Connecting Software Metrics across Versions to Predict Defects

Yibin Liu^{1,2,*}, Yanhui Li^{1,2,*,†}, Jianbo Guo^{3,*}, Yuming Zhou^{1,2}, Baowen Xu^{1,2,†}
1. State Key Laboratory for Novel Software Technology, Nanjing University, China
2. Department of Computer Science and Technology, Nanjing University, China
3. Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing, China
yibinliu93@foxmail.com, {yanhuili, zhouyuming, bwxu}@nju.edu.cn, jianboguo@outlook.com

***Abstract*—Accurate software defect prediction could help software practitioners allocate test resources to defect-prone modules effectively and efficiently. In the last decades, much effort has been devoted to build accurate defect prediction models, including developing quality defect predictors and modeling techniques. However, current widely used defect predictors such as code metrics and process metrics could not well describe how software modules change over the project evolution, which we believe is important for defect prediction. In order to deal with this problem, in this paper, we propose to use the Historical Version Sequence of Metrics (HVSM) in continuous software versions as defect predictors. Furthermore, we leverage Recurrent Neural Network (RNN), a popular modeling technique, to take HVSM as the input to build software prediction models. The experimental results show that, in most cases, the proposed HVSM-based RNN model has significantly better effort-aware ranking effectiveness than the commonly used baseline models.**

I. Introduction

Fixing software defects is a very important part of software maintenance that consumes a huge amount of time and effort [8]. The preliminary step of bug fixing is to identify the potential locations of bugs in a software project. In the last decades, many software defect prediction models [17], [18], [23], [27], [55], [56], [63], [65] have been proposed to identify defect-prone modules, which could help software engineers test and debug software more effectively and efficiently. In order to achieve accurate defect prediction, it is essential to use quality defect predictors and modeling techniques to build the prediction models.

There are two main aspects of concern in the establishment of prediction models: one is proposing new views to collect metrics or selecting proper metrics to improve the prediction performance; the other is introducing new classification techniques that can perform better and comparing them with previously used techniques.

In terms of metrics for defect prediction, most existing studies concentrate on two types of metrics: code metrics and process metrics. The code metrics can well describe the static characteristics of a file in a given version, and a proper classifier could group those similar files together to distinguish buggy files from clean ones. CK features [5] and McCabe features [32] are the commonly studied static metrics which are extracted from code static properties (e.g., dependencies, function and inheritance counts). Recently, Wang et al. [56] leveraged Deep Belief Network (DBN) to automatically learn semantic features which is a more complex kind of code metrics to capture the semantic difference of source code. Process metrics have been provided to describe the change information in project's development. Bird et al. [3] examined the effect of ownership in defect prediction and provided process metrics that measure the contribution of developers in projects. Mockus and Weiss [37] studied the performance of metrics measuring the change of a project in predicting the risk of new changes. Arisholm et al. [1] and Rahman and Devanbu [45] both investigated the predictive power of different types of metrics including code and process metrics, and drew the same conclusion that using process metrics can significantly improve the performance of prediction. McIntosh et al. [33] also studied the predictive power of code review coverage, participation and expertise, and found them effective in predicting defects. In most cases, process metrics are version-duration, which means that they are extracted from two adjacent versions to describe the change of files between the two versions.

Meanwhile, researchers have applied many machine learning classification algorithms to build prediction models in software defect prediction [9], [54]. Wang and Li [57] constructed a Naive Bayes based model to predict software defects on MDP datasets, which has better performance than decision tree based learner J48. Sun et al. [52] compared the performance of several types of classification algorithms over the 14 NASA datasets, and found that Random Forest is not significantly better than the others. Zhang et al. [65] proposed a connectivity-based unsupervised classifier, Spectral Clustering (SC), and compared the performance of SC with supervised classifiers using data from 26 projects.

This paper deviates from existing studies in two important ways. **Firstly,** we propose a new way to construct predictors based on software metrics: we connect a file's metrics in several continuous versions together in ascending order of version (Figure 1). We call this new predictor **Historical Version Sequence of Metrics** (HVSM). Compared with code metrics and process metrics, HVSM has the following advantages:

- HVSM provides a new and more complete perspective for engineers and managers to consider and explain the trend of how the files change over the project's evolution.

* These authors contributed equally to this work
† Corresponding authors

978-1-5386-4969-5/18/$31.00 © 2018 IEEE

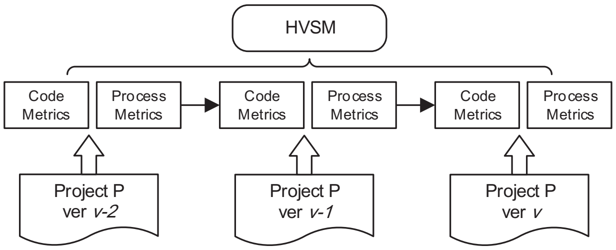


Fig. 1. An overview of different types of metrics

- HVSM employs code metrics to describe files' static data or process metrics to describe change data. Additionally, the sequence of HVSM can reveal the files' comprehensive change history which are not included in most process metrics.

Secondly, we bring in a new classification technique, Recurrent Neural Network (RNN) [29], to process the HVSMs. Since a long lived file may exist in more versions, and has a longer HVSM than others, the HVSM set of the project may contains variable-length HVSMs. The existing techniques [7], [52], [57], [65] of defect prediction cannot directly handle variable-length metrics in HVSMs. An RNN is a powerful graphical model for sequential data inputs, which uses the internal state to memorize previous inputs and handle variable length inputs.

Our approach consists of three steps: a) constructing the historical version sequence of each file in our projects; b) extracting the HVSM of each file from its version sequence; (c) leveraging RNN to predict file-level defects using the extracted HVSMs (details are in Figure 3 and Section III).

The main contributions of this paper are listed as follows:

- We provide a novel view to reveal the static and change information of a file in the version sequence of a project, which is expressed in the form of HVSM in defect prediction.
- We leverage HVSM with the help of a classification technique RNN to improve the performance in within-project defect prediction (WPDP) and evaluate the result of our approach compared with 7 typical classifiers on 9 open source projects from PROMISE. The SK test and Win/Tie/Loss evaluation shows that our approach outperforms other techniques in effort-aware scenarios.

The rest of this paper is organized as follows. Section II describes the background and related work on defect prediction and RNN. Section III introduces our approach to extract HVSMs and applies RNN to perform defect prediction. Section IV shows the experimental setup. Section V evaluates the performance of our approach against other techniques. Some discussions and threats to validity of our work are presented in section VI and VII. We conclude our work in section VIII.

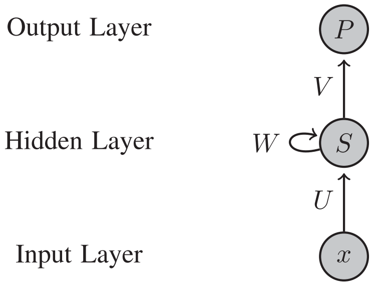


Fig. 2. A RNN model with one hidden layer.

II. Background and Related Work

This section provides the background and related work of file-level defect prediction and recurrent neural network.

A. Typical Defect Prediction Process

Defect prediction on file-level has been studied by many prior works [9], [18], [27], [35], [40]–[42], [44], [48], [56], [59]. The typical process of file-level defect prediction is as follows: firstly, labeling data as buggy or clean based on bug reports for each file; secondly, collecting corresponding metrics of these files as features; thirdly, training prediction models using machine learning classifiers with the input of instances with features and bug labels; Finally, predicting whether a new instance is buggy or clean using trained models.

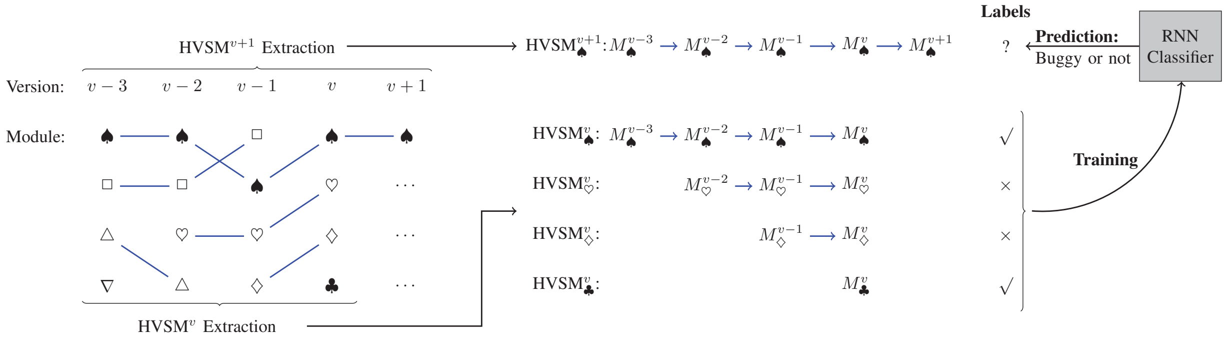
There are two main scenarios of defect prediction, one of them is within-project defect prediction (WPDP) [6], [27], [30], [34], [41], [56]. In WPDP, researchers always train classifiers using the data in an older version and predict defects in a newer version within the same project. The other one is cross-project defect prediction (CPDP) [14], [16], [42], [56]. The CPDP problem is motivated by the fact that many companies lack the training data in practice. A typical solution for CPDP is to apply prediction models that are built using data from a different source project.

