How Well Do Change Sequences Predict Defects? Sequence Learning from Software Changes

Ming Wen, Rongxin Wu, and Shing-Chi Cheung

Abstract—Software defect prediction, which aims to identify defective modules, can assist developers in finding bugs and prioritizing limited quality assurance resources. Various features to build defect prediction models have been proposed and evaluated. Among them, process metrics are one important category. Yet, existing process metrics are mainly encoded manually from change histories and ignore the sequential information arising from the changes during software evolution. Are the change sequences derived from such information useful to characterize buggy program modules? How can we leverage such sequences to build good defect prediction models? Unlike traditional process metrics used for existing defect prediction models, change sequences are mostly vectors of variable length. This makes it difficult to apply such sequences directly in prediction models that are driven by conventional classifiers. To resolve this challenge, we utilize Recurrent Neural Network (RNN), which is a deep learning technique, to encode features from sequence data automatically. In this paper, we propose a novel approach called Fences, which extracts six types of change sequences covering different aspects of software changes via fine-grained change analysis. It approaches defects prediction by mapping it to a sequence labeling problem solvable by RNN. Our evaluations on 10 open source projects show that Fences can predict defects with high performance. In particular, our approach achieves an average F-measure of 0.657, which improves the prediction models built on traditional metrics significantly. The improvements vary from 31.6 to 46.8 percent on average. In terms of AUC, Fences achieves an average value of 0.892, and the improvements over baselines vary from 4.2 to 16.1 percent. Fences also outperforms the state-of-the-art technique which learns semantic features automatically from static code via deep learning.

Index Terms—Defect prediction, process metrics, sequence learning

1 Introduction

Software defect prediction models have been proposed to identify defective program modules automatically [1], [2], [3], [4], [5], [6], [7], [8], [9], [10]. The prediction results enable quality assurance (QA) teams to prioritize limited resources to inspect those modules that are most likely defective. Software metrics play crucial roles in building defect prediction models [11], [12]. Various software metrics have been designed to distinguish buggy modules from clean ones. These metrics mostly fall into two categories: static code metrics and process metrics. Static code metrics, which measure software complexities, are extracted from source files [2], [13], [14], [15], [16]. Representative static code metrics include lines of code [16], class dependencies [14] as well as conceptual cohesions [15] and the count of member functions or inheritances [13]. Process metrics are extracted from software change histories [4], [6], [9], [10], [12], [17] such as code churn [9], [18], ownerships [10], [12], [19] and the complexity of changes [4].

Earlier studies found that process metrics outperform static code metrics in defect prediction [9], [12]. It is because the changes of static code metrics caused by software evolution are often small. The use of static code metrics therefore suffers from stagnation in prediction models [12]. Process metrics can intrinsically avoid this shortcoming, since they are extracted from change histories and evolve over time. A software change can be measured towards different aspects, such as the *meta information* (e.g., time, developer and code churn) and *semantic information* (i.e., semantic operations of the change). A series of changes to a source file may be committed during software evolution, and these committed changes are leveraged to collectively encode process metrics. Common practices to encode these metrics include: counting the number of a certain property like *bug fixing changes* and *distinct developers* [6], [9], [10], taking the maximum/minimum/average of a certain value like *added lines*, *deleted lines* and *change intervals* [9], and computing the entropy to measure some complexity like *code scattering* [12] and the *change complexity* [4].

However, existing process metrics are manually encoded and do not leverage the sequential information of software changes, which contains useful information to distinguish buggy instances from clean ones. Let us illustrate it by an example. Suppose there are plenty of source

• *The authors are with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong, China.*
E-mail: {mwenaa, wurongxin, scc}@cse.ust.hk.

Manuscript received 7 Sept. 2017; revised 5 Sept. 2018; accepted 28 Sept. 2018. Date of publication 16 Oct. 2018; date of current version 12 Nov. 2020. (Corresponding author: Rongxin Wu).
Recommended for acceptance by M. Di Penta.
Digital Object Identifier no. 10.1109/TSE.2018.2876256

TABLE 1
A Motivating Example

Pattern	Semantic Types of the Last Four Changes				Process Metrics	Buggy Ratio		
					COND [17]	xalan 2.5	camel 1.4	poi 2.5
1	♠	⊙	⊙	⊙	1	0.79	0.16	0.50
2	⊙	♠	⊙	⊙	1	0.86	0.14	0.63
3	⊙	⊙	♠	⊙	1	0.91	0.29	0.85
4	⊙	⊙	⊙	♠	1	1.00	0.41	0.92

♠ *denotes semantic operation related to* if-conditions.
⊙ *denotes the other operations.*

files that each received exactly one semantic operation of *if-conditions* during the last four committed changes. The traditional process metric which counts the number of changes that alter *if-conditions* (e.g., COND [17]), will produce the same values as 1 for all these files. However, there can be four possible change sequence patterns where the change with *if-conditions* may take place in any of the four changes as shown in Table 1. To study if the bug characteristics among these four patterns of change sequences are different, we collect all the source files from the projects xalan 2.5, camel 1.4 and poi 2.5, and calculate the buggy ratios of the source files that satisfy any of these four patterns. As we can see from Table 1, the buggy ratios of different patterns vary within the same project. This result motivates us to extract more informative features by considering the sequence data embedded in software changes. Moreover, we also observe that the buggy ratio of the same pattern also varies across projects. This result indicates that change sequence patterns may not be generalized to different projects. This motivates us to encode project-specific sequence features via learning from the projects' change history automatically.

There are no existing studies on whether the sequence data derived from software changes can be used for defect prediction and how the data can be leveraged. Besides, program modules usually receive divergent number of changes, thus resulting in unequal lengths of the change sequences. This makes it difficult to build defect prediction models from change sequences using conventional classification algorithms (e.g., *Naive Bayes*, *Decision Tree* and *Logistic Regression* and so on), which assume features are characterized by vectors of the same length. To address the issue, we propose a novel approach called FENCES, which conducts de**FE**ct predictio**N** via encoding features from **C**hang**E** **S**equences automatically by leveraging Recurrent Neural Networks (RNN). RNN is a deep learning technique that can automatically derive latent features from sequence data and has been widely used to resolve the well-known sequence labeling problems [20], such as handwriting recognition [21], speech recognition [21], sentiment classification [20], [22] and also software engineering tasks [23], [24]. Yet, there can be many types of sequence data derivable from a project but not all of them are meaningful. Referring to existing process metrics, FENCES extracts five different types of sequence data, which describe the meta information of changes, from software change histories. Besides, FENCES designs another type of sequence that encodes the semantic information of changes through fine-grained change analysis [25]. It then applies RNN with the structure of Long Short Term Memory (LSTM) [26], [27], [28] to encode features from these six types of sequences by training them simultaneously.

We compare FENCES with three baselines which leverage traditional process metrics in 10 open source projects, and find that our proposed model which learns features from sequence data automatically outperforms all the baselines significantly. Besides, we also compare FENCES with the state-of-the-art work that also leverages deep learning technique to learn features automatically from static code. The experimental results show that our approach achieves better performance on average.

In summary, this paper makes the following major contributions:

- We propose a novel approach FENCES, which leverages deep learning to encode features automatically from change sequence data. Such sequence data has not yet been used by existing works for software defect prediction. To the best of our knowledge, we are the first to apply RNN to extract features for defect prediction.
- We extract six types of sequences from change histories and build prediction models on them by leveraging RNN. Specifically, we design a type of sequence that tracks all the semantic information of each change via fine-grained change analysis. We find that all these sequences can predict defects with high performance.
- Our evaluation results show that our proposed approach FENCES significantly outperforms all baselines that leverage traditional process and static code metrics. It also outperforms the state-of-the-art technique that leverages deep learning techniques to learn static code metrics automatically.

The rest of the paper is structured as follows. Section 2 presents the background and motivation. Section 3 explains the details of our proposed approach. Section 4 describes our experimental setup, which covers the datasets, baselines, evaluation metrics and parameter settings. Section 5 presents our experimental evaluation. Section 6 discusses related work and Section 7 concludes our work.

2 BACKGROUND AND MOTIVATION

This section introduces the general process of defect prediction and summarizes the process metrics that are widely used and evaluated in existing prediction models. It then discusses the limitations of existing process metrics as well as the challenges of leveraging the sequential information extracted from code repositories for defect predictions. It finally explains how to label sequences using RNN.

2.1 Software Defect Prediction Process

The general process of defect prediction [2], [7], [9], [10], [29], [30] includes four steps. First, program modules (e.g., package, file, class and so on) of a project are collected and labeled. A program module is labeled as *buggy* if any post-release defects have been discovered. It is labeled as *clean* otherwise. Second, each collected program module's features such as process metrics [9], [10] and code metrics [2], [29] are extracted from the code repository or source codes.

TABLE 2
Categories of Process Metrics and Representative Metrics in Each Category

Id	Category	Representative Metrics		Notion
1	Autorship	OWN	[6], [12], [19], [31]	Owners' Contribution
		EXP	[6], [12], [19]	Developers' Experience
		DDEV	[9], [12]	Number of Distinct Developers
		ADEV	[9]	Number of Active Developers
2	Change Type	NREFAC	[9]	Number of Refactoring Changes
		NBF	[6], [8], [9]	Number of Bug Fixing Changes
3	Change Interval	MAXI / MINI / AVGI	[6]	Max / Min / Average Time Gap between Two Sequential Changes
4	Code Churn	ADD	[9], [12]	Lines of Code Added
		DEL	[9], [12]	Lines of Code Deleted
		HCM	[4]	Entropy of Multiple Changes
5	Co-Change	MCO/ ACO	[9]	Max / Average Number of Files Co-Changed in a Change
		NADEV/ NDDEV	[12], [32]	Properties of the Co-Changed Files such as Ownership
6	Semantic Change Type	COND	[17]	Number of Condition Expression Changes
		ELSE	[17]	Number of `Else`-Part Changes

Third, a classification algorithm uses instances, which combine features and labels, to train a prediction model for the project. This step requires a *training dataset*. Finally, the trained model is applied to a new instance and predicts if it is *buggy* or *clean*. The performance of the prediction model is then evaluated by a *testing dataset*. Software features are essential in building effective prediction models [9], [12]. Various software features have been proposed to boost the performance of prediction models, and process metrics are one important category that are widely adopted by existing models [4], [6], [9], [10], [12].

2.2 Change Sequences and Process Metrics

A program module can receive a number of changes during a time period (e.g., the period between two releases). The chronological order of these changes forms a sequence for each program module. Software changes can be characterized in multiple properties such as *code churn* and *authorship*. For each of these properties, we can construct a separate sequence following the same chronological order of software changes. For example, suppose a program module has received five changes during a period, and the developers who committed the changes are Andy, David, Andy, Olivier, David in the chronological order. After indexing Andy, David and Olivier as 1, 2, 3, we can compose a sequence $\langle 1, 2, 1, 3, 2\rangle$ in the aspect of authorship. The process metric `ddev` [9], [10], [27], which counts the number of distinct developers in the change sequence, will produce the feature value of 3 for this example. Besides the property of authorship, process metrics can also be encoded from the change sequences by leveraging other properties of a change. We group existing process metrics [4], [6], [9], [10], [12], [17], [19] into six categories, which separately measure *authorship*, *type*, *time interval*, *code churn*, *coupling* and *semantic* aspect of changes for program modules. Table 2 lists these categories along with some representative metrics for each category. Specifically, Owner's contribution (i.e., *OWN*) is measured by the percentage of the lines authored by the highest contributor of a source file [12]. For example, if a source file of 100 lines is contributed by three different developers, and

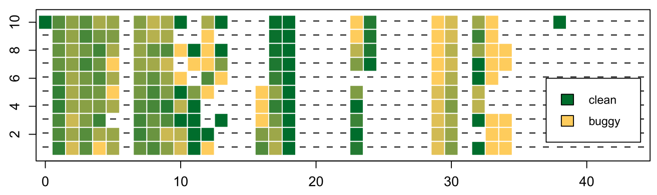


Fig. 1. Buggy ratios of source files touched by each developer in the latest ten changes. the greener, the higher clean ratio, the yellower, the higher buggy ratio. '-' Means no statistical data.

the number of authored lines of these three developers are 50, 30 and 20 respectively, then the metric *OWN* is valued as 0.5 (i.e., 50 / 100). A developer' experience is measured by the percentage of lines he authored in the project at a given point of time. The metric *EXP* measures the geometric mean of the experiences of all the developers that have contributed to a source file [12]. Metric *HCM* measures the distributions of the modified lines for the multiple changes committed within a given period of time [4].

2.3 Motivation of Using Change Sequences

Existing process metrics are handcrafted from change histories, such as picking up the max or computing the entropy. However, these traditional process metrics are incapable of capturing the information embedded in the change sequences as shown by the example in Table 1. Here, we conduct another study to show that the buggy characteristics can be captured by sequential information. As pointed out by the example in the previous subsection, the metric `ddev`, which counts the number of distinct developers in change histories, has widely been used by existing defect prediction models [9], [10], [27]. However, `ddev` neither identifies developers nor captures the evolution of their involvement. In contrast, this data is preserved by the sequence of authorship. Researchers have found that developers have diverse coding styles and experience levels, leading to different defect patterns [7], [19], [31], [33]. Here, we sampled 45 distinct developers from the *camel* project. For each developer, we extracted all the source files (including *buggy* and *clean* ones) that were changed by the developer in the latest n changes of the source files. Then, we calculated the *buggy ratios* using the following equation:

$$buggy\ ratio = \frac{\#buggy\ files}{\#buggy\ files\ +\ \#clean\ files}. \tag{1}$$

The results for the 45 developers are shown in Fig. 1 for $n = 10$. The x-axis represents different developers, and the y-axis represents the latest n changes. The result indicates that the defect patterns for different developers diverge a lot. Some developers write code with high quality (e.g., Developer 18 and 24), while some developers have high chances to introduce bugs (e.g., Developer 33). The probability that the code written by a developer is buggy may change over the time. For example, developer 23 wrote buggy code at early changes but wrote comparatively clean code at later changes. This motivates us to track developers' information in defect prediction. However, this information is lost by simply counting the number of distinct developers (`ddev`) or active developers (`adev`) [12]. The sequence of

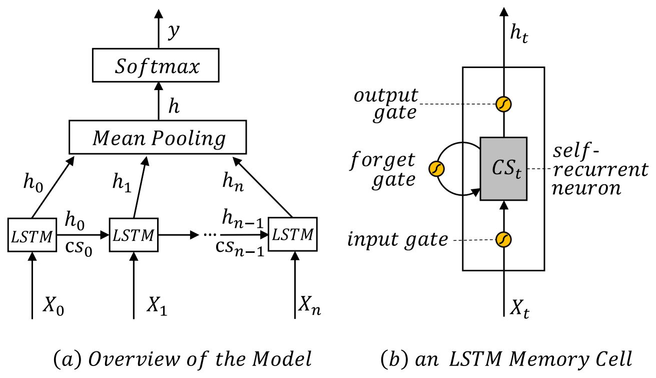


Fig. 2. LSTM Cell and Overview of RNN with LSTM.

authorship, which keeps track of program modules' authorship evolution, can preserve such information.

