An Empirical Study on Progressive Sampling for Just-in-Time Software Defect Prediction

Xingguang Yang*†, Huiqun Yu*‡✉, Guisheng Fan*✉, Kang Yang*, Kai Shi§
*Department of Computer Science and Engineering, East China University of Science and Technology, Shanghai 200237, China
†Shanghai Key Laboratory of Computer Software Evaluating and Testing, Shanghai 201112, China
‡Shanghai Engineering Research Center of Smart Energy, Shanghai, China
§Alibaba Group, Hangzhou, China

CEUR-WS.org/Vol-2511/QuASoQ-02.pdf

***Abstract*—Just-in-time software defect prediction (JIT-SDP) is an active research topic in the field of software engineering, aiming at identifying defect-inducing code changes. Most existing JIT-SDP work focuses on improving the prediction performance of the model by improving the model. However, a frequently ignored problem is that collecting large and high quality defect data sets is costly. Specifically, when labelling the samples, experts in the field are required to carefully analyze the defect report information and log of code modification, which requires a lot of effort. Therefore, how to build a high-performance JIT-SDP model with a small number of training samples is an issue worth studying, which can reduce the size of the defect data sets and reduce the cost of data sets acquisition. This work thus provides a first investigation of the problem by introducing a progressive sampling method. Progressive sampling is a sampling strategy that determines the minimum number of training samples while guaranteeing the performance of the model. However, progressive sampling requires that the learning curve of the prediction model be well behaved. Thus, we validate the availability of progressive sampling in the JIT-SDP issue based on six open-source projects with 227417 changes. Experimental results demonstrate that the learning curve of the prediction model is well behaved. Therefore, the progressive sampling is feasible to tackle the JIT-SDP problem. Further, we investigate the optimal training sample size derived by progressive sampling for six projects. Empirical results demonstrate that a high-performance prediction model can be built using only a small number of training samples. Thus, we recommend adopting progressive sampling to determine the size of training samples for the JIT-SDP problem.**

***Index Terms*—Just-in-time, software defect prediction, progressive sampling, mining software repositories**

I. Introduction

Defects in the software system can cause huge losses to companies [1]. Although software quality activities (such as source code checking and unit testing) can reduce the number of defects in software, they require a lot of testing resources. Therefore, how to release a high-quality software project with limited testing resources is a huge challenge in the field of software engineering [2]. Software defect prediction is an effective method. Developers use machine learning or statistical learning methods to identify the defect-proneness of program modules in advance, thereby investing more limited testing resources into defect-prone modules [2].

Corresponding Authors: Huiqun Yu (yhq@ecust.edu.cn), Guisheng Fan (gsfan@ecust.edu.cn)

Just-in-time software defect prediction(JIT-SDP) is a more fine-grained defect prediction method, which is made at change-level rather than module-level(e.g., function, file, and class) [1]. In the software development process, once the developer submits a modification to the software code, the defect prediction model will predict the defect-proneness of the code. If the change is predicted to be buggy, the corresponding developer will be assigned to check the change. Therefore, JIT-SDP has the advantages of fine granularity, instantaneity, and traceability [3], and has been adopted by many companies such as Lucent [4], BlackBerry [5], Cisco [6], etc.

Recently, JIT-SDP has received extensive attention and research. The main research work focuses on model building [7] [8], feature selection [1] [9], data annotation [10], etc. However, few studies have focused on the cost of acquiring defect data sets. Specifically, in order to obtain high-quality defect data sets, experts in specific fields are required to analyze version control systems (SVN, CVS, Git, etc.) and defect tracking systems (Bugzlla or Jira) during the data annotation phase [3]. Therefore, constructing an accurate defect data set is costly [11]. In the field of software engineering data mining [12] [13], researchers found the following relationship between the size of the data sets and the performance of the prediction model: When the data set size is small, the accuracy of the prediction model increases significantly as the size of the data increases. When the data sets size exceeds a certain number, adding more data does not lead to higher prediction performance. Therefore, how to build a high performance prediction model with fewer training samples for JIT-SDP is a problem worth studying, which brings two advantages:

- Firstly, reducing the size of training samples can reduce the cost of data sets labeling.
- Secondly, when using complex learning algorithms such as deep learning algorithms [8], reducing the size of the training data can significantly reduce the time required for model training.