In this study, we focus on improving the performance in WPDP with our approaches.

B. Recurrent Neural Network

A recurrent neural network (RNN) is a powerful graphical model for sequential data, which can handle variable length inputs or outputs for different demands, like speech recognition, video analysis and natural language translation [11], [26], [29]. Recently, RNN has been applied in related software engineering problems, especially in obtaining an API usage sequence for a given natural language query based on RNN Encoder-Decoder [12].

Figure 2 shows the main structure of RNN, which contains one input layer, several self-connected hidden layers and one output layer. RNN processes information from a sequential input $x^1, x^2, ..., x^T$, where T is the length of the sequential data, by incorporating it into the hidden state S $(S^1, S^2, ..., S^T)$ that is passed through time, and outputs $P = P^T$ in the end which is combined with the corresponding training target y to get the loss L to train the network. U, V and W are the weight matrix


Labels
HVSM^{v+1} Extraction
HVSM$^{v+1}_{♠}$: $M^{v-3}_{♠} \rightarrow M^{v-2}_{♠} \rightarrow M^{v-1}_{♠} \rightarrow M^{v}_{♠} \rightarrow M^{v+1}_{♠}$
?
Prediction:
Buggy or not
RNN Classifier
Version: $v-3$ $v-2$ $v-1$ v $v+1$
Module:
♠ ♠ □ ♠ ♠
□ □ ♠ ♡ ...
△ ♡ ♡ ♢ ...
▽ △ ♢ ♣ ...
HVSM$^{v}_{♠}$: $M^{v-3}_{♠} \rightarrow M^{v-2}_{♠} \rightarrow M^{v-1}_{♠} \rightarrow M^{v}_{♠}$ √
HVSM$^{v}_{♡}$: $M^{v-2}_{♡} \rightarrow M^{v-1}_{♡} \rightarrow M^{v}_{♡}$ ×
HVSM$^{v}_{♢}$: $M^{v-1}_{♢} \rightarrow M^{v}_{♢}$ ×
HVSM$^{v}_{♣}$: $M^{v}_{♣}$ √
Training
HVSM^{v} Extraction

(a) Historical Version Sequence **(b) Historical Version Sequence of Metrics** **(c) RNN**

Fig. 3. Overview of our proposed HVSM-based defect prediction

between input and hidden layer, the weight matrix between hidden and output layer, and the weight matrix between hidden layer and itself at adjacent time steps respectively. The hidden state S^t at every time step t of the sequence is modeled as follows:

$$S^t = \begin{cases} tanh(Ux^t + b), if\ t = 1 \\ tanh(Ux^t + WS^{t-1} + b), if\ t \in \{2, 3, ..., T\} \end{cases} \tag{1}$$

where $tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ and the vector b are the bias parameters. Current activations S^t in the hidden layer are determined by both current input x^t and previous hidden layer activations S^{t-1}, which makes the network's internal state **memorize** previous inputs. The memory power of RNN is the core reason we select RNN to deal with sequential inputs. And the output, i.e. the probability for $y = 1$, is:

$$P^T = sigmoid(VS^T + c) \tag{2}$$

where $sigmoid(c) = \frac{1}{1+e^{-c}}$ and c is the bias parameter. Once P^T is obtained, the loss L is computed as:

$$L = -ylogP^T - (1-y)log(1-P^T) + \frac{\lambda}{2}||\omega||_2^2 \tag{3}$$

where $\frac{\lambda}{2}||\omega||_2^2$ is the so called Tikhonov regularization to avoid over-fitting on the training set:

$$||\omega||_2^2 = \sum_{i,j} U_{ij}^2 + \sum_{i,j} V_{ij}^2 + \sum_{i,j} W_{ij}^2$$

RNN can be trained to minimize the loss L using gradient descent with a technique known as back-propagation [58], [61]. We first initialize U, V and W randomly, then set b and c to $\mathbf{0}$, and update them by the iteration process:

$$\begin{cases} U_{ij}(l+1) &= U_{ij}(l) - \eta\frac{\partial L}{\partial U_{ij}} \\ V_{ij}(l+1) &= V_{ij}(l) - \eta\frac{\partial L}{\partial V_{ij}} \\ W_{ij}(l+1) &= W_{ij}(l) - \eta\frac{\partial L}{\partial W_{ij}} \\ b_k(l+1) &= b_k(l) - \eta\frac{\partial L}{\partial b_k} \\ c(l+1) &= c(l) - \eta\frac{\partial L}{\partial c} \end{cases} \tag{4}$$

where l is the l^{th} iteration, η is the learning rate, U_{ij}, V_{ij}, W_{ij} are the i^{th} row, j^{th} column entry of matrix U, V, W respectively, b_k is the k^{th} entry of bias vector b.

III. Our Approach

In this work, we focus on the sequential information of files across versions. For files that exist in several continuous versions, the comprehensive sequential information in change history is useful when performing defect prediction. We define **Historical Version Sequence of Metrics (HVSM)** to highlight the version sequence information of files. With HVSM, a classification technique RNN is used to predict defective files. Our approach consists of three major steps: (1) constructing file's historical version sequence (2) extracting HVSMs from files' version sequences, (3) leveraging RNN to predict defects using the extracted HVSM.

A. Constructing Historical Version Sequence

Considering the development of a project, files change across versions. As shown in Figure 3(a), each symbol represents a file in the project and they exist in different versions from $v-3$ to $v+1$. The connection lines between versions indicate the sequential change of the same file. We sort all versions containing a file a in ascending order, and call it a's historical version sequence. From Figure 3, taking version v as the current version, the version sequence of file ♠ lasts from $v-3$ until now, while file □ has its version sequence ending in the previous version $v-1$. In our approach, we consider the files with the same name and the same path as one specific file when processing file's version sequence.

It should be noticed that, in common, files always exist in continuous versions. To summarize, we define 3 types of files at the time of a specific version v:

- **Developing File** (♠, ♡, ♢): the files that are created in previous versions, and still exist in current version.
- **Newborn File** (♣): The files that are created in current version.
- **Dead File** (▽, □, △): The files that exist in previous versions and disappear in current version.

The high percentage of developing files (DFs) will make the historical information more complete when constructing historical version sequence. In section IV-C, we will show the percentage of DFs in our studied projects.

For the project shown in Figure 3, assume that engineers want to predict defects in version $v+1$ using the versions before. A common practice [56] is to use files in version v to build the training set for a classifier. Code metrics and process metrics extracted from these files are frequently used in defect prediction. Considering the change history of files in a project, process metrics apparently provide more information than code metrics. Prior works extract process metrics from code ownership [3], change frequency [1], [45], developer experience [37] and etc. Others also extract change metrics from version history [1] or use the metric difference between versions as features [22].

The approaches above do take the change and process information of files into account, but most of the metrics are version-duration, which means that the metric considers only the information between two adjacent versions and cannot reveal the historical information of whole version sequence.

