dataset, different sizes of samples (consisting of 5% to 50% of source files) are chosen. The classification accuracy is evaluated by using the Recall (the percentage of defective programs detected), Precision (the percentage of actual defective programs in the reported cases), and F-measure (the harmonic mean of Precision and Recall).

TABLE VII. DEFECT PREDICTION BASED ON SAMPLES (JDT.CORE)

Sample Size	Co-Forest			Logistic Regression			Naive Bayes			Decision Tree		
	Recall	Precision	F-measure	Recall	Precision	F-measure	Recall	Precision	F-measure	Recall	Precision	F-measure
68	75.0%	71.2%	0.73	70.6%	70.0%	0.70	64.6%	74.7%	0.69	62.1%	64.2%	0.63
5%	73.8%	70.8%	0.72	63.7%	64.4%	0.64	67.6%	73.3%	0.70	70.8%	69.2%	0.70
10%	73.6%	73.2%	0.73	62.6%	65.5%	0.64	61.3%	77.3%	0.68	70.5%	71.2%	0.70
15%	74.6%	73.6%	0.74	62.0%	65.9%	0.64	58.4%	79.5%	0.67	70.7%	72.0%	0.71
20%	74.4%	74.4%	0.74	63.0%	67.0%	0.65	54.6%	81.2%	0.65	71.0%	71.7%	0.71
25%	75.4%	74.2%	0.75	64.8%	67.8%	0.66	54.4%	81.8%	0.65	71.5%	72.1%	0.72
30%	75.4%	75.2%	0.75	66.4%	69.2%	0.68	53.2%	82.4%	0.65	72.1%	72.9%	0.72
40%	75.7%	74.8%	0.75	69.7%	70.4%	0.70	51.3%	83.0%	0.63	72.4%	72.4%	0.72
50%	75.5%	75.6%	0.75	69.9%	72.1%	0.71	50.6%	84.1%	0.63	72.6%	73.6%	0.73