2.4 Challenges

Although sequences of software changes contain useful information of buggy characteristics, generating effective features from the sequences is challenging. First, these sequences have unequal lengths. Different program modules usually receive different number of changes, resulting in divergent lengths of the corresponding change sequences. This makes it difficult to unify all sequences into simple prediction models. Second, transforming the sequences with uneven lengths into the fixed-length features is impracticable. One possible solution is to treat every possible subsequences of length n as the feature set $\langle ss_1, ss_2, \ldots, ss_n \rangle$. The value of a feature ss_i is then determined by whether ss_i exists in the given sequences. However, whether the best length n in a project can be generalized to other projects is uncertain. Moreover, the number of features could be huge due to the vast amount of all subsequences with length n. Take the change type sequence as an example. If we classify changes into four types (i.e., *corrective*, *perfective*, *adaptive* and *inspection* [34]), there are 4^n possible subsequences, and thus the number of features increases exponentially with n.

To resolve these two challenges, we leverage recurrent neural networks to process arbitrary sequences of inputs and label the sequence automatically by learning the latent features. Specifically, we approach the problem of defect prediction by mapping it to a sequence labeling problem that predicts how likely a source file with a specific sequence is buggy.

2.5 Sequence Labeling with RNN

RNN has been successfully and widely applied in many learning tasks that involve sequence data such as sentence sentiment analysis [20], [22], machine translation [35], [36], speech recognition [21] and software engineering tasks [23], [24]. There are several variants of RNN. We adopt the variant with the Long Short Term Memory architecture [26], [27], [28] since it is shown to be effective for sequence labeling problems such as sentiment analysis [20] for its capability of learning long-term dependencies. Unlike traditional RNN, the LSTM variant introduces a new structure called a *memory cell* as shown in Fig. 2b. An LSTM memory cell comprises four major components: a self-recurrent neuron, an input gate, a forget gate and an output gate. Each cell stores its own state, which is denoted as CS_t. This design determines what information in the sequence to store, forget and output, which allows keeping the long-term temporal contextual information for the sequence. In this way, salient temporal patterns can be remembered over arbitrarily long sequences. We adopt the standard LSTM model [26], [27], [28] in this study. Fig. 2a shows an overview of the model, which is composed of a single LSTM layer followed by a *mean pooling* and a *softmax* layer. The input of the model is a sequence X, each element X_t of which could either be a single feature value or a vector encoding a set of features. Sequence X is then fed to the LSTM layer with each of its element X_t accepted by an LSTM cell. Therefore, the LSTM layer contains the same number of LSTM cells as the length of the input sequence. The model then processes the element X_t from the input sequence one by one. Specifically, the state of a memory cell $\boldsymbol{CS}_t$ is affected by the sequence input at t, the output of the previous cell $\boldsymbol{h}_{t-1}$ and the state of the previous cell $\boldsymbol{CS}_{t-1}$. The output of the LSTM cell at t is computed by the sequence input at t, the state of the cell $\boldsymbol{CS}_t$, and the output of the previous cell $\boldsymbol{h}_{t-1}$ as shown in Equation (2), where $\boldsymbol{b}$ denotes bias vectors and $\boldsymbol{W}$ denotes weights.

$$\begin{aligned} \boldsymbol{CS}_t &= \mathcal{G}(\boldsymbol{W}_1 \cdot [\boldsymbol{h}_{t-1},\ \boldsymbol{X}_t],\ \boldsymbol{CS}_{t-1},\ \boldsymbol{b}_1) \\ \boldsymbol{h}_t &= \mathcal{F}(\boldsymbol{W}_2 \cdot [\boldsymbol{h}_{t-1},\ \boldsymbol{X}_t],\ \boldsymbol{CS}_t,\ \boldsymbol{b}_2). \end{aligned} \tag{2}$$

The computations of function $\mathcal{F}$ and $\mathcal{G}$ comprise several steps, which are determined by the design of the four parts of the LSTM cell [26], [27], [28]. The outputs of all the units $\boldsymbol{h}_t$ are aggregated by a mean pooling layer to $\boldsymbol{h}$. The final output of the model is a vector $\boldsymbol{y}$ indicating the probabilities of the sequence belonging to each class. In our approach, the number of the classes is set to 2, either *buggy* or *clean*. The parameters $\boldsymbol{b}$ and $\boldsymbol{W}$ are learned during training to minimize the error rate of the output $\boldsymbol{y}$.

3 Approach

In this study, we propose a novel approach called Fences, which conducts de**FE**ct predictio**N** via encoding features from **C**hang**E** **S**equences automatically. Fences works at the **source file** level. Fig. 3 shows its overview. Given the change history (denoted as a sequence of changes $S = \langle c_0, c_1, \ldots, c_n \rangle$) of a source file, traditional approaches first encode features from it by heuristics such as taking the *sum*, *max* or computing the *entropy*, and then feed these features to traditional machine learning classifiers (e.g., *Logistic Regression* and *Naive Bayes*) to predict defects. In contrast, our approach first conducts change analysis for each of the changes, and then extracts sequences from the analysis based on a pre-defined property set of the changes. Defects are predicted by leveraging the RNN model as shown in Fig. 2, which is capable of labeling sequences by learning features from these sequences automatically. Both of the traditional process metrics-based defect prediction approaches and Fences leverage the change histories to predict defects. However, the ways to encode features from the change histories differ them from each other significantly.

3.1 Change Analysis

Given a specific version of a project, for each source file m, we extract a sequence of changes that modify m from the

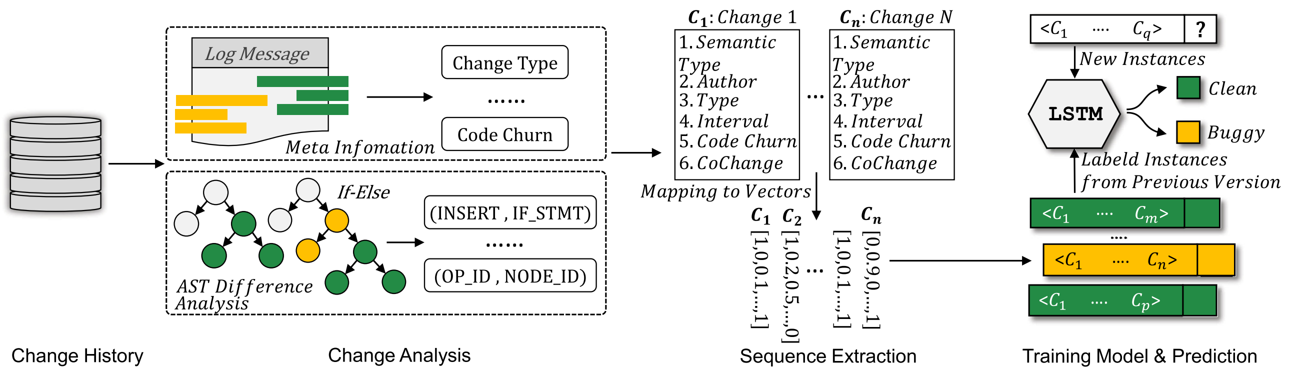


Fig. 3. Overview of FENCES.

prior version to the given version. We then conduct change analysis to obtain the meta and semantic information for each change in the extracted change sequence for each m.

Table 2 in Section 2.2 summarizes the commonly used process metrics [4], [6], [8], [9], [19], [27], [29]. Five out of the six categories are related to the meta information of changes. These metrics concern different aspects of software changes, and are widely leveraged by existing-process metrics based defect prediction models [4], [6], [10], [12]. Therefore, we extract the meta information of each change through analyzing the change logs and the textual contents generated by version control systems (e.g., *Git*). Specifically, we extract five different kinds of meta information: *authorship*, *change type*, *change interval*, *code churn* and *co-change*.

A change may perform multiple semantic operations on the source code, such as altering condition expressions, inserting parameters and so on. The information that characterizes the semantic effects of these operations in a change provides useful hints to buggy suspiciousness as shown by our previous example in Table 1 and related work [17], [37]. Therefore, we conduct fine-grained analysis to extract semantic information for each change. Fine-grained change analysis tracks down to the level of single source code expressions by analyzing the differences between the ASTs (*Abstract Syntax Trees*) of two versions preceding and succeeding to the change. One of the merits of using AST to represent source code is that we can abstract away formatting changes which have no implication of buggy patterns. Our fine-grained change analysis captures the semantic information of changes by means of ⟨OPERATION KIND, AST NODE TYPE⟩ pairs, where each distinct pair corresponds to a *change semantic type*. There are four main kinds of OPERATION KIND, which are `Change`, `Add`, `Delete` and `Move`. The AST NODE TYPE represents the type of the changed AST node. We keep all but those types referring to non-source code changes (e.g., modification of JavaDoc). In total, we keep 84 distinct node types. As a result, our study captures the semantic information of changes by means of 336 $(84 * 4)$ change semantic types. Examples of such semantic types are ⟨CHANGE, INFIX_EXPRESSION⟩, ⟨INSERT, EXPRESSION_STATEMENT⟩ and so on. We leverage the state-of-the-art tool GumTree [25] to analyze the differences between two ASTs.

3.2 Sequences Extraction

Based on the meta and semantic information of changes extracted as described in the previous section, we extract six change sequences of different types for each source file as explained below. To ease explanation, let c_i denote a software change that modified a source file p in its change history, and $S = \langle c_0, c_1, \ldots, c_n \rangle$ denote a chronologically ordered sequence of changes in p's change history.

Change Semantic Types. As described in the previous section, there are 336 distinct change semantic types. A software change may perform multiple semantic operations on source code, and thus contain multiple semantic types. To represent the semantic types of each change c_i, we index each change semantic type first, and then use a vector $\boldsymbol{Sem}_i = \langle m_0, m_1, \ldots, m_{335} \rangle$ in which m_j $(j \leq 335)$ denotes the number of the semantic type with the index j made by change c_i. Therefore, the sequence of change semantic type for p is $S_{sem} = \langle \boldsymbol{Sem}_0, \boldsymbol{Sem}_1, \ldots, \boldsymbol{Sem}_n \rangle$ where $\boldsymbol{Sem}_i$ $(i \leq n)$ is the vector recording the semantic information of change c_i.

Authorship. Authorships of source files have been found to be correlated with software defects [7], [10], [19]. Our study in Section 2 further confirms that buggy characteristics exist in the sequences of authorship. Therefore, we leverage the sequences of authorship for defect prediction. We first index all the distinct developers, and the sequence of authorship is $S_{au} = \langle au_0, au_1, \ldots, au_n \rangle$ where au_i $(i \leq n)$ is the index of the developer who committed the change c_i.

Change Types. Changes can be classified into multiple types. The count of some change types has been leveraged as a feature for defect prediction as listed by the second category in Table 2 [6], [8], [9]. Inspired by this, we propose to leverage the sequence of change types in our defect prediction model. Following the existing heuristic [34] which infers the types from the commit logs, we characterize each software change's type by means of four purposes: *corrective*, *perfective*, *adaptive* and *inspection*. Since a change can serve multiple purposes, we use a vector $\boldsymbol{T}_i = \langle t_0, t_1, t_2, t_3 \rangle$ to record the change type of a change c_i, in which t_0 is equal to 1 if the change serves a *corrective* purpose and 0 otherwise, and similar cases for the other three types. The sequence of change types for p is $S_{type} = \langle \boldsymbol{T}_0, \boldsymbol{T}_1, \ldots, \boldsymbol{T}_n \rangle$ where $\boldsymbol{T}_i$ $(i \leq n)$ records the change types of c_i.

Change Intervals. A change interval is the time gap between two successive changes of the same source file. Program modules that have been changed frequently (i.e., with short change intervals) likely contain faults [6]. Inspired by this, we leverage the sequence of change intervals in our defect prediction model. We extract this information as follows: the sequence of change interval for p is $S_{inter} = \langle in_0, in_1, \ldots, in_n \rangle$

where in_i $(i \leq n-1)$ is the change interval (in days) between the change c_i and c_{i+1}, and in_n is the interval between the last change and the time for prediction. The time for prediction is when developers leverage FENCES to conduct defect prediction. For instance, A common scenario is that a developer leverages FENCES to predict defects at the source file level once a new version is going to be released, then the time for prediction is the release date of the new version.

Code Churn. The process metrics measuring code churns are widely adopted in defect prediction models [4], [6], [9]. *add* and *del* measure the lines of code that are added or deleted in a change, and *add−del* measures the code churn [9]. Our defect prediction model leverages the code churn information using a vector $\boldsymbol{CC} = \langle add, del, add-del\rangle$ to record such information of a change. The sequence of code churn for p is $\boldsymbol{S}_{cc} = \langle \boldsymbol{CC}_0, \boldsymbol{CC}_1, \ldots, \boldsymbol{CC}_n\rangle$ where $\boldsymbol{CC}_i$ $(i \leq n)$ represents the code churn information of change c_i.

Co-Change. Multiple source files may be co-changed in one single change. Co-change source files are more likely to have similar buggy patterns, and therefore co-change information has been leveraged in defect prediction models [1], [8], [9], [12]. We first index all the source files in the software, and then use a vector $\boldsymbol{CO} = \langle co_0, co_1, \ldots, co_m\rangle$ to record the co-change information of a change. Element co_i is equal to 1 if the source file indexed with i is co-changed with the targeting file in the change, and 0 otherwise. The sequence of co-changed source files for p is $\boldsymbol{S}_{co} = \langle \boldsymbol{CO}_0, \boldsymbol{CO}_1, \ldots, \boldsymbol{CO}_n\rangle$ where $\boldsymbol{CO}_i$ $(i \leq n)$ is the vector recording the co-change information of change c_i.

In summary, given a source file p and its change sequence $\boldsymbol{S}$, we extract six sequences, which are different types and are denoted as $\boldsymbol{S}_{sem}$, $\boldsymbol{S}_{au}$, $\boldsymbol{S}_{type}$, $\boldsymbol{S}_{inter}$, $\boldsymbol{S}_{cc}$ and $\boldsymbol{S}_{co}$. All these six sequences have the same length n, which is the number of changes made to p.

3.3 Sequence Discretization and Normalization

The sequences in our defect prediction approach can be classified into two categories based on the data types of elements in a sequence. The first category includes the sequences $\boldsymbol{S}_{au}$, $\boldsymbol{S}_{type}$ and $\boldsymbol{S}_{co}$, and the data type of elements in this category of sequences is *nominal*. The values for nominal data type are provided to distinguish one object from another. For example, for elements in the sequence of authorship, "1" represents a developer A, and "2" represents a developer B. The other category includes the sequences $\boldsymbol{S}_{sem}$, $\boldsymbol{S}_{inter}$ and $\boldsymbol{S}_{cc}$ and the data types of element in this category of sequences is *numerical* type, which count the number of a certain property. The values for numerical data type are integer numbers, which are measurable and meaningful for ordering, e.g., change interval between two sequential changes in terms of the number of days and the number of the occurrences of a certain semantic type. Different nominal data values have distinct meaning, while the magnitude of differences between numerical data values is blurred. For example, *interval* values 36 and 37 are different, but they measure change intervals in close proximity. The wide range of the numerical values hinder the discovery of sequence patterns [38]. To handle the side-effect caused by the numerical values, *discretization* techniques have been proposed and widely used in data mining or machine learning tasks [38], [39]. Studies showed that prediction tasks can benefit from discretization [38]: (1) discretization can lead to improved predictive accuracy; (2) rules with discrete values are normally shorter and more understandable. We adopt a classical approach *equal-frequency* [39] to discrete our numeric sequences. First, it extracts all the elements of a given type of sequences and sorts these elements. Then, given a pre-defined k, it divides the sorted values into k ranges so that each range contains nearly the same number of elements. To determine the discretization value of a new element, it checks which range covers this element and assigns the index of the covered range as the discretization value to it. In this work, we set k to 50 empirically.