B. Extracting HVSM

Here we introduce the **Historical Version Sequence of Metrics (HVSM)** as mentioned at the beginning of this section to highlight the sequential information of file's changes across versions. For a specific length len, HVSM joins a file's metrics of at most len continuous versions traced back from version v, and groups the metrics in ascending order of versions. For convenience, we introduce some symbols: for a given file a in version x, we denote M_a^x as its metric set; for a in a studied version v, we define $HVSM_a^v$ as:

$$HVSM_a^v = M_a^{v_o} \to M_a^{v_o+1} \to ... \to M_a^v$$

where v_o is the first version during the last len versions, that a exists. And we denote $T_a^v = v - v_o + 1$ as the number of versions file a exists from v_o to v, and call it the length of $HVSM_a^v$. For example in Figure 3, let $len = 4$, the file ♡ has $T_{\heartsuit}^v = 3$. Assuming the metric set $M_{\heartsuit}^x$ containing 10 metrics, then its $HVSM_{\heartsuit}^v$ includes $10 \times 3 = 30$ metric values. It should be noticed that $1 \leq T_a^v \leq len$. For a specific software project, we denote $HVSM^v$ as the set containing the HVSMs of all files in version v of this project:

$$HVSM^v = \{HVSM_a^v | a \in \text{all files in version } v\}$$

and $m^v = |HVSM^v|$ denotes the number of files in version v of the project. Then we define the length of $HVSM^v$ as the number len of concerned versions: $length(HVSM^v) = len$

Take the project in Figure 3 as an example. For $len = 4$, the $HVSM^v$ that contains HVSMs of the four listed files ♠, ♡, ♢ and ♣ are shown in the right part of the figure. The dead files ∇, □ and △ do not exist in version v so they are not included in $HVSM^v$. When predicting defects in $v+1$, the test set is also built with HVSM as the example $HVSM_{\spadesuit}^{v+1}$ shown in the figure.

C. Applying RNN to HVSM

In this section, we will introduce how to apply a proper classification technique, which is RNN in this paper, to the data of HVSMs to improve the performance of defect prediction.

In our study, for file a in a given version v, the input of typical classifiers will be the metric set of a, i.e. M_a^v. When it comes to RNN, the input should be the HVSM of a, i.e. $HVSM_a^v$. It should be noticed that RNN can handle sequential data (see Section II-B), so RNN will still work if the length of each file's HVSM differs in training set or test set.

Figure 4 illustrates the training process. First, we obtain a training sample $HVSM_a^v$ from the training set and then unfold RNN along the input sequence $HVSM_a^v$ according to its length T_a^v. Then, a forward pass is done: we let each metric set $M_a^{v-T_a^v+t}$ of $HVSM_a^v$ be the input of RNN at each time step, i.e.

$$x^t = M_a^{v-T_a^v+t} \tag{5}$$

where $t \in \{1, 2, ..., T_a^v\}$ and then compute hidden state $S^1, S^2, ..., S^{T_a^v}$ successively according to equation 1 as well as obtain the probability that the file a in version v has bugs $P^{T_a^v}$ at last. Finally, in backward pass, we obtain the loss L_a and compute its gradient $\frac{\partial L_a}{\partial \omega}$ on $HVSM_a^v$ with back-propagation technique, where ω represents the weight parameters $U_{ij}, V_{ij}, W_{ij}, b_k$ and c in equation 4. After obtaining gradient on all the training samples, we compute the gradient needed in equation 4 by

$$\frac{\partial L}{\partial \omega} = \frac{1}{m_{tr}^v} \sum_a \frac{\partial L_a}{\partial \omega} \tag{6}$$

where m_{tr}^v is the number of training samples. Then the new weights U, V, W, b and c can be obtained by the iterations in equation 4.

Once we complete the training process, predictions on the test set could be made by a similar process: for any instance $HVSM_b^v$ in the test set, first do a T_b^v-length unfolding, and then compute the probability $P^{T_b^v}$ that file b in version v has bugs via forward pass.

The most important thing to note is that in the RNN model, the same weights are reused at every time step (parameters sharing), which makes our RNN able to handle HVSMs with variable length. For example, during training process, for file ♡ in version v, $T_{\heartsuit}^v = length(HVSM_{\heartsuit}^v) = 3$, then we get a $3-$length unfolding RNN to process it. During the prediction process, for file ♢ in version v, $T_{\diamondsuit}^v = length(HVSM_{\diamondsuit}^v) = 2$, then we get a $2-$length unfolding RNN to process it.

IV. Experimental Setup

We conduct several experiments to study the performance of RNN using HVSMs and compare it with existing classification techniques.

A. Collected Datasets

We use data from 9 projects in the PROMISE data repository to make it available to replicate and verify our experiment. Each project has several versions and code metrics with clear defect information. In this work, each file's defect information is labeled as 0 (clean file) or 1 (buggy file). As shown in Table I, the length of each project's version sequence varies from 3 versions to 5 versions, which makes it available to extract

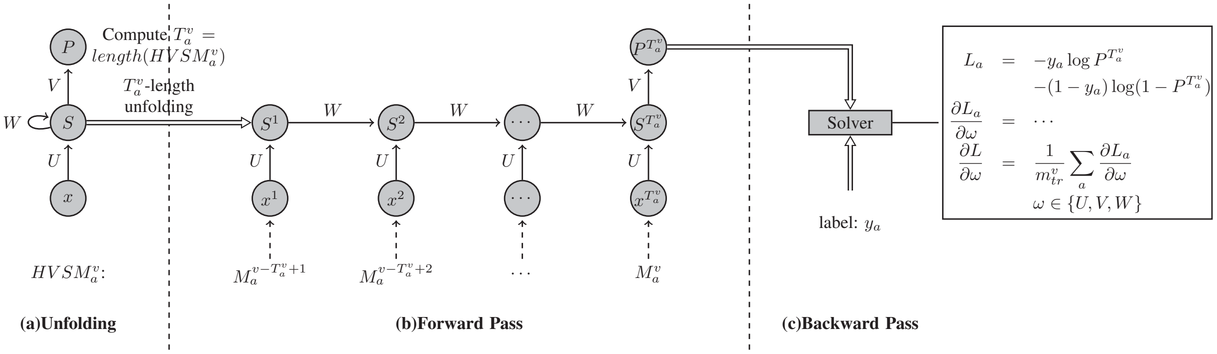


Fig. 4. Training process of RNN on a HVSM

HVSM with $len > 1$ when predicting WPDP. The data of these projects was investigated and shared by Jureczko and Madeyski [19] and also used by others' work [4], [13], [56].

TABLE I
INFORMATION OF COLLECTED PROJECTS

Project	Collected Versions	Avg. #Files	Avg. KLOC	Avg. %Bugs
ant	1.3, 1.4, 1.5, 1.6, 1.7	338	100	19.6%
camel	1.0, 1.2, 1.4, 1.6	696	78	18.9%
jedit	3.2, 4.0, 4.1, 4.2, 4.3	350	160	19.6%
log4j	1.0, 1.1, 1.2	150	27	50.4%
lucene	2.0, 2.2, 2.4	261	72	54.9%
poi	1.5, 2.0, 2.5, 3.0	345	99	49.8%
velocity	1.4, 1.5, 1.6	213	54	58.5%
xalan	2.4, 2.5, 2.6	804	314	36.6%
xerces	init, 1.2, 1.3, 1.4	411	140	38.3%

B. Baseline Classifiers

In this work, 7 typical classifiers are used as baselines to compare the performance with our RNN techniques. Most of the typical classifiers are studied by previous work [7], [43], [45], [52], [57] in defect prediction. The 7 classifiers are Naive Bayes (NB), Logistic Regression (LR), k-Nearest Neighbor (KNN), Random Forest (RF), C5.0 decision tree (C5.0), standard Neural Network (NN) and C4.5-like decision trees (J48).

C. Model Construction

For each project, we have 3 to 5 collected versions, which construct the version sequence of this project. In order to observe the performance of our HVSM, the studied versions are those with enough number of previous versions, which provide HVSM with enough length.