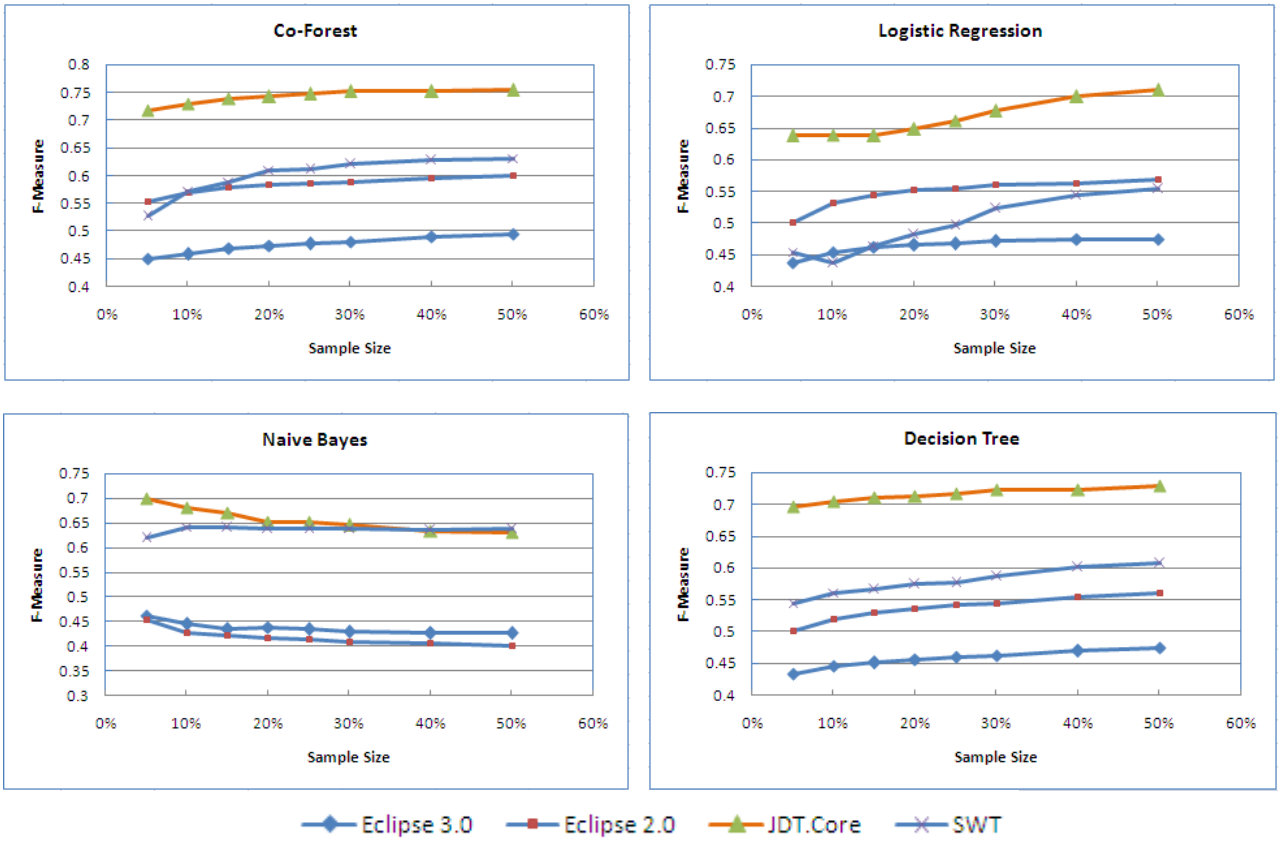

Figure 5. The comparsions of F-meaures with different sample sizes

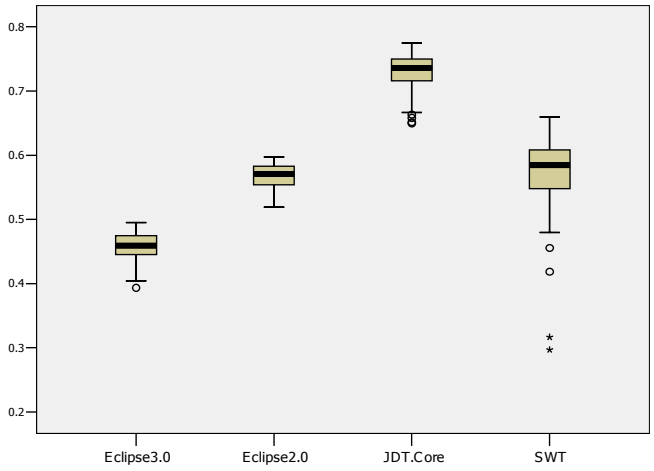

Figure 6. The box-polts of the F-measures obtained from 100*10% samples (Co-Forest)

B. *Evaluation Results*

Table VII shows the results of the defect prediction models for JDT.Core project, which are obtained from 5% to 50% samples (as well as a 68 sample consisting of 68 source files), using four classifiers. All classifiers achieve good prediction results with different sample sizes (with F-measure between 0.63 and 0.75). The semi-supervised learning method Co-Forest achieves consistently better prediction results than other classifiers. For all sample sizes, the Recalls of Co-Forest are above 73%, the Precisions are above 70%, and the F-measures are above 0.72. These results are considered satisfactory.

Table VII also shows that for all classifiers, increasing sample sizes do not necessarily lead to better prediction results. For example, for the Co-Forest method, when increasing sample size from 5% to 20%, the F-measure is only improved by 0.02. When the sample size is further increased to 50%, the F-measure is essentially not changed.

To show the generality of the results, for each sample size we also perform the prediction 100 times. Figure 5 shows the average F-measures obtained by different classifiers. Clearly, we can see that smaller samples can achieve similar prediction accuracy as larger samples do. For example, when using the Decision Tree classifier to predict defects in SWT project, increasing sample size from 5% to 10% only increases the F-measure by 0.02, further increasing sample size from 10% to 50% only increases the F-measure by 0.04. The prediction accuracy tends to become stable. These results show that building a model with more data does not necessarily lead to significant performance improvement. However, the savings of cost resulted from data reduction is significant. Figure 5 also shows that for all studied projects, in general Co-Forest achieves better results than other classifiers.