In order to reduce the use of training samples, this paper first introduces progressive sampling into the study of JIT-SDP problems. We conduct experiment on the change-level defect data sets from six open source projects with 227417 changes. The main contributions of this paper are as follows:

Copyright © 2019 for this paper by its authors.
Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

- We introduce progressive sampling to the JIT-SDP study to determine the optimal training sample size and reduce the cost of defect data acquisition. However, progressive sampling requires that the learning curve of the prediction model be well behaved in coarse granularity. Therefore we conduct a large-scale empirical study based on the defect data sets from six open-source projects. The experimental results show that the learning curve of the prediction model is well behaved. So progressive sampling is efficient for JIT-SDP.
- We further investigate the optimal training sample size derived by progressive sampling based on six open-source projects. The experiment uses the random forest to establish a prediction model and uses AUC to evaluate the performance of the model. Empirical results show that using progressive sampling can significantly reduce the number of training samples used while guaranteeing the performance of the prediction model. Therefore, we recommend that in the practical application of JIT-SDP, using progressive sampling can effectively reduce the amount of training samples and reduce the cost of model building.

The rest of the paper is organized as follows: The related work is described in Section II. Section III introduces the progressive sampling and it's application in the scenario of JIT-SDP. Experimental setup is described in the Section IV. Section V introduces the experimental results and discussion. Section VI introduces the threats to validity. Conclusions and future work is described in the Section VII.

II. Related Work

A. Just-in-Time Software Defect Prediction

JIT-SDP is a special method for predicting software defects. Unlike traditional defect prediction, JIT-SDP is performed at change-level, which has finer granularity. Mockus and Weiss [4] first proposed the idea of JIT-SDP, and they designed a number of change metrics to predict whether changes are defect-inducing or clean. Recently, Kamei et al. [1] performed a large-scale empirical study in JIT-SDP. They collected eleven data sets from six open-source projects and five commercial projects. Their experimental results show that their prediction model can achieve 68% accuracy and 64% recall. Moreover, they find that 35% defect-inducing changes can be identified using only 20% of the effort.

Subsequently, researchers proposed various methods to improve the performance of the prediction model for JIT-SDP. Chen et al. [14] designed two objects through the benefit-cost analysis, and formalized the JIT-SDP problem into a multi-objective optimization problem. They proposed a method called MULTI based on NSGA-II [15]. The experimental results show that MULTI can significantly improve the effort-aware prediction performance for JIT-SDP. Furthermore, Yang et al. [16] found that the MULTI method is more biased towards the benefit object in the optimal solution selection. Therefore, they proposed a benefit-priority optimal solution selection strategy to improve the performance of the MULTI method. Cabral et al. [17] first found that JIT-SDP suffers from class imbalance evolution. Their proposed approach can obtain top-ranked g-means compared with state-of-the-art methods.

B. Progressive Sampling

Weiss and Tian pointed out that in the field of data mining, data acquisition is one of the main costs of the process of building a classification model [18]. Therefore, reducing the use of training data while guaranteeing the performance of the prediction model can reduce the cost of model building. In solving the actual classification task, using fewer training samples can still get a high prediction model. Thus, Provost et al. [19] proposed progressive sampling method. Progressive sampling continuously increases the number of training samples by the iterative method. Currently progressive sampling has been widely used in the field of software engineering data mining. For example, in the study of performance prediction for configurable software, obtaining data sets is costly. Thus, Sarkar et al. [12] used progressive sampling to determine the optimal number of training samples.

In the JIT-SDP study, obtaining high-quality defect data sets is costly, and it requires specialists in specific fields to thoroughly analyze defect report information and code modification logs [11]. The most existing JIT-SDP study only focuses on improving the performance of the prediction model, but ignores the cost of defect data sets acquisition. Therefore, this paper first introduces progressive sampling into the JIT-SDP scenario.