In WPDP, files in an older version a are used as training set to predict defects in a newer version b, and the two versions are in the same project. For the metrics used in training set and test set, typical classifiers should perform defect prediction using M^v in version v while RNN uses $HVSM^v$. For example, considering the situation that researchers want to predict defects in **ant 1.7** with extracted metrics and labeled bugs in **ant 1.6** and former versions. Table II lists the metrics used by typical classifiers and our approach in this situation. Typically, classifiers use metrics in the training set **ant 1.6** without historical information in former versions, while our approach RNN use $HVSM^{1.6}$ which contains the sequential information from **ant 1.3** to **ant 1.6** as training data. In order to include more historical information, this study selects the **last two versions** in each project's version sequence as the

TABLE II
AN EXAMPLE OF METRICS AND VERSIONS INCLUDED FOR TYPICAL CLASSIFIERS AND RNN

		RNN	Typical Classifiers
Training Set	Metric set Versions [1]	$HVSM^{1.6}$ ant 1.3, 1.4, 1.5, 1.6	$M^{1.6}$ ant 1.6
Test Set	Metric set Versions	$HVSM^{1.7}$ ant 1.3, 1.4, 1.5, 1.6, 1.7	$M^{1.7}$ ant 1.7

[1] refers to the versions included in the corresponding metric set

TABLE III
AN OVERVIEW OF HVSMS EXTRACTED AS TRAINING AND TEST SET IN EACH PROJECT

Project (Tr->T)[1]	HVSM	Version Sequence	Avg. len [2]	#Files	#DF[3]	%DF
ant 1.6->1.7	$HVSM^{1.6}$ $HVSM^{1.7}$	1.3,1.4,1.5,1.6 1.3,1.4,1.5,1.6,1.7	2.6 2.3	351 745	293 355	83.5% 47.7%
camel 1.4->1.6	$HVSM^{1.4}$ $HVSM^{1.6}$	1.0,1.2,1.4 1.0,1.2,1.4,1.6	2.0 2.7	872 965	577 857	66.2% 88.8%
jedit 4.2->4.3	$HVSM^{4.2}$ $HVSM^{4.3}$	3.2,4.0,4.1,4.2 3.2,4.0,4.1,4.2,4.3	3.2 2.4	367 492	291 225	79.3% 45.7%
log4j 1.1->1.2	$HVSM^{1.1}$ $HVSM^{1.2}$	1.0,1.1 1.0,1.1,1.2	1.9 2.0	109 205	98 117	89.9% 57.1%
lucene 2.2->2.4	$HVSM^{2.2}$ $HVSM^{2.4}$	2.0,2.2 2.0,2.2,2.4	1.8 2.2	247 340	192 235	77.7% 69.1%
poi 2.5->3.0	$HVSM^{2.5}$ $HVSM^{3.0}$	1.5,2.0,2.5 1.5,2.0,2.5,3.0	2.4 3.1	385 442	314 382	81.6% 86.4%
velocity 1.5->1.6	$HVSM^{1.5}$ $HVSM^{1.6}$	1.4,1.5 1.4,1.5,1.6	1.7 2.6	214 229	155 210	72.4% 91.7%
xalan 2.5->2.6	$HVSM^{2.5}$ $HVSM^{2.6}$	2.4,2.5 2.4,2.5,2.6	1.9 2.6	803 885	689 766	85.8% 86.6%
xerces 1.3->1.4	$HVSM^{1.3}$ $HVSM^{1.4}$	init,1.2,1.3 init,1.2,1.3,1.4	2.2 2.2	453 588	433 328	95.6% 55.8%

[1] Tr denotes the training set and T denotes the test set.
[2] refers to $length(HVSM)$ (see Section III-B).
[3] refers to Developing Files. For a given version v, developing files were created before version v, and still exist in v (see Section III-A).

training and test set in WPDP, and make the length of HVSM long enough. Table III shows the extracted HVSMs of the two selected versions as training and test set in WPDP in each project. The table also shows the number and ratio of developing files in each selected version. The high average percentage of developing files insures that HVSM will cover most of the files in a project's version history.

In order to train our defect prediction models, we use the implementations of the 7 typical classifiers provided by R packages. RNN is manually implemented using MATLAB since there is no suitable RNN packages with the data in the forms of HVSM as the input. We use the MATLAB code **fming.m** written by Carl Edward Rasmussen[1] to realize the iteration process. To be noticed that some classifiers, like RF and NN based classifiers (RNN and NN) may have randomness in prediction, this paper repeat the classification process 10 times for each of these classifiers and report the mean value.

This paper also applies the automated hyper-parameter optimization on the classification techniques introduced by Tantithamthavorn et al. [54] using `caret R` [25] package.

D. Metrics

In our study we use code metrics and process metrics for our classification techniques and HVSM that RNN uses.

1) Code Metrics: Code metrics used for classification techniques in this study are extracted and investigated by Jureczko and Madeyski [19] in previous work. According to the paper, there are in total 20 code metrics including the common used LOC (lines of code) in addition to the other 19 metrics suggested by Chidamber and Kemerer [5], Henderson-Sellers [15], Bansiy and Davis [2], Tang et al. [53], Martin [31], and McCabe [32].

2) Process Metrics: Different from code metrics which are static metrics within each release of projects, process metrics measure the change information of files during a period of time. In this study, 4 change metrics studied by Nagappan and Ball [39] are extracted as process metrics. **ADD** and **DEL** measure the lines of code added or deleted in a specific file from last release to current release. **CADD** and **CDEL** are cumulative lines of code added or deleted during the whole version sequence until a specific release. These metrics are studied as effective predicting indicators by previous works [37]–[39].

In total, there are 20 code metrics and 4 process metrics involved in this study.

E. Evaluation

There are many evaluation measures in the field of defect prediction. In this study, we mainly focus on the performance of classification techniques in effort-aware scenarios.

1) Cost-Effectiveness: When performing defect prediction, classifiers always rank the files by their probability of being defective. Sometimes practitioners do not have enough resources to inspect the whole project, they prefer to check those files with small size and high fault-proneness. In this situation, a cost-effective model would rank the files in descending order of their bug density. The effort-aware ranking effectiveness of classification techniques in defect prediction is always evaluated by cost-effectiveness (CE) curve, which is widely used in prior works [30], [45], [63]. Figure 5 shows an example of the CE curve. In the figure, x-axis represents the cumulative percentage of LOC of the files, and y-axis is the cumulative percentage of bugs detected by the selected files. For a prediction model A, we sort the files in descending orders of $P(buggy)/LOC$, where $P(buggy)$ is the predicted probability of a file being defective. A CE curve of model A plots proportion of defects truly detected against proportion of LOC coming from the ordered set of files. We use the following formula introduced by Arisholm et al. [1] to calculate CE :

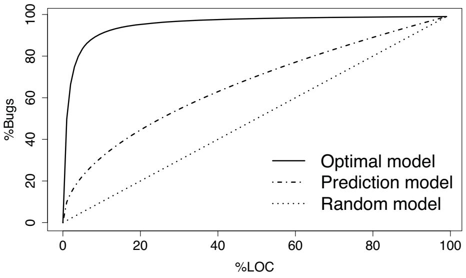


Fig. 5. An example of CE curve

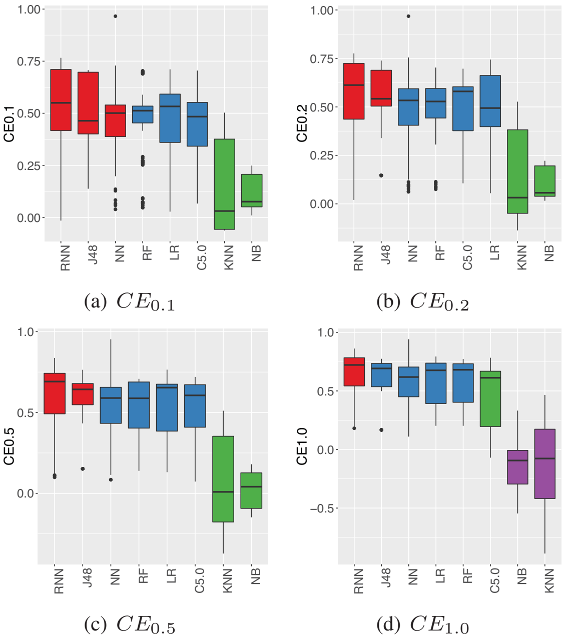
$$CE_\pi = \frac{Area_\pi(M) - Area_\pi(Random)}{Area_\pi(Optimal) - Area_\pi(Random)}$$

Where $Area_\pi(A)$ is the area under the curve of model A (M, $Random$ or $Optimal$) for a given π percentage of LOC. In random model, files are randomly selected to inspect, while in optimal model, files are ranked in descending order according to their actual bug densities. The larger CE_π means a better ranking effectiveness. The cut-off π varies from 0 to 1 indicating the percentage of LOC that we inspect. In this work, we report CE_π at π = 0.1, 0.2, 0.5 and 1.0.

2) Scott-Knott Test: Scott-Knott (SK) test [49] (using the 95% confidence level) is also applied in this paper to group classifiers into statistically distinct ranks. The SK test recursively ranks the evaluated classifiers based on hierarchical clustering analysis. It clusters the evaluated classifiers into two groups based on evaluation metrics, and recursively executes within each rank until no significant distinct groups can be created [9]. The SK test has been used in prior works [9], [21], [36], [65] to compare the performance of different classifiers.