Different sequences have different ranges of values, in order to facilitate training all these sequences together, we normalize the values element-wise for all the six types of sequences to the range of 0 to 1. Data normalization is widely used by existing prediction tasks [6], [12], [37], and the classical *min-max* normalization technique is adopted in our approach. Be noted that, for those sequences of nominal types, the normalized values can still be used as token identifiers since unique tokens still have distinct values [37].

3.4 Model Training and Prediction

For each given source file, we extract six sequences of different types. These sequences reflect divergent properties of software changes, and we leverage all of them in our model. In order to enable all these sequences be trained by the LSTM model as introduced in Section 2.5 simultaneously, we combine these sequences by concatenation. Specifically, we create a sequence $\boldsymbol{X} = \langle \boldsymbol{X}_0, \boldsymbol{X}_1, \ldots, \boldsymbol{X}_n\rangle$ for each of the source file, in which $\boldsymbol{X}_i$ is a vector encoding a set of properties by concatenating all the elements from the six types of extracted sequences. Specifically, $\boldsymbol{X}_i = \boldsymbol{Sem}_i \oplus au_i \oplus \boldsymbol{T}_i \oplus in_i \oplus \boldsymbol{CC}_i \oplus \boldsymbol{CO}_i$. $\boldsymbol{X}$ is then fed to the LSTM model as described in Fig. 2a. The number of nodes in the output layer is set to 2, and thus the output $\boldsymbol{y}$ of the model is a vector which contains two values $\langle y_c, y_b\rangle$, in which y_c indicates the suspiciousness of the source file of being clean, and y_b of being buggy. We label the sequence as buggy if y_b is greater than y_c, and the corresponding of the source file is regarded as buggy accordingly. However, the LSTM model needs to be well trained before it can label change sequences correctly. Specifically, we train the model by leveraging the sequences of changes committed in the *older version* as well as the label of the sequences. After the LSTM is trained, we can feed arbitrary sequences (e.g., those extracted from *newer version*) to it and obtain the predicted labels. Defects are predicted based on the obtained labels.

For the two types of nominal change sequences $\boldsymbol{S}_{au}$ and $\boldsymbol{S}_{co}$, the newer version might introduce new developers and new source files, which have not been indexed in the older version. Therefore, we need to use a default value to represent those un-indexed instances that have not been observed from the older version. For example, suppose a source file has received five changes during the development period of the newer version, and the developers who committed the changes are Andy, David, Andy, Olivier, David in the chronological order. If Olivier is newly involved to the project starting from the newer version, and has not been indexed in the older version, we will use *Unknown* to represent it in the sequence $\boldsymbol{S}_{au}$ if the model

TABLE 3
Subjects for Evaluation

Project	Description	Selected Versions	Avg Num of Developers	Number of Changes	Number of Hunks	Number of Files	Buggy Ratios (%)	Ratio of Modified Files (%)
ant	Java based build tool	1.5, 1.6, 1.7	58	989/814/2091	16943/7719/17022	283/351/745	10.9/26.1/22.3	98.9/99.7/99.5
camel	Enterprise integration framework	1.2, 1.4, 1.6	275	251/666/747	4303/6425/6188	608/872/965	35.5/16.5/19.4	97.7/89.6/71.1
jEdit	Text editor designed for programmers	3.2, 4.0, 4.1	37	87/245/299	1957/6270/3763	272/306/321	33.1/24.5/25.0	65.8/65.7/68.9
log4j	Logging library for Java	1.0, 1.1, 1.2	21	30/108/193	494/788/1998	135/109/205	25.2/33.9/92.2	88.1/59.6/83.9
lucene	Text search engine library	2.0, 2.2, 2.4	89	254/204/318	997/1793/2811	195/247/340	46.7/58.3/58.8	62.1/91.5/82.1
xalan	A library for transforming XML files	2.4, 2.5	33	612/563	6123/5741	723/803	15.2/48.2	93.1/94.1
xerces	XML parser	1.2, 1.3	34	437/175	5020/1245	440/453	16.1/15.0	94.1/41.9
ivy	Dependency management library	1.4, 2.0	23	552/348	4413/2492	241/352	6.6/11.4	100/65.9
synapse	Data transport adapters	1.1, 1.2	36	448/59	4421/374	222/256	26.0/33.6	100/42.2
poi	Java library to access Microsoft format files	1.5, 2.5, 3.0	40	132/274/159	1427/2949/3333	237/385/442	59.5/64.4/63.6	99.2/80.3/99.1

learned from an older version is leveraged to predict defects for the newer version. As a result, the authorship sequence for this source file is Andy, David, Andy, Unknown, David. FENCES will index all distinct developers into *Integers*, and "Unknown" will be indexed to the default value 0. However, if the newer version is used to train a prediction model, and the model is leveraged to predict defects for future versions subsequent to the newer version, the developer Olivier will then be indexed and kept in the authorship sequence S_{au}.

4 EXPERIMENTAL DESIGN

4.1 Datasets

To facilitate the replication and verification of our experiments, we use the publicly available benchmark from the PROMISE data repository [40] to evaluate the performance of FENCES. Specifically, we leverage the same benchmark dataset as a recent study [37]. The benchmark contains all the Java open source projects whose version numbers are provided (10 projects with 28 versions) from the PROMISE data repository [40]. The version number is required, since our approach and process metrics need to extract change histories based on that information. We exclude synapse 1.0, since synapse 1.0 is the initial version and has no preceding change histories to construct the process metrics for both our approach and previous works [10], [12]. As a result, 10 Java projects and 27 versions are kept in our evaluation. Table 3 gives the descriptions of these projects. For each of the version, for instance *ant 1.6*, we need to retrieve the development history of this version in order to build our prediction model. For *ant 1.6*, we extract all the changes committed after the release of the previous version (i.e., *ant 1.5*) to the current version. Be noted that the release time of a version is extracted from the corresponding source control tags, which can be obtained via leveraging build-in command such as *"git tag"*. Specifically, *ant 1.5* was released at *2002-07-09 15:05:30* and *ant 1.6* was released at *2003-12-18 12:46:41*. Therefore, we extract all the changes committed between these two timestamps in order to learn sequence features for version *ant 1.6*.

Table 3 also shows the number of developers involved in these versions, the number of changes of the versions, the number of source files and the buggy ratios for each project version. Each subject received 0.1K $\sim$ 1.2K changes during the version on average and contained 150 $\sim$ 1,046 files with an average buggy ratio of 13.4 $\sim$ 49.7 percent. Table 3 also shows the ratio of modified files, from which we can tell that 84.3 percent of the source files have been modified by at least one change during the developmenet of a release version on average. Be noted that source files are labeled as buggy or clean provided by the PROMISE benchmark. Specifically a source file is labeled as buggy if it contains at least one bug after the release of the version. A source file is considered to contain a bug if the fixing commits of the bug modified the source file. A file can be modified at one or more places, and each of them is called a hunk or a delta. A hunk is a group of contiguous lines that are changed along with contextual unchanged lines [41]. Therefore, we also display the information of the number of hunks. Fig. 4 plots the number of changes and hunks made to each source file within a release for the 27 versions of the 10 projects. On average, each source file received 4.75 changes (varying from 1.50 to 13.80 cross different versions and projects) and 12.82 hunks (varying from 3.17 to 58.42 cross different versions and projects). In particular, the number of changes of a source file represents the length of the sequence extracted for the source file. We further investigated the distribution of the lengths of those change sequences as displayed in Fig. 5. In shows that 84.3 percent of the source files have been modified by at least one change on average, and thus the lengths of their sequences are greater than 1. The length of 44.1 percent of the source files is greater than 5, and the length of 23.0 percent of the source files is greater than 10. Such results show that the majority of source files have been modified by substantial changes during a release version, which provide rich information for our approach to learn buggy patterns automatically.

4.2 Baselines

To evaluate the effectiveness of our proposed approach, we compare it with several baselines, including those leveraging traditional features (both *process* and *static code* metrics) and the state-of-the-art work which leverages deep learning to learn semantic features from static code automatically [37]. These baselines are described as follows.

FENCES builds prediction models from change sequences, and traditional process metrics also leverage change histories to encode features. Therefore, we first compare our approach with the models built on different sets of process metrics, which cover the main categories summarized in Section 2.2. The first feature set is proposed by Rahman et al. [12], which covers a set of 14 process metrics. This feature set mainly focuses on the *experiences* of the developers and *co-change* histories of source files. We refer to it as *RH* in our experiment.

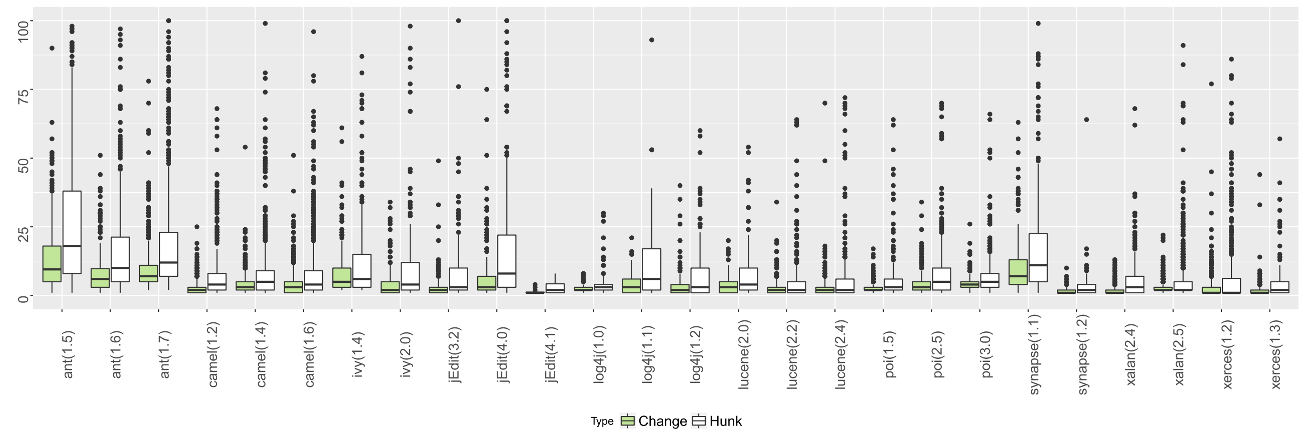


Fig. 4. The number of changes and hunks of a source file within a release.

The second feature set is proposed by Moser et al. [9], which includes 17 process metrics extracted from change histories. These metrics measure the *code churn*, *change types* and *co-change* information of software changes. We refer to this feature set as *MO*. The third feature set is SCC [17]. It mainly focuses on the *semantic types* of software changes. Category 6 in Table 2 shows some examples. We refer to it as *SCC*. In total, there are 34 distinct metrics covering the process metrics popularly used for evaluation by existing work [9], [10], [12], [17], [32]. They are listed in Table 4 with their abbreviated names and short descriptions.

Since static code metrics play a significant role in the domain of defect prediction [2], [13], [14], [16], we also compare our approach with the models built on static code metrics. We select the *CK* metrics set [13], which contains 20 features including lines of code, operand and operator counts, number of methods in a class, the position of a class in inheritance tree, and McCabe complexity measures [42], and so on. The details of these features can be referred to an existing study [43]. These features are available in PROMISE dataset, and are widely leveraged and evaluated by existing work [29], [30], [44], [45]. We also compare with the state-of-the-art work, which also leverages deep learning techniques to **A**utomatically learn **S**emantic **F**eatures from static code [37], and we denote it as *ASF* in this work. ASF works towards different directions compared with our approach since our approach is to encode features automatically from change histories instead of static code.

4.3 Evaluation Method

We construct datasets for different versions of each selected project. To conduct the evaluation, we use the dataset (e.g., change histories) from an *older version* of a project as the *training dataset*, and the dataset from a *newer version* of a project as the *testing dataset*. To examine the performance of the baseline features in order for comparison, we build defect prediction models using three different classifiers: *Logistic Regression* (LR), *Naive Bayes* (NB), *ADTree* (ADT) and *RandomForest* (RF). These four classifiers are widely used by existing work [9], [10], [12], [17], [32], [37]. Recent studies [46], [47] find that the selection of classifiers have less impact on the performance of defect prediction than the selection of software metrics. Therefore, the adoption of the four representative classifiers, which cover a range of different techniques, is capable of evaluating the performance of the process metrics, static code metrics as well as

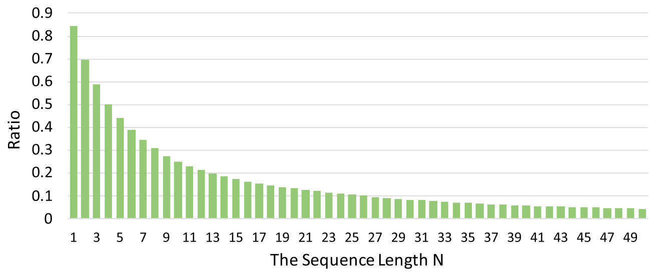


Fig. 5. The ratio of source files whose change sequence's length is greater than N.

TABLE 4
Traditional Process Metrics and the Corresponding Brief Descriptions

Metric Name	Description	Source
COMM	Commit Count	RH
ADEV/DDEV	Active/Distinct Developer Count	RH
ADD/DEL	Normalized Lines Added/Deleted	RH
OWN	Owner's Contributed Lines	RH
MINOR	Minor Contributor Count	RH
SCTR	Changed Code Scattering	RH
NADEV	Neighbor's Active Developers Count	RH
NDDEV	Neighbor's Distinct Developers Count	RH
NCOMM	Neighbor's Commit Count	RH
NSCTR	Neighbor's Change Scattering	RH
OEXP	Owner's Experience	RH
EXP	All Committer's Experience	RH
COMM, DDEV, ADD, DEL are used as defined above		MO
NREFAC	Number of Refactoring Changes	MO
NBF	Number of Bug Fixes Changes	MO
MADD/AADD	Maximum/Average Lines Added	MO
MDEL/ADEL	Maximum/Average Lines Deleted	MO
CC/MCC/ACC	All/Maximum/Average Code Churn	MO
MCO/ACO	Maximum/Average Number of Co-Change	MO
AGE/WAGE	Age/Weighted Age of the File	MO
CDEL	Changes Altering Declaration of a Class	SCC
OSTATE	Insertion or Deletion of Object States	SCC
FUNC	Insertion or Deletion of Functionality	SCC
MDECL	Changes Altering Declaration of Methods	SCC
STMT	Insertion or Deletion of Statements	SCC
COND	Changes Altering Condition Expressions	SCC
ELSE	Insertion or Deletion of *else*-parts	SCC

the automatically learned semantic features. We use the WEKA [48] toolkit for these classifiers. The imbalanced instances of clean and buggy data in our dataset are found to degrade the performance of prediction classifiers [49]. For instance, the buggy ratios are usually less than 50 percent in our dataset as shown in Table 3. Therefore, suggested by an existing study [37], we perform the re-sampling technique, i.e., SMOTE [50], on our traning dataset for both FENCES and conventional classifiers.