Figures 6 shows the box-plots for the F-measures obtained from 100*10% samples using Co-Forest. The median F-measures of 100 predictions for Eclipse 3.0, Eclipse 2.0, JDT.Core, and SWT projects are 0.46, 0.57, 0.74, and 0.58 respectively. For all projects, the interquartiles (ranges between upper quartile and lower quartile, i.e., the ranges between the 75th percentile point and the 25th percentile point) are all narrow (less than 0.06). The ranges between upper tails and lower tails of the box-plots are also narrow (less than 0.15). There are only a few data points (less than 4) fall below the lower tail. These results show that for large samples, the Co-Forest classifier can achieve good defect prediction accuracy with high confidence. For the SWT project, the variations among different predictions are larger (as reflected by the wider ranges in box-plots). However, the lower quartile is 0.55, therefore among the 100 predictions, 75 predictions have F-measures higher than 0.55.

VI. SAMPLE-BASED SOFTWARE QUALITY CONTROL

In software quality control, an organization often sets a quality standard for the percentage of defective programs. If the percentage of defective programs in a project under examination is above certain allowable level, a decision can be made to reject the project due to its poor quality. Such quality control measure is especially useful for outsourced projects and third-party testing.

Based on the percentages of defective programs obtained from samples, classical hypothesis testing can be applied to test if the quality of the whole project is acceptable with respect to certain quality standard. For example, we can formulate null and alternative hypotheses on the desired percentage of defective programs, and perform a hypothesis testing via a t-test. If the result of the t-test concludes with high confidence that the percentage of defective programs in the whole project is above the allowable level (say, 30%), a decision can be made to reject the project.

To optimize sample-based software quality control, we need not test all programs in a sample before we make decision on rejection or acceptance of the software quality. A sampling plan called the *sequential probability ratio test* (SPRT) [10] can help us achieve an early determination of software quality, thus saving considerable data collection efforts. SPRT is a specific sequential hypothesis test. Unlike classical hypothesis testing, SPRT can reach a conclusion according to certain threshold values before all data is collected and analyzed.

In SPRT, the hypotheses are specified as follows:

$$H0: p \le p0$$

$$H1: p \ge p1$$

, where $p0$ is the lower bound below which the quality should be accepted, $p1$ is the upper bound above which the quality should be rejected. To test the above hypotheses, the following ratio is calculated after making m observations $x_1, x_2, \ldots x_m$:

$$f_m = \prod_{i=1}^{m} \frac{\Pr(X_i = x_i \mid p = p_1)}{\Pr(X_i = x_i \mid p = p_0)} = \frac{{p_1}^{d_m}(1-p_1)^{m-d_m}}{{p_0}^{d_m}(1-p_0)^{m-d_m}},$$

where $d_m = \sum_{i=1}^{m} x_i$. Note that the SPRT ratio is often found with the use of logarithms. The stopping rule of SPRT is as follows:

- $B < f_m < A$: continue sampling
- $f_m \le B$: accept H0
- $f_m \ge A$: accept H1

, where A and B are specified based on the desired type I error α (false negative ratio) and type II error β (false positive ratio) as follows:

$$A = \frac{(1-\beta)}{\alpha}, B = \frac{\beta}{(1-\alpha)}$$

As an example, assuming the thresholds $p0 = 0.1$ and $p1 = 0.3$, the allowed errors $\alpha = 0.05$, and $\beta = 0.2$. Thus $A = 16$ and $B = 0.21$. Suppose we have tested a random sample of 17 JDT.Core programs with the following results: 00010010001001010 (1 indicates a defective program and 0 indicates non-defective program). We can calculate the ratio $f_{17} = 11.91$, which is below A and above B, therefore we continue sampling and testing. Suppose the next sampled program turns out to be a defective program, we obtain $f_{18} = 35.73$, which is above A. Therefore we accept H1 and reject the project because it does not satisfy our quality standard.

The SPRT process can be also graphically represented, such as the one shown in Figure 7. The rejection line and the acceptance line define the three regions (Reject, Accept, Continue). Decisions are made depending on the region the actual number of defective programs falls in. The derivation of rejection and acceptance lines from parameters $p0$, $p1$, α, β can be found at [10]. In Figure 7, the number of defective programs after sampling 18 programs is 6, which enters the Reject region. Note that using SPRT, the decision of rejection/acceptance is made only after the testing of a few programs, thus the time and cost required for software quality control can be significantly reduced.