3) Win/Tie/Loss: To further compare the performance of classifiers apart from average value or average rank, we also apply Win/Tie/Loss results which is also used for performance comparison between different techniques by prior works [24], [41], [51]. For each project, we repeat the model training of RNN and other techniques that have randomness 10 times and have 10 scores of the performance for each technique. For each

[1] http://learning.eng.cam.ac.uk/carl/



(a) $CE_{0.1}$

(b) $CE_{0.2}$

(c) $CE_{0.5}$

(d) $CE_{1.0}$

Fig. 6. The boxplots of $CE_{\pi}, \pi = 0.1, 0.2, 0.5, 1.0$ values of RNN and 7 baseline classifiers using only code metrics. Different colors represents different Scott-Knott test ranks (from top down, the order is red, blue, green, purple).

technique that has no randomness (like logistic regression), the result is copied 10 times to make it comparable with RNN. After that, Wilcoxon signed-rank test [60] together with Cliff's delta δ [47] is conducted to compare the performance of RNN and other classification techniques. For a baseline technique A, if RNN outperforms A in a given project according to the Wilcoxon signed-rank test ($p < 0.05$), and the magnitude of the difference between RNN and A is not negligible according to Cliff's delta δ ($\delta \geq 0.147$), we mark the RNN as a 'Win'. In contrast, RNN is marked as a 'Loss' compared with technique B when $p < 0.05$ and $\delta \leq -0.147$, which means that B outperforms RNN in a project with statistical significance. Otherwise, the case is marked as a 'Tie'. Finally, we count the Wins, Ties and Losses for RNN against each technique in each project. This Win/Tie/Loss evaluation shows that the number of projects in which RNN outperforms other techniques with statistical significance.

V. Results

This section provides our experimental results. We focus on comparing our approach, RNN using HVSM, with other classifiers in within-project defect prediction (WPDP), and answer the following research question:

A. *RQ1: Does RNN with HVSM outperform other techniques in WPDP using code metrics?*

Generally speaking, our approach outperforms other techniques in effort-aware scenarios evaluated by CE, and the result is supported by SK test and Win/Tie/Loss results.

Figure 6 shows an overview of our approach comparing with other classifiers. The boxplots show the distribution of CE_{π} values of each classifier in the studied datasets. Different colors of the boxplot indicate different tiers that a classifier is ranked by SK test (using the 95% confidence level). The SK result shows that our approach ranks the first (red boxplots) under all the evaluation metrics.

Table IV shows the detailed comparison of CE_{π} values of the top four techniques. Considering the average value, RNN has the best performance among the top four techniques under all the evaluation metrics. Highlighted by bold font, Our approach achieved the best performance in no less than 6 (out of 9) datasets evaluated by different CE_{π}. In addition, we also provide average rank (AR) [18], [65] of each technique over all the projects. AR can well reflect how a classifier outperforms others with little influence by extreme values in a few dataset which is also adopted by other works [6], [28]. In the view of AR, our approach has 2.2, 2.2, 2.2 and 1.6 under $CE_{0.1}$, $CE_{0.2}$, $CE_{0.5}$ and $CE_{1.0}$ respectively, which are the best among the top 4 techniques. The superior AR shows a better applicability of our approach in different projects compared with other classifiers.

In order to further compare the classifiers, we also apply the Win/Tie/Loss indicator with the help of Wilcoxon signed-rank test and Cliff's delta δ. The Win/Tie/Loss result shows whether RNN is significantly better or not when compared with other classifiers. Our approach achieves at least 6 'Win' and no more than 3 'Loss' against a specific classifier in all the cases, which means that RNN outperforms others in at least 6 (out of 9) datasets with statistical significance. This result supports the better performance of RNN compared with baseline classifiers.

In summary, our approach outperforms baseline techniques using code metrics as training data in effort-aware scenarios evaluated by CE. The result is supported by SK test and Win/Tie/Loss evaluation.

B. *RQ2: Does our approach outperform other techniques in WPDP using both code and process metrics?*

In addition to code metrics, process metrics are also effective predictor in defect prediction. This paper also provides performance of RNN and typical classifiers using both code and process metrics as training data.

1) RQ2a: How is the performance of RNN and typical classifiers using both code and process metrics?: In this section, the training data for typical classifiers includes 24 metrics (20 code metrics and 4 process metrics), and in HVSM used by RNN, metric set in each version also consists of these 24 metrics.

Similar with RQ1, Figure 7 shows an overview of RNN comparing with other classifiers. Evaluated by $CE_{0.1}$, $CE_{0.5}$ and $CE_{1.0}$, our approach ranks the first and has significant distinction with most of the typical techniques. When it comes to $CE_{0.2}$, RNN still ranks at the top but is together with other 5 classifiers, which means that RNN has similar performance with them. Nevertheless, RNN has the best average $CE_{0.2}$ (0.534) over 9 datasets according to Table V and is 10% more than the second technique J48 (0.484). The SK result together

TABLE IV
THE CE_π VALUES OF THE TOP 4 CLASSIFIERS USING **CODE METRICS** IN WPDP. BOLD FONT HIGHLIGHTS THE BEST PERFORMANCE.

	$CE_{0.1}$				$CE_{0.2}$				$CE_{0.5}$				$CE_{1.0}$			
Target (Tr->T)	RNN	J48	NN	RF	RNN	J48	NN	RF	RNN	J48	NN	RF	RNN	J48	NN	LR
ant 1.6->1.7	0.023	**0.138**	0.09	0.069	0.049	**0.147**	0.098	0.096	0.125	0.152	0.127	**0.154**	**0.209**	0.167	0.158	0.202
camel 1.4->1.6	0.354	0.307	**0.369**	0.267	**0.396**	0.339	0.39	0.317	**0.462**	0.433	0.446	0.394	**0.533**	0.498	0.517	0.507
jedit 4.2->4.3	**0.438**	0.401	0.391	0.46	0.455	**0.508**	0.372	0.479	0.496	**0.643**	0.378	0.584	0.537	**0.734**	0.43	0.392
log4j 1.1->1.2	**0.757**	0.702	0.692	0.569	**0.715**	0.689	0.641	0.525	**0.706**	0.672	0.586	0.462	**0.719**	0.692	0.588	0.32
lucene 2.2->2.4	**0.757**	0.707	0.705	0.696	**0.767**	0.711	0.73	0.699	**0.781**	0.721	0.752	0.706	**0.806**	0.731	0.784	0.794
poi 2.5->3.0	**0.6**	0.463	0.505	0.511	**0.67**	0.579	0.575	0.62	**0.737**	0.568	0.637	0.687	**0.781**	0.536	0.681	0.779
velocity 1.5->1.6	**0.538**	0.495	0.508	0.528	**0.571**	0.542	0.528	0.566	0.694	0.679	0.653	**0.7**	**0.764**	0.757	0.733	0.737
xalan 2.5->2.6	**0.548**	0.464	0.514	0.533	**0.612**	0.504	0.582	0.593	**0.671**	0.548	0.648	0.664	**0.708**	0.537	0.686	0.692
xerces 1.3->1.4	**0.701**	0.697	0.5	0.474	**0.764**	0.739	0.523	0.458	**0.824**	0.764	0.537	0.411	**0.848**	0.774	0.525	0.676
Avg.	**0.524**	0.486	0.475	0.456	**0.555**	0.529	0.493	0.484	**0.611**	0.576	0.529	0.529	**0.656**	0.603	0.567	0.567
AR	2.2	3.9	3.6	4.1	2.2	3.3	4.1	4.2	2.2	3.7	4.3	3.6	1.6	3.9	4.0	3.4
Win/Tie/Loss	—	7/1/1	7/1/1	7/1/1	—	7/0/2	7/1/1	6/2/1	—	7/0/2	7/2/0	6/0/3	—	8/0/1	7/2/0	7/2/0

[1] Tr denotes the training set version and T denotes the test set version.

TABLE V
THE CE_π VALUES OF THE TOP 4 CLASSIFIERS USING BOTH **CODE AND PROCESS METRICS** IN WPDP. BOLD FONT HIGHLIGHTS THE BEST PERFORMANCE.