4.4 Evaluation Metrics

Precision, Recall and F-measure are the conventional performance measures for defect prediction, which are widely adopted by existing studies [9], [12], [29], [30], [37], [51], and they can be calculated as follows:

$$precision = \frac{true\ positive}{true\ positive + false\ positive} \tag{3}$$

$$recall = \frac{true\ positive}{true\ positive + false\ negative} \tag{4}$$

$$F\text{-}measure = \frac{2 * precision * recall}{precision + recall}. \tag{5}$$

True positive represents the number of instances predicted as buggy that are truly buggy. *False positive* means the number of instances predicted as buggy that are not buggy actually, and *false negative* indicates the number of instances predicted as non-buggy but are actually buggy. For both precision and recall, the higher the value, the better the performance of the prediction model. However, precision and recall have a natural trade-off. Therefore, F-measure combines both of them by the weighted harmonic mean. We, specifically, use F-measure at the default cutoff 0.5 (F_{50}) to evaluate the performance of defect prediction models. The higher F-measure value, the better performance of the corresponding prediction model.

However, these metrics are threshold-dependent and require a particular threshold for evaluation. Therefore, they are sensitive to the thresholds used as cut-off parameters [52]. Since these metrics can be affected by the class imbalance problem [52], we also adopt another threshold-independent, non-parametric metric, i.e., Area Under the ROC Curve (AUC), for evaluation. The Receiver Operating Characteristic (ROC) plots the true positive rate against false positive rate. ROC illustrates the performance of a classifier model when its discrimination threshold is varied. To more comprehensively measure and compare the performance of defect prediction models built on our proposed features and other set of features, we report the AUC, which is a single scalar value that measures the overall performance of a prediction model. AUC is widely adopted in the defect prediction studies [2], [3], [6], [9], [12], [53]. The AUC value of 1 represents a perfect classifier, while a random classifier's AUC value is expected as 0.5. We follow the guideline of Gorunescu [54] to interpret the performance of AUC: 0.90 to 1.00 as excellent prediction, 0.80 to 0.90 as a good prediction, 0.70 to 0.80 as a fair prediction and a poor prediction for values below 0.70.

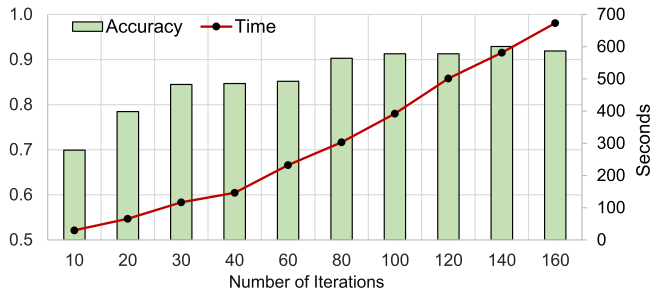


Fig. 6. Accuracy and time of different num of iterations.

4.5 Parameter Settings for LSTM Models

We build our LSTM model based on the well-known framework-*tensorflow* [55] with the structure of *BasicLSTMCell* and the learning algorithm of *AdamOptimizer*. Three major parameters, i.e., *learning rate*, *hidden units* and *iterations*, are needed to be set. To tune these parameters, we use 3 experiment sets, camel ($1.2 \rightarrow 1.4$), poi ($2.5 \rightarrow 3.0$) and log4j ($1.1 \rightarrow 1.2$) since these projects represents the large, median and small sizes of the total subjects in terms of the number of source files. We train our models on a CentOS server with 32 Intel Xeon E5-2450 processors and 196 GB physical memory. Be noted that GPU is not leveraged for the model training in our experiments for fair comparison with baselines in terms of their computation time. For the learning rate, we select 5 values: 0.001, 0.005, 0.01, 0.05 and 0.1. We try different number of hidden units including 8, 16, 64, 128, 256, 512 and 1024. The number of iterations in our study represents how many times we go through all the instances for the training set during learning, and we experiment values varying from 10 to 160. These three parameters are trained together and we find that the model can achieve the best performance with an aggressive learning rate of 0.1 and the number of hidden unit of 128. As for the number of iterations, the training accuracy increases slowly when it reaches 80. Fig. 6 plots the average values of the accuracy and time for different iterations. With the increasing of the iterations, it also takes more time for training. To balance the accuracy and time, we select 100 for the number of iterations in our final model. It takes 400 seconds for training on average as shown in Fig. 6. Moreover, the memory space cost for our proposed approach is less than 1 GB. This indicates that our approach to learning features from sequence data is applicable in practice. The selected parameters setting (i.e., *learning rate* = 0.1, *hidden unit* = 128 and *iterations* = 100) is applied to all the rest of projects.

4.6 Parameter Settings for Conventional Classifiers

As revealed by a recent study, the performance of prediction models built on conventional classifiers can be improved, if their parameters are fine tuned [56]. Therefore, in order to make comparisons fair, we also tune parameters for the baselines, which are those prediction models built on different conventional classifiers: *Logistic Regression*, *Naive Bayes*, *ADTree* and *RandomForest*. Referring to the recent study [56] and the documentations of WEKA, the following parameters of those classifiers can be tuned as shown in Table 5. For *Logistic Regression*, it does not require parameter settings [56]. For *Naive Bayes*, we seleted all the configurable parameters and the candidate values listed in the study [56]

TABLE 5
Classifiers and their Candidate Parameter Values

Classifier	Parameter Name	Candidate Values*
Logistic Regression	**D**egree **I**nteraction	DI={**1**}
Naive Bayes	**L**aplace **C**orrection	LC={**0**}
	Distribution **T**ype	DT={True,**False**}
ADTree	The Number of **B**oosting **I**terations	BI={**10**, 20, 30, 40, 50}
Random Forest	The Number of **C**lassification **T**rees	CT={**10**, 20, 30, 40, 50}

**bold values are the default parameters*

TABLE 6
Selected Parameters for Different Feature Set

Metrics Set	LR	NB	ADT	RF
RH	DI={1}	LC={0}, DT={False}	BI={50}	CT={50}
MO	DI={1}	LC={0}, DT={False}	BI={20}	CT={40}
SCC	DI={1}	LC={0}, DT={True}	BI={20}	CT={10}
All	DI={1}	LC={0}, DT={False}	BI={40}	CT={50}

(i.e., Laplace Correction and Distribution Type). For *ADTree*, the parameter required to be set is the number of boosting iterations with the default value of 10 according to the documentation of WEKA. Referring to the recent study [56], we set the candidates of the parameter of boosting iterations to 10, 20, 30, 40 and 50. For *Random Forest*, we need to set the number of classification trees [56]. The same as the recent study, we set the candidate values to 10, 20, 30, 40 and 50 [56]. Similar to tuning hyper-parameters of FENCES as described in the previous subsection, we select the same set of three projects to tune parameters of conventional classifiers, which are camel ($1.2 \rightarrow 1.4$), poi ($2.5 \rightarrow 3.0$) and log4j ($1.1 \rightarrow 1.2$). Specifically, we run conventional classifiers based on different combinations of candidate values, and then select those parameters that achieve the optimum performance averaged over the three subjects in terms of AUC. We adopt those parameters such selected, as shown in Table 6, in our subsequent experiments. The feature set "All" denoted in Table 6 represents the feature set combining all features together, including both process metrics and static code metrics (see RQ4 in Section 5 for more details).

5 EXPERIMENTAL RESULTS

In this section, we present the evaluation results of our proposed approach. We aim at answering the following five research questions.

RQ1: Does FENCES *outperform those defect prediction models built from traditional process metrics?*

FENCES builds defect prediction models from change sequences via leveraging RNN to encode features from change histories automatically. Therefore, we first compare FENCES with those models built from tradition process metrics. Specifically, we conduct 13 sets of defect prediction experiments as shown in Tables 7, 8, 9 and 10, each of which leverages two versions of the datasets in the same project. The dataset in the older version is used for training while the one in newer version is used for testing. For the three traditional process metrics we choose (RH, MO and SCC as described in Section 4.2), we build models on them using four different machine learning classifiers, i.e., *Logistic Regression*, *Naive Bayes*, *ADTree* and *RandomForest*. Be noted that for those three project versions used for parameter tuning (as described in Sections 4.5 and 4.6), we did not compare their results in this RQ.

Tables 7, 8 and 9 show the results in terms of the *Precision*, *Recall* and *F-measure* respectively for our approach and all baseline models for the 13 sets of experiments. It shows that FENCES achieves high performance in terms of the three evaluation metrics. The precision of our approach varies from 0.365 to 0.828, the recall varies from 0.620 to 0.902 and

TABLE 7
Comparisons between FENCES and Models Built from Traditional Process Metrics (*Precision*)

Project	Tr→Te	Fences	RH				MO				SCC			
			LR	NB	ADT	RF	LR	NB	ADT	RF	LR	NB	ADT	RF
Ant	1.5→1.6	59.4	29.9	68.6	45.5	48.7	28.6	69.6	49.6	53.5	50.9	56.9	48.4	45.8
	1.6→1.7	54.5	75.3	52.0	38.2	44.4	28.3	38.4	25.7	34.7	67.5	60.7	67.3	46.5
camel	1.4→1.6	43.9	36.2	40.7	33.6	43.6	23.5	29.0	26.0	27.3	31.3	31.7	26.6	27.5
jEdit	3.2→4.0	47.2	30.8	40.0	41.0	41.9	40.7	37.3	23.3	38.9	56.1	73.8	46.1	43.4
	4.0→4.1	67.6	49.6	53.1	54.2	54.7	38.4	42.9	43.2	55.2	52.5	52.3	50.7	45.3
log4j	1.0→1.1	72.5	48.6	39.8	35.1	80.0	52.0	44.8	40.4	36.5	35.8	35.2	35.1	80.0
lucene	2.0→2.2	60.3	63.6	83.5	67.9	75.4	62.2	75.0	60.4	59.8	89.0	84.6	77.8	77.8
	2.2→2.4	69.8	61.3	75.3	69.9	71.6	71.4	77.4	70.9	65.7	76.1	90.2	78.8	76.5
xalan	2.4→2.5	53.4	62.0	51.3	55.2	55.0	48.5	49.2	52.8	51.1	63.4	63.4	57.6	57.9
xerces	1.2→1.3	36.5	41.7	55.4	47.1	31.1	18.5	27.4	25.1	32.4	35.1	46.4	36.6	37.1
ivy	1.4→2.0	54.5	15.3	29.0	23.8	29.2	16.3	16.3	24.1	23.0	23.8	39.4	30.7	33.7
synapse	1.1→1.2	45.6	35.5	63.8	45.2	50.3	39.2	50.6	54.1	40.5	49.2	55.2	43.6	50.4
poi	1.5→2.5	82.8	74.3	67.2	49.1	86.9	64.4	66.2	80.0	83.7	90.5	90.5	89.7	88.9
Average		59.8	48.0	55.4	46.6	54.8	40.9	48.0	44.3	46.3	55.5	60.0	53.0	54.7
Improvement of Average			24.5%	8.0%	28.3%	9.0%	46.1%	24.5%	35.0%	29.0%	7.8%	-0.4%	12.8%	9.3%
Median		54.5	48.6	53.1	45.5	50.3	39.2	44.8	43.2	40.5	52.5	56.9	48.4	46.5
Improvement of Median			12.3%	2.7%	20.0%	8.5%	39.2%	21.8%	26.3%	34.6%	4.0%	-4.1%	12.8%	17.2%

All numbers are in ***percentage*** *(%).* ***Tr*** *denotes the training version, and* ***Te*** *denotes the testing version.*

TABLE 8
Comparisons between Fences and Models Built from Traditional Process Metrics (*Recall*)

Project	Tr→Te	Fences	RH				MO				SCC			
			LR	NB	ADT	RF	LR	NB	ADT	RF	LR	NB	ADT	RF
Ant	1.5→1.6	62.0	75.1	23.8	40.6	53.3	80.7	45.7	52.3	61.5	48.1	26.1	39.7	51.8
	1.6→1.7	72.3	26.1	62.7	79.6	73.6	88.4	61.8	89.7	89.3	26.4	31.1	41.5	50.5
camel	1.4→1.6	70.0	40.7	28.5	12.1	8.4	66.0	51.4	50.6	48.6	24.1	12.0	45.9	40.3
jEdit	3.2→4.0	82.2	16.9	95.5	94.1	82.9	62.0	8.5	15.6	37.3	60.6	36.1	68.4	72.6
	4.0→4.1	64.9	78.1	75.6	72.9	74.5	76.9	82.2	70.4	63.5	54.7	50.8	53.4	59.7
log4j	1.0→1.1	78.4	63.1	81.5	89.2	10.8	39.2	38.5	72.4	38.9	85.1	88.9	89.2	10.8
lucene	2.0→2.2	80.4	66.8	18.0	44.8	40.8	48.3	30.2	80.0	65.1	24.6	14.9	38.3	34.3
	2.2→2.4	89.6	67.0	30.1	49.1	47.7	26.6	13.6	43.8	53.3	31.2	15.1	36.3	43.6
xalan	2.4→2.5	90.2	17.5	99.8	74.1	59.5	31.9	99.7	84.5	99.5	18.3	11.6	30.9	32.5
xerces	1.2→1.3	70.5	16.4	16.6	23.4	30.6	85.1	74.1	42.7	35.9	19.7	16.6	22.8	25.3
ivy	1.4→2.0	63.2	59.0	76.8	82.0	82.3	62.3	93.9	78.8	88.0	51.6	65.1	68.0	75.0
synapse	1.1→1.2	87.2	56.6	13.4	44.5	30.2	48.0	11.5	32.4	40.8	9.9	10.5	19.3	26.5
poi	1.5→2.5	90.0	71.4	79.4	23.2	61.5	95.6	46.5	58.4	62.6	45.4	23.5	41.7	48.3
Average		76.8	50.4	54.0	56.1	50.5	62.4	50.6	59.4	60.3	38.4	30.9	45.8	44.0
Improvement of Average			52.6%	42.4%	36.9%	52.3%	23.2%	51.9%	29.4%	27.3%	99.9%	148.3%	67.8%	74.8%
Median		78.4	59.0	62.7	49.1	53.3	62.3	46.5	58.4	61.5	31.2	23.5	41.5	43.6
Improvement of Median			32.8%	25.0%	59.7%	47.0%	25.9%	68.7%	34.1%	27.4%	151.6%	233.7%	88.8%	79.9%