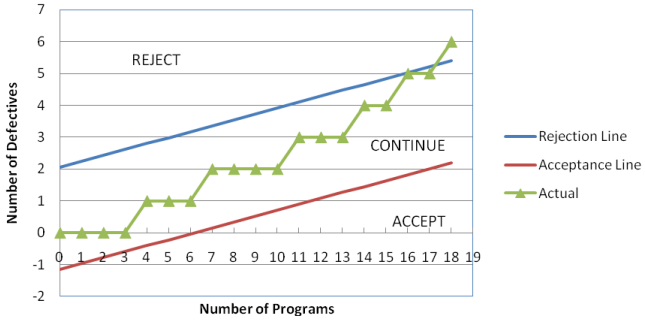

Figure 7. Graphical Representation of SPRT

VII. RELATED WORK

A. Statistical Approaches to Software Quality Assurance

Statistical Quality Control (SQC) has been widely used in traditional industrial engineering and has now become an integral part of Total Quality Management. In software engineering area, efforts have been made to apply SQC to software quality assurance. As early as in 1980, Cho proposed

to define standards for software defect ratio, and to determine the acceptance of software quality via sampling [4]. Thelin et al. [21] applied sampling technique to improve software inspections. They proposed to sample design documents to increase the efficiency of an inspection process.

Statistical Process Control (SPC) [10] is an important SQC measure that has been applied to software process improvement [9, 12]. SPC uses various control charts and sampling plans to help project managers evaluate if a process is under control with respect to variation. The control charts are usually 3-sigma charts, where upper control limit (UCL) and lower control limit (LCL) specify 3 standard deviation of the sample mean. If a process stays within the limits, it is assumed that the process is under control. Otherwise, the process may be out of control and a search of assigned causes should be initiated.

Although there are related works on applying statistical techniques to software quality assurances, we believe that this area is generally not well explored and lacks of empirical studies. There is still ampler room for further research. In this paper, we describe sample-based program quality estimation, discussing issues such as defect prediction, the impact of sample size, and SPRT-based sampling plan. It is also interesting to explore the synergy between our method and the related work (such as SPC and Six Sigma [1]), aiming for a complete solution for statistical software quality control.

B. Related Work on Defect Prediction

The ability to predict software quality is important for software quality improvement and project management. Over the years, many predictions have been proposed to predict defect totals and defect-proneness. For example, Compton and Withrow [6] proposed a LOC-based polynomial regression model to predict the number of defects in a software system. In our previous work [26], we analyzed the Eclipse defect data over a 3-years period, and found that the growth of the number of defects can be well modeled by polynomial functions. Furthermore, we can predict the number of future Eclipse defects based on the nature of defect growth. Kim et al. [14] predict defects by caching locations that are adjusted by software change history data. Their method can predict 73%-95% faults based on 10% files. Menzies et al. [16] build classification models for predicting defect-proneness using the Naive Bayes classifier (with probability of detection 71%), but we pointed out that their model is unsatisfactory when precision is concerned [23]. Zimmermann and Nagappan [29] proposed models based on the structural measurement of dependency graphs. Hassan [11] found that the more complex the changes to a file, the higher the chance the file will contain faults. In our prior work, we also proposed code complexity-based defect-prediction models [25, 27].

The current defect prediction models are not without critics. These models are all built using historical data from past projects, assuming that the new projects share the same product/process characteristics and their defects can be predicted based on previous data. This assumption may not be always valid. Firstly, the historical data may not be available (and reliable) for all projects. Secondly, even some previous project data is available, whether it can construct effective prediction models for completely new projects remains to be verified. This is particularly of concern in a rapid changing environment, where the requirements, development personnel, organizational structure, or even process changes. Recently Zimmermann et al. [30] also ran 622 cross-project predictions for 12 real-world applications, their results indicate that cross-project prediction is a serious challenge, i.e., simply using models built from other projects does not lead to accurate predictions.

Our methods adopt a statistical approach to quality estimation. In this study we do not attempt to improve prediction accuracy or determine the best subset of metrics as defect predictors. Instead our method is to show that a small sample can build cost-effective defect prediction models. Compared to the related work, our method has the following advantages:

- Does not require historical or other projects' data.
- Supports the prediction of defect-prone programs as well as defect totals, proportions and distributions.
- Requires only a small sample of programs.