	$CE_{0.1}$				$CE_{0.2}$				$CE_{0.5}$				$CE_{1.0}$			
Target (Tr->T)	RNN	RF	NN	J48	RNN	J48	RF	NN	RNN	RF	J48	NN	RNN	RF	J48	NN
ant 1.6->1.7	0.064	0.056	**0.068**	0.039	0.054	0.064	**0.072**	0.064	0.089	**0.14**	0.102	0.082	0.165	**0.221**	0.08	0.115
camel 1.4->1.6	**0.348**	0.302	0.334	0.281	**0.377**	0.293	0.334	0.34	**0.442**	0.418	0.361	0.396	**0.51**	0.48	0.371	0.456
jedit 4.2->4.3	**0.365**	0.267	0.348	0.276	**0.355**	0.354	0.333	0.326	0.392	0.475	**0.526**	0.345	0.476	0.583	**0.647**	0.408
log4j 1.1->1.2	**0.737**	0.612	0.619	0.672	**0.721**	0.609	0.558	0.604	**0.707**	0.537	0.589	0.55	**0.709**	0.516	0.601	0.549
lucene 2.2->2.4	0.723	0.725	0.64	**0.741**	0.736	**0.751**	0.722	0.668	**0.763**	0.738	0.76	0.696	**0.795**	0.776	0.774	0.713
poi 2.5->3.0	**0.575**	0.543	0.536	0.492	**0.66**	0.608	0.647	0.631	0.735	**0.736**	0.697	0.706	0.779	**0.78**	0.694	0.741
velocity 1.5->1.6	**0.545**	0.508	0.506	0.435	**0.559**	0.481	0.542	0.526	0.638	**0.679**	0.562	0.602	0.72	**0.753**	0.569	0.685
xalan 2.5->2.6	0.55	**0.603**	0.525	0.417	0.604	0.46	**0.651**	0.588	0.67	**0.7**	0.5	0.652	0.704	**0.721**	0.431	0.684
xerces 1.3->1.4	0.69	0.495	0.485	**0.697**	0.737	**0.739**	0.498	0.507	**0.811**	0.439	0.764	0.53	**0.833**	0.23	0.774	0.52
Avg.	**0.511**	0.457	0.451	0.450	**0.534**	0.484	0.484	0.473	**0.583**	0.540	0.540	0.507	**0.632**	0.562	0.549	0.541
AR	1.7	3.5	4.1	4.0	2.1	3.6	3.1	4.1	2.2	2.6	3.6	4.6	1.7	2.4	4.3	4.3
Win/Tie/Loss	—	5/3/1	6/3/0	7/1/1	—	5/3/1	6/2/1	5/4/0	—	4/1/4	5/3/1	6/3/0	—	4/1/4	8/0/1	7/2/0

with average values supports the better performance of our approach compared with typical techniques.

Table V also lists the Win/Tie/Loss results under each CE_π. In most cases, RNN has more than half (5 out of 9) 'Win' against other techniques. When compared with RF, RNN has better average value, but 4 'Win' and 4 'Loss' under $CE_{0.5}$ and $CE_{1.0}$, which means that RNN is not significantly better than RF in most datasets. This result indicates that RNN may not be suitable to use both code metrics and process metrics.

2) RQ2b: How is the performance of RNN using only code metrics comparing with typical classifiers using both code metrics and process metrics?: Comparing the performance of RNN between Table IV and Table V, it is clear that RNN has better performance using HVSM built with only code metrics. Since process metrics are more difficult to achieve than code metrics, it is meaningful to compare the performance of RNN using only code metrics with typical classifiers using both code and process metrics. In this section, HVSM uses metric set in each version that consists of only the 20 code metrics, which is the same as the experiment in RQ1.

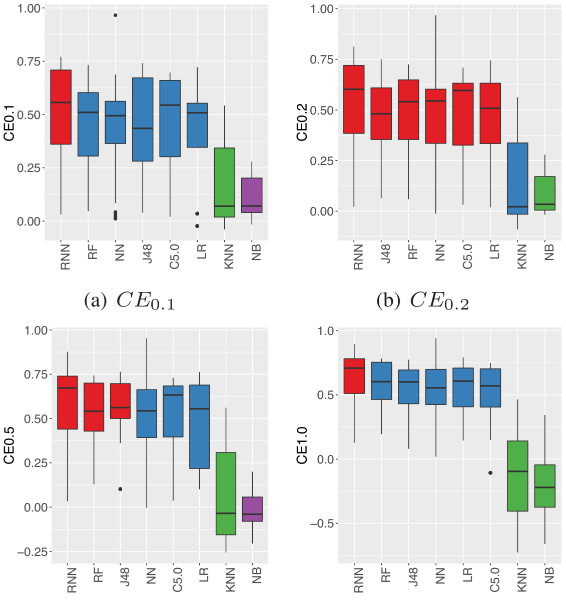
According to the results in RQ2a, RF performs the best among the 7 typical techniques evaluated by each CE_π, so this paper selects RF on behalf of typical techniques to compare with RNN. Table VI shows the detailed results of RNN (using only code metrics) and RF (using both code and process metrics) under each CE_π. From the table we can see that RNN has at least 6 (out of 9) better performance under different CE_π. When it comes to Win/Tie/Loss, RNN has 7 'Win' in $CE_{0.1}$ and $CE_{0.2}$ and 5 'Win' in $CE_{0.5}$ and $CE_{1.0}$, the number of 'Loss' is no more than 2. This result shows that RNN (using only code metrics) outperforms RF (using both code and process metrics) with statistical significance in more than half of the datasets.

Generally speaking, RNN has better performance compared with typical techniques using both code and process metrics. Furthermore, RNN using HVSM built with only code metrics outperforms baseline classifiers trained with both code and process metrics with statistical significance.

VI. DISCUSSION

A. Fairness of Training Data

In our approach, RNN predicts defects in test set using HVSM as its training set. According to the definition of HVSM, it has access to the history of metrics in previous versions, while the training set of typical classifiers does not. The comparison between our approach and other techniques seems unfair. To be noticed that, this paper is proposed to draw attention to using the historical information in previous versions (like HVSM) instead of using data in just one single version as typical techniques do in WPDP. The result



(a) $CE_{0.1}$ (b) $CE_{0.2}$

(c) $CE_{0.5}$ (d) $CE_{1.0}$

Fig. 7. The boxplots of $CE_\pi, \pi = 0.1, 0.2, 0.5, 1.0$ values of RNN and 7 baseline classifiers using both code and process metrics. Different colors represents different Scott-Knott test ranks (from top down, the order is red, blue, green, purple).

TABLE VI
THE CE PERFORMANCE OF RNN USING HVSM BUILT WITH ONLY CODE METRICS COMPARING WITH RF USING BOTH CODE AND PROCESS METRICS. BOLD FONT HIGHLIGHTS THE BETTER PERFORMANCE BETWEEN RNN AND RF.

	$CE_{0.1}$		$CE_{0.2}$		$CE_{0.5}$		$CE_{1.0}$	
Target (Tr->T)	RNN	RF	RNN	RF	RNN	RF	RNN	RF
ant 1.6->1.7	0.023	**0.056**	0.049	**0.072**	0.125	**0.14**	0.209	**0.221**
camel 1.4->1.6	**0.354**	0.302	**0.396**	0.334	**0.462**	0.418	**0.533**	0.48
jedit 4.2->4.3	**0.438**	0.267	**0.455**	0.333	**0.496**	0.475	0.537	**0.583**
log4j 1.1->1.2	**0.757**	0.612	**0.715**	0.558	**0.706**	0.537	**0.719**	0.516
lucene 2.2->2.4	**0.757**	0.725	**0.767**	0.722	**0.781**	0.738	**0.806**	0.776
poi 2.5->3.0	**0.6**	0.543	**0.67**	0.647	**0.737**	0.736	**0.781**	0.78
velocity 1.5->1.6	**0.538**	0.508	**0.571**	0.542	**0.694**	0.679	**0.764**	0.753
xalan 2.5->2.6	0.548	**0.603**	0.612	**0.651**	0.671	**0.7**	0.708	**0.721**
xerces 1.3->1.4	**0.701**	0.495	**0.764**	0.498	**0.824**	0.439	**0.848**	0.23
Avg.	**0.524**	0.457	**0.555**	0.484	**0.611**	0.540	**0.656**	0.562
Win/Tie/Loss	—	7/0/2	—	7/0/2	—	5/2/2	—	5/2/2

in Section V shows that our approach outperforms typical techniques using data in a single previous version in WPDP, and this section will show the result of typical techniques using data in the whole version sequence which is more fair comparing with our approach.

In this section, the training data of typical classifiers are mixed-up files in the whole version sequence. For example, when predicting defects in **ant 1.7**, the training set of typical classifiers consists of all files in the previous versions (**ant 1.3, 1.4, 1.5, 1.6**). Table VII shows a clear review of the results comparing RNN with typical classifiers using mixed-up training data. For training data of typical techniques, the metric set they use still has two types: (1) only code metrics (cm), (2) both code and process metrics (cm+pm). According to the result of RQ2b, RNN uses HVSM built with only code metrics in the following result. From the table we can see that with the whole version sequence included in the training data, typical classifiers still have worse average performance than RNN. This result supports the usefulness of sequential information that HVSM has, which is not included in the simple mixed-up training data used by typical techniques.