*All numbers are in **percentage** (%). **Tr** denotes the training version, and **Te** denotes the testing version.*

TABLE 9
Comparisons between Fences and Models Built from Traditional Process Metrics (*F-measure*)

Project	Tr→Te	Fences	RH				MO				SCC			
			LR	NB	ADT	RF	LR	NB	ADT	RF	LR	NB	ADT	RF
Ant	1.5→1.6	60.6	42.8	35.3	42.9	50.9	42.2	55.1	50.9	57.2	49.5	35.8	43.6	48.6
	1.6→1.7	62.2	38.7	56.8	51.6	55.4	42.8	47.4	40.0	50.0	37.9	41.1	51.3	48.4
camel	1.4→1.6	54.0	38.3	33.5	17.8	14.1	34.7	37.1	34.4	35.0	27.2	17.5	33.7	32.7
jEdit	3.2→4.0	60.0	21.8	56.4	57.1	55.6	49.1	13.8	18.7	38.1	58.3*	48.5	55.1	54.3
	4.0→4.1	66.2	60.7	62.4	62.1	63.1	51.2	56.4	53.5	59.1	53.6	51.5	52.0	51.5
log4j	1.0→1.1	75.3	54.9	53.4	50.4	19.0	44.7	41.4	51.8	37.7	50.4	50.4	50.4	19.0
lucene	2.0→2.2	68.9	65.1	29.6	54.0	53.0	54.4	43.1	68.8*	62.3	38.6	25.3	51.3	47.6
	2.2→2.4	78.5	64.1	43.0	57.6	57.3	38.7	23.1	54.1	58.9	44.2	25.8	49.7	55.5
xalan	2.4→2.5	67.1	27.3	67.7*	63.3	57.1	38.5	65.9*	65.0*	67.5*	28.4	19.7	40.2	41.6
xerces	1.2→1.3	48.1	23.6	25.6	31.2	30.8	30.4	40.0	31.6	34.1	25.2	24.5	28.1	30.1
ivy	1.4→2.0	58.5	24.3	42.1	36.9	43.1	25.9	27.7	36.9	36.5	32.6	49.1	42.3	46.5
synapse	1.1→1.2	59.9	43.6	22.2	44.9	37.8	43.1	18.8	40.5	40.7	16.5	17.6	26.8	34.7
poi	1.5→2.5	86.2	72.8	72.8	31.5	72.0	77.0	54.6	67.5	71.6	60.5	37.3	56.9	62.6
Average		65.7	44.5	46.2	46.3	46.9	44.1	40.3	47.2	49.9	40.2	34.2	44.7	44.1
Improvement of Average			47.7%	42.1%	41.9%	40.1%	49.0%	62.8%	39.0%	31.6%	63.3%	92.2%	46.8%	48.8%
Median		62.2	42.8	43.0	50.4	53.0	42.8	41.4	50.9	50.0	38.6	35.8	49.7	47.6
Improvement of Median			27.4%	26.9%	8.3%	3.0%	27.3%	31.7%	7.1%	9.2%	41.4%	52.5%	9.8%	14.5%

*All numbers are in **percentage** (%). **Tr** denotes the training version, and **Te** denotes the testing version; Superscript $\star$ denotes the improvement is **not significant**.*

the F-measure varies from 0.481 to 0.862. Especially, for all the 13 sets of experiments, Fences outperforms all the baseline models in terms of F-measure with respect to both the average value or median value. For example, when we use the dataset of *ant* 1.5 for training and the dataset of *ant* 1.6 for testing, the improvement of our approach over the second best approach (the model using *RandomForest* with the feature set *MO*) is 5.9 percent. In terms of F-measure with the average value, The best model using *Logistic Regression* and *Naive Bayes* is built on the feature set *RH*, and the improvements of our approach over them are 47.7 and 42.1 percent For both *ADTree* and *RandomForest*, the best performance is achieved using the feature set *MO*, and the improvements of our approach over them are 39.0 and 31.6 percent respectively.

Since the tools for the baseline approaches are unavailable [9], [12], [17], we implemented them by ourselves and reproduced their results. We observe that the reproduced results of SCC in our experiments differ to the values reported in the prior study [17]. We found that two major factors contribute to the performance differences between our experimental results and prior experimental results.

TABLE 10
Comparisons between FENCES and Models Built from Traditional Process Metrics (*AUC*)

Project	Tr→Te	Fences	RH				MO				SCC			
			LR	NB	ADT	RF	LR	NB	ADT	RF	LR	NB	ADT	RF
Ant	1.5→1.6	88.3	58.0	67.7	84.0	73.4	66.9	77.2	82.8	77.9	73.1	74.9	84.9	68.5
	1.6→1.7	86.8	81.4	81.4	85.1	82.7	73.7	76.1	80.0	83.5	61.2	82.7	88.4	74.0
camel	1.4→1.6	89.7	65.3	65.6	86.3	65.7	59.5	61.2	64.7	61.4	65.6	76.4	75.2	68.2
jEdit	3.2→4.0	91.2	39.4	82.8	97.3	76.7	65.9	37.0	60.1	65.4	77.8	87.8	81.7	76.0
	4.0→4.1	93.4	83.9	84.8	93.0	86.6	73.2	84.0	82.2	81.0	77.1	84.3	88.5	77.6
log4j	1.0→1.1	90.2	77.3	86.0	89.5	78.3	61.2	59.1	85.6	58.7	87.0	89.0	89.5	89.3
lucene	2.0→2.2	92.9	57.5	62.4	78.8	66.2	53.9	64.5	71.0	52.2	82.3	91.3	87.1	77.9
	2.2→2.4	89.4	47.5	64.4	74.0	60.2	50.0	60.3	69.5	57.4	75.5	89.8	84.8	72.3
xalan	2.4→2.5	89.9	53.9	67.6	80.2	64.1	63.5	91.3	82.5	66.5	75.4	82.9	83.3	77.0
xerces	1.2→1.3	84.1	49.5	71.3	88.2	63.0	68.8	74.4	72.3	69.8	67.9	90.8	83.9	75.6
ivy	1.4→2.0	89.9	71.1	84.0	93.1	88.1	72.0	95.2	91.4	85.2	62.2	89.5	87.5	87.0
synapse	1.1→1.2	87.7	55.7	66.4	69.4	65.8	54.4	55.8	78.9	61.0	78.4	80.0	91.5	82.9
poi	1.5→2.5	90.2	71.3	57.5	86.9	83.3	46.8	53.0	77.0	80.0	81.8	93.1	84.4	86.6
Average		89.2	62.4	72.5	85.0	73.4	62.3	68.4	76.8	69.2	74.3	85.6	85.4	77.9
Improvement of Average			42.8%	23.0%	4.8%	21.5%	43.1%	30.4%	16.1%	28.8%	20.1%	4.2%	4.4%	14.4%
Median		89.9	58.0	67.7	86.3	73.4	63.5	64.5	78.9	66.5	75.5	87.8	84.9	77.0
Improvement of Median			55.1%	32.7%	4.2%	22.5%	41.6%	39.4%	13.9%	35.2%	19.0%	2.4%	5.9%	16.8%

All numbers are in ***percentage*** *(%).* ***Tr*** *denotes the training version, and* ***Te*** *denotes the testing version*

First, the two experiments use different sets of subjects whose buggy ratios can greatly affect the prediction performance. The average buggy ratio of our subjects is 26.0 percent, while this ratio is 43.0 percent for the subjects used by SCC [17], which is much higher than our subjects. If the subjects have similar buggy ratio, our implemented baseline approach can achieve similar performance. Take the subject poi 2.5 as an example. The buggy ratio of this subject is 40.7 percent which is close to the one of the subjects used by SCC [17]. The reproduced F-measure is 0.626, which is close to the values reported by SCC [17]. Second, the two experiments have different settings since they have different prediction goals. In the setting of our experiments, we label a source file as defect when it has at least a bug following existing studies [37], [45]. However, in the experimental setting of the original paper that proposed SCC [17], a source file is labeled as buggy only when it has more number of bugs than the median value, which means the labeled buggy-files are highly-defective. The subjects used in their experiments are reported to contain more number of bugs on average than those used in our experiments. For example, a project with 278 source files (e.g., *Compare*) can have 665 reported bugs. Therefore, even though the policy which uses the median number of bugs as the cut-off point to label a source file as buggy or not is adopted, it still achieves an average buggy ratio of 43.0 percent.

On average, FENCES achieves the F-measure as 0.657 and outperforms the baseline models significantly. The improvement of our approach over the baseline models is at least 40.1, 31.6, and 46.8 percent for the traditional process metric sets RH, MO, and SCC respectively in terms of the average value. In terms of the median value, the improvements made by our approach are at least 3.0, 7.1, and 9.8 percent for the traditional process metric sets RH, MO, and SCC respectively. Fig. 7 plots the results in terms of F-measure for the 13 runs of each experiments, we then apply the Mann-Whitney U-Test [57] to test if the improvements of F-measure are significant. The results show that for the majority of cases, the improvements are significant ($p\text{-}value < 0.05$). The few cases (6 out of 156), for which no significant improvements have been observed, are marked with start * in Table 9.

Fig. 10 shows the results evaluated in terms of AUC. It shows that FENCES makes *good* predictions (AUC $\geq$ 0.8) for all the 13 experiments, and 5 of them are *excellent* (AUC $\geq$ 0.9), which shows the effectiveness of our proposed tool FENCES. Averaged over all the 13 experiments, FENCES achieves the optimum performance with an AUC value of 0.892. The improvements of FENCES over the baseline models are significantly, which are at least 4.8, 16.1, and 4.2 percent for the traditional process metric sets RH, MO, and SCC respectively in terms of the average value. In terms of the median value, the improvements made by FENCES are at least 4.2, 13.9, and 2.4 percent for the three sets of traditional process metrics respectively. In terms of AUC with the average value, The best model using all the four different classifiers is built on the feature set *SCC*, and the improvements of our approach over them are 20.1, 4.2, 4.4 and 14.4 percent respectively.

Based on the above results, we can draw the following conclusion:

Change sequences can predict defects. Defect prediction model built from change sequences outperforms those built from traditional process metrics significantly (varying from 31.6 to 46.8 percent on average in terms of F-measure and 4.2 to 16.1 percent on average in terms of AUC).

RQ2: Does FENCES *outperform those defect prediction models built from static code metrics?*

Another important direction of defect prediction is to build models from metrics encoded from static source

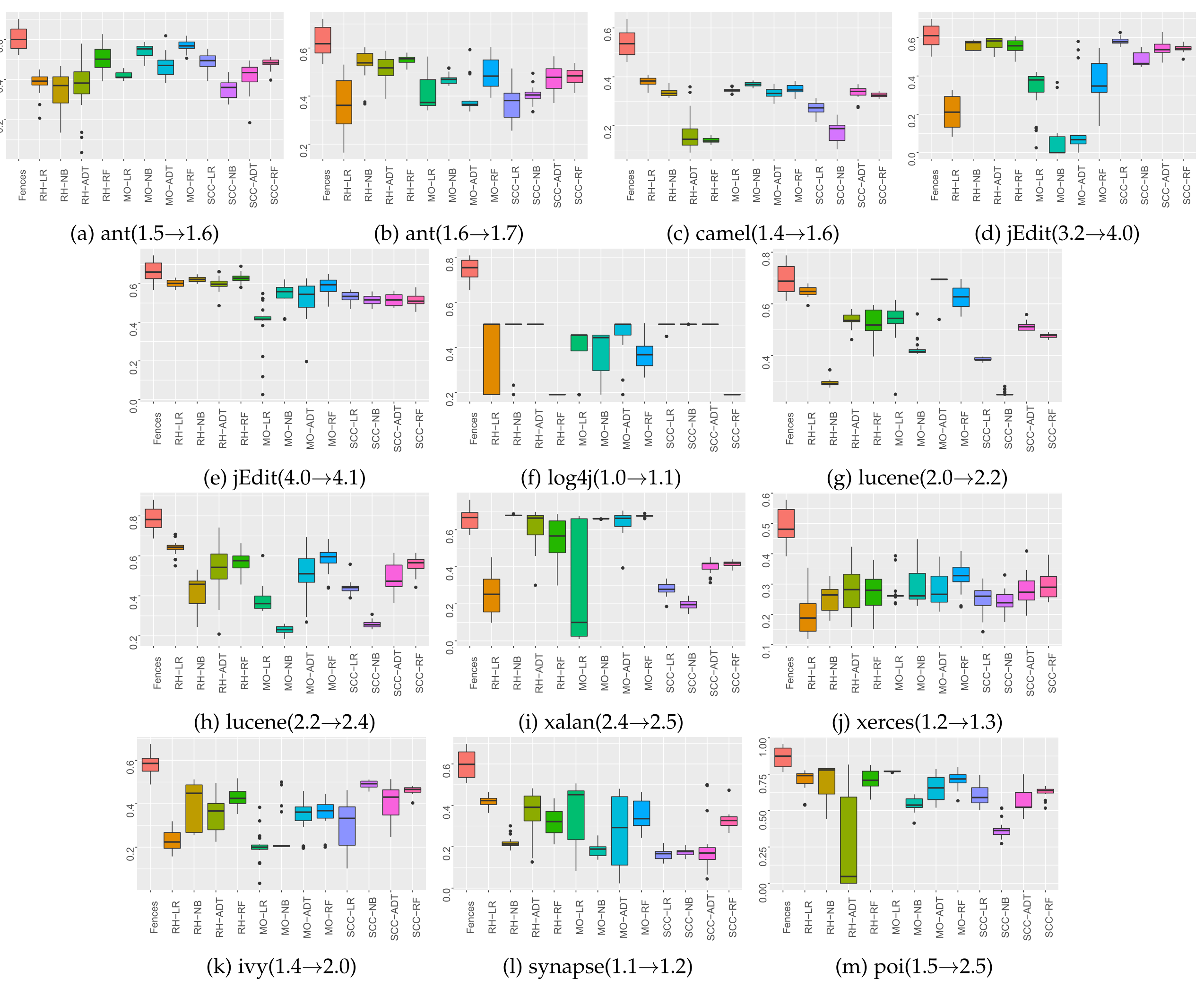


Fig. 7. Boxplots of the results of fences and models built from traditional process metrics (*F-measure*).

code [2], [13], [14], [37], [52]. Especially, a deep learning technique was used to learn semantic features from source code automatically recently [37]. This research question aims to compare our approach with this category of approaches. Specifically, we take the models built on two feature sets CK and ASF (see Section 4.2) as the baselines. CK represents the traditional static code metrics which are encoded from source code by heuristic. ASF was proposed by Wang et al.[37], which is a number of semantic features automatically learned via *Deep Belief Networks*. Table 11 shows the comparison results in terms of F-measure. We directly use the results reported in the related work [37] for the baseline models built on ASF, because we do not have the implementation of the tool to extract ASF and reproducing the results by ourselves may introduce threats to validity. Since the original work did not evaluate ASF using the classifier of *RandomForest* [37], we also do not compare the results under this setting. Besides, the results on the subject *log4j* (1.1 → 1.2) was not reported [37], we then exclude it in this comparison. Be noted that ASF leveraged 5 different subjects to tune parameters and the authors kept the results for these subjects in their evaluation [37]. Therefore, we also run the experiments for those project versions used for parameter tuning (as described in Sections 4.5 and 4.6) and compare their results with ASF. As a result, there are 15 sets of experiments in total. For each set of experiment, FENCES builds prediction models using the same settings as the one used in RQ1.