III. Progressive Sampling

A. Basic Concept of Progressive Sampling

Progressive sampling is a popular sampling strategy that has been used for various learning models [13]. Progressive sampling is an iterative process whose basic idea is to generate an array of integers $n_0, n_1, n_2, ..., n_k$. Each integer n_i indicates that the training samples with size of n_i are used to build the prediction model at the *i*th iteration. According to the calculation of the number of training samples in each iteration, progressive sampling can be classified as arithmetic progressive sampling and geometric progressive sampling [20]. The size of training samples for two progressive sampling is calculated as shown in Eq. (1) and Eq. (2), respectively, where n_0 represents the initial training sample size and a determines the growth rate of the training samples. It can be seen that the main difference between the two kinds of progressive sampling is that the geometric progressive sampling has a larger growth rate than the arithmetic progressive sampling, and is suitable for the prediction model with high algorithm complexity. Since the machine learning algorithm used in this paper is the random forest [21], the training time of the model is short, so it is suitable to use arithmetic progressive sampling.

$$n_i = n_0 + i * a \tag{1}$$

$$n_i = n_0 * a^i \tag{2}$$

Copyright © 2019 for this paper by its authors.
Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

Learning curve. The learning curve [19] describes the prediction performance of the prediction model at different training sample sizes and can clearly characterize the learning process of progressive sampling. Typical learning curve is shown as Fig.1, where the *x*-axis represents the training sample size and the *y*-axis represents the prediction performance of the model. A well behaved learning curve is monotonically non-decreasing and contains three regions: In the first region, the model performance increases rapidly as the training sample size increases; in the second region, the model performance increases slowly as the training sample size increases; in the third region, adding more training samples will not significantly improve the performance of the model.

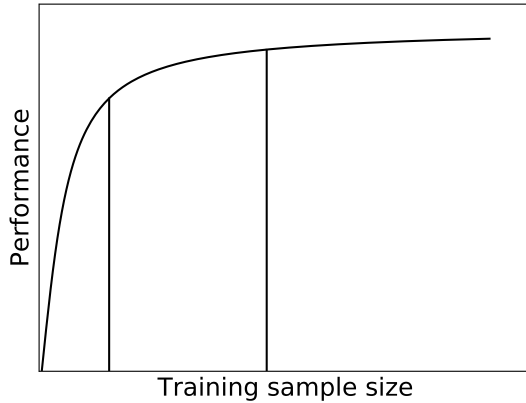


Fig. 1. Learning Curve

B. The Process of Progressive Sampling in the JIT-SDP

Progressive sampling is widely used in various software engineering related studies, such as performance prediction of configurable software [12], etc. In this paper, we first introduce progressive sampling into the JIT-SDP scenario. The detailed process is shown in Fig. 2, which involves four steps as following:

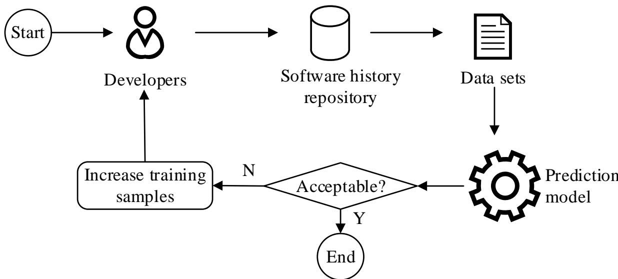


Fig. 2. The process of progressive sampling in the JIT-SDP

1) Developers mine metrics related to software changes from the software history repositories.
2) Domain experts label samples as defect-inducing or clean by analyzing defect reporting information and code modification logs in version control systems, and build defect data sets. Because of the high cost of this process, only a few number of changes are labeled.
3) Based on the existing data sets, a machine learning algorithm is used to build a defect prediction model.
4) After the prediction model is evaluated, it is required to determine whether the performance of the model is acceptable. If the performance is not acceptable, the more training samples will be collected according to the rules of progressive sampling.

We use following Algorithm 1 to describe more formally the application process of progressive sampling in JIT-SDP.

Algorithm 1: The progressive sampling for JIT-SDP

Input: initial sample size: n_0; growth factor: a; termination threshold: $threshold_AUC$
Output: prediction model: $model$;

```
1  begin
2      # Build data sets with n_0 samples
3      D = mining_software_repository(n_0)
4      while true do
5          # Split data sets into training and test sets
6          train_set, test_set = train_test_split(D)
7          # Build prediction model based on machine
            learning methods
8          model = Random_Forest(train_set)
9          # Model evaluation
10         AUC = Evaluation(model, test_set)
11         # Whether the model is acceptable
12         if AUC > threshold_AUC then
13             return model
14         end
15         else
16             new_D =
                mining_software_repository(a)
17             D = D ∪ new_D
18         end
19     end
20 end
```

We denote an instance of a code change as $X = \{x_1, x_2, ..., x_m\}$, where $x_1, x_2, ..., x_m$ represent the m metrics of the change X. An example of the change X is denoted as $(\mathbf{x}, Y)$, where **x** represents the values of metrics and Y represents whether the change is buggy or clean. If the change X is identified as buggy, then Y will be marked as 1, otherwise it is marked as 0. The defect data sets D for a specific project are composed of a set of examples X, where $X \subseteq D$.