TABLE VII
THE AVERAGE CE_π VALUE OF RNN COMPARING WITH TYPICAL CLASSIFIERS USING MIXED-UP TRAINING DATA. THE TECHNIQUES ARE RANKED BY DESCENDING ORDER OF THEIR AVERAGE VALUE OF THE 4 LISTED RESULTS

Technique	$CE_{0.1}$	$CE_{0.2}$	$CE_{0.5}$	$CE_{1.0}$
RNN	0.524 [1]	0.555	0.611	0.656
LR(cm)	0.515	0.545	0.592	0.629
LR(cm+pm)	0.511	0.536	0.584	0.620
NN(cm)	0.495	0.522	0.570	0.612
NN(cm+pm)	0.480	0.503	0.545	0.587
RF(cm+pm)	0.486	0.497	0.543	0.559
J48(cm)	0.486	0.504	0.539	0.537
RF(cm)	0.456	0.474	0.511	0.518
C5.0(cm+pm)	0.414	0.436	0.464	0.428
C5.0(cm)	0.396	0.412	0.430	0.378
J48(cm+pm)	0.384	0.383	0.397	0.383
KNN(cm+pm)	0.148	0.129	0.048	-0.203
NB(cm+pm)	0.140	0.119	0.022	-0.166
NB(cm)	0.127	0.105	0.015	-0.133
KNN(cm)	0.119	0.088	-0.006	-0.255

[1] The result is the average value of each technique's performance in the 9 datasets.

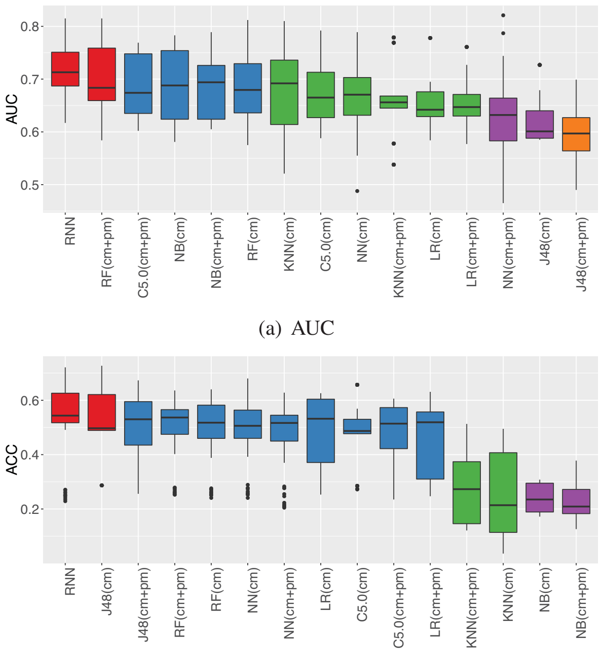
B. Performance Evaluated by Other Measures

In addition to CE_π, this paper also provides performance of RNN and typical techniques evaluated by other 2 measures: AUC and ACC. According to the result of RQ2b, RNN uses HVSM built with code metrics only in the following results.

AUC The *Area Under the ROC Curve (AUC)* [62] is calculated from the *Receiver Operator Characteristic (ROC)* curve. AUC is a threshold-independent performance metric that plots the false positive rate ($\frac{FP}{FP+TN}$) against the true positive rate ($\frac{TP}{TP+FN}$). It measures how a classifier can discriminate between buggy and clean files, and the higher AUC value indicates a better performance. AUC is a widely used evaluation metric that was adopted by many works [9], [10], [28], [35], [40], [41], [46], [50], [54]. Figure 8(a) shows the boxplots of performance of RNN and typical techniques under AUC. RNN is ranked at the top by SK test with the best average AUC over the tested datasets, and has distinct advantages compared with most of the typical classifiers.

ACC In addition to CE, ACC [20], [64] is another commonly used indicators that describe the effort-aware ranking effectiveness of a classification technique. ACC denotes the recall of defective files when using 20% of the entire efforts according to its rank. Figure 8(b) shows the performance of RNN and typical techniques under ACC. It can be seen that RNN is at the top rank under SK test. This further supports the result in Section V.

In summary, evaluated by AUC and ACC, RNN still has better performance compared with other techniques.



(a) AUC

(b) ACC

Fig. 8. The boxplots of AUC and ACC values of 7 baseline classifiers using 2 types of metric sets(code metrics only(cm), code and process metrics(cm+pm)) and RNN. Different colors represents different Scott-Knott test ranks (from top down, the order is red, blue, green, purple, orange).

VII. Threats to Validity

Project selection In this work, we select 9 open-source projects which have been used by prior works. The code metrics are extracted and validated by M. Jureczko [19]. The process metrics can be directly calculated from the source code of each project. Furthermore, it is recommended that more projects and metrics should be tested using our approach, and the result may vary.

Techniques selection Most of the classification techniques that we use are commonly investigated in defect prediction literature. The study on more techniques is required. In addition, the classifier that we use to process HVSM is RNN, which is one of the techniques that can handle sequential data. Replication studies using different classifiers to process HVSM may prove fruitful.

Study replication These typical techniques that we use as baselines are implemented using R packages. The open source implementation of our RNN model can be accessed online (we provide the source code in Matlab, see Section IX for detail). Besides, randomness in some classifiers including RNN will make replication a little different from our result.

VIII. Conclusions

Accurate software defect prediction could help software practitioners allocate test resources to defect-prone modules effectively and efficiently. In the last decades, much effort has been devoted to build accurate defect prediction models, including developing quality defect predictors and modeling techniques. However, current widely used defect predictors such as code metrics and process metrics could not well describe how software modules change over the project evolution, which we believe is important for defect prediction. In order to deal with this problem, we propose to use the Historical Version Sequence of Metrics (HVSM) in continuous software versions as defect predictors. Furthermore, we leverage Recurrent Neural Network (RNN), a popular modeling technique, to take HVSM as the input to build software prediction models.

Our evaluation on 9 open source projects shows that our approach outperforms 7 baseline classifiers. We examine the results mainly in effort-aware scenarios measured by cost-effectiveness(CE). The Win/Tie/Loss evaluation with Wilcoxon signed-rank test and Cliff's delta δ, and Scott-Knott test are also applied to support our results. In most cases, the proposed HVSM-based RNN model has a statistically significantly better effort-aware ranking effectiveness than the commonly used baseline models. In summary, our contributions are as follows:

- **Providing HVSM to highlight the historical trend that files change in version sequence.** HVSM can describe a file's changing information in sequence by joining its metrics in a specific number of continuous historical versions.
- **Leveraging a proper technique, RNN, to handle HVSM in defect prediction.** We apply RNN to HVSM to perform within-project defect prediction. The comparison between RNN and other baseline classifiers shows that our approach has better performance with statistical significance in effort-aware scenarios. In addition, it is suggested to use code metrics to build HVSM that RNN uses in order to achieve better performance.

In the future, we would like to extend our approach to more projects in defect prediction. In addition, we encourage future works to apply different techniques to HVSM or using the information provided by HVSM to improve the performance of typical techniques.

IX. Repeatability

We provide the datasets (including version datasets for typical classifiers and HVSM datasets for our RNN) and Matlab source code that used to construct our RNN model at https://github.com/againcy/2018Saner_RNN.

Acknowledgement

The authors would like to thank the anonymous reviewers for their constructive comments. This work was supported by the National Key Basic Research and Development Program of China (2014CB340702), the National Natural Science Foundation of China (Grant No. 61403187, 61772259, 61472175, 61472178, 61772263), the Natural Science Foundation of Jiangsu Province of China (Grant No. BK20140611).

References

[1] E. Arisholm, L. C. Briand, and E. B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1):2–17, 2010.

[2] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Softw. Eng.*, 28(1):4–17, Jan. 2002.

[3] 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 *ACM Sigsoft Symposium and the European Conference on Foundations of Software Engineering*, pages 4–14, 2011.

[4] G. Canfora, A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella. Defect prediction as a multiobjective optimization problem. *Software Testing Verification and Reliability*, 25(4):426–459, 2015.

[5] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[6] M. D'Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, 2012.

[7] K. O. Elish and M. O. Elish. Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software*, 81(5):649–660, 2008.

[8] L. Erlikh. Leveraging legacy system dollars for e-business. *It Professional*, 2(3):17–23, 2000.