The results show that FENCES achieves better performance than CK, which encodes features from source code by heuristics, in all the subjects. On average, our approach achieves improvements of 33.7, 46.3 and 46.4 percent over CK for classifiers *Logistic Regression*, *Naive Bayes* and *ADTree* in terms of the average value respectively. In terms of the median value, the improvements made by FENCES are 16.2, 34.6 and 36.7 percent for LR, NB and ADT respectively. When compared with the metrics encoded automatically via deep learning, our approach achieves better performance for most of the cases (12 out of the 15 experiments) as shown in Table 11. Take the subject *poi* (1.5 → 2.5) as an example. The F-measure of our approach is 0.862, while the best F-measure achieved by ASF is 0.770. Averaged on the 15 experiments, our approach achieves better performance, and the improvements are 6.6, 7.1 and 0.5 percent for *Logistic Regression*, *Naive Bayes* and *ADTree* respectively. In terms of the median results, FENCES achieves similar results as ASF. On average, FENCES achieves better performance, and the improvement over static metric baselines ranges from 0.5-46.4 percent. Based on that, we draw the following conclusion:

TABLE 11
F-measure Comparisons between FENCES and Models Built from Static Metrics (Both Traditional Metrics and Metrics Automatically Learned via Deep Learning)

Project	Versions	FENCES	ASF			CK		
	Tr→Te		LR	NB	ADT	LR	NB	ADT
ant	1.5→1.6	60.6	**91.6**	63.0	91.4	50.6	56.0	45.3
	1.6→1.7	62.2	92.5	**96.1**	94.2	54.3	52.2	47.0
camel	1.2→1.4	49.5	59.8	45.9	**78.5**	36.3	30.7	40.2
	1.4→1.6	**54.0**	34.2	48.1	37.4	34.6	26.5	38.3
jEdit	3.2→4.0	**60.0**	55.2	58.3	57.4	54.5	48.6	46.6
	4.0→4.1	**66.2**	62.3	60.9	61.5	56.4	54.8	44.8
log4j	1.0→1.1	**75.3**	68.2	72.5	70.1	53.5	68.9	45.5
lucene	2.0→2.2	**68.9**	63.0	63.2	65.1	59.8	50.0	48.4
	2.2→2.4	**78.5**	62.9	73.8	77.3	69.4	37.8	58.8
xalan	2.4→2.5	**67.1**	56.5	45.2	59.5	54.0	39.8	50.5
xerces	1.2→1.3	**48.1**	47.5	38.0	41.1	26.0	33.3	23.6
ivy	1.4→2.0	**58.5**	34.8	34.4	35.0	24.0	38.9	30.0
synapse	1.1→1.2	**59.9**	42.3	57.9	58.3	31.6	50.8	48.4
poi	1.5→2.5	**86.2**	66.4	77.0	64.0	50.3	32.3	43.5
	2.5→3.0	**80.6**	78.3	77.7	80.3	74.5	46.2	55.6
Average		**65.0**	61.0	60.8	64.7	48.7	44.5	44.4
Median		62.2	62.3	60.9	**64.0**	53.5	46.2	45.5

Tr *denotes the training version and* ***Te*** *for the testing version.*
LR*: Logistic Regression,* ***NB****: Naive Bayes,* ***ADT****: ADTree.*

Defect prediction model built from change sequences outperforms those built from traditional static code metrics significantly, and it achieves better performance than the models built from automatically learned semantic features on average.

RQ3: Does FENCES *outperform those defect prediction models built by combining all metrics?*

It is feasible to combine all traditional metrics to build defect prediction models. As such, we explore whether our approach outperforms the baseline models built from all these metrics (34 distinct process metrics plus 20 distinct static code metrics). As pointed out by earlier studies (e.g., [1]) that too many features can lead to overfitting (caused by high-dimensional problem) and degrade the prediction performance on new instances. Therefore, we conduct feature selection in building the baseline models. Specifically, we adopt Wrapper Subset Evaluation provided by WEKA [48] to select features with the *best first search* algorithm. Fig. 8 shows the values of Precision, Recall, F-measure and AUC for our approach and the four baseline models using *Logistic Regression*, *Naive Bayes*, *ADTree* and *RandomForest* respectively. It shows that FENCES outperforms the models built from all the traditional metrics on average. Specifically, the improvement of precision varies from 2.0 to 28.5 percent, the improvement of recall varies from 31.6 to 51.3 percent, the improvement of F-measure varies from 21.6 to 66.3 percent, and the improvement of AUC varies from 11.8 to 29.6 percent for the four different classifiers on average. We then apply the Mann-Whitney U-Test [57] to test if the improvements are significant, and the *p-value* is shown in Fig. 8 (displayed in blue at the top of each box-plot). It shows that the improvements of FENCES over traditional metrics using different classifiers are significant ($p < 0.05$) in terms of both F-measure and AUC. Based on these results, we can draw the following conclusion:

Defect prediction model built from change sequences outperforms those built from all traditional metrics (both process and static code metrics), and the improvements are significant.

RQ4: How is the performance of each sequence?

FENCES leverages six different types of sequences, and encodes features from these sequences automatically via RNN. These different sequences are S_{sem}, S_{au}, S_{type}, S_{inter}, S_{cc} and S_{co}. This research question aims to evaluate the performance of each sequence in predicting defects. We used one single sequence at a time to build a defect prediction model instead of concatenating them together as described in Section 3.4. Specifically, we let X be S_{sem}, S_{au}, and S_{type} and so on separately. Fig. 9a plots the F-measure of the models built from each type of sequence individually and the model built by concatenating all types of sequences together. The results show that the models based on single sequence data achieves an average of F-measure from 0.498 to 0.554 individually, which outperform most of the defect prediction models built from traditional process metrics. On average, the highest performance is achieved by the sequence of *co-change*, followed by the sequence of *semantic*. However, no significant differences are observed between the performance of each individual sequence by the Mann-Whitney U-Test [57] with $p < 0.05$. Moreover, the model built by concatenating all types of sequence performs significantly better than that on individual sequence and the baseline ($p \leq 0.01$). This suggests that it is better to build defect prediction models based on all types of sequences in practice. We further investigated the effects of discretization, as

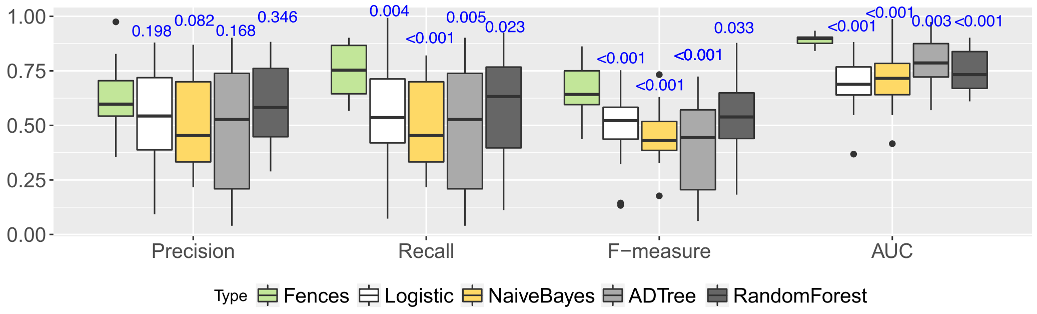
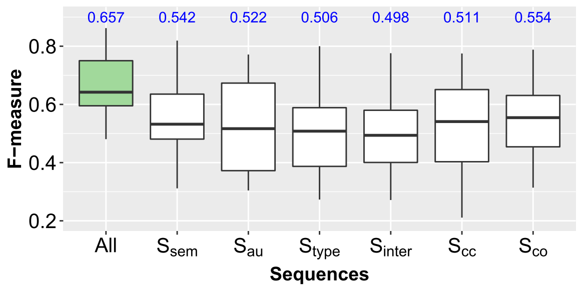
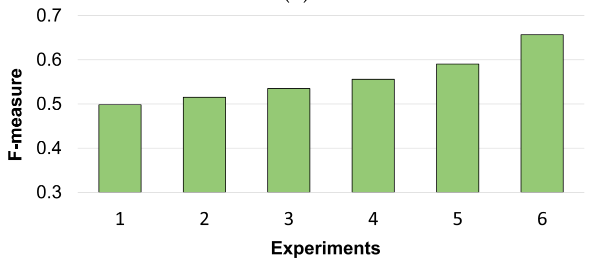


Fig. 8. Comparisons between FENCES and models built from combination of both static code and process metrics.



(a)



(b)

Fig. 9. Evaluations of different sequences. The mean value is displayed in blue at the top in Figure (a).

described in Section 3.3, on the performances of individual sequences. Recall that the process of discretization has been applied to the three types of sequences, S_{sem}, S_{inter} and S_{cc}. We also evaluated the performance of these sequences without applying the process of discretization, and found that the performance can be improved by 6.2, 6.3 and 8.1 percent on average respectively if discretization has been applied. These results show that discretization is important and can improve the performance of FENCES.

To further investigate whether each type of sequence contributes to the overall performance of our approach, we build our models by incrementally adding one type of sequence each time in the order of S_{inter}, S_{type}, S_{cc}, S_{au}, S_{sem} and S_{co}. This order is set based on the performance of each type of sequence. Fig. 9b shows the performance of our models with this incremental setting (the number i in x-axis represents the setting that the ith sequence has been added to the model). As we can see, the F-measure is incrementally improved. The results indicate that each type of sequence indeed contributes to the overall performance of FENCES.

The different sequences we extracted characterize different aspects of software changes, and these aspects actually have also been characterized by traditional process metrics. For example, traditional approaches uses *MCC* (i.e., the maximum code churn) to measure the code churn of software changes, which is measured by sequence S_{cc} in our study. Table 12 shows the corresponding traditional process metrics for each change sequence. In particular, we are interested in whether each change sequence can outperforms the prediction models built from its corresponding traditional process metrics here. Specifically, we build prediction models based on the corresponding traditional process metrics using the classifier *RandomForest*, and then compare its performance with that of each sequence. We choose RandomForest since it achieves the optimum performance for different metric sets on average as shown in Table 9. Fig. 10 shows the experimental results in terms of F-measure, and we can see

TABLE 12
The Corresponding Traditional Features for Each Sequence

Sequence	Corresponding Traditional Features' Names
Authorship (S_{au})	ADEV, DDEV, OWN, MINOR, OEXP, EXP
Co-Change (S_{co})	MCO, ACO
Code Churn (S_{cc})	ADD, DEL, MADD, AADD, MDEL, ADEL, CC, MCC, ACC
Interval (S_{inter})	AGE, WAGE
Semantic (S_{sem})	CDEL, OSTATE, FUNC, MDECL, STMT, COND, ELSE
Type (S_{type})	COMM, NREFAC, NBF

The descriptions of all the traditional features can be referred in Table 4

for all the six change sequences, our approach outperforms the models built from its corresponding process metrics with significant differences (p-$value$ < 0.05). These results demonstrate the effectiveness of sequencing software changes in predicting defects.

Based on the above results, we can draw the following conclusion:

Each type of sequence contributes to the overall performance of our model, and the best performance is achieved when all sequences are combined. Besides, the performance of each change sequence is better than the prediction models built from the corresponding traditional metrics.

RQ5: How is the performance of the prediction models when combining our sequence features with traditional process metrics together?

In RQ4, we investigated the performance of each sequence extracted from the change history. Specifically, we let X be S_{sem}, S_{au} and so on separately and feed it to the LSTM model. The LSTM model is trained from the *older version* and can be applied to the sequences extracted from the *newer version*. The output $\mathbf{y}$ of the LSTM model (i.e., as described in Section 2.5) when feeding a new sequence indicates the buggy proneness of the sequence. In the previous research question, the value $\mathbf{y}$ is leveraged to predict defects directly. Actually, this value can be regarded as a learned latent feature for each sequence as well. For example, the output $\mathbf{y}$ could be a pair of $\langle 0.2, 0.8\rangle$, which indicates the sequence has a probability of 0.8 to be buggy and a probability of 0.2 to be clean. The value of 0.8, which indicates the buggy proneness of the sequence, can be regarded as the encoded feature from the sequence.

The key difference between the feature such encoded and traditional process metrics is depicted in Fig. 11. Actually,

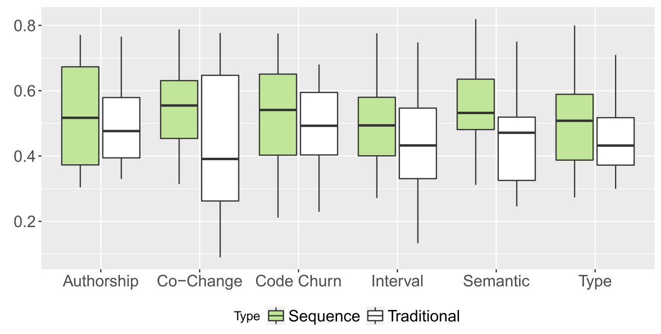


Fig. 10. Comparisons between each sequence and the corresponding traditional features in terms of F-measure.

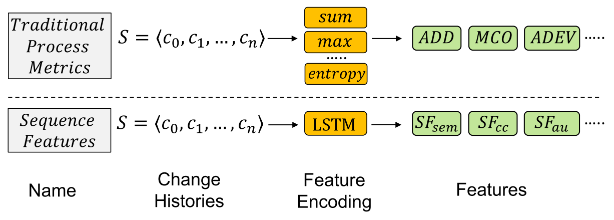


Fig. 11. Encoding features from change histories.

both of the traditional process metrics and sequence features are encoded from the change histories. Traditional process metrics are hard coded using heuristics such as picking up the *average*, *max* or *entropy* of the change sequences as shown in Fig. 11. For instance, *MADD* is the maximum number of lines added among all the changes in sequence S while *AADD* is the average value. However, the effectiveness of such heuristics for different projects with divergent properties cannot be promised. Our intuition is that the buggy patterns for different projects are divergent, and thus using a unified hard coded heuristic to encode features cannot achieve the optimum performance. Instead, we need to learn from change histories to know how to encode better features. Specifically, we encode features automatically using the LSTM model learned from the change histories of *older versions*. The features such encoded are project-specific since the encoding mechanism is learned project by project. Such an encoding mechanism is practical since it only requires the buggy instances and labels of the *older versions* as input, which is the same as that of traditional classifiers. Once the encoder is learned from older versions, it can be applied to encode features on newer versions of the same projects.