The proposed sample-based defect prediction method is independent of the attributes used for model construction. In this study, we only use static code attributes to build the model. As suggested in [11, 18, 29], using more types of attributes, such as changes and dependency metrics, could improve the accuracy of the prediction. We will investigate more types of attributes in our future work.

VIII. THREATS TO VALIDITY

In our approach, we draw a random sample from the population of programs. To ensure proper statistical inference and to ensure the cost-effectiveness of the proposed method, the population size should be large enough. Therefore, the proposed method is suitable for large-scale software systems.

The simple random sampling method requires that each individual in a sample to be collected entirely by chance with the same probability. Selection bias may be introduced if the program sample is collected simply by convenience, or from a single developer/team. The selection bias can lead to non-sampling errors (errors caused by human rather than sampling) and should be avoided. In our experiments, to ensure the randomness we use the statistical package SPSS, which can select a random sample based on computer generated (pseudo) random numbers.

The defect data for a sample is collected through quality assurance activities such as software testing, static program checking and code inspection. As the sample will be used for prediction, these activities should be carefully carried out so that most of defects can be discovered. Incorrect sample data may lead to incorrect estimates of the population.

In our experiments, we used the public Eclipse defect dataset. Although this dataset has been used by many other studies [17, 24, 27, 28, 30], our results may be affected if the dataset is flawed (e.g., there are problems in bug data collection and recording [2]). Also, Eclipse is an open source project. It is desirable to replicate the experiments on industrial, in-house developed projects to further evaluate their validity. This will be our important future work.

IX. CONCLUSIONS

Statistical inference permits us to draw conclusions about a population based on a sample that is quite small in comparison to the size of population, thus improving cost-effectiveness. In this paper, we propose simple random sampling based methods for software quality estimation (with respect to defects). From a small sample of programs we can successfully estimate defect totals, proportions and distributions for the entire program population in a project. We have also found the minimum sample size with respect to error rate and confidence level. A sample as little as 68 programs can achieve good estimation accuracy.

Samples can also be used to construct cost-effective defect prediction models, which can then predict defect-proneness of an un-sampled program. We find that smaller samples can achieve similar prediction accuracy as larger samples do. We also find that the semi-supervised learning method called Co-Forest performs better in sample-based defect prediction. Samples can be used to make decisions on rejection or acceptance of program quality too. We propose to use the SPRT sampling plan, which allows a decision to be made only after testing a few programs.

We believe our sample-based quality estimation methods can help project managers achieve an early yet accurate understanding of the overall program quality, thus help them better controls SQA activities and make rational decisions. Unlike current defect prediction methods, the proposed method does not require historical defect data, or data from other companies. Our experiments show that the proposed methods are simple yet effective.

In future, we will apply our methods to industrial practices and evaluate their effectiveness. We will also explore if other sampling methods, such as multi-stage stratified sampling can improve the estimation accuracy.

ACKNOWLEDGMENTS

We thank Prof. Zhi-Hua Zhou for providing us the code of Co-Forest, and for his valuable comments on the paper. We also thank Sung Kim at HKUST for his comments. This research is supported by the State Laboratory of Novel Software Technology at Nanjing University, the Key Laboratory of High Confidence Software at Peking University, and the Chinese NSF grants 60703060 and 90718022.

REFERENCES

[1] K. Arul and H. Kohli, Six Sigma for Software Application of HypothesisTests to Software Data, *Software Quality Journal* (12), Kluwer Academic, 2004.

[2] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. T. Devanbu, Fair and Balanced? Bias in Bug-Fix Datasets, *Proc. ESEC/FSE 2009,* Amsterdam, Netherlands.

[3] L. Briand and I. Wieczorek, Resource modeling in software engineering. *Encyclopedia of Software Engineering*, Wiley, pp. 1160-1196, 2002.

[4] C. K. Cho, An Introduction to *Software Quality Control*, Wiley, 1980.

[5] O.Chappelle, B. Scholkopf, and A. Zien, eds., *Semi-Supervised Learning*. MIT Press, Cambridge, MA, 2006.

[6] T. Compton and C. Withrow, Prediction and Control of Ada Software Defects, *J. Systems and Software*, vol. 12, pp. 199-207, 1990.

[7] J. Devore, *Probability and Statistics for Engineering and the Sciences*, Duxbury Press, 1995.