In the beginning, developers need to mine n_0 samples from the software history repositories and build data sets D (Line 3). The data sets are then split into training and test sets (Line 6). The the defect prediction model is built and evaluated based on a machine learning algorithm (Line 8-10). Our experiment uses the random forest to build a prediction model and evaluate the model using AUC. If the performance of the model exceeds the threshold $threshold_AUC$, the progressive

Copyright © 2019 for this paper by its authors.
Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

sampling terminates the iteration (Line 12-13). Otherwise, it is necessary to further collect a samples from the software history repositories to increase the size of the data sets (Line 16-17).

IV. Experimental Setup

This paper introduces progressive sampling into the JIT-SDP problem and designs the following two research questions:

- RQ1: Whether the progressive sampling is feasible in the JIT-SDP scenario?
- RQ2: What is the optimal training sample size to establish a high performance prediction model by adopting progressive sampling?

The experimental hardware environment is *Intel(R) Core(TM)I7-7700 CPU RAM: 8G*. The programming environment used in the experiment is *python3.2*.

A. Data Sets

The data sets used in the experiment were provided by Kamei et al. [1] and are widely used in the field of JIT-SDP [7] [8] [9]. The data sets are collected from six open source projects, namely *Bugzilla(BUG)*, *Columba(COL)*, *Eclipse JDT(JDT)*, *Eclipse Platform(PLA)*, *Mozilla(MOZ)*, and *PostgreSQL(POS)*, with a total of 227417 changes. The number of defective changes, defect rate, and data collection period for each subject system are shown in Table I.

TABLE I
The basic information of data sets

Project	Period	#defective changes	#changes	%defect rate
BUG	1998/08/26∼2006/12/16	1696	4620	36.71%
COL	2002/11/25∼2006/07/27	1361	4455	30.55%
JDT	2001/05/02∼2007/12/31	5089	35386	14.38%
MOZ	2000/01/01∼2006/12/17	5149	98275	5.24%
PLA	2001/05/02∼2007/12/31	9452	64250	14.71%
POS	1996/07/09∼2010/05/15	5119	20431	25.06%

In order to accurately predict defects for software changes, Kamei et al. [1] designed 14 metrics. These metrics can be divided into five dimensions: diffusion, size, purpose, history, and experience. The specific description information is shown in Table II.

B. Prediction Model

Similar to previous research [22], the experiment uses the random forest algorithm to build prediction models [21], because previous studies have shown that random forest is highly robust, accurate and stable on JIT-SDP issues [23], and exceed other modeling techniques [24].

Random forest is an ensemble learning algorithm based on decision tree. Different from the conventional decision tree, the base learner randomly selects a subset of the attributes in each node's attribute set, and then selects an optimal attribute from the subset. Random forest algorithm is simple, easy to implement, and has low computational overhead, and is widely used in various learning tasks.

TABLE II
The description of metrics

Dimension	Metric	Description
Diffusion	NS	Number of modified subsystems
	ND	Number of modified directories
	NF	Number of modified files
	Entropy	Distribution of modified code across each file
Size	LA	Lines of code added
	LD	Lines of code deleted
	LT	Lines of code in a file before the change
Purpose	FIX	Whether or not the change is a defect fix
History	NDEV	Number of developers that changed the files
	AGE	Average time interval between the last and the current change
	NUC	Number of unique last changes to the files
Experience	EXP	Developer experience
	REXP	Recent developer experience
	SEXP	Developer experience on a subsystem

C. Performance Indicators

The test samples can be divided into true positive(TP), false negative(FN), false positive(FP), and true negative(TN) according to the labels of the samples and the prediction results. The confusion matrix of the classification results is shown in the Table III. JIT-SDP is a binary classification problem. Common evaluation indicators include precision, recall, accuracy, etc. However, since the defect data sets are usually class-imbalanced, these threshold-based evaluation indicators are sensitive to threshold settings. Therefore, threshold independent evaluation indicators should be used [25].