In this research question, we are going to investigate whether the sequence features such encoded complement to traditional process metrics, and whether they can improve the performance of prediction models when they are combined with traditional process metrics. Specifically, for the six change sequences S_{inter}, S_{type}, S_{cc}, S_{au}, S_{sem} and S_{co}, we denote the *sequence features* such encoded as SF_{inter}, SF_{type}, SF_{cc}, SF_{au}, SF_{sem} and SF_{co} respectively. We then compare the prediction models built using all the traditional process metrics described in Table 4 with the models built using the combination of our sequence features and those traditional metrics. In total, there are 34 traditional process metrics and 6 sequence features. We then feed these features to traditional classifiers to predict defects. Similarly, feature selection is conducted to alleviate the side-effect of too many features. Specifically, the Wrapper Subset Evaluation provided by WEKA [48] is adopted to select features with the *best first search* algorithm the same as the experimental setting in RQ3.

Fig. 12 shows the performance of AUC and F-measure only using traditional process metrics (denoted as AUC.T or FM.T in the figure) and those using traditional metrics combined with sequence features (marked as AUC.S or FM.S) for each of the four classifiers. In terms of AUC, the improvement achieved by combining the sequence features is 2.9, 5.7, 5.4 and 3.8 percent for *Logistic Regression*, *Naive Bayes*, *ADTree* and *RandomForest* respectively. In terms of F-measure, the improvements achieved by adding the sequence features are 33.6, 22.2, 17.3 and 14.3 percent for *Logistic Regression*, *Naive Bayes*, *ADTree* and *RandomForest* respectively. These

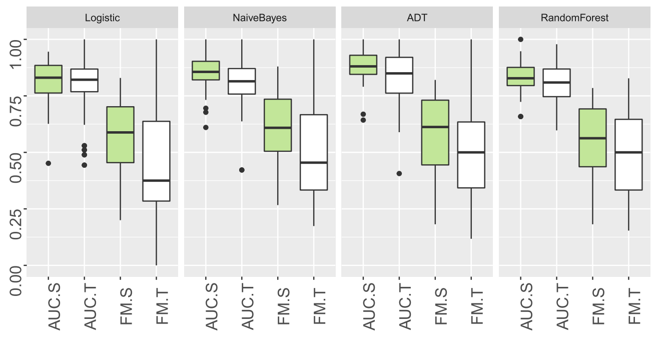


Fig. 12. Prediction results with and without sequence features.

results show that our proposed sequence features encoded via sequence learning can improve the performance of defect prediction models that are built with only traditional process metrics, and the improvements are significant ($p < 0.05$ using the Mann-Whitney U-Test [57]). These results indicate that the sequence features are actually complementary to existing traditional process metrics and can further improve the performance of prediction models built from traditional process metrics.

Based on these results, we can draw the following conclusion:

> Our sequence features are complementary to the traditional process metrics, and the performance of prediction models can be improved significantly when sequence features and process metrics are combined together.

6 Discussions

6.1 Computation Time of Fences

Fences can predict defects with better performance compared with traditional process metrics-based approaches. However, as shown in Fig. 6, it takes a few minutes to train a prediction model. Here, we compare Fences with the baselines in terms of their time efficiency, and Table 13 shows the results. Be noted that the computation time for the baseline is evaluated on the metrics set *RH* with the classification algorithm of *Logistic Regression*. The computation time of other metrics using other classifiers is at the same level as shown in Table 13. The results show that it takes 353.58 (i.e., varying from 90.01 to 1023.48) seconds for Fences to train a defect prediction model on average. Once the model has been trained, it only takes less one second for Fences to predict defects. For prediction models using traditional classification algorithms, it takes less than one second to train a model using the Weka toolkit. Although the training time of Fences is much longer than that of traditional models, it still takes only a few minutes for it to finish training a model. We believe it is acceptable and practical to take a few minutes to train a model for defect prediction.

6.2 Why are Change Sequences Better?

Our evaluation results reveal that the defect prediction models built from change sequence features outperform those built from traditional features significantly. It is because that change sequences contain rich information to distinguish between buggy instances and clean instances as supported by our motivating examples, and thus use deep learning

TABLE 13
Comparison between Fences and Baselines in Terms of Time Efficiency

Project	Tr→Te	Fences		Traditional	
		Training	Testing	Training	Testing
Ant	1.5->1.6	167.14	<0.01	0.43	<0.01
	1.6->1.7	357.25	<0.01	0.11	<0.01
camel	1.2->1.4	311.52	<0.01	0.11	<0.01
	1.4->1.6	279.44	<0.01	0.12	<0.01
jEdit	3.2->4.0	258.02	<0.01	0.10	<0.01
	4.0->4.1	309.14	<0.01	0.09	<0.01
log4j	1.0->1.1	261.90	<0.01	0.06	<0.01
	1.1->1.2	338.14	<0.01	0.06	<0.01
lucene	2.0->2.2	198.59	<0.01	0.08	<0.01
	2.2->2.4	1023.48	<0.01	0.04	<0.01
xalan	2.4->2.5	662.09	<0.01	0.05	<0.01
xerces	1.2->1.3	413.83	<0.01	0.07	<0.01
ivy	1.4->2.0	424.30	<0.01	0.06	<0.01
synapse	1.1->1.2	90.01	<0.01	0.04	<0.01
poi	1.5->2.5	430.32	<0.01	0.06	<0.01
	2.5->3.0	482.12	<0.01	0.07	<0.01

models to distill such information can help build effective models to predict defects. To further understand why change sequences work, we investigate the sequence of S_{inter}, which measures the time (in terms of days or weeks) between two sequential changes. Although S_{inter} does not achieve the optimum performance among all the six types of sequences, it still outperforms most of the models built from traditional metrics.

The metric Interval is the corresponding traditional process metric used in defect prediction models [6]. The intuition behind it is that software modules are more likely to be buggy when they are more frequently changed [6]. The *maximum*, *minimum* and *average* values of Interval among all the changes are encoded as metrics for a source file in existing models [6]. However, source files are *continuously* being changed during a development period, and these three traditional metrics cannot capture the Interval's continuity nature. We illustrate this by conducting the following empirical study.

The unequal lengths of change sequences make it challenging to explore Interval sequences' bug characteristics. To overcome this, we leverage the concept of n-gram to investigate the buggy patterns of subsequences with a fixed length rather than the original sequences. An n-gram is a contiguous sequence of the n items from a given sequence. For instance, given a sequence of $\langle A, B, A, C, B\rangle$, there are three 3-grams for this sequence, which are $\langle A, B, A\rangle$, $\langle B, A, C\rangle$ and $\langle A, C, B\rangle$. The n-gram representation facilitates us to identify the buggy characteristics among the sequences. We then study the buggy ratios (see Equation 1) of the n-gram sequences to investigate which n-grams are more likely to be buggy. We selected the 3-gram 5-gram for investigation. Table 14 shows the top five n-grams which have the highest buggy ratio and clean ratio. The clean ratio is calculated as $1-$*buggy ratio*. The smaller the index value, the shorter the time interval. In these sequence, 0 indicates the two consecutive changes are committed in the same day. For the sequence of 0:0:13, there are four changes, the gap between the 1st and 2nd one is 0, and the gap between

TABLE 14
Top n-Grams of Interval Sequences

rank	3-gram		5-gram	
	buggy	clean	buggy	clean
1	0:0:13	0:2:1	0:0:4:0:0	18:12:17:3:9
2	5:0:0	14:0:10	2:2:0:0:0	15:7:19:10:7
3	1:1:0	3:9:7	0:0:0:3:0	8:9:15:20:10
4	9:0:0	0:3:9	7:0:0:0:0	18:22:7:21:22
5	0:0:7	9:15:20	0:1:8:6:6	19:1:18:28:19

the 2nd and 3rd is also 0, which indicates these three changes are made within the same day. If we look at those top ranked buggy 3-grams, there are consecutive small values in the sequences. For example, 5:0:0 has two consecutive zeros and so does 9:0:0. However, these have not been observed for top-ranked clean instances. Therefore, these results indicate that source files that are frequently changed (i.e., consecutive small values) within a short period of time (i.e., small values such as 0 and 1) are more likely to contain faults. At the same time, the cleanest sequences tend to correlate with a sequence of large values. This indicates that clean files are less frequently changed than the buggy ones. Such information can be captured by our approach while cannot be captured by traditional process metrics.

The limitation of traditional metrics is that it can not differentiate buggy instances from clean ones under certain circumstances. For example, the metric which measures the *minimum* value of Interval [6] will produce the same value of 0 for both the buggy 3-gram and clean 3-gram that are ranked as top 1. However, we can see that the instances of the top-1 buggy 3-gram are *continuously* being frequently changed (e.g., with two consecutive 0 values) during the development period. The same cases for the other top n-grams (e.g., top-2 of 3-gram). Such information is lost if we encode the features by taking the *minimum* values while it is kept in the change sequences. By leveraging the sequence learning technique RNN, Fences is able to capture more buggy characteristics than traditional metrics. This explains why Fences works better than those prediction models built from traditional metrics.

6.3 Sequencing Static Code Metrics

Currently, Fences only leverages the sequences of process metrics to predict defects since process metrics can naturally form a sequence following a chronological order. Focusing on one type of metrics is a common practice in defect prediction [5], [6], [9], [12], [14], [15], [17], [47]. Another important category of metrics in defect prediction is static code metrics, which can also form sequences. For example, we can measure the code metrics for each revision of a source file and take these values as sequences. However, forming static code metrics in this way may still suffer from the stagnation problem [12]. For example, static code metric measuring the number of methods of a source file may not change during software evolution if no changes insert new methods. Forming this static code metric into a sequence will result in sequences with all the same values. This may counteract the effect of using sequences. Nevertheless, transferring static code metrics into sequences is worthing exploring in the future.

7 Threats to Validity

Our experiments are subject to several threats to validity.

Reproducibility of the Results. The reported results of our approach is averaged over 20 runs for each experiment in order to avoid potential randomness of the learning algorithms. For the comparative analysis, we compare our approach with multiple baselines. To avoid introducing threats to validities while reproducing the results, we directly use the reported results in the existing study [37] for AFS and CK since we adopt the same dataset from the PROMISE repository for evaluations. For the baselines of process metrics RH, MO, SCC and all the metrics combined, we reproduce the results by ourselves. We carefully extract all features as described by the original studies [9], [12], [17]. We then leveraged the widely used tool WEKA to reproduce the results. The same, the reported results of baselines are averaged over 20 runs for each experiment in order to avoid the randomness of the re-sampling technique (i.e., as described in Section 4.3) and feature selections (i.e., as described in RQ3 and RQ5).

Subject Selection Bias. A benchmark dataset, which contains 10 popular open source projects and 27 versions, is used in our evaluations. All these projects are collected from the PROMISE repository, and have been widely used and evaluated by existing studies [37], [58]. The selected projects have a large variance in the buggy ratio, the number of changes, and the number of files. However, it is still possible that the subjects are not sufficiently generalizable to represent all the projects. Evaluation on more subjects will increase the confidence for our conclusions. Besides, all the 10 projects used in our evaluation are Java projects, which poses the threat that our results might not be generalizable to projects of other languages. To validate if our findings are generalizable to projects of other languages, experiments on these projects are required. We leave this as our future work.

Data Sufficiency and Overfitting. Deep learning systems require a large number of data for training in order to obtain good models, and thus another potential threat is that our training data might not be sufficient. To avoid such threat, we use the benchmark dataset, which has already been leveraged to train effective deep learning models in predicting defects [37]. Our results agree with the previous results [37], where the models built via deep learning outperform traditional models. This indicates the dataset is adequate to build good deep learning models. Obtaining more labeled data for training is always desired, which is left as our future work.

Overfitting is a threat common to all prediction models. To avoid such threat, we used the basic L2 regularization [59] to the loss function to better avoid overfitting. L2 regularization is a technique to impose constraints on the weights within the nodes of LSTM. In particular, it adds "squared magnitude" of the weights as the penalty term.

8 Related Work

8.1 Software Metrics in Defect Prediction

Software metrics are essential in building effective prediction models [12]. Therefore, many research studies target at designing more effective metrics [2], [4], [9], [12], [13], [14], [33], [37], [60], [61]. Metrics frequently used in defect prediction can be divided into two categories: static code metrics and process metrics. There are also other metrics such as measuring the popularity of source files via investigating the discussions in e-mail archives [62] and metrics derived from organization structures of a company [63].

Static code metrics are mainly extracted from source files which measure properties of source code. The intuition is that more complex program modules are more likely to be error-prone. For example, size metrics such as LOC have been shown to be correlated with defects [16]. McCabe et al. [42] proposed the cyclomatic metric to measure complexity of source code structure, which has been leveraged in defect prediction models [9], [64]. Characteristics of Object-Oriented languages in terms of inheritance, coupling and cohesion, are also designed and widely used in defect prediction studies [2], [13], [15], [33], [65]. Recently, Wang et al. proposed to learn semantic features from static code automatically via deep learning [37].

Process metrics are another category of features used in defect prediction. Process metrics measure the histories of change activities for program modules. The intuition of using process metrics in defect prediction is that bugs are caused by changes [2], [9], [41]. Different categories of process metrics describe different properties of software changes, such as code churn [9], [12], co-change information [8], [9], [12], [66], authorship [9], [10], [19], [31], and behavioral interaction [33]. These metrics are shown to be effective in defect prediction. Ratzinger et al. proposed to leverage relative values of process metrics to predict defects [61]. For instance, they compute the ratio of the number of bug fixes changes within a period of time with respect to the total number of changes. Besides, some studies showed that process metrics perform better than code metrics in defect prediction [9], [12]. However, all these process metrics compress features into a single metric while our approach leverages sequences of these features to predict defects.

Most of existing defect prediction models leverage static code metrics and process metrics as features and feed these features to conventional classifiers for prediction. Different from these studies, our approach builds defect prediction models from change sequences and leverages RNN to learn features automatically.

8.2 Defect Prediction Approaches

Despite the different categories of metrics as introduced above, many machine learning algorithms have been used in defect prediction. The most common used machines learning algorithms are supervised learning algorithms, including Support Vector Machines [67], Naive Bayes [9], [68], Logistic Regression [6], [9], [12], Decision Tree [9], [60], [69], [70] and Dictionary Learning [44]. Although there are various kinds of supervised learning algorithms, Arisholm et al. [11] found that the performance of prediction depends more on the categories of metrics than the learning algorithms.

To deal with the situations that historical training data is not sufficient for supervised learning, different learning techniques have been introduced. We summarize these techniques into three major categories. The first category is active learning. It interactively queries the developers to obtain the labels for a small sample set of new instances and uses the sample data for training [71]. The second category is cross-project defect prediction [30], [58]. The basic idea of this category of approach is to transfer instances with labels from other projects for the target project [30], [58]. The third category of learning techniques is unsupervised learning. This category of learning techniques enjoys the advantage that requires no training datasets [51], [72], [73]. The main process for unsupervised defect prediction is to group software instances into clusters, and then label each cluster as buggy or clean. Zhong et al. [73] proposed to use k-means and *neural-gas* clustering in defect prediction. However, their approach requires experts to label whether each cluster is buggy manually. Nam et al. [51] proposed to label the clusters using the thresholds set on certain selected metrics. Zhang et al. [72] proposed to use a connectivity-based unsupervised classifier for defect prediction. Their results show that the connections between buggy and clean instances are weaker than that among buggy instances and that among clean ones.

