Inference of development activities from interaction with uninstrumented applications

Lingfeng BAO
Zhejiang University

Zhenchang XING
Australian National University

Xin XIA
Zhejiang University

David LO
Singapore Management University, davidlo@smu.edu.sg

Ahmed E. HASSAN
Queen's University

DOI: https://doi.org/10.1007/s10664-017-9547-8

Published in Empirical Software Engineering, 2017 September, Pages 1-39
http://doi.org/10.1007/s10664-017-9547-8

Inference of development activities from interaction with uninstrumented applications

Lingfeng Bao[1] · Zhenchang Xing[2] · Xin Xia[1,3] · David Lo[4] · Ahmed E. Hassan[5]

Abstract Studying developers' behavior in software development tasks is crucial for designing effective techniques and tools to support developers' daily work. In modern software development, developers frequently use different applications including IDEs, Web Browsers, documentation software (such as Office Word, Excel, and PDF applications), and other tools to complete their tasks. This creates significant challenges in collecting and analyzing developers' behavior data. Researchers usually instrument the software tools to log developers' behavior for further studies. This is feasible for studies on development activ-

Communicated by: Massimiliano Di Penta

✉ Xin Xia
xxia@zju.edu.cn

Lingfeng Bao
lingfengbao@zju.edu.cn

Zhenchang Xing
zhenchang.xing@anu.edu.au

David Lo
davidlo@smu.edu.sg

Ahmed E. Hassan
ahmed@cs.queensu.ca

[1] College of Computer Science and Technology, Zhejiang University, Hangzhou, China

[2] Research School of Computer Science, Australian National University, Canberra, Australia

[3] Faculty of Information Technology, Monash University, Melbourne, Australia

[4] School of Information Systems, Singapore Management University, Singapore, Singapore

[5] School of Computing, Queen's University, Kingston, Canada

ities using specific software tools. However, instrumenting all software tools commonly used in real work settings is difficult and requires significant human effort. Furthermore, the collected behavior data consist of low-level and fine-grained event sequences, which must be abstracted into high-level development activities for further analysis. This abstraction is often performed manually or based on simple heuristics. In this paper, we propose an approach to address the above two challenges in collecting and analyzing developers' behavior data. First, we use our ActivitySpace framework to improve the generalizability of the data collection. ActivitySpace uses operating-system level instrumentation to track developer interactions with a wide range of applications in real work settings. Secondly, we use a machine learning approach to reduce the human effort to abstract low-level behavior data. Specifically, considering the sequential nature of the interaction data, we propose a Condition Random Field (CRF) based approach to segment and label the developers' low-level actions into a set of basic, yet meaningful development activities. To validate the generalizability of the proposed data collection approach, we deploy the ActivitySpace framework in an industry partner's company and collect the real working data from ten professional developers' one-week work in three actual software projects. The experiment with the collected data confirms that with initial human-labeled training data, the CRF model can be trained to infer development activities from low-level actions with reasonable accuracy within and across developers and software projects. This suggests that the machine learning approach is promising in reducing the human efforts required for behavior data analysis.

Keywords Software development · Developers' interaction data · Condition Random Field

1 Introduction

Researchers in the software engineering community have significant interest in understanding developers' behavior with a variety of objectives; for example, to investigate the capabilities of the developers (von Mayrhauser and Vans 1997; Corritore and Wiedenbeck 2001; Lawrance et al. 2013), the developers' information needs in developing and maintaining software (Li et al. 2013; Wang et al. 2011; Ko et al. 2006), how developers collaborate (Koru et al. 2005; Dewan et al. 2009), and what we can do to improve the performance of developers (Ko and Myers 2005; Hundhausen et al. 2006; Robillard et al. 2004; Duala-Ekoko and Robillard 2012).

Whereas previous researches have frequently produced impressive results in their own environments, their data collection and analysis methods cannot always be easily transferred to a new study context.

1. Existing studies usually focus on the developers' behavior *within* IDEs. However, today's software development practices involve many other applications (e.g., web applications, office software) specializing in different tasks, such as web search, Q&A sites, social media, and documentation.
2. The majority of studies require *application-specific support* for collecting the developers' behavior data. For example, Vakilian et al. (2012) rely on an Eclipse IDE feature to record the refactorings that are applied by developers. In the majority of other cases, researchers instrument specific software tools used in their studies to collect the required data. Such instrumentation is usually non-trivial and time-consuming to repeat.
3. Existing studies typically focus on a *specific development activity* such as feature location (Wang et al. 2011), online search (Li et al. 2013), debugging (Lawrance et al.

2013), and program comprehension (Ko et al. 2006). Researchers develop activity-specific heuristics to analyze and abstract developers' behavior. A new study context could render these activity-specific heuristics irrelevant.

In this paper, our objective is to investigate whether we can use a machine learning approach to infer a *set* of basic development activities in *real work settings* from a developer's interactions with applications in the *entire* working environment *without application-specific support*. Inference of the basic development activities from the developer's low-level actions could provide a foundation for further exploratory analysis of developers' high-level behavior patterns in software development tasks. To achieve our objective, there are two research challenges that must be addressed.

1. Developers use many desktop and web applications in their work. We need to monitor the developers' interaction with a wide range of applications at a sufficiently low level to obviate application-specific support while collecting appropriately detailed information for inferring the development activities at a higher-level of abstraction.
2. Software development involves a wide range of development activities, such as coding, debugging, testing, searching, and documentation. These activities frequently interleave with one another. Furthermore, developers can perform development activities in various ways. Inferred models of development activities and inference algorithms must adapt to this behavioral complexity and variety.

To address the first challenge, we resort to our ActivitySpace framework Bao et al. (2015a, b). The ActivitySpace framework aims to support the inter-application information needs in software development. It consists of an *Activity Tracking* infrastructure for tracking the actions of a developer in the entire working environment using operating-system windows and accessibility APIs. Compared with application-specific instrumentation, ActivitySpace can collect the developer interactions with many commonly used desktop and web applications in software development, such as IDEs (e.g., Eclipse, Visual Studio), web browsers (e.g., Firefox, Chrome, IE), and text editors (e.g., Notepad, Notepad++). While a developer is working in an application, ActivitySpace unobtrusively logs a time series of action records. Each action record logs the mouse or keyboard action, relevant window information, and focused UI component information. For example, when a developer is viewing a Java source file *example.java* in Eclipse and he/she clicks the *Open Resource* menu, ActivitySpace generates an action record including the current working window information (i.e., the Java file "example.java") and the focused UI information ("Open Resource" menu). Due to the efficiency of operating-system APIs and the ActivitySpace's design to separate the activity tracking infrastructure from the activity analysis components, activity tracking incurs negligible overhead in logging developers' actions and does not interfere with their normal work (Bao et al. 2015b).

By analyzing the captured time series of action records, we can infer the intentions and activities of the developers. For example, when a developer is debugging in Eclipse, he/she would usually set breakpoints and perform various stepping actions through one or more source files. As such, the recorded time-series data would contain a sequence of time-stamped mouse click actions (e.g., click editor ruler to set breakpoints, click stepping button on the toolbar) and/or keyboard shortcut actions (e.g., stepping shortcut "F5" or "F6"). The record would also log the name of the source files that was being debugged. Given a time series of these action records, we can infer that the developer was debugging the program using some simple rules, for example, by looking for debugging-related mouse or keyboard actions.

However, the recorded time-series data will never be this clean. First, there will be many noise actions in the recorded data, such as meaningless or wrong mouse clicks or keyword shortcuts. Furthermore, the debugging actions will most likely interleave with many other actions. For example, the developer may edit an "if" condition during debugging, or briefly read an online API specification. How can we distinguish the period when the developer debugs the software from those when the developer performs other development activities that may also involve low-level debugging, code editing, and web browsing actions? For example, for the period during which a developer is coding, he/she may integrate a code snippet from the Web into the project and execute the code to ensure that the reused code functions as expected. As another example, for the period during which the developer is reading an online tutorial, he/she may copy the sample code in the tutorial and run the code in the IDE to confirm its action. Although these development activities have similar low-level actions, the developer's intention would be considerably different, i.e., debugging versus coding versus web browsing. Heuristics-based rules that only examine individual actions and not the surrounding behavior context would likely be unable to address this behavioral complexity and variety.

We hypothesize that if the time series of low-level action records collected by ACTIVITYSPACE can be partitioned into logical segments that are assigned with meaningful semantic labels, it could be significantly easier to reveal the developer's intention and activities by examining the surrounding actions before and after an individual action. For example, if copying-pasting a piece of code from a web browser to an IDE occurs in between many coding actions, we could consider this copy-paste action as part of a coding activity. Conversely, if the copy-paste action occurs in between many web page browsing actions, we could consider the copy-paste action as part of a web browsing activity to learn something online. Therefore, to address the second challenge, we propose a framework for partitioning time-series action records into meaningful segments and giving each segment a label that represents the collective meaning of the actions in the segment. Knowing these action segments and their semantic labels could facilitate researchers and practitioners to mine and discover the developer's behavior patterns at a higher-level of abstraction.

There are several techniques to segment and label sequential data; for example, Hidden Markov Model (HMM) (Rabiner 1989), Maximum Entropy Markov Model (MEMM) (McCallum et al. 2000), and Conditional Random Field (CRF) (Lafferty et al. 2001). CRF outperforms both HMM and MEMM on many real-world sequence segmentation and labeling tasks (Lafferty et al. 2001; Pinto et al. 2003). CRF has been successfully applied to segment and label sequential action lists of a real-time strategy game named *StarCraft* (Gong et al. 2012), which is similar to the action records collected by ACTIVITYSPACE. Hence, we adopt CRFs to automatically segment and label the time series of action records. First, we abstract action records collected by ACTIVITYSPACE by extracting a set of features from the raw data. Then, we obtain the CRF training data by manually labeling the low-level action records with high-level development activities by a segmentation-based annotation process which essentially examines actions and their surrounding actions to infer relevant development activities. We use this training data to train a CRF classifier. Finally, we use the CRF classifier to automatically predict the development-activity label of unseen low-level action records.

The main contributions of this paper are:

1. We propose a CRF-based approach that aims to segment and label the time-series low-level action records that are collected by our ACTIVITYSPACE framework.

2. We evaluate the proposed approach using week-long data from ten professional developers in a real work setting.

The remainder of the paper is structured as follows. Section 2 describes the proposed approach including data collection, feature selection and CRF model training. Section 3 reports our experiment results to validate the proposed approach. Section 4 discusses threats to the validity of our research results. Section 6 reviews related work. Section 7 concludes the paper and discusses future directions.

2 Our CRF-Based Framework

Figure 1 presents an overview of our CRF-based framework. We use our ActivitySpace framework to collect developers' behavioral data when they interact with applications in their working environment. We refer to the collected time-series of action records as the *action list* in this work. The proposed framework is divided into two phases: a model training phase and a prediction phase. In the model training phase, we manually annotate the collected action lists using a set of development-activity labels that reflect the developers' intention in the activities. In this work, we focus on six basic types of development activities: coding, debugging, testing, navigation, search, and documentation. Note that our approach is general and can be applied to different labeling schema in other studies for different purposes. We use the labeled action lists as the training data to train a classifier based on CRF. In the prediction phase, we use the trained CRF classification model to automatically assign development-activity labels to the actions in the unseen action lists.