The experiment uses AUC to evaluate the prediction performance of the model [1] [22]. AUC (Area Under Curve) is the area under the ROC curve. The ROC (Receiver Operating Characteristic) curve is drawn as follows: First, the test examples are sorted in descending order according to the probability that the prediction is positive; then the test examples are regarded as positive classes one by one, and the true positive rate (TPR) and false positive rate (FPR) are calculated each time; using TPR as the ordinate and FPR as the abscissa, one point of the ROC curve is obtained, and these points are connected to obtain the ROC curve.

$$TPR = \frac{TP}{TP + FN} \tag{3}$$

$$FPR = \frac{FP}{TN + FP} \tag{4}$$

TABLE III
Confusion matrix

Actual value	Prediction result: Positive	Prediction result: Negative
Positive	TP	FN
Negative	FP	TN

D. Data Preprocessing

In order to improve the prediction performance of the model, according to the recommendations of Kamei et al. [1], we conduct the following preprocessing on the data sets:

Copyright © 2019 for this paper by its authors.
Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

1) Remove highly correlated metrics. Since NF and ND, REXP and EXP are highly correlated, ND and REXP are excluded. Since LA and LD are highly correlated, LA and LD are normalized by dividing by LT. Since LT and NUC are highly correlated with NF, LT and NUC are normalized by dividing by NF.
2) Logarithmic transformation. Since most metrics are highly skewed, each metric(except for fix) performs a logarithmic transformation.
3) Dealing with class imbalance. The data sets used in the experiment are class-imbalanced, i.e., the number of defect-inducing changes is far more than the number of clean changes. Therefore, we perform random undersampling on the training set. By randomly removing the clean changes, the number of defect-inducing changes is the same as the number of clean changes.

V. Experimental Results and Discussion

This section answers the questions raised in Section IV through experiments.

A. Analysis for RQ1

Motivation. Progressive sampling is an effective means of determining the optimal training sample size, and is widely used in the field of software engineering [12]. This paper first introduces progressive sampling into the JIT-SDP problem to determine the optimal training sample size for the prediction model. However, in practical applications, progressive sampling requires that the learning curve of the prediction model be well behaved [19]. The basic characteristic of a well behaved learning curve is that the slope of the learning curve is monotonically non-increasing at the level of coarse granularity [19]. Therefore, we aim to verify whether progressive sampling is feasible on JIT-SDP issues through empirical research.

Approach. The experiment uses the six open source projects introduced in the Section IV as the research object to explore whether the learning curve of the JIT-SDP model is well behaved. Because the prediction model is based on a fast training random forest algorithm, the arithmetic progressive sampling method is adopted. To plot a learning curve for each data sets, we divided the data sets into two parts: 50% as the training pool for the constructing training sets and 50% as the test sets for the model evaluation. The two parameters of the arithmetic progressive sampling are as follows:

- $n_0 = |training\ pool| * 1\%$
- $a = |training\ pool| * 1\%$

Since progressive sampling requires the learning curve to be well behaved in coarse grain size, the granularity of our parameter settings is large. The initial number of training samples is 1% of the total number of training pools, and 1% of the number of training samples is added per iteration.

Findings. The experimental results are shown in Fig. 3, which contains six subgraphs, each of which represents a learning curve for a project, where the horizontal axis represents the size of training samples and the vertical axis represents the performance of the prediction model.

As can be seen from Fig. 3, the learning curve for each system is well behaved, which is generally monotonically non-decreasing. Although the learning curve fluctuates in local areas, the general trend is monotonically non-decreasing. Therefore, we can draw a conclusion that progressive sampling is feasible in the JIT-SDP problem.

B. Analysis for RQ2

Motivation. The Section V-A has proven that progressive sampling is feasible in the JIT-SDP problem. However, what is the optimal training sample size to establish a high performance prediction model by adopting progressive sampling is a question worth studying. If a high-performance prediction model can be built with very few training samples, then only a small number of data sets need to be labelled during the construction of the defect data sets, which can greatly reduce the cost of data sets acquisition. Therefore, it is necessary to further investigate the optimal training sample size based on progressive sampling for JIT-SDP.