Recently, various deep learning algorithms have been leveraged in defect prediction [37], [74]. Yang et al. [74] proposed to use Deep Belief Networks (DBN) to learn new features from existing manually encoded features. Wang et al. [37] then proposed to apply DBN to learn semantic features automatically. Their approach first parses source code using Abstract Syntax Tree (AST) into a vector of AST nodes, and then feeds the vectors of AST nodes to DBN after encoding to generate features automatically. Their results show that the automatically learned semantic features can improve the performance of defect prediction. These approaches are orthogonal to our approach, which encodes features automatically from change histories instead of static code. To the best of our knowledge, we are the first to use RNN to extract features from change sequences.

9 Conclusion

In this paper, we study whether sequences derived from change histories can predict defects. However, change sequences are not directly applicable to prediction models driven by conventional classifiers due to the unequal lengths of sequences. Thus, we propose a novel approach called Fences, which leverages RNN, a deep learning technique for labeling sequences, to encode features from change sequences. Fences extracts six different types of sequences from change histories covering the key properties of a change. It then performs defect prediction by mapping it to a sequence labeling problem that aims at predicting whether a source file with a specific sequence is buggy or clean. Our evaluation results on 10 open source projects show that change sequences provide valuable information for defect prediction. Specifically, our approach outperforms those prediction models built on traditional metrics significantly, with the average improvement varying from 31.6 to 46.8 percent in terms of F-measure and 4.2 to 16.1 percent in terms of AUC. In the future, we plan to extend our study to more projects, including commercial software systems.

Acknowledgments

This work is supported by the Hong Kong RGC/GRF grant 16202917 and 2018 MSRA collaborative research fund.

References

[1] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *Proc. 7th IEEE Working Conf. Mining Softw. Repositories*, 2010, pp. 31–41.

[2] M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: A benchmark and an extensive comparison," *Empirical Softw. Eng.*, vol. 17, pp. 531–577, 2012.

[3] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall, "Method-level bug prediction," in *Proc. ACM-IEEE Int. Symp. Empirical Softw. Eng. Meas.*, 2012, pp. 171–180.

[4] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proc. 31st Int. Conf. Softw. Eng.*, 2009, pp. 78–88.

[5] A. E. Hassan and R. C. Holt, "The top ten list: Dynamic fault prediction," in *Proc. 21st IEEE Int. Conf. Softw. Maintenance*, 2005, pp. 263–272.

[6] H. Hata, O. Mizuno, and T. Kikuno, "Bug prediction based on fine-grained module histories," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 200–210.

[7] T. Jiang, L. Tan, and S. Kim, "Personalized defect prediction," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2013, pp. 279–289.

[8] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 489–498.

[9] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 181–190.

[10] F. Rahman and P. Devanbu, "Ownership, experience and defects: a fine-grained study of authorship," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 491–500.

[11] E. Arisholm, L. C. Briand, and E. B. Johannessen, "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," *J. Syst. Softw.*, vol. 83, no. 1, pp. 2–17, 2010.

[12] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 432–441.

[13] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–793, Jun. 1994.

[14] R. Harrison, S. J. Counsell, and R. V. Nithi, "An evaluation of the mood set of object-oriented software metrics," *IEEE Trans. Softw. Eng.*, vol. 24, no. 6, pp. 491–496, Jun. 1998.

[15] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 287–300, Mar./Apr. 2008.

[16] H. Zhang, "An investigation of the relationships between lines of code and defects," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2009, pp. 274–283.

[17] E. Giger, M. Pinzger, and H. C. Gall, "Comparing fine-grained source code changes and code churn for bug prediction," in *Proc. 8th Working Conf. Mining Softw. Repositories*, 2011, pp. 83–92.

[18] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proc. 27th Int. Conf. Softw. Eng.*, 2005, pp. 284–292.

[19] 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 *Proc. 19th ACM SIGSOFT Symp/13th Eur. Conf. Found. Softw. Eng.*, 2011, pp. 4–14.

[20] A. Graves, *Supervised Sequence Labelling*. New York, NY, USA: Springer, 2012.

[21] A. Graves and N. Jaitly, "Towards end-to-end speech recognition with recurrent neural networks," in *Proc. 31st Int. Conf. Int. Conf. Mach. Learn.*, 2014, pp. 1764–1772.

[22] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proc. ACL*, 2011, pp. 142–150.

[23] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2014, pp. 419–428.

[24] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Proc. 12th Working Conf. Mining Softw. Repositories*, 2015, pp. 334–345.

[25] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, “Fine-grained and accurate source code differencing,” in *Proc. 29th ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2014, pp. 313–324.

[26] F. A. Gers, J. Schmidhuber, and F. Cummins, “Learning to forget: Continual prediction with lstm,” *Neural Comput.*, 2000, pp. 2451–2471.

[27] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, “Predicting fault incidence using software change history,” *IEEE Trans. Softw. Eng.*, vol. 26, no. 7, pp. 653–661, Jul. 2000.

[28] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Comput.*, no. 8, pp. 1735–1780, 1997.

[29] 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 Softw. Eng.*, vol. 17, pp. 375–407, 2010.

[30] J. Nam, S. J. Pan, and S. Kim, “Transfer defect learning,” in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 382–391.

[31] E. Giger, M. Pinzger, and H. Gall, “Using the gini coefficient for bug prediction in eclipse,” in *Proc. 12th Int. Workshop Principles Softw. Evol./7th Annu. ERCIM Workshop Softw. Evol.*, 2011, pp. 51–55.

[32] F. Rahman, D. Posnett, I. Herraiz, and P. Devanbu, “Sample size vs. bias in defect prediction,” in *Proc. 9th Joint Meet. Found. Softw. Eng.*, 2013, pp. 147–157.

[33] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In, “Micro interaction metrics for defect prediction,” in *Proc. 19th ACM SIGSOFT Symp./13th Eur. Conf. Found. Softw. Eng.*, 2011, pp. 311–321.

[34] R. Purushothaman and D. E. Perry, “Toward understanding the rhetoric of small source code changes,” *IEEE Trans. Softw. Eng.*, vol. 31, no. 6, pp. 511–526, Jun. 2005.

[35] S. Liu, N. Yang, M. Li, and M. Zhou, “A recursive recurrent neural network for statistical machine translation,” in *Proc. Assoc. Comput. Linguistics*, 2014, pp. 1491–1500.

[36] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to sequence learning with neural networks,” in *Proc. NIPS*, 2014, pp. 3104–3112.

[37] S. Wang, T. Liu, and L. Tan, “Automatically learning semantic features for defect prediction,” in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 297–308.

[38] H. Liu, F. Hussain, C. L. Tan, and M. Dash, “Discretization: An enabling technique,” *Data Mining Knowl. Discovery*, vol. 6, no. 4, pp. 393–423, 2002.

[39] J. Dougherty, R. Kohavi, M. Sahami, et al., “Supervised and unsupervised discretization of continuous features,” in *Proc. 12th Int. Conf. Int. Conf. Mach. Learn.*, 1995, pp. 194–202.

[40] 2016. [Online]. Available: http://openscience.us/repo/defect/ck, Accessed on: Aug. 8, 2016.

[41] M. Wen, R. Wu, and S.-C. Cheung, “Locus: Locating bugs from software changes,” in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2016, pp. 262–273.

[42] T. J. McCabe, “A complexity measure,” *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Jul. 1976.

[43] Z. He, F. Peters, T. Menzies, and Y. Yang, “Learning from open-source projects: An empirical study on defect prediction,” in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas.*, 2013, pp. 45–54.

[44] X.-Y. Jing, S. Ying, Z.-W. Zhang, S.-S. Wu, and J. Liu, “Dictionary learning based software defect prediction,” in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 414–423.

[45] T. Zimmermann, R. Premraj, and A. Zeller, “Predicting defects for eclipse,” in *Proc. 3rd Int. Workshop Predictor Models Softw. Eng.*, 2007, Art. no. 9.

[46] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, “A systematic literature review on fault prediction performance in software engineering,” *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, pp. 1276–1304, Nov./Dec. 2012.

[47] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič, “Software fault prediction metrics: A systematic literature review,” *Inf. Softw. Technol.*, vol. 55, no. 8, pp. 1397–1418, 2013.

[48] 2016. [Online]. Available: http://www.cs.waikato.ac.nz/ml/weka/, Accessed on: Aug. 8, 2016.

[49] Y. Kamei, A. Monden, S. Matsumoto, T. Kakimoto, and K.-i. Matsumoto, “The effects of over and under sampling on fault-prone module detection,” in *Proc. 1st Int. Symp. Empirical Softw. Eng. Meas.*, 2007, pp. 196–204.

[50] M. Tan, L. Tan, S. Dara, and C. Mayeux, “Online defect prediction for imbalanced data,” in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, 2015, pp. 99–108.

[51] J. Nam and S. Kim, “Clami: Defect prediction on unlabeled datasets,” in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2015, pp. 452–463.

[52] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, “Towards building a universal defect prediction model with rank transformed predictors,” *Empirical Softw. Eng.*, vol. 21, pp. 2107–2145, 2015.

[53] F. Rahman, D. Posnett, and P. Devanbu, “Recalling the imprecision of cross-project defect prediction,” in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, Art. no. 61.

[54] F. Gorunescu, *Data Mining: Concepts, Models and Techniques*, vol. 12. New York, NY, USA: Springer, 2011.

[55] (2016). [Online]. Available: https://www.tensorflow.org/, Accessed on: Aug. 8, 2016.

[56] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, “Automated parameter optimization of classification techniques for defect prediction models,” in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng.*, 2016, pp. 321–332.

[57] H. B. Mann and D. R. Whitney, “On a test of whether one of two random variables is stochastically larger than the other,” *Ann. Math. Statist.*, pp. 50–60, Mar. 1947.

[58] G. Canfora, A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, “Multi-objective cross-project defect prediction,” in *Proc. IEEE 6th Int. Conf. Softw. Testing Verification Validation*, 2013, pp. 252–261.

[59] A. Y. Ng, “Feature selection, l 1 vs. l 2 regularization, and rotational invariance,” in *Proc. 21st Int. Conf. Mach. Learn.*, 2004, Art. no. 78.

[60] A. Bernstein, J. Ekanayake, and M. Pinzger, “Improving defect prediction using temporal features and non linear models,” in *Proc. 9th Int. Workshop Principles Softw. Evol.: Conjunction 6th ESEC/FSE Joint Meet.*, 2007, pp. 11–18.

[61] J. Ratzinger, H. Gall, and M. Pinzger, “Quality assessment based on attribute series of software evolution,” in *Proc. 14th Working Conf. Reverse Eng.*, 2007, pp. 80–89.

[62] A. Bacchelli, M. DAmbros, and M. Lanza, “Are popular classes more defect prone?” in *Proc. Int. Conf. Fundam. Approaches Softw. Eng.*, 2010, pp. 59–73.

[63] N. Nagappan, B. Murphy, and V. Basili, “The influence of organizational structure on software quality,” in *Proc. ACM/IEEE 30th Int. Conf. Softw. Eng.*, 2008, pp. 521–530.

[64] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, “Benchmarking classification models for software defect prediction: A proposed framework and novel findings,” *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 485–496, Jul./Aug. 2008.

[65] M. Jureczko and D. Spinellis, “Using object-oriented design metrics to predict software defects,” in *Proc. Models Methodology Syst. Dependability*, 2010, pp. 69–81.

[66] M. D’Ambros, M. Lanza, and R. Robbes, “On the relationship between change coupling and software defects,” in *Proc. 16th Working Conf. Reverse Eng.*, 2009, pp. 135–144.

[67] K. O. Elish and M. O. Elish, “Predicting defect-prone software modules using support vector machines,” *J. Syst. Softw.*, vol. 81, pp. 649–660, 2008.

[68] W. Tao and L. Wei-Hua, “Naive bayes software defect prediction model,” in *Proc. Int. Conf. Comput. Intell. Softw. Eng.*, 2010, pp. 1–4.

[69] T. A. M. Khoshgoftaar and N. Seliya, “Tree-based software quality estimation models for fault prediction,” in *Proc. 8th IEEE Symp. Softw. Metrics*, 2002, pp. 203–214.

[70] P. Knab, M. Pinzger, and A. Bernstein, “Predicting defect densities in source code files with decision tree learners,” in *Proc. Int. Workshop Mining Softw. Repositories*, 2006, pp. 119–125.

[71] M. Li, H. Zhang, R. Wu, and Z.-H. Zhou, “Sample-based software defect prediction with active and semi-supervised learning,” *Automated Softw. Eng.*, vol. 19, pp. 201–230, 2012.

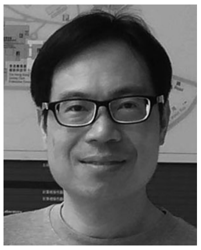
[72] F. Zhang, Q. Zheng, Y. Zou, and A. E. Hassan, “Cross-project defect prediction using a connectivity-based unsupervised classifier,” in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 309–320.

[73] S. Zhong, T. M. Khoshgoftaar, and N. Seliya, “Unsupervised learning for expert-based software quality estimation.” in *Proc. 7th IEEE Int. Symp. High Assurance Syst. Eng.*, 2004, pp. 149–155.

[74] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, “Deep learning for just-in-time defect prediction,” in *Proc. IEEE Int. Conf. Softw. Quality Rel. Secur.*, 2015, pp. 17–26.

Ming Wen received the BSc degree from the College of Computer Science and Technology, Zhejiang University (ZJU), in 2014. He is currently working toward the PhD degree at the Department of Computer Science and Engineering, Hong Kong University of Science and Technology (HKUST). His research interests include program analysis, mining software repositories, fault localization and repair. More information about him can be found at: http://home.cse.ust.hk/~mwenaa/

Rongxin Wu received the PhD degree from HKUST, in 2017. He is a post-doctoral research fellow with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology (HKUST). His research interests include program analysis, software security, and mining software repository. His research work has been regularly published in top conferences in the research communities of program languages and software engineering, including POPL, PLDI, ICSE, FSE, ISSTA and ASE and so on. He has served as a reviewer in reputable international journals and a program committee member in several international conferences. He has ever received ACM SIGSOFT Distinguished Paper award. More information about him can be found at: http://home.cse.ust.hk/~wurongxin/



Shing-Chi Cheung received the doctoral degree in computing from the Imperial College London. He joined the Hong Kong University of Science and Technology (HKUST) where he is a professor of computer science and engineering, in 1994. He founded the CASTLE research group at HKUST and co-founded in 2006 the International Workshop on Automation of Software Testing (AST). He was the general chair of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014). He was an editorial board member of the *IEEE Transactions on Software Engineering* (TSE, 2006-9). His research interests focus on the quality enhancement of software for mobile, web, deep learning, open-source and end-user applications. He is an ACM distinguished scientist. More information about his CASTLE research group can be found at http://sccpu2.cse.ust.hk/castle/people.html