[8] N. Fenton and N. Ohlsson, Quantitative Analysis of Faults and Failures in a Complex Software System, *IEEE Trans. Software Engineering*, 26 (8), pp. 797-814, 2000.

[9] W. Florac, A. Carleton and J. Barnard, Statistical Process Control: Analyzing a Space Shuttle Onboard Software Process, *IEEE Software*, July/August 2000.

[10] E. Grant and R. Leavenworth, *Statistical Quality Control*, McGraw-Hill, 1988.

[11] A. E. Hassan, Predicting Faults Using the Complexity of Code Changes, *Proc.ICSE'09*, Vancouver, Canada, May 2009.

[12] P. Jalote and A. Saxena, Optimum Control Limits for Employing Statistical Process Control in Software Process, *IEEE Trans. on Software Engineering*, 28(12), 2002.

[13] A. Koru and H. Liu, Building Effective Defect-Prediction Models in Practice, *IEEE Software*, 22(6), 23-29, 2005.

[14] S. Kim, T. Zimmermann, E. Whitehead Jr., A. Zeller, Predicting Faults from Cached History, *Proc. ICSE'07*, Minneapolis, USA, 2007

[15] M. Li and Z.-H. Zhou. Improve computer-aided diagnosis with machine learning techniques using undiagnosed samples. *IEEE Trans. on Systems, Man and Cybernetics - Part A: Systems and Humans*, 2007, 37(6): 1088-1098. The Co-Forest tool is available at: http://lamda.nju.edu.cn/datacode/CoForest.htm

[16] T. Menzies, J. Greenwald and A. Frank, Data Mining Static Code Attributes to Learn Defect Predictors, *IEEE Trans. Software Engineering, 32(11), pp. 1-12, 2007.*

[17] R. Moser, W. Pedrycz and G. Succi, A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction, *Proc. ICSE 2008*, Leipzig, Germany, 2008.

[18] N. Nagappan, T. Ball, and A. Zeller, Mining Metrics to Predict Component Failures, *Proc.ICSE'06*, Shanghai, China, May 2006.

[19] T. Ostrand, E. Weyuker and R. Bell, Predicting the Location and Number of Faults in Large Software Systems, *IEEE Trans. Software Engineering*, 31 (4), pp. 340-355, 2005.

[20] J. Rice, *Mathematical Statistics and Data Analysis*, second edition, Duxbury Press, 1995.

[21] T. Thelin, H. Petersson, P. Runeson, and C. Wohlin, Applying sampling to improve software inspections. *J. Systems and Software*, 73(2), 2004.

[22] WEKA: http://www.cs.waikato.ac.nz/ml/weka/

[23] H. Zhang and X. Zhang, Comments on "Data Mining Static Code Attributes to Learn Defect Predictors", *IEEE Trans. on Software Engineering*, 33(9), pp. 635-636, 2007.

[24] H. Zhang, On the Distribution of Software Faults, *IEEE Trans. on Software Engineering*, 34(2), pp. 301-302, 2008.

[25] H. Zhang, X. Zhang and M. Gu, Predicting Defective Software Components from Code Complexity Measures, *Proc. 13th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*, Dec 2007, Melbourne, Australia.

[26] H. Zhang, An Initial Study of the Growth of Eclipse Defects, *Proc. 5th Working Conference on Mining Software Repositories (MSR 2008)*, Leipzig, Germany, May 2008.

[27] H. Zhang, An Investigation of the Relationships between Lines of Code and Defects, *Proc. ICSM'09*, Edmonton, Canada, September 2009.

[28] T. Zimmermann, R. Premraj and A. Zeller, 2007. Predicting Defects for Eclipse, *Proc. PROMISE'07*, Minneapolis, USA.

[29] T. Zimmermann and N. Nagappan, Predicting Defects using Network Analysis on Dependency Graphs, *Proc.ICSE 2008*, Leipzig, Germany.

[30] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, B. Murphy. Cross-project Defect Prediction: A Large Scale Experiment on Data vs. Domain vs. Process, *Proc. ESEC/FSE 2009*, Amsterdam, Netherlands.

[31] Z.-H. Zhou. When semi-supervised learning meets ensemble learning. *Proc.the 8th Int. Workshop on Multiple Classifier Systems (MCS'09)*, Reykjavik, Iceland, 2009. (keynote)