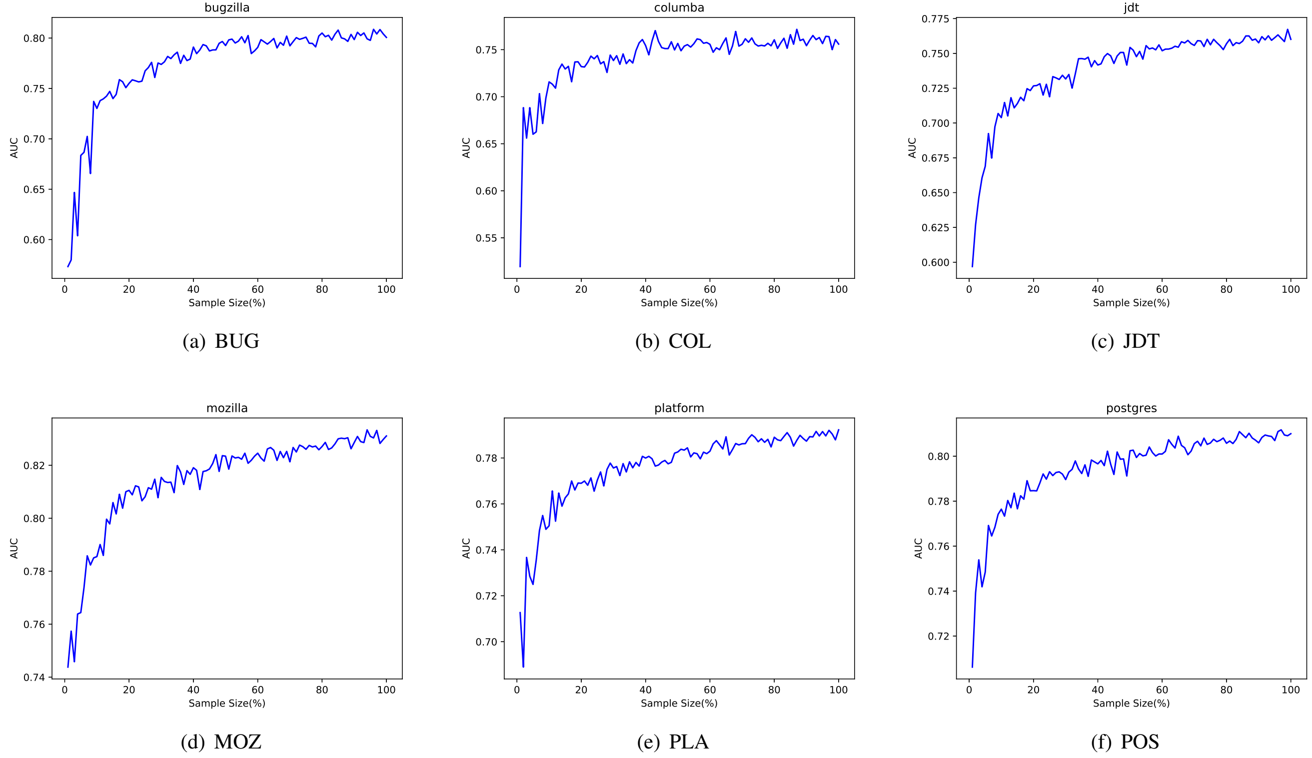
Approach. The experiment uses the data sets of the six projects introduced in Section IV. The prediction model is built based on the random forest, and the optimal training sample size is calculated based on the arithmetic progressive sampling. The experimental data sets are divided into two parts: 50% as a training pool for generating training sets and 50% as test sets for model evaluation. The parameters of the arithmetic progressive sampling are as follows: First, the initial sample size should be set small, so n_0 is set to 0.5% of the size of training pool. Second, since the training time of the model is short, the number of samples added at each iteration should not be too large. The experiment sets the growth rate a to 20. The threshold in the progressive sampling is used to determine whether the performance of the model is acceptable. Threshold settings are usually given by experts in a particular field. Previous studies have shown that the AUC value of the JIT-SDP model is usually not lower than 0.75 [22]. Therefore, the threshold of acceptable performance $threshold_AUC$ is set to 0.75, i.e., once the AUC value of the prediction model is greater than or equal to 0.75, the progressive sampling terminates the iteration and returns the value of training sample size.