2.1 Data Collection

To improve the generalizability of the proposed framework for different working environments, we resort to operating-system level instrumentation. In particular, we use our ActivitySpace framework to collect Human-Computer-Interaction (HCI) data while a developer is interacting with different applications in his working environment. To obviate

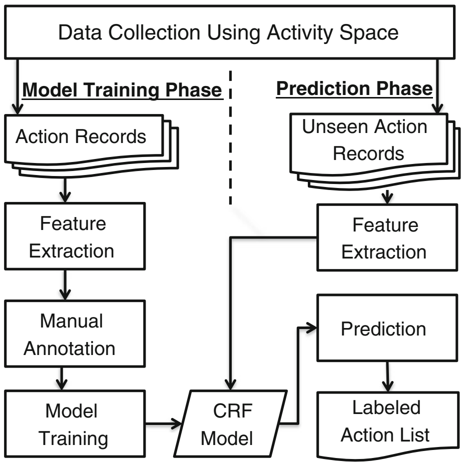


Fig. 1 The CRF-based framework

application specific support, ACTIVITYSPACE uses operating-system (OS) window APIs and accessibility APIs to record the HCI data. As the developer is interacting with an application, ACTIVITYSPACE generates a time series of action records (see Table 1 for an example). Each action record has a time stamp down to millisecond precision. An action record is composed of event type, basic window information collected using OS window APIs, and focused UI component information that the application exposes to the operating system through accessibility APIs. Operating-system APIs incur the minimal overhead for data collection. Furthermore, ACTIVITYSPACE simply logs the actions of interest without performing any time-consuming analysis; hence, it does not interfere with the user's normal work (see Section 3.3.2 for a pilot study of ACTIVITYSPACE performance).

ACTIVITYSPACE can monitor three types of mouse events (mouse move, mouse wheel, and mouse click) and two types of keyboard events (normal keystrokes such as alphabetic and numeric keys and shortcut keystrokes such as "Ctrl+F" (Search or Find), "Ctrl+O" (Eclipse shortcut key for quick outline)). Basic window information includes the position of the mouse or cursor, the title and boundary of the focused application window, the title of the root parent window of the focused application window, and the process name of the application. If the event type is mouse click, ACTIVITYSPACE uses accessibility APIs to extract the focused UI component information: *UI Name*, *UI Type*, *UI Value*, and *UI Boundary* of the focused UI component, and the *UI Name* and *UI Type* of *the root parent UI component*.

Basic window information and the focused UI information can facilitate the inference of the developer's action. For example, we can infer that the action at T_1 in Table 1 occurs in an Eclipse application window, and the action at T_n occurs in a Firefox application window. The collected window information and the focused UI information indicate that at time T_1 the developer selected the file "JSTreeDao.java" in "Project Explorer" in Eclipse, and at time T_n the developer searched *java calendar* on Google in Firefox.

Table 1 Examples of action records that are collected by ACTIVITYSPACE

Timestamp	T1	...	Tn
Event	Mouse Click		KeyInput: "Ctrl+V"
Cursor Position	(143, 254)		(595, 262)
Window Title	N/A		java calendar - Google Search - Mozilla Firefox
Window Boundary	(6, 105, 495, 1008)		(0, 0, 1920, 1040)
Parent Window Title	Java – Project/package/TimelineExample.java - Eclipse		N/A
Process Name	eclipse.exe	...	firefox.exe
UI Name	JSTreeDao.java		Search
UI Type	tree item		combo box
UI Value	N/A		java calendar
UI Boundary	(123, 249, 205, 267)		(136, 121, 706, 140)
Parent UI Name	Project Explorer		java calendar - Google Search - Mozilla Firefox
Parent UI Type	Pane		Window

It is important to note that operating-system windows and accessibility APIs are standard interfaces built in modern operating systems for assistive applications. Existing HCI studies (Hurst et al. 2010; Chang et al. 2011) and our own survey of accessibility support in commonly used software applications on three popular desktop operating systems[1] confirm that a wide range of applications expose their internal HCI data to operating-system accessibility APIs. Therefore, operating-system level instrumentation such as ACTIVITYSPACE allows our data collection method to be applied in working environments with different applications and data collection requirements.

2.2 Feature Selection

As can be observed in Table 1, the HCI data collected by ACTIVITYSPACE is rather detailed and fine-grained. To facilitate the understandability and manual labelling of the raw HCI data, we preprocess the raw HCI data by abstracting and extracting a set of features. This set of features is also used to train the CRF classifier.

For each action record in the raw HCI data, we extract the following features:

- **Application**: this can be directly inferred from process name, such as Eclipse, Firefox.
- **AppType**: we categorize common-used applications into six predefined application types: IDE, Web Browser, Office, Text Editor, PDF Reader and Others.
- **WindowName**: this can be extracted from the title of the application window or its parent window. We leverage regular expressions to extract the window name.
- **DocType**: the extracted window information usually contains the file or document on which the developer is working. We use regular expressions to extract the extension of the file (e.g., ".java"). The DocType can reveal the type of information on which a developer is working (e.g., "java" for source file and "xml" for configuration). If AppType is Web Browser, the DocType is set to "WebPage".
- **IDEContext**: words in the window title may indicate the context where the developer is working. For example, in Eclipse, the window title contains perspective information, such as "Java", "Java EE" or "Debug". Similarly, the window title of Microsoft Visual Studio contains the words "Debugging" when the developer is debugging. We leverage regular expressions to extract such context information for IDE applications. For other AppTypes, this feature is set to "NA" (not applicable).
- **EventType**: mouse event or keystroke event.
- **KeySequence**: for a sequence of keystroke events, if all the events are normal keystroke events (alphabetic and numeric keys), we aggregate all keyboard input into a key sequence. Function keys or shortcut keys are maintained as separate keyboard input. For mouse events, this feature is set to "NA".
- **UIType, UIName, UIValue, ParentUI**: these features can be directly extracted from the HCI data.

Table 2 shows an example of an action list and the extracted features for each action in the list. From this action list, we can learn that the developer was editing the source file "pricingoptionSearch.xhtml" in Eclipse. Then, he switched to Chrome to access the modified web page and attempted input on the web page. Then, he switched back to Eclipse. He used shortcut key "Ctrl+Shift+R" to activate the "Open Resource" dialog in Eclipse to locate another file containing pricing options. Then, he used shortcut key "Ctrl+F" to search

[1] http://baolingfeng.weebly.com/accessibility-survey.html

Table 2 An example of action list

Time	Application	AppType	WindowName	DocType	IDEContext	EventType	KeySequence	UIType	UIName	UIValue	ParentUI	Label
t1	eclipse	IDE	pricingoptionSearch.xhtml	xhtml	Java EE	mouse	NA	edit	NA	Long Text	pane	C
t2	eclipse	IDE	pricingoptionSearch.xhtml	xhtml	Java EE	keyinput	Ctrl+C	NA	NA	NA	NA	C
t3	eclipse	IDE	pricingoptionSearch.xhtml	xhtml	Java EE	mouse	NA	thumb	Position	NA	pane	C
t4	eclipse	IDE	pricingoptionSearch.xhtml	xhtml	Java EE	keyinput	*int p* =	NA	NA	NA	NA	C
t5	eclipse	IDE	pricingoptionSearch.xhtml	xhtml	Java EE	keyinput	Ctrl+S	NA	NA	NA	NA	C
...												
t6	chrome	Browser	Pricing Options - DO2	WebPage	NA	mouse	NA	edit	address bar	*http://...*	tool bar	T
t7	chrome	Browser	Pricing Options - DO2	WebPage	NA	keyinput	*www.*	NA	NA	NA	NA	T
...												
t8	eclipse	IDE	home.xhtml	xhtml	Java EE	keyinput	Ctrl+Shift+R	NA	NA	NA	NA	N
t9	eclipse	IDE	Open Resource	NA	NA	keyinput	*pricingoption*	NA	NA	NA	NA	N
t10	eclipse	IDE	pricingoptionSearch.xhtml	xhtml	Java EE	mouse	NA	edit	NA	Long Text	pane	N
t11	eclipse	IDE	pricingoptionSearch.xhtml	xhtml	Java EE	mouse	NA	thumb	Position	NA	pane	N
t12	eclipse	IDE	pricingoptionSearch.xhtml	xhtml	Java EE	keyinput	Ctrl+F	NA	NA	NA	NA	N
t13	eclipse	IDE	Find/Replace	NA	NA	keyinput	*option*	NA	NA	NA	NA	N
t14	eclipse	IDE	Find/Replace	NA	NA	mouse	NA	button	Close	NA	dialog	N
...												
t15	iexplorer	Browser	ajax... - Stack Overflow	WebPage	NA	mouse	NA	button	close	NA	window	W
...												
t16	eclipse	IDE	pricingoptionSearch.xhtml	xhtml	Debug	keyinput	F8	NA	NA	NA	NA	D
t17	eclipse	IDE	pricingoptionSearch.xhtml	xhtml	Debug	mouse	NA	edit	NA	Long Text	pane	D
t18	eclipse	IDE	pricingoptionSearch.xhtml	xhtml	Debug	keyinput	F8	NA	NA	NA	NA	D

the keyword "option" in file *pricingoptionSearch.xhtml*. Next, he read a web page regarding AJAX on STACKOVERFLOW. Finally, he debugged the "pricingoptionSearch.xhtml" file in the Eclipse debug perspective.

2.3 Manual Annotation

We then review the collected data and summarize the developers' activities into the following six high-level development-activity labels:

- **Coding (C)**: developers' coding activities can occur in IDEs or other text editors. When the developers are writing code, the ACTIVITYSPACE will collect many key input events in the IDEs or text editors.
- **Debugging (D)**: developers' debugging activities refer to not only using debugging tools in IDEs, but also activities such as viewing bug reports in documentation software or web browsers.
- **Testing (T)**: developers' testing activities refer to executing the applications with certain test inputs, and recording or examining test results in documentation software or web browsers. Note that the application under test can be a desktop application or a web application (e.g., a J2EE web application in our experiment).
- **Navigation (N)**: developers' navigation activities refer to browsing, searching and locating resources in IDEs, such as browsing source files in the "Project Explorer", searching some words in a file, or searching classes or methods.
- **Web Browsing (W)**: developers' Web browsing activities refer to searching and reading web pages in web browsers.
- **Documentation (U)**: the developers' documentation activities refer to reading or writing project-specific documents (such as requirement documents, design models, test cases) in documentation software (such as Office Word, Excel, and PDF applications). In practice, developers may also use web browsers or IDEs to read or write project-specific documents.

During the development of coding schema, we note the key actions that are commonly associated with relevant development activities. For example, when developers are "coding", several keystroke events will be generated then they press the shortcut "Ctrl+S" to save the changes. Another example is that developers usually use shortcut "Ctrl+F" or "Ctrl+H" in Eclipse to begin code search, which could be indicators of "navigation" activities.

Given an action list such as the one in Table 2, we go through the action list and segment the list into a list of action fragments. The principle of segmentation is that the majority of a developer's actions in a fragment are associated with the same type of development activity. The principle of determining fragment boundaries is to search for distinct transitions between development activities. Once we have the initial list of action fragments, we go through the list again to determine if adjacent action fragments can be merged into one action fragment or an action fragment can be further segmented into more fine-grained fragments. Finally, we annotate each action in a fragment with the development-activity label to which the majority of the actions in the fragment belong.