[9] B. Ghotra, S. McIntosh, and A. E. Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE), 16-24 May 2015*, volume vol.1, pages 789–800, 2015.

[10] 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*, page 171, 2012.

[11] A. Graves, A. R. Mohamed, and G. Hinton. Speech Recognition with Deep Recurrent Neural Networks. In *Proceedings of the 2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, number 3, pages 6645–6649, 2013.

[12] X. Gu, H. Zhang, D. Zhang, and S. Kim. Deep API Learning. In *Proceedings - the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2016.

[13] Z. He, F. Peters, T. Menzies, and Y. Yang. Learning from open-source projects: An empirical study on defect prediction. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, Maryland, USA, October 10-11, 2013*, pages 45–54, 2013.

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

[15] B. Henderson-Sellers. *Object-Oriented Metrics, measures of Complexity*. Prentice Hall, 1996.

[16] S. Hosseini, B. Turhan, and M. Mäntylä. Search based training data selection for cross project defect prediction. In *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE 2016, pages 3:1–3:10, New York, NY, USA, 2016. ACM.

[17] T. Jiang, L. Tan, and S. Kim. Personalized defect prediction. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 279–289, 2013.

[18] X. Y. Jing, S. Ying, Z. W. Zhang, S. S. Wu, and J. Liu. Dictionary learning based software defect prediction. In *Proceedings of the 36th International Conference on Software Engineering*, pages 414–423, 2014.

[19] M. Jureczko and L. Madeyski. Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10, pages 9:1–9:10, New York, NY, USA, 2010. ACM.

[20] 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. 39(6):757–773, 2013.

[21] H. Khalid, M. Nagappan, E. Shihab, and A. E. Hassan. Prioritizing the Devices to Test Your App on: A Case Study of Android Game Apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 610–620, 2014.

[22] S. Kim, E. J. Whitehead, and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.

[23] S. Kim, T. Zimmermann, E. J. W. Jr., and A. Zeller. Predicting faults from cached history. In *Proceeding of the 1st Annual India Software Engineering Conference, ISEC 2008, Hyderabad, India, February 19-22, 2008*, pages 15–16, 2008.

[24] E. Kocaguneli, T. Menzies, J. Keung, D. R. Cok, and R. J. Madachy. Active learning and effort estimation: Finding the essential content of software effort estimation data. *IEEE Trans. Software Eng.*, 39(8):1040–1053, 2013.

[25] M. Kuhn. caret: Classification and regression training. Technical report, 2015. http://CRAN.R-project.org/package=caret.

[26] S. J. Lee, K. C. Kim, H. Yoon, and J. W. Cho. Application of fully recurrent neural networks for speech recognition. In *Proceedings of 1991 International Conference on Acoustics, Speech, and Signal Processing*, number 1, pages 77–80 vol.1, 1991.

[27] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In. Micro Interaction Metrics for Defect Prediction. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 311–321, 2011.

[28] S. Lessmann, S. Member, B. Baesens, C. Mues, and S. Pietsch. Benchmarking Classification Models for Software Defect Prediction : A Proposed Framework and Novel Findings. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 34(4):485–496, 2008.

[29] Z. C. Lipton, J. Berkowitz, and C. Elkan. A critical review of recurrent neural networks for sequence learning. *Computer Science*, 2015.

[30] W. Ma, L. Chen, Y. Yang, Y. Zhou, and B. Xu. Empirical analysis of network measures for effort-aware fault-proneness prediction. *Information & Software Technology*, 69:50–70, 2016.

[31] R. Martin. Oo design quality metrics - an analysis of dependencies. In *Proceeding of Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics*, OOPSLA94, 1994.

[32] T. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):20–26, 1976.

[33] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):2146–2189, 2016.

[34] T. Menzies, J. Greenwald, and A. Frank. Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Transactions on Software Engineering*, 33(1):2–14, 2007.

[35] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. Defect prediction from static code features: Current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, 2010.

[36] N. Mittas and L. Angelis. Ranking and clustering software cost estimation models through a multiple comparisons algorithm. *IEEE Transactions on Software Engineering*, 39(4):537–551, 2013.

[37] A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[38] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. pages 181–190, 2008.

[39] N. Nagappan and T. Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *Proceedings - 1st International Symposium on Empirical Software Engineering and Measurement, ESEM 2007*, pages 364–373, 2007.

[40] J. Nam and S. Kim. CLAMI : Defect Prediction on Unlabeled Datasets. In *Proceedings of International Conference on Automated Software 2015*, 2015.

[41] J. Nam and S. Kim. Heterogeneous Defect Prediction. In *Proceeding of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2015)*, pages 508–519, 2015.

[42] J. Nam, S. J. Pan, and S. Kim. Transfer defect learning. In *Proceedings - International Conference on Software Engineering*, pages 382–391, 2013.

[43] A. Panichella, C. V. Alexandru, S. Panichella, A. Bacchelli, and H. C. Gall. A search-based training algorithm for cost-aware defect prediction. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pages 1077–1084, New York, NY, USA, 2016. ACM.

[44] 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 1 – 11, 2008.

[45] 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.

[46] 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 2012*, page 1, 2012.

[47] J. Romano, J. D. Kromrey, and J. Coraggio. Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohen's d indices the most appropriate choices? 2006.

[48] G. Scanniello, C. Gravino, A. Marcus, and T. Menzies. Class level fault prediction using software clustering. In *Preceedings of 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 640–645, 2013.

[49] A. J. Scott and M. Knott. A Cluster Analysis Method for Grouping Means in the Analysis of Variance. *Biometrics*, 30(3):507–512, 1974.

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

[51] Q. Song, J. Ni, and G. Wang. A fast clustering-based feature subset selection algorithm for high-dimensional data. *IEEE Trans. Knowl. Data Eng.*, 25(1):1–14, 2013.

[52] Z. Sun, Q. Song, and X. Zhu. Using coding-based ensemble learning to improve software defect prediction. *IEEE Transactions on Systems Man and Cybernetics Part C*, 42(6):1806–1817, 2012.

[53] M.-H. Tang, M.-H. Kao, and M.-H. Chen. An empirical study on object-oriented metrics. In *Proceedings of the 6th International Symposium on Software Metrics*, METRICS '99, pages 242–, Washington, DC, USA, 1999. IEEE Computer Society.

[54] C. Tantithamthavorn, S. Mcintosh, A. E. Hassan, and K. Matsumoto. Automated Parameter Optimization of Classification Techniques for Defect Prediction Models. In *Proceedings - ICSE 2016*, 2016.

[55] J. Wang, B. Shen, and Y. Chen. Compressed C4.5 models for software defect prediction. In *Proceedings of 12th International Conference on Quality Software, Xi'an, China, August 27-29, 2012*, pages 13–16, 2012.

[56] S. Wang, T. Liu, and L. Tan. Automatically Learning Semantic Features for Defect Prediction. In *Proceedings of International Conference on Software Engineering*, 2016.

[57] T. Wang and W. H. Li. Naive bayes software defect prediction model. In *Computational Intelligence and Software Engineering (CiSE), 2010 International Conference on*, pages 1–4, 2010.

[58] P. J. Werbos. Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1(4):339–356, 1988.

[59] E. J. Weyuker, T. J. Ostrand, and R. M. Bell. Using developer information as a factor for fault prediction. In *Proceedings - ICSE 2007 Workshops: Third International Workshop on Predictor Models in Software Engineering, PROMISE'07*, 2007.

[60] F. Wilcoxon. Individual comparisons of grouped data by ranking methods. *Journal of economic entomology*, 39(6):269, 1946.

[61] R. J. Williams and D. Zipser. Backpropagation. chapter Gradient-based Learning Algorithms for Recurrent Networks and Their Computational Complexity, pages 433–486. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1995.

[62] S. Wu and P. Flach. A scored AUC metric for classifier evaluation and selection. In *Proceedings of the ICML 2005 Workshop on ROC Analysis in Machine Learning*, 2005.

[63] Y. Yang, M. Harman, J. Krinke, S. Islam, and D. Binkley. An Empirical Study on Dependence Clusters for Effort-Aware Fault-Proneness Prediction. *Ase*, pages 296–307, 2016.

[64] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung. Effort-aware just-in-time defect prediction : Simple unsupervised models could be better than supervised models. In *Proceedings of FSE 2016: ACM SIGSOFT International Symposium on the Foundations of Software Engineering*.

[65] F. Zhang, Q. Zheng, Y. Zou, and A. E. Hassan. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 309–320, 2016.