- $n_0 = |training\ pool| * 0.5\%$
- $a = 20$
- $threshold_AUC = 0.75$

For better generalizability of experimental results, and to counter random observation bias, the entire experiment is repeated 100 times.

Findings. The experimental results are shown in Fig. 4, which describes box plots of optimal training sample sizes for six projects. Table IV shows the median of optimal training sample size for six projects, where the second column represents the median of optimal training sample sizes calculated from 100 experimental results, and the third column represents

Copyright © 2019 for this paper by its authors.
Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).



(a) BUG (b) COL (c) JDT

(d) MOZ (e) PLA (f) POS

Fig. 3. The learning curves for six projects

the ratio of the optimal training sample size to the total number of data sets.

As can be seen from the Table IV, the optimal training sample size between each system has a large difference. In particular, for system MOZ, PLA, and POS, the proportion of the optimal training sample size to the total data sets is very low (less than 3%), i.e., a JIT-SDP model with high performance can be obtained by using less than 3% data sets as training sets.

Therefore, the use of progressive sampling is important for specific projects. Empirical studies have shown that using only a small number of samples can build a high performance prediction model. We recommend using progressive sampling to determine the number of training samples to reduce the cost of building defect data sets while preserving the performance of the model.

TABLE IV
THE MEDIAN OF OPTIMAL TRAINING SAMPLE SIZE FOR SIX PROJECTS

Project	Optimal training sample size#	Rate%
BUG	361	7.81
COL	931	20.89
JDT	6778	19.15
MOZ	565	0.57
PLA	1890	2.94
POS	271	1.32

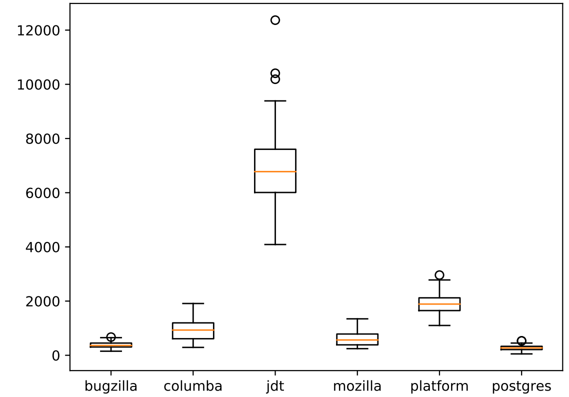


Fig. 4. Optimal training sample size

VI. THREATS TO VALIDITY

External validity. Threats to external validity are mainly from the data sets used in the experiment. Although data sets are widely used in JIT-SDP research [1] [22] [7], whether the experimental conclusions can be extended to other project data sets remains to be further verified. Therefore, more data sets have yet to be mined to verify the generalization of experimental results.

Construct validity. The threats to construct validity are

Copyright © 2019 for this paper by its authors.
Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

mainly considered whether the evaluation indicator used in our experiment can accurately reflect the prediction performance of the prediction model. The experiment uses AUC to evaluate the JIT-SDP model, which is also widely adopted by previous research [22].

Internal validity. The threats to internal validity are mainly from experimental code. Our experimental code is written in python. In order to reduce errors in the code, we used mature libraries and carefully checked the code of the experiment.

VII. Conclusions and Future Work

In this paper, we first introduce progressive sampling for the JIT-SDP. Progressive sampling is a commonly used sampling strategy that progressively increases the number of training samples to determine the optimal number of training samples.The experiment is conducted on six open source projects with 227417 changes. Our prediction model is built based on the random forest algorithm and evaluated by AUC.

Large-scale empirical studies demonstrate that progressive sampling is feasible in the JIT-SDP scenario. Moreover, experimental results show that the optimal training sample size derived by progressive sampling is very small. Especially, the proportion of training samples to the total number of data sets is less than 3% on the projects MOZ, PLA, and POS. Therefore, we suggest that progressive sampling can be used in the practical application of JIT-SDP to determine the optimal number of samples, thereby reducing the number of training samples and reducing the cost of acquiring data sets.

In the future, we plan to design a more intelligent progressive sampling method. We aim to further reduce the training sample size by selecting samples more intelligently so that progressive sampling can reach the termination condition earlier. Secondly, in order to further verify the generalization of the experimental conclusions, we hope to collect more data sets to improve the reliability of the experimental conclusions.