It is important to note that although individual actions contain important hints for inferring the high-level development activities, the development activities as defined in the above coding schema cannot be reliably determined by looking at the actions and involved applications in the actions individually. For example, as part of a debugging activity, the developer can read the bug report in documentation software, whereas as part of a documentation activity, the developer can read a requirement specification in documentation software. Similarly, the developer can use a web browser to search the Web (as part of a Web browsing activity), read documents (as part of a documentation activity), or even test the web application (as part of a testing activity). Furthermore, the developer can navigate between source files during coding, debugging, or to locate feature implementation. To reliably infer the high-level development activities from the action list, we must examine the surrounding context before and after an individual action. This is why we adopt the above segmentation-based annotation process which essentially examines actions and their surrounding context to determine relevant development activities. This is also why we propose to use context-sensitive data analysis methods such as CRF to infer development activities from the low-level actions.

2.4 CRF Classifier for Development Activities

CRF is a type of discriminative undirected probabilistic graphical model that can be applied to segment and label sequential data, such as natural language text or biological sequences (Berger et al. 1996; Durbin et al. 1998). In contrast to classical classifiers that predict a label for a single sample without regard to neighboring samples, CRF can consider context. CRF relaxes the independence assumptions that are required by HMM, which defines a joint probability distribution over label sequences given an observation sequence, and also avoid the label bias problem, which is a weakness of MEMM.

In this work, we adopt linear-CRF methods (Lafferty et al. 2001). Formally, given an action list $\boldsymbol{X} = (a_1, a_2, ..., a_m)$ with m actions (i.e., an observation sequence), a segmentation S of $\boldsymbol{X}$ is defined as k non-overlapping segments, denoted as $S = \{s_1, s_2, ..., s_k\}$, where for $1 \leq i \leq k$, $s_i = (a_{b_i}, a_{b_{i+1}} ..., a_{b_i+v_i})$, $1 \leq b_i \leq m$ and $\sum_{i=1}^{k} v_i = m - k$. Let $\boldsymbol{L} = \{l_1, l_2, ..., l_h\}$ be a set of h unique labels. In this work, the labels are the six development-activity labels as defined in Section 2.3. Our problem is to identify all the segments $s_1, s_2, ..., s_k$, and assign a label $l \in \boldsymbol{L}$ for each segment s_i, $1 \leq i \leq k$. The manual annotation step produces a label sequence $\boldsymbol{Y}$ for a given observation sequence $\boldsymbol{X}$. A set of observation sequences together with the corresponding label sequences constitute the training data for training the linear-CRF classifier. To train the CRF classifier, we use not only features of the current action record, but also features of the neighboring action records as defined in Section 2.2. Further details regarding linear-CRF can be found in Lafferty et al. (2001).

3 Experiment

In this section, we introduce our research questions, describe our experimental setup and analyze the experiment results.

3.1 Evaluation Metrics

In this paper, we train a CRF classifier to segment and label developers' action lists. Given an action list $\boldsymbol{X}$ consisting of N actions and a set of action labels $L = \{\lambda_1, \lambda_2, ..., \lambda_h\}$, we use C_λ to represent the number of actions correctly assigned the label λ by the CRF classifier. The total number of actions correctly assigned is represented as $C = \sum_{\lambda=1}^{h} C_\lambda$. The correctness of the assignment is determined against the human labels of the action list (i.e., ground truth).

We use accuracy to evaluate the overall performance, i.e., the number of correctly classified actions over the total number of actions: $accuracy = \frac{C}{N}$.

We also use precision, recall and F1-score to evaluate the performance for each label. For one label λ, we use TP_λ, TN_λ, FP_λ, and FN_λ to represent the number of true positives, true negatives, false positives, and false negatives, respectively. We calculate precision, recall and F1-score as follows:

– **Precision:** the proportion of actions that are correctly labeled as λ among those labeled as λ, i.e.,

$$precision_\lambda = \frac{TP_\lambda}{TP_\lambda + FP_\lambda} \tag{1}$$

– **Recall:** the proportion of actions that are correctly labeled as λ, i.e.,

$$recall_\lambda = \frac{TP_\lambda}{TP_\lambda + TN_\lambda} \tag{2}$$

– **F1-score:** a summary measure that combines both precision and recall. It evaluates if an increase in precision (recall) outweighs a reduction in recall (precision), i.e.,

$$F1 - score_\lambda = 2 \times \frac{precision_\lambda \times recall_\lambda}{precision_\lambda + recall_\lambda} \tag{3}$$

As multiple action labels are predicted in our study, we also compute macro-averaged precision, recall and F1-score, which can provide a view of the general prediction performance. Macro-averaged metrics weigh all the classes equally, regardless of how many instances they include. Macro-averaged metrics can be calculated as follows:

$$M_{macro} = \frac{1}{h} \sum_{\lambda=1}^{h} M_\lambda \tag{4}$$

where M indicates an evaluation measure, i.e. precision, recall and F1-score, and h is the number of classes.

We also compute the Kappa statistic (Fleiss 1971), which is a measure of agreement between the predictions and the actual labels. Kappa metrics can also be interpreted as a comparison of the overall accuracy of a classifier to the expected accuracy of a random guess classifier (as defined in the RQ1 in Section 3.2). The greater the Kappa metric, the better the classifier is compared to the random guess classifier. The interpretations for Kappa values are displayed in Table 3.

Table 3 The interpretations for Kappa values

Kappa value	Interpretation
< 0	Poor agreement
[0.01, 0.20]	Slight agreement
[0.21, 0.40]	Fair agreement
[0.41, 0.60]	Moderate agreement
[0.61, 0.80]	Substantial agreement
[0.81, 1.00]	almost perfect agreement

3.2 Research Questions

In this work, we propose a CRF-based approach to abstract low-level developers' actions into a set of basic, yet meaningful development activities. To investigate the effectiveness and applicability of the proposed CRF-based approach, we are interested to answer the following research questions:

RQ1 *How effective is the proposed CRF-based approach for segmenting and labeling developers' action lists? Further, how much improvement can the proposed approach achieve over a heuristic-rules based method and an ordinary classifier (i.e., SVM)?*

In this paper, we trained the CRF classifier to segment and label the developers' action lists. The CRF implementation that we used was CRF++.[2] In this study, we defined three baseline classifiers for comparison: Support Vector Machine (SVM), rule-based prediction, and random weighted classifier.

SVM is a popular classification algorithm for analyzing data and has been used in many previous software engineering research studies (e.g., Anvik et al. 2006; Maiga et al. 2012; Sun et al. 2010; Thung et al. 2012; Tian et al. 2012; Le and Lo 2013). SVM is trained using the same training data and the same set of features as used for the CRF classifier. Compared to SVM, CRF considers the neighboring context; hence, it can typically achieve superior performance on sequential data classification compared to SVM.

To investigate the benefits of a machine learning approach over a hand-crafted heuristic-rules based method, we designed the following heuristic rules to infer development activities from the action lists:

– **Coding**: all key input actions in IDEs or in code files in text editors, except some keyboard shortcuts for specific functions, e.g., searching shortcut "Ctrl+F" for the majority of applications, stepping shortcut "F5" or "F6" in Eclipse.
– **Debugging**: all actions in the Debug perspective of IDE, key input actions using debugging keyboard shortcuts,[3] such as "F5" or "F6" in Eclipse, and all actions in the Bugzilla and bug report documents.
– **Testing**: all actions in the tested applications and the testing documents. In our collected data, all the tested applications are web applications and we identify them using keywords, such as the name or URL of the website, "localhost".
– **Navigation**: all other actions except *coding* and *debugging* actions.
– **Web Browsing**: all actions in websites other than Bugzilla web pages and the web pages of tested application.
– **Documentation**: all actions in the documents other than bug report documents and testing documents.

We developed a Python script according to these heuristic rules and applied this script to our collected data to infer development activities.

[2]CRF++: Yet another CRF toolkit http://crfpp.sourceforge.net/

[3]Note that during the manual annotation process (see Section 2.3), we find all keyboard shortcuts in our collected data and categorize them according to their functions.

A random weighted classifier randomly predicts a certain label according to its probability of occurrence in the data. If the proportion of class λ instances in the training data is x_λ, we will compute the accuracy, precision, recall and F1-score as follows:

$$Accuracy = \sum_{\lambda=1}^{h} x_\lambda^2 \tag{5}$$

$$Precision_\lambda = Recall_\lambda = F1 - score_\lambda = x_\lambda \tag{6}$$

Note that the macro-averaged metrics of a random weighted classifier are equal to the probability of picking a certain class (i.e. 1/6 in our study).

In our experiment, we regarded half a working day as a working session for each participant. Hence, we divided the collected action list into ten folds (i.e., working sessions) for each participant (see our collected data in Section 3.3.3). To preserve the sequential nature of the data, the action records in every fold were order preserved and the order of consecutive folds were also preserved. Then we used the first n folds as the training data and the remaining $10 - n$ folds as the testing data (n from 1 to 9). An evaluation metric of a classifier is the average of nine rounds of testing. To evaluate the performance of the proposed CRF-based classifier and the three baseline classifiers, we performed three analyses: 1) We compared the accuracy, Kappa metric, and macro-averaged precision/recall/F1-score of the different classifiers on all the data as a whole. We also compared the precision, recall, and F1-score of the different classifiers on different labels of the whole data. 2) We compared the accuracy, Kappa metric, and macro-averaged precision/recall/F1-score of the different classifiers for each participant. 3) We compared the precision, recall, and F1-score of the different classifiers on different labels of each participant's data.

RQ2 *How does the size of the training data affect the performance of the CRF classifier? What proportion of the training data could achieve the best performance?*

The size of the training data could have a critical impact on the classification result. In general, the larger the amount of training data, the better the performance of the trained classifier. However, sometimes increasing the training data does not produce a better performing classifier. Therefore, we wanted to investigate the impact of different sizes of training data on classification performance. To answer this question, we conducted two experiments with different methods of controlling the size of the training data. First, we used the accuracies of nine rounds of testing for each participant obtained in RQ1 to answer this question. That is, the size of training data ranges from 0.5 to 4.5 days of the action list of each participant. Then, to investigate the effort of manual annotation required for different participants, we controlled the size of the training data by the number of action records to be labeled. We set the number of action records in the training data from 100 to 1,000 with a step size of 100 (i.e., nine trials). Then, we calculated the accuracy of each trial and compared the results. In both experiments, the order of the action records was preserved.

RQ3 *Can the CRF classifier be applied to cross-developer prediction? That is, can we use the CRF classifier trained using one (or some) developer's action list(s) to classify the action list of another developer?*

Although developers' behavior and actions are frequently different, we hypothesize that if two developers are in a similar working context, such as in the same project or performing similar tasks, then the action lists of the different developers may share some common characteristics. Therefore, cross-developer prediction could be possible. To answer this question, we used one participant's action list as the training data, and an other participants' action lists as the test data to evaluate the performance of the CRF. We also used all the other participants' action lists as the training data, and used the remaining participant's action list as the test data to evaluate the performance of the CRF. We used the accuracy as the evaluation metric in this experiment. We also reported the Kappa metrics of cross-developer prediction.

3.3 Experimental Setup

In this work, our objective was to investigate the effectiveness and applicability of the proposed approach for analyzing professional developers' behavior data in real software development settings. Therefore, we deployed our ActivitySpace framework (Bao et al. 2015a) in our industry partner company and collected and annotated ten professional developers' one-week working data in three software projects. Then, we described the participant developers, the raw behavior data collected, and the manual annotation process to generate the ground-truth development activity labels for responding to the above research questions. The complete experiment procedure is presented in Fig. 2.

3.3.1 Participants

We recruited ten professional developers from Hengtian, an IT company in China. Hengtian is an outsourcing company with more than 2,000 professional developers. It primarily undertakes outsourcing projects for US and European corporations (e.g., StateStreet Bank, Cisco, and DST). Table 4 summarizes the demographic information of the ten participating developers. Each of the ten recruited developers (S1, S2, ..., S10) had at least two years of working experience. All of them used Java as a programming language in their daily work. They were from three different projects: participant developers S1–S5 were from a finance system re-engineering project, S6–S8 were from an e-commerce website project, and S9 and S10 were from a project that developed and maintained an internal information management system for the company. These ten developers' activities could be different based on the tasks for which they are responsible, such as bug fixes, front-end web development, database development, and business analysis. This made our dataset more diverse. In addition to providing working data for the study, these ten participants were also consulted to

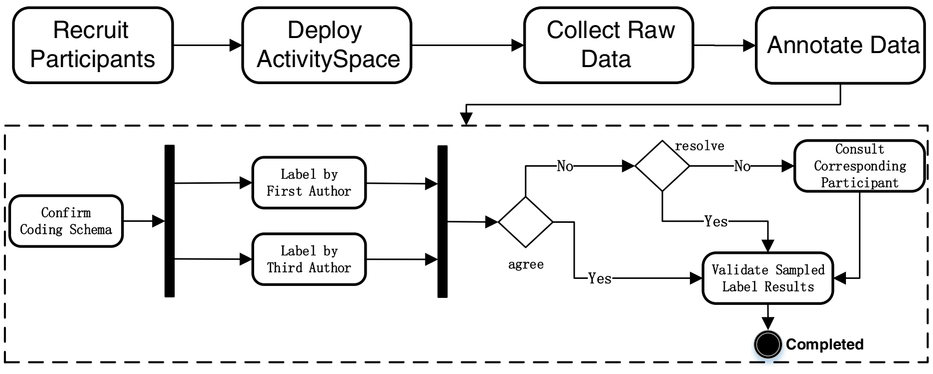


Fig. 2 The experiment procedure

Table 4 The demographics information of the 10 participant developers

	Project	Project description	Experience	Responsibility
S1	P1	A financesystem	2 years	Web Development
S2	P1	re-engineering projec	4 years	Java Development, Database Development
S3	P1		3 years	Java Development, Database Development
S4	P1		4 years	Web Development, Database Development
S5	P1		2 years	Web Development
S6	P2	An e-commerce	4 years	Web Development
S7	P2	website project	3 years	Web Development
S8	P2		3 years	Quality Assurance, Business Anlysis
S9	P3	An internal management	5 years	Database Development
S10	P3	system of company	2 years	Deployment

confirm the coding schema of development activities, resolve any disagreements between annotators, and validate the label results (See Section 3.3.4).

3.3.2 ActivitySpace *Deployment Settings*

We deployed our ActivitySpace tool on the participants' computers and collected their working data throughout one week. In this study, we configured ActivitySpace to track the developers' interactions with the applications commonly used in developers' daily work, including web browsers (e.g., *Firefox, Chrome, Internet Explorer*), document editors (and/or readers) (e.g., *Word, Excel, PowerPoint, Adobe Reader, Foxit Reader, Notepad, Notepad++*), and IDEs (e.g., *Eclipse, Visual Studio*). We configured ActivitySpace to collect keyboard events and mouse-click events for focused UI components. ActivitySpace was configured to ignore mouse-move events and mouse-wheel events. The developers had the option of terminating ActivitySpace's data tracking when they were working on proprietary data and/or applications from the outsourcing company. This option only influenced the amount of data collected, not the developers' working behavior, because they used the same working environments for proprietary or non-proprietary data and applications.

We conducted a pilot study to verify the runtime performance of ActivitySpace. We modified the code of ActivitySpace to log the time spent on recording one action record. We deployed the modified ActivitySpace on the computer of one developer that participated in our study and collected 1 h of his working data. ActivitySpace collected 426 action records. The average and standard deviation time was 0.08 ± 0.04 s per action record. Note that this time statistic included the time spent on logging the time, which is an overhead of the modified ActivitySpace. We also asked the developer to monitor memory usage during the data collection. The developer reported that the memory usage of ActivitySpace was stable (approximately 40MB RAM). The configuration of the developer's computer was Windows 7, 8GB RAM and Intel(R) Core(TM) i5-4570 CPU. This pilot study confirmed that ActivitySpace did not interfere with developers' normal work.

3.3.3 Raw Interaction Data

For the efficiency of action logging, ActivitySpace did not perform any analysis of the logged actions during the logging process. We post-processed the logged raw data to determine the effective working hours for each developer. Effective working hours refer to the time when a developer continuously performed actions on his/her computer. If a developer did not perform an action on the computer for a period (30 min in this study), we did not regard that period as effective working time. Furthermore, we only collected the developers' actions that were related to the project. Actions such as reading news on news websites (e.g., weibo.com) were eliminated. All developers ran ActivitySpace on their computer for five working days. As summarized in Table 5, after the data preprocessing, we collected approximately 240 h of effective working data, which contained more than 50,000 action records. We regarded half a working day as a working session for each developer and computed the average and standard deviation of the action records per working session (see the last column in Table 5). For different developers, because their tasks and behavior were different, the number of action records varied from approximately 3,600 to approximately 6,800.

3.3.4 Manual Annotation

To confirm the developed coding schema (i.e., the six kinds of development-activity labels, see Section 2.3), we emailed a survey of the developed coding schema to 20 developers in Hengtian, including the ten participants in our study and ten additional developers. In the document, we first listed the definition of activity labels (as defined in Section 2.3) to the developers. Then, we asked two questions: 1) What are the major activities in your working time? 2) Do you think our coding schema covers all the activities in your working time? Eighteen of 20 developers felt that our coding schema covers their major activities in working time. Three sample responses are provided as follows:

– *My job is in charge of Java development, the majority of my work is coding and fixing bugs. I also search for solutions when I have technical problems. Sometimes I will maintain the documents in my project.*

Table 5 The overview of raw data

	EffectiveHour	#ActionRecord	#Session (mean ± std)
S1	22.24	4,342	427.6 ± 16.1
S2	20.95	5,902	588.3 ± 25.3
S3	19.42	3,983	385.0 ± 47.4
S4	32.31	6,814	683.2 ± 32.6
S5	23.49	4,760	481.1 ± 21.0
S6	23.97	5,082	508.4 ± 17.5
S7	21.13	5,222	517.8 ± 14.3
S8	30.90	4,890	488.3 ± 13.2
S9	24.80	5,358	539.0 ± 73.5
S10	21.58	3,654	361.3 ± 36.3
Total	240.78	50,006	498.0 ± 96.6

- *I am a database administrator. I write numerous SQL scripts and also test many scripts submitted by other developers. To solve exceptions, I usually search for answers on the internet or refer to the official documentation.*
- *I am a QA and responsible for testing three projects in the company. I think my tasks belong to "Testing", "Navigation", "Web Browsing" and "Documentation", because I are not required to write much code and only write simple scripts sometimes.*

Two developers identified some limitations of our coding schema, which was as follow:

- *I think this coding schema is for developers. However, I am not only a developer but also a project manager. I spend a considerable amount of my time on email and project management. These management activities may not be covered in your coding schema.*
- *I agree that my major development activities are covered in the coding schema. However, if I use other applications, e.g. email client, message sender, to communicate with other people, to what label do these activities belong?*

These two developers' questions are concerned with management and communication activities in the broader context of software development. As our focus in this study is on development-related activities, we consider our coding schema sufficient for this study. We leave the coding-schema limitations identified by the two developers as a future work.

To ensure a high level of rigor in our manual annotation process, the first and third authors, who both had more than six years programming experience, manually labeled the collected action records. The manual annotation process was divided into 10 annotation sessions, i.e., labeling the action list of one participant per session. The entire manual annotation process required approximately 80 h, including manual annotation and validation of the annotation results with the participant developers.

The annotators followed the segmentation-based annotation process and principles described in Section 2.3. During an annotation session, the two annotators independently annotated the same action list. We used Fleiss Kappa (Fleiss 1971) to measure the agreement between the two annotators. The overall Kappa value (see the interpretations for Kappa values in Table 3) between the two annotators on all action lists was 0.61 which indicates a substantial agreement between the annotators. After completing the manual annotation process, the two annotators discussed their disagreements to reach a common decision. We had two types of disagreements. First, the majority of the disagreements were due to the boundary of adjacent segments, i.e., which segments several actions close to segment boundary should belong to. The two annotators resolved such boundary disagreements by reviewing and discussing the relatedness of those close-to-boundary actions and adjacent segments. Second, some segments contained a mix of similar numbers of different types of actions. The two annotators could disagree on action labels in such segments. If they could not resolve the disagreement, the corresponding developers were consulted. When consulting the developers, we provided them with code files, web pages and/or other documents in the action records, and the screenshots that ActivitySpace captured when logging the action actions. This helped the developers recall their activities (Safer and Murphy 2007). Then, we asked the developers to explain what they did in these activities and why they did these activities. With the developers' explanation of their activities and intentions behind these activities, the two annotators made a final decision on action labels.

In our annotation approach, we annotated each action in a segment with the development-activity label to which the majority of the actions in the segment belong. To confirm the validity of this majority-voting heuristic and ensure the quality of the ground-truth labels for the experiments, we validated our action labeling results by interviewing the participant developers. Considering the volume of the data, we did not validate the entire annotation results. Instead, we randomly sampled segments of labeled action records that contained approximately 10% of action records for each participant. As the goal of this validation was to confirm the majority-voting labeling heuristic, we excluded segments with similar numbers of different types of actions which could require human judgment to determine action labels for the actions in the segments.

Recall that we showed the definition of the activity labels to the developers when they were asked to confirm the developed coding schema. Before validating the action labeling results with each participant, we described the definition of activity labels to the developers once more, including their feedbacks to our coding schema to refresh their memory. Then, for each sampled segment, we provided the corresponding developer with code files, web pages and/or other documents, and the screenshots that ActivitySpace captured in that segment. We asked the developer to confirm whether the collected data were consistent with their work at the time the data were collected. Reviewing the documents and screenshots logged in a segment helps the developer recall his/her activities in the segment (Safer and Murphy 2007). Subsequently, we asked the developer to explain what activities they did in the segment and why they did these activities. Following the developer's explanation of his/her activities, we asked the developer to identify one major development activity (according to our coding schema) in the segment. The interviews with the developers required approximately 6 h. Finally, we compared the participants' reported major development activities for the sampled action segments with our action labeling results to confirm the validity of our annotation results. For the sampled segments, all the major activities reported from participants agreed with our annotation results. This confirms the validity of our majority-voting action labeling heuristic and the quality of ground-truth labels for the experiments.

3.4 Ground-Truth Development Activity Labels

Table 6 shows the total number of action records and the corresponding percentage in different types of development activities for the ten participants. We can see that different developers had different distributions of development activities in their work. The number

Table 6 The number and percentage of each class of development activity by different participants

	Coding	Debugging	Testing	Navigation	WebBrowsing	Documentation
S1	1874 (43.16%)	354 (8.15%)	1231 (28.35%)	705 (16.24%)	160 (3.68%)	18 (0.41%)
S2	727 (12.32%)	430 (7.29%)	1263 (21.4%)	2694 (45.65%)	251 (4.25%)	537 (9.1%)
S3	1044 (26.21%)	456 (11.45%)	443 (11.12%)	1877 (47.13%)	81 (2.03%)	82 (2.06%)
S4	1503 (22.06%)	62 (0.91%)	1505 (22.09%)	2282 (33.49%)	1274 (18.7%)	187 (2.74%)
S5	2319 (48.72%)	293 (6.16%)	1278 (26.85%)	733 (15.4%)	135 (2.84%)	2 (0.04%)
S6	1537 (30.24%)	712 (14.01%)	1878 (36.95%)	709 (13.95%)	157 (3.09%)	89 (1.75%)
S7	1197 (22.92%)	999 (19.13%)	2047 (39.2%)	716 (13.71%)	201 (3.85%)	62 (1.19%)
S8	702 (14.36%)	127 (2.6%)	1191 (24.36%)	241 (4.93%)	921 (18.83%)	1708 (34.93%)
S9	1679 (31.34%)	0 (0%)	1759 (32.83%)	691 (12.9%)	1212 (22.62%)	17 (0.32%)
S10	387 (10.59%)	313 (8.57%)	783 (21.43%)	1994 (54.57%)	177 (4.84%)	0 (0%)
Total	12969 (25.93%)	3746 (7.49%)	13378 (26.75%)	12642 (25.28%)	4569 (9.14%)	2702 (5.4%)

of action records labeled in three types of development activities, namely coding (25.93%), testing (26.75%) and navigation (25.28%), was more than the total number of action records labeled in the other three development activities. 5%–10% of the action records were labeled in debugging for specific participants, including S1, S2, S3, S5, S6, S7, and S10. Participants S4, S8 and S9 had minimal or no action records labeled in debugging. We found that this was because S4 was mainly responsible for front-end development, S8's main work was testing and business analysis, and S9 was responsible for database development during the week of our data collection. We further confirmed with S4 that he usually used a log to observe and verify code change and he rarely used IDE debug features. Therefore, there were only a small number of debugging action records for participant S4.

The roles of all participants. except participant S8, in our study were developers in the studied company. In this company, developers mainly write software code and unit test code, whereas other documents such as requirement documents, bug reports, and test cases are written by other people. For example, a business analyst is responsible for eliciting requirements from clients and writing the requirement documents. During the week we collected the data, only participant S8 performed business analysis tasks. Hence, there were very minimal documentation actions in our collected data. As documentation actions are not representative of the work performed by the participants, we ignored them in our experiments.

3.5 Results

3.5.1 RQ1: Effectiveness of CRF—Performance for the Whole Data

Tables 7, 8 and 9 show the confusion matrixes of CRF, the heuristic-rules method, and the SVM baseline for the whole data of all the ten participants, respectively. For each approach, the confusion matrix was constructed by accumulating the confusion matrixes of all rounds of validation. Based on these confusion matrixes, we computed the overall accuracy and Kappa metric of CRF, the heuristic-rules method, and the SVM baseline for the whole data, as shown Table 10. To compute the accuracy of the random weighted classifier on the whole data, we set x_λ be equal to the average proportion of instances in the whole data. The accuracy is equal to 0.221, which is lower than those of the other classifiers. Table 11 shows the precision, recall, and F1-score for each label using CRF, the heuristic-rules method, the SVM baseline and the random weighted classifier for the whole data. The last column of Table 11 is the corresponding macro-averaged metrics.

Table 7 Confusion matrix of CRF for the whole data

		Predicted label					
		Coding	Debugging	Testing	Navigation	Web browsing	Documentation
Actual label	Coding	42593	2615	1258	11014	452	55
	Debugging	3047	9187	1125	5187	138	2
	Testing	2132	833	47175	508	10072	670
	Navigation	6621	2691	775	45773	376	62
	Web browsing	439	173	5926	170	14736	92
	Documentation	541	3149	511	212	383	4356

Table 8 Confusion matrix of heuristic-rules for the whole data

		Predicted label					
		Coding	Debugging	Testing	Navigation	Web browsing	Documentation
Actual label	Coding	24168	4450	454	24677	3630	608
	Debugging	1625	11676	275	4007	1075	28
	Testing	25	812	28854	184	29949	1566
	Navigation	14253	1386	443	39171	500	545
	Web browsing	17	15	4015	36	17329	124
	Documentation	6	0	10	14	265	8857

The accuracies and the macro-averaged metrics demonstrate that the proposed CRF classifier achieved the best performance on the whole data, the performance of the heuristic-rules method was in the middle, and the SVM baseline performed the poorest. Except for the macro-averaged recall where the proposed CRF classifier and the heuristic-rules method were virtually identical, our CRF classifier achieved significantly superior results for all other metrics. Furthermore, the classification results of the proposed CRF classifier had substantial agreement with the actual labels, whereas the heuristic-rules method had only moderate agreement, and the SVM baseline had only slight agreement.

Regarding the precision, recall, and F1-score for each label in the whole data, CRF had the highest precision on all labels except for the precision of the heuristic-rules method for the label "Debugging". CRF had the highest recall on three labels: "Coding", "Testing" and "Navigation", whereas the heuristic-rules method had the highest recall on the other three labels. CRF achieved the highest F1-score on all labels except for the "Debugging" and "Documentation" labels using the heuristic-rules method. The better performance of the heuristic-rules method for the label "Debugging" can be attributed to the fact that the majority of the debugging activities were performed in debugging perspective, and thus could be easily inferred by heuristic rules. A similar reason can be extended to the better performance of the heuristic-rules method for the label"Documentation". That is, developers performed the majority of the documentation activities in Office or PDF software, which can be easily inferred.

Table 9 Confusion matrix of SVM for the whole data

		Predicted label					
		Coding	Debugging	Testing	Navigation	Web browsing	Documentation
Actual label	Coding	22706	50	21793	10248	1041	2149
	Debugging	7097	11	6738	4167	16	657
	Testing	18018	947	27183	4394	4480	6368
	Navigation	9372	50	20813	22920	487	2656
	Web browsing	5748	681	8156	3197	2441	1313
	Documentation	2110	1695	299	662	3398	988

Table 10 Accuracies and Kappa values of CRF, heuristic-rules and SVM for the whole data

	Accuracy	Kappa value
CRF	0.728	0.651 (substantial)
RULE	0.578	0.476 (moderate)
SVM	0.339	0.133 (slight)

3.5.2 RQ1: Effectiveness of CRF - Performance for Each Participant

Table 12 presents the accuracy of the proposed CRF classifier, the heuristic-rules method, the SVM baseline, and the random weighted classifier for each participant. The reported accuracy of a classifier is the average accuracy of nine rounds of testing. The standard deviation of the accuracies of nine rounds of testing are also shown in Table 12. The random weighted classifier achieved the lowest average accuracy, but in some cases its accuracies were larger than those of the SVM baseline, e.g., the participant S1 and S7.

The results indicate that our CRF classifier had the best accuracies for all participants, except participant S8. For the heuristic-rules method, the accuracies of all participants were not overly stable: the lowest was 0.386 for participant S1 and the highest was 0.740 for participant S8. For participant S8, we found that the participant performed many documentation activities. The majority of documentation activities were performed in Office or PDF software, which could be easily identified by heuristic rules. The heuristic-rules method also demonstrated an acceptable result for participant S2, because this participant performed many navigation activities, which can be easily identified by heuristic rules such as the "Find and Search" window. However, the accuracies of the other participants in general were not very good. The average accuracy of the heuristic-rules method across all the participants was 0.560, while the average accuracy of the proposed CRF classifier was 0.761.

For the SVM baseline, the accuracies were all below 0.6 (only 0.325 on average). Except for participant S1, SVM was less effective than the heuristic-rules method. This could be because SVM does not consider the behavior context and local features alone cannot reliably

Table 11 Precisions, recalls, f1-scores and the corresponding macro-averaged metrics for each label using CRF, heuristic-rules, SVM and random weighted classifier for the whole data

		Coding	Debugging	Testing	Navigation	Web browsing	Documentation	Marco-averaged
Precision	CRF	0.769	0.493	0.831	0.728	0.563	0.832	0.703
	RULE	0.603	0.637	0.847	0.575	0.329	0.755	0.624
	SVM	0.349	0.003	0.320	0.503	0.206	0.070	0.242
	Random	0.258	0.083	0.273	0.250	0.096	0.041	0.167
Recall	CRF	0.735	0.492	0.768	0.813	0.684	0.476	0.661
	RULE	0.417	0.625	0.470	0.696	0.805	0.968	0.663
	SVM	0.392	0.001	0.443	0.407	0.113	0.108	0.244
	Random	0.258	0.083	0.273	0.250	0.096	0.041	0.167
F1-score	CRF	0.751	0.492	0.798	0.768	0.618	0.605	0.672
	RULE	0.493	0.631	0.605	0.630	0.467	0.848	0.612
	SVM	0.369	0.001	0.371	0.450	0.146	0.085	0.237
	Random	0.258	0.083	0.273	0.250	0.096	0.041	0.167

Table 12 Accuracies of CRF, heuristic-rules, SVM and random weighted classifier for each participants

	CRF	RULE	SVM	Random
S1	0.808 ± 0.071	0.386 ± 0.029	0.420 ± 0.342	0.347 ± 0.044
S2	0.756 ± 0.066	0.691 ± 0.023	0.100 ± 0.148	0.297 ± 0.035
S3	0.720 ± 0.054	0.524 ± 0.049	0.515 ± 0.170	0.349 ± 0.039
S4	0.808 ± 0.089	0.637 ± 0.046	0.355 ± 0.266	0.273 ± 0.045
S5	0.863 ± 0.024	0.419 ± 0.019	0.470 ± 0.349	0.393 ± 0.027
S6	0.772 ± 0.057	0.542 ± 0.028	0.434 ± 0.316	0.283 ± 0.032
S7	0.730 ± 0.100	0.644 ± 0.033	0.287 ± 0.282	0.346 ± 0.079
S8	0.616 ± 0.128	0.740 ± 0.051	0.140 ± 0.184	0.260 ± 0.021
S9	0.724 ± 0.197	0.437 ± 0.042	0.414 ± 0.235	0.299 ± 0.069
S10	0.809 ± 0.052	0.616 ± 0.035	0.584 ± 0.279	0.388 ± 0.022
Mean	0.761 ± 0.084	0.564 ± 0.036	0.372 ± 0.257	0.323 ± 0.041

classify actions. Furthermore, the results show that the standard deviation of the heuristic-rules method was the least, since its performance depended on only the defined heuristic rules but not the size of the training data. The standard deviation of CRF was considerably less than that of the SVM baseline, which indicates that CRF is more robust than the SVM baseline. Our results show that a machine learning approach cannot always outperform heuristic rules, unless the machine learning approach is well designed to consider the characteristics of the data.

We also calculated the accuracy improvement using the proposed CRF classifier compared with the heuristic-rules method or the SVM baseline, i.e., $improvement = \frac{A_{crf} - A_m}{A_m}$, where A is the accuracy and m can be either heuristic-rules method or SVM baseline. The average accuracy improvements over the heuristic-rules method, the SVM baseline and the random weighted classifier were 34.93%, 104.52%, and 135.60%, respectively. For the heuristic-rules method, the accuracy improvement for participant S1 was the greatest (109.32%), and the accuracy improvement for participant S8 is the least (−16.76%). This is because participant S1 was responsible for web development; he/she switched between different tasks frequently, such as coding, testing, and navigation. In such a complex working behavior, heuristic rules cannot work well. Conversely, participant S8 performed many documentation activities that can be easily inferred by heuristic rules, for example, by software application used. For the SVM baseline, the accuracy improvement for the participant S2 was the greatest (655.38%), and the accuracy improvement for the participant S10 was the least (38.57%). To measure whether the improvement was statistically significant, we also apply Wilcoxon signed-rank test (Wilcoxon 1945), and the p-values on the heuristic-rules method and the SVM baseline are 1.85E-12 and 1.83E-16 respectively, which indicates that the improvement was statistically significant at a confidence level of 99%.

Table 13 shows the Kappa values of CRF, the heuristic-rules method, and the SVM baseline for each participant. We confirmed that the Kappa values of CRF were superior compared to those of the heuristic-rules method and SVM baseline for all participants, except participant S8. The agreements between the predictions using CRF and the actual labels for all participants were at least moderate. Eight out of ten Kappa values using CRF were above 0.60, which indicates substantial agreement. However, the agreement levels using the heuristic-rules method were fair and moderate for all the participants, except

Table 13 Kappa statistics of CRF, heuristic-rules and SVM for each participant

	CRF	RULE	SVM
S1	0.73 (substantial)	0.24 (fair)	0.19 (slight)
S2	0.64 (substantial)	0.57 (moderate)	−0.08 (poor)
S3	0.56 (moderate)	0.36 (fair)	0.22 (fair)
S4	0.74 (substantial)	0.52 (moderate)	0.14 (slight)
S5	0.8 (substantial)	0.28 (fair)	0.18 (slight)
S6	0.68 (substantial)	0.43 (moderate)	0.17 (slight)
S7	0.64 (substantial)	0.55 (moderate)	0.00 (slight)
S8	0.51 (moderate)	0.63 (substantial)	−0.06 (poor)
S9	0.63 (substantial)	0.29 (fair)	0.14 (slight)
S10	0.69 (substantial)	0.46 (moderate)	0.32 (fair)

substantial for S8. The agreement levels using the SVM baseline were the poorest. For participants S2 and S8, the Kappa values of the SVM baseline were actually less than zero, which indicates poor agreement.

To compare the performance of CRF, the heuristic-rules method, SVM baseline, and random weighted classifier for each participant, we also calculated the macro-averaged precisions, recalls and F1-scores of different classifiers for each participant (see Table 14). We found that the macro-averaged precisions of CRF were greater than those of the heuristic-rules method for all participants except S8. For participant S8, the macro-averaged precision of CRF was virtually the same as that of the heuristic-rules method (0.66 *vs.* 0.67). Although the macro-averaged recalls of the heuristic-rules method for some participants (e.g., S2, S3) were greater than those of CRF, but the majority of the macro-averaged F1-scores of CRF were greater than those of the heuristic-rules method except for participants S2 and S8. For participants S2 and S8, the macro-averaged F1-scores of CRF were essentially the same as those of the heuristic-rules method (0.63 *vs.* 0.64, and 0.65 *vs.* 0.67). The macro-averaged metrics of the SVM baseline and random weighted classifier were all less than those of CRF

Table 14 Macro-averaged precisions, recalls and f1-scores of CRF, heuristic-rules and SVM for each participant

	Marco-precision				Marco-recall				Marco-F1-score			
	CRF	RULE	SVM	Random	CRF	RULE	SVM	Random	CRF	RULE	SVM	Random
S1	0.847	0.580	0.428	0.167	0.746	0.454	0.335	0.167	0.779	0.423	0.454	0.167
S2	0.698	0.651	0.132	0.167	0.664	0.676	0.124	0.167	0.631	0.644	0.228	0.167
S3	0.623	0.560	0.462	0.167	0.471	0.588	0.248	0.167	0.633	0.522	0.542	0.167
S4	0.790	0.600	0.442	0.167	0.769	0.649	0.378	0.167	0.801	0.638	0.489	0.167
S5	0.798	0.476	0.470	0.167	0.680	0.512	0.282	0.167	0.806	0.385	0.536	0.167
S6	0.789	0.559	0.384	0.167	0.621	0.626	0.256	0.167	0.636	0.511	0.504	0.167
S7	0.751	0.632	0.284	0.167	0.519	0.744	0.167	0.167	0.685	0.635	0.331	0.167
S8	0.659	0.670	0.117	0.167	0.680	0.700	0.151	0.167	0.645	0.669	0.208	0.167
S9	0.737	0.527	0.351	0.167	0.644	0.622	0.296	0.167	0.712	0.491	0.490	0.167
S10	0.796	0.525	0.583	0.167	0.634	0.628	0.363	0.167	0.705	0.584	0.669	0.167

and the heuristic-rules method. Overall, the proposed CRF achieved the best performance for the majority of the participants.

3.5.3 RQ1: Effectiveness of CRF—Performance for Each Label

We used box plot to compare the results of precision, recall, and F1-score for each label using CRF, the heuristic-rules method, SVM, and the random weighted classifier on each participant's data—see Figs. 3, 4 and 5, respectively. To plot the boxplot, we used the average metrics of nine rounds of testing for each participant. However, in some rounds of testing, there existed very minimal or no action records labeled by one label in the test data of a participant. For example, when we used the first seven working sessions of action records to predict the remaining three working sessions of action records for participant S2, there were only six "Documentation" action records in the S2's testing data. For such rarely occurring labels, the performance metrics of CRF, the heuristic-rules method, and SVM baseline were invalid, Therefore, we ignored these data when plotting the boxplot.

Figure 3 shows that the precisions of CRF were the best for all labels, and that the precisions of heuristic-rules were superior to those of SVM for all labels. There were some cases where heuristic-rules or SVM provided improved performance compared to CRF, for example, for the label "Web Browsing", participant S7 using the heuristic-rules method and participant S4 using SVM (i.e., the outliers in Fig. 3e). However, the median values (the band inside the box) of CRF were all greater than those of the heuristic-rules method and SVM. For the label "Debugging", the median value of CRF was marginally greater than that of the heuristic-rules method. However, in some cases, the precision of CRF for the label "Debugging" was less than that of the heuristic-rules method (see Fig. 3b). This is because

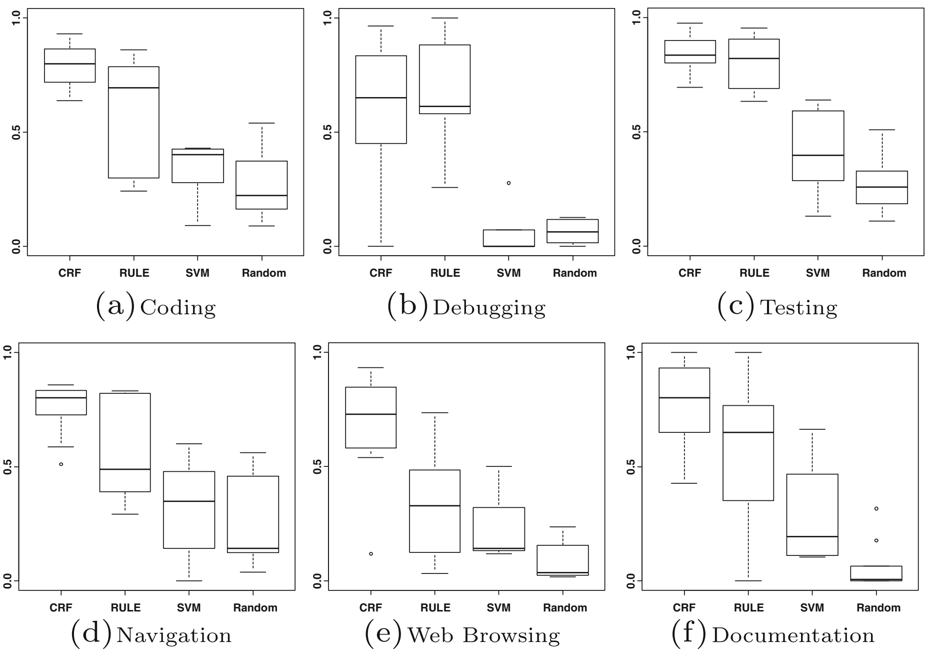


Fig. 3 Precisions for each label using CRF, heuristic-rules, SVM and random weighted classifier

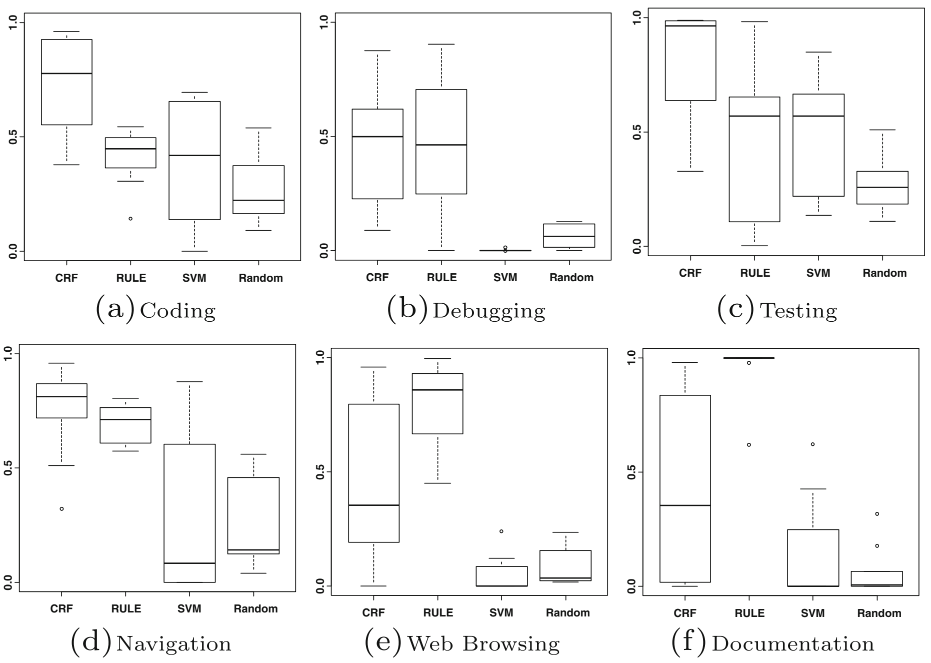


Fig. 4 Recalls for each label using CRF, heuristic-rules, SVM and random weighted classifier

developers performed the majority of debugging activities in the debugging perspective of IDEs, which can be easily inferred by heuristic rules.

For the results of recall, CRF demonstrated superior performance compared to the heuristic-rules method on all labels except the label "Web Browsing" and "Documentation", whereas the SVM baseline provided the poorest performance on all labels (see Fig. 4). For the label "Web Browsing" and "Documentation", we used heuristic-rules to address the majority of web pages and documents that existed in our collected data based on the data observation. Therefore, the heuristic-rules method achieved improved performance compared to CRF. However, this may not be generalizable on new data. For the result of F1-score, which combines precision and recall, CRF demonstrated superior performance compared to the heuristic-rules method except for the label "Debugging" and "Documentation", whereas the SVM baseline had the poorest F1-score on all labels (see Fig. 5). In some cases, the performance of SVM was unacceptable. For example, for the label "Debugging", the F1-scores of SVM are considerably less than those of CRF and the heuristic-rules. This could be because the SVM baseline does not consider context information when inferring developers' activities.

Table 15 lists the maximum, minimum, median, mean and standard deviation of F1-scores for each label. The maximum F1-scores of CRF are all greater than those of the baseline classifiers. The minimum F1-scores of all classifiers are typically small, because there could be no action records for a particular label when the size of the training data is small. In such cases, the F1-score of random weighted classifier is zero. The heuristic-rules method achieved considerably greater minimum F1-scores than other classifiers on labels "Debugging", "Navigation", and "Documentation". This is because the heuristic rules are not directly influenced by the size of the training data. The median and mean values

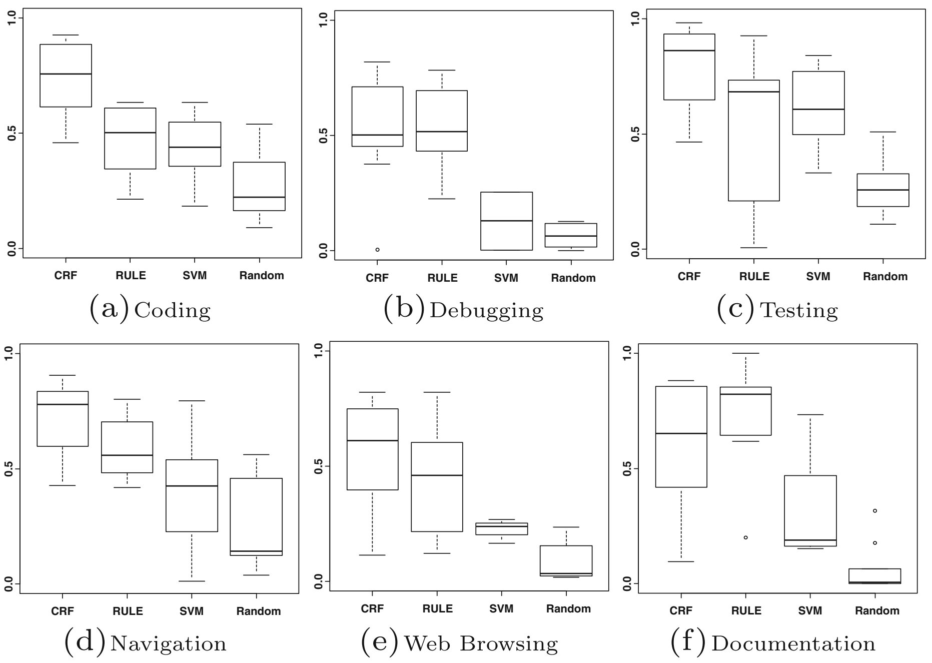


Fig. 5 F1-Scores for each label using CRF, heuristic-rules, SVM and random weighted classifier

of F1-scores of CRF are all greater than those of the baseline classifiers except for the label "Documentation" using the heuristic-rules method. The random weighted classifier achieves the smallest standard deviation, because its results depend on only the proportion of instances that belong to a class in the testing data. The standard deviations of CRF are all greater than those of the heuristic-rules method except for the label "Testing". This is because the results of the heuristic-rules method depend on its defined rules, not the size of the training data. The standard deviations of CRF are less than those of the SVM baseline except for the labels "Debugging" and "Web Browsing".

To measure whether the F1-score improvement of the proposed CRF classifier was significant over the heuristic-rules method and SVM baseline, we applied the Wilcoxon signed-rank test (Wilcoxon 1945) for each label on F1-score (see the last column in Table 15). We also calculated Cliff's delta,[4] which is a non-parametric effect size measure, to show the effect size of the difference between the two groups (see the last second column in Table 15). For the heuristic-rules method, the p-values for four labels, i.e. "Coding", "Debugging", "Testing" and "Navigation", were less than 0.05 and the Cliff's deltas were at least at the small effectiveness level, which means that the improvement of the proposed CRF classifier was statistically significant at the confidence level of 95%. However, the improvement for the other two labels, i.e., "Web Browsing" and "Documentation" was not statistically significant, because there existed participants for whom the F1-scores of the heuristic-rules method were greater than those of CRF. For the SVM baseline, the p-values were all less than 0.05 and the Cliff's deltas were all at the large effectiveness level, which

[4]Cliff defines a delta of less than 0.147, between 0.147 to 0.33, between 0.33 and 0.474, and above 0.474 as negligible, small, medium, and large effect size, respectively.

Table 15 Results of test statistics of F1-scores for each label

	Approach	max	min	median	mean	std	δ	pvalue
Coding	CRF	0.963	0.222	0.783	0.733	0.190	–	–
	RULE	0.666	0.143	0.489	0.472	0.139	0.778	0.002
	SVM	0.894	0.003	0.498	0.443	0.294	0.901	0.002
	Random	0.642	0.013	0.222	0.262	0.154	0.549	<0.001
Debugging	CRF	0.983	0.006	0.668	0.594	0.268	–	–
	RULE	0.827	0.225	0.526	0.560	0.159	0.188	0.020
	SVM	0.253	0.004	0.128	0.128	0.176	0.969	0.004
	Random	0.215	0.000	0.046	0.062	0.060	0.608	<0.001
Testing	CRF	1.000	0.085	0.877	0.788	0.230	–	–
	RULE	0.951	0.003	0.697	0.552	0.305	0.407	0.049
	SVM	0.996	0.000	0.814	0.638	0.365	0.580	0.010
	Random	0.692	0.022	0.251	0.264	0.124	0.333	<0.001
Navigation	CRF	1.000	0.057	0.792	0.726	0.190	–	–
	RULE	0.851	0.319	0.560	0.589	0.137	0.531	0.010
	SVM	0.964	0.001	0.644	0.504	0.338	0.827	0.002
	Random	0.611	0.022	0.147	0.263	0.189	0.587	<0.001
Web browsing	CRF	0.955	0.008	0.634	0.561	0.268	–	–
	RULE	0.887	0.089	0.455	0.434	0.232	−0.012	0.500
	SVM	0.607	0.002	0.072	0.204	0.241	0.875	0.004
	Random	0.395	0.000	0.041	0.083	0.089	0.600	<0.001
Documentation	CRF	1.000	0.029	0.790	0.673	0.301	–	–
	RULE	1.000	0.200	0.860	0.788	0.190	−0.469	0.992
	SVM	0.955	0.003	0.254	0.367	0.372	0.611	0.016
	Random	0.480	0.000	0.007	0.061	0.117	0.471	<0.001

Measurements are reported in the following columns: maximum, minimum, median, mean, standard deviation (std), Cliff's delta (δ) and p-value

means that the improvement of the proposed CRF classifier was statistically significant at the confidence level of 95%.

3.5.4 RQ2: The Effect of the Size of Training Data

Table 16 shows the accuracy of all participants predicted by the CRF classifier over different sizes of the training data ranging from 0.5 to 4.5 working days. Table 17 shows the accuracy of all participants predicted by the CRF classifier over the different sizes of the training data ranging from 100 to 1,000. For each participant, we also calculated the mean and standard deviation of the accuracies (see the last column in these two tables).

From these two tables, we found that the performance of the CRF classifier was significantly stable for some participants (i.e., S1, S2, S5, and S10). Their standard deviations were small. Given a extremely small size of training data, their accuracies were high. For example, when the size of the training data was only 100 action records, the accuracy of participant S1 was greater than 0.70. For other participants, when the size of the training data increased, the accuracy became stable. However, the required sizes of training data were

Table 16 Accuracies of different sizes of the training data (working session)

Day	0.5	1.0	1.5	2.0	2.5	3.0	3.5	4.0	4.5	Mean ± Std
S1	0.80	0.82	0.80	0.80	0.78	0.75	0.71	0.86	0.96	0.81 ± 0.07
S2	0.74	0.72	0.73	0.72	0.71	0.73	0.72	0.86	0.88	0.76 ± 0.06
S3	0.65	0.63	0.76	0.77	0.77	0.77	0.71	0.73	0.69	0.72 ± 0.05
S4	0.60	0.77	0.78	0.82	0.84	0.87	0.86	0.83	0.91	0.81 ± 0.08
S5	0.85	0.85	0.86	0.85	0.85	0.83	0.89	0.89	0.90	0.86 ± 0.02
S6	0.68	0.72	0.75	0.75	0.74	0.80	0.81	0.85	0.84	0.77 ± 0.05
S7	0.54	0.73	0.73	0.73	0.69	0.68	0.75	0.84	0.89	0.73 ± 0.09
S8	0.36	0.50	0.57	0.58	0.73	0.70	0.71	0.65	0.74	0.62 ± 0.12
S9	0.45	0.58	0.61	0.60	0.58	0.93	0.92	0.95	0.91	0.72 ± 0.19
S10	0.77	0.78	0.79	0.76	0.79	0.82	0.79	0.89	0.90	0.81 ± 0.05

different. For participants S3, S4, S6, and S7, when the number of action records in training data was 500 or greater, the accuracies became stable with small standard deviation. However, for participants S8 and S9, it required considerably more training data. For participant S8, when the size of the training data was greater than 2.5 days, the accuracy value became stable and equal to ∼0.7. The accuracy values of participant S9 were high (i.e., greater than 0.9) when the size of training data was greater than three days. We found that the size of the training data where the accuracy became stable depended on the label distribution in the training data and testing data. For example, there was no action record labeled by "Coding" in the previous 500 action records of participant S9 and the majority of actions were labeled as "Testing" and "Web Browsing". Therefore, the accuracies of participant S9 were extremely low (i.e., less than 0.3) when the size of the training data was less than 500 action records.

Overall, we confirmed the CRF classifier was robust and effective with small training data. However, to achieve acceptable performance, the labels in the training data must be diverse. To decrease the effort of manual annotation in practice, rather than labeling behavior data by consecutive working sessions (which could be uniform), we observed the action

Table 17 Accuracies of different sizes of the training data (number of action records)

	100	200	300	400	500	600	700	800	900	1000	Mean ± Std
S1	0.70	0.78	0.80	0.80	0.80	0.81	0.81	0.81	0.82	0.81	0.79 ± 0.03
S2	0.70	0.72	0.73	0.73	0.73	0.74	0.73	0.73	0.74	0.73	0.73 ± 0.01
S3	0.46	0.51	0.54	0.65	0.64	0.64	0.63	0.63	0.65	0.68	0.60 ± 0.07
S4	0.50	0.50	0.52	0.51	0.61	0.59	0.58	0.61	0.70	0.71	0.58 ± 0.07
S5	0.84	0.85	0.85	0.85	0.85	0.84	0.84	0.84	0.85	0.85	0.85 ± 0.00
S6	0.39	0.55	0.64	0.66	0.68	0.71	0.72	0.73	0.73	0.73	0.65 ± 0.10
S7	0.58	0.58	0.58	0.52	0.52	0.52	0.65	0.62	0.71	0.74	0.60 ± 0.07
S8	0.37	0.34	0.36	0.35	0.36	0.33	0.34	0.32	0.32	0.52	0.36 ± 0.05
S9	0.29	0.26	0.23	0.26	0.45	0.44	0.43	0.54	0.54	0.57	0.40 ± 0.12
S10	0.58	0.73	0.75	0.78	0.77	0.79	0.79	0.79	0.80	0.80	0.76 ± 0.06

Table 18 Accuracies of the cross-person prediction

	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	Mean ± Std.
S1	–	0.44	0.48	0.65	0.89	0.43	0.34	0.51	0.62	0.53	0.54 ± 0.15
S2	0.66	–	0.65	0.60	0.76	0.38	0.40	0.71	0.41	0.73	0.59 ± 0.14
S3	0.80	0.63	–	0.70	0.83	0.66	0.77	0.35	0.64	0.63	0.67 ± 0.13
S4	0.58	0.64	0.58	–	0.60	0.35	0.31	0.67	0.53	0.69	0.55 ± 0.13
S5	0.89	0.41	0.44	0.62	–	0.46	0.38	0.37	0.65	0.50	0.52 ± 0.16
S6	0.77	0.74	0.76	0.64	0.80	–	0.84	0.68	0.56	0.55	0.70 ± 0.10
S7	0.49	0.73	0.66	0.57	0.48	0.80	–	0.63	0.61	0.71	0.63 ± 0.10
S8	0.58	0.42	0.46	0.49	0.61	0.42	0.32	–	0.56	0.43	0.48 ± 0.09
S9	0.75	0.66	0.59	0.68	0.79	0.63	0.62	0.54	–	0.70	0.66 ± 0.07
S10	0.79	0.60	0.65	0.71	0.84	0.70	0.67	0.35	0.43	–	0.64 ± 0.15
All others	0.90	0.81	0.75	0.70	0.90	0.84	0.81	0.71	0.71	0.74	0.79 ± 0.074

distribution in working sessions and tried to prepare the training data with diverse labels from non-consecutive working sessions.

3.5.5 RQ3: Cross-Developer Prediction

In the experiment, we used each participant's action list as the training data to train a CRF model to predict the label of the other nine participants' actions. We also used all other participants' action lists as the training data to train a CRF model to predict the label of the remaining participant's actions. Tables 18 and 19 show the accuracies and Kappa values of cross-developer prediction using the proposed CRF classifier, respectively. For each cell in these two tables except the last column and the last row, the row participant indicates the training data and the column participant is the test data. In the last column of these two tables, each cell is the mean and standard deviation of the accuracy values (or Kappa values) of using the participant's data in a row to predict the other participants in the column. In

Table 19 Kappa values of the cross-person prediction

	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	Mea ± Std.
S1	–	0.42	0.44	0.64	0.89	0.40	0.31	0.48	0.60	0.49	0.52 ± 0.16
S2	0.64	–	0.63	0.59	0.75	0.35	0.38	0.69	0.39	0.71	0.57 ± 0.15
S3	0.79	0.62	–	0.69	0.83	0.65	0.76	0.32	0.62	0.59	0.65 ± 0.14
S4	0.56	0.63	0.55	–	0.59	0.32	0.28	0.65	0.51	0.67	0.53 ± 0.13
S5	0.90	0.40	0.43	0.62	–	0.44	0.35	0.34	0.64	0.47	0.51 ± 0.17
S6	0.76	0.73	0.74	0.63	0.80	–	0.83	0.67	0.55	0.51	0.69 ± 0.10
S7	0.47	0.72	0.63	0.56	0.46	0.79	–	0.62	0.60	0.68	0.61 ± 0.10
S8	0.56	0.40	0.42	0.48	0.60	0.40	0.29	–	0.55	0.38	0.45 ± 0.09
S9	0.74	0.65	0.56	0.68	0.79	0.62	0.60	0.52	–	0.67	0.65 ± 0.08
S10	0.78	0.59	0.63	0.71	0.84	0.69	0.66	0.32	0.41	–	0.63 ± 0.16
All others	0.90	0.81	0.74	0.70	0.91	0.83	0.81	0.69	0.69	0.72	0.78 ± 0.08

the last row of these two tables, each cell is the accuracy (Kappa values) of using all other participants' data to predict a particular participant in the column.

The accuracy results of cross-person prediction using CRF are acceptable. In the results using one participant to predict the other participant, there are many cases where the accuracy results were close to or higher than 60%. The mean accuracy of using the model trained by the action lists of participant S6 to predict other participants' actions was the greatest (i.e., 70%). Using all other participants to predict the remaining one, all accuracy results were greater than 0.70. This shows that cross-developer prediction using CRF is feasible. However, we would like to point out that there were only ten developers in our experiment and they all used Java as the main programming language. More data from different developers and projects is required to verify whether the cross-developer prediction is feasible in general. We are considering to collect more data from different developers in the future.

Furthermore, the accuracy of cross-developer prediction can indicate the similarity between two developers' behavior or working context. For example, the accuracy results of cross-developer prediction between participants S1 and S5 are approximately 90%. Indeed, we find S1 and S5 are working on the same module of the same project. They closely collaborated in their daily work. This indicates opportunities for mining and transferring best practices (usually not explicitly documented) among developers.

From the Kappa metrics in Table 19, we can observed that the results of the proposed classifier trained using all other participants' data for predicting the remaining participant had substantial agreement with the actual labels for six participants, and almost perfect agreement for four participants (S1, S5, S6, and S7). This is because all other participants' data exposes the classifier to diverse behavior and labels for accurate prediction. For the results of the proposed classifier trained using one participants' data for predicting others, 31.1% of the cases had moderate agreement and 51.1% had substantial or virtually perfect agreement. For 17.8% of the cases with below-moderate agreement, it was mainly because the training data of one participant did not include the labels in the other participant's data. For example, using S8's data to predict S7's data, the agreement was only 0.29 (fair). This is because S8 performed many "Documentation" actions, whereas S7 does had only a small number of "Documentation" actions in his working data.

4 Discussion

In this section, we introduce implications based on the proposed approach and experiment results:

4.1 Mining and Using Behavior Patterns for Software Development

Software artifacts (e.g., code, bug reports) are well archived, and many approaches have been proposed to mine and use the knowledge in software artifacts to assist software development. Conversely, behavior data in software development tasks are discarded, even though they have great potential to improve software development practices. In fact, a developer study by Wang et al. (2011) discovered distinct phases and recurring patterns in the process of feature location tasks. They further show that the phases and patterns discovered from senior developers' behavior can be taught to junior developers. Once junior developers learn better feature location practices, their performance in feature location tasks can be improved. The study by Wang et al. was conducted in controlled experiments and their behavior pattern discovery and analysis were performed manually. Our study echoes their findings in real development tasks, and we show the potential of using a machine learning approach to infer phases and behavior patterns in the process of software development tasks.

To verify this potential, we chose two participants (S1 and S2) and collected one more week's of interaction data in their work. Then, we used the trained CRF classifier to predict their activities in that week. We found that there were more action records labeled in "Coding" and "Web Browsing" (37% and 35%, respectively) than other classes of actions for participant S1, whereas for participant S2, "Coding", "Testing" and "Debugging" were the three most common action records accounting for 34%, 29%, and 22% of the total number of action records, respectively. We let the two developers examine the prediction results. They agreed that our analysis result was consistent with their real development activities. S1 stated *"I was required to implement a new function that used a Java library I had never used before. Therefore, during this week I read several online tutorials and developed a demo application to learn this library."* Consequently, he expended considerable amount of time on "Coding" and "Web Browsing". S2's tasks in that week were mainly bug fixing because several bugs were reported in the module for which he was responsible. Hence, he expended the majority of his time on investigating the root causes of the bugs and validating his code changes.

Our study indicates that a knowledge repository of behavior data can be as useful as an artifact repository for software development. The proposed approach can be the first step towards validating the feasibility and applicability of a knowledge repository of behavior data for software development. Such a behavior knowledge repository can not only archive the behavior data, but also be mined to discover good or bad practices in software development tasks. As such, this knowledge repository could complement the current artifacts-based evaluation of developers' performance with new behavior-oriented aspects. It could facilitate addressing some key challenges in software engineering knowledge management, such as the best practice of completing tasks in a timely fashion despite the departure of key developers. Behavior patterns in the knowledge repository could enable a more systematic transfer of best practices between developers, for example, by recommending better practices to improve a developer's current less-effective practices.

4.2 Deployment Challenges of Behavior Tracking Systems

The importance of studying developers' behavior has long been recognized in many studies. However, compared with software artifacts that have been well studied, behavior data in software engineering has been much less explored. This can be attributed to the difficulties in collecting and analyzing behavior data. The proposed approach is an attempt to mitigate these difficulties in the study of developers' behavior. First, by operating-system level instrumentation, it can be relatively easy to adapt our approach in different study settings. Secondly, using a machine learning approach can reduce the amount of human efforts required to manually examine the data.

We would like to highlight some technical prerequisites in deploying the proposed approach in real work settings. First, although operating-system level instrumentation improves the generalizability of data collection in different working environments, it remains necessary to investigate the applications' support for operating-system accessibility APIs before deploying the data collection framework in a specific working environment, similar to the survey we conducted to test the generalizability of our ACTIVITYSPACE framework[1] for different applications and operating systems. The focus is to confirm what information applications expose to the accessibility APIs and whether the exposed information is sufficient for further abstraction. Secondly, although a machine learning approach can potentially reduce the human efforts for analyzing behavior data in the end, the human labeling of behavior data is a mandatory step in the beginning. The goal is to obtain high-quality training data with effective behavior diversity and precise labels. These initial human efforts remain significant. A way to mitigate this effort is to devise statistical methods to estimate how much data must be annotated by observing the overall data required to train a high-quality machine learning model.

We also experienced interesting ethical debates regarding deploying a behavior data analysis tool such as the proposed approach in the company. The company managers appreciated the value of behavior analysis and a behavior knowledge repository to improve the project management and developers' practices in their daily work. Developers also acknowledged the potentials of behavior analysis for supporting their information needs in their work (Bao et al. 2015b). However, they are reluctant to openly release their working data with others. With the growth of the open source movement, developers have significantly less problem sharing their code. Traditionally, developers share how they code in video tutorials. Recently, experienced developers share how they code in real time using live programming broadcast on Reddit. This suggests that it is not completely impossible for developers to share their behavior data if the developer's privacy is protected and the data is used in a constructive manner to facilitate developer growth.

5 Threats to Validity

Threats to Internal Validity One of the threats to internal validity is the annotation errors in the manually labeled action list. We collected more than 50,000 action records from ten developers' daily work. Owing to the large quantity of action records to be labeled and human misunderstanding of developers' activities, there could exist annotation errors. To minimize errors, two human annotators participated in labeling all the records and expended approximately 80 h to complete this task. The two annotators independently annotated the same data; then, the disagreements were resolved by discussion and assistance from developers. Furthermore, we sampled 10% of the action records to let developers verify that our annotation results were accurate. In retrospect, we could improve the participants' validation step by intentionally planting false information (e.g., annotations that are obviously false), and verify if the participants identify these errors when they validate the data.

Another threat to internal validity is the definition of the development-activity labels in our coding schema. In this work, we considered only six basic development activities: *coding*, *debugging*, *testing*, *navigation*, *web search*, and *documentation*. However, developers' behavior can be more complex, which could involve behaviors not covered by the defined six development activities. There could also potentially be activities that are more difficult to recognize, especially when the annotators lack domain-specific knowledge. Furthermore, there could be levels of ambiguities among the six development activities, which could lead to the erroneously labeling results. According to our experiment results, we believe that the six development activities in our schema are reasonable and can address the developers' daily working behavior in our collected data. However, for other studies with different purposes or involving special activities, other labels could be defined for training the CRF model.

Threats to External Validity The major threat to external validity is the generalizability of our results. In this study, we collected approximately 240 effective working-hours behavior data from ten developers' one-week work; these 10 developers were from three different projects. Although the data was collected in real work settings, we acknowledge the limitation of the data and the limitation of the developers. In the future, we will reduce this threat by collecting more data from more developers and projects.

Threats to Construct Validity In this study, we use accuracy, precision, recall, and F1-scores as the main evaluation metrics; these are also widely used by past software engineering studies to evaluate the effectiveness of a prediction technique (Nguyen et al. 2012; Wu et al. 2011; Xia et al. 2013). Thus, we believe there is minimal threat to construct validity.

6 Related Work

Developers' Behavior Analysis There exist many studies that investigate and model developers' behavior in software development tasks such as debugging (von Mayrhauser and Vans 1997; Lawrance et al. 2013; Sillito et al. 2005), feature location (Wang et al. 2011), program comprehension (Corritore and