Acknowledgment

This work is partially supported by the NSF of China under grants No.61772200 and 61702334, Shanghai Pujiang Talent Program under grants No. 17PJ1401900. Shanghai Municipal Natural Science Foundation under Grants No. 17ZR1406900 and 17ZR1429700. Educational Research Fund of ECUST under Grant No. ZH1726108. The Collaborative Innovation Foundation of Shanghai Institute of Technology under Grants No. XTCX2016-20.

References

[1] 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 Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2013.

[2] Z. Li, X. Jing, and X. Zhu, "Progress on approaches to software defect prediction," *IET Software*, vol. 12, no. 3, pp. 161–175, 2018.

[3] L. Cai, Y. Fan, M. Yan, and X. Xia, "Just-in-time software defect prediction:a road map," *Journal of Software*, vol. 30, no. 5, pp. 1288–1307, 2019.

[4] A. Mockus and D. M. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal*, vol. 5, no. 2, pp. 169–180, 2000.

[5] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, "An industrial study on the risk of software changes," in *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering, FSE*, p. 62, 2012.

[6] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *37th IEEE/ACM International Conference on Software Engineering, ICSE*, pp. 99–108, 2015.

[7] Q. Huang, X. Xia, and D. Lo, "Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction," *Empirical Software Engineering*, vol. 24, no. 5, pp. 2823–2862, 2019.

[8] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi, "Deepjit: an end-to-end deep learning framework for just-in-time defect prediction," in *Proceedings of the 16th International Conference on Mining Software Repositories, MSR*, pp. 34–45, 2019.

[9] J. Liu, Y. Zhou, Y. Yang, H. Lu, and B. Xu, "Code churn: A neglected metric in effort-aware just-in-time defect prediction," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM*, pp. 11–19, 2017.

[10] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes," *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 641–657, 2017.

[11] X. Chen, L. Wang, Q. Gu, Z. Wang, C. Ni, W. Liu, and Q. Wang, "A survey on cross-project software defect prediction methods," *Journal of Computer*, vol. 41, no. 1, pp. 254–274, 2018.

[12] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki, "Cost-efficient sampling for performance prediction of configurable systems (T)," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE*, pp. 342–352, 2015.

[13] A. Lazarevic and Z. Obradovic, "Data reduction using multiple models integration," in *Principles of Data Mining and Knowledge Discovery, 5th European Conference, PKDD*, pp. 301–313, 2001.

[14] X. Chen, Y. Zhao, Q. Wang, and Z. Yuan, "MULTI: multi-objective effort-aware just-in-time software defect prediction," *Information & Software Technology*, vol. 93, pp. 1–13, 2018.

[15] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[16] X. Yang, H. Yu, G. Fan, and K. Yang, "An empirical studies on optimal solutions selection strategies for effort-aware just-in-time software defect prediction," in *The 31st International Conference on Software Engineering and Knowledge Engineering, SEKE*, pp. 319–424, 2019.

[17] G. G. Cabral, L. L. Minku, E. Shihab, and S. Mujahid, "Class imbalance evolution and verification latency in just-in-time software defect prediction," in *Proceedings of the 41st International Conference on Software Engineering, ICSE*, pp. 666–676, 2019.

[18] G. M. Weiss and Y. Tian, "Maximizing classifier utility when there are data acquisition and modeling costs," *Data Mining and Knowledge Discovery*, vol. 17, no. 2, pp. 253–282, 2008.

[19] F. J. Provost, D. D. Jensen, and T. Oates, "Efficient progressive sampling," in *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD*, pp. 23–32, 1999.

[20] G. H. John and P. Langley, "Static versus dynamic sampling for data mining," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining, KDD*, pp. 367–370, 1996.

[21] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[22] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, "Studying just-in-time defect prediction using cross-project models," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2072–2106, 2016.

[23] Y. Jiang, B. Cukic, and T. Menzies, "Can data transformation help in the detection of fault-prone modules?," in *Proceedings of the 2008 Workshop on Defects in Large Software Systems, held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA*, pp. 16–20, 2008.

[24] Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams, and A. E. Hassan, "Revisiting common bug prediction findings using effort-aware models," in *26th IEEE International Conference on Software Maintenance, ICSM*, pp. 1–10, 2010.

[25] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 1–18, 2017.

Copyright © 2019 for this paper by its authors.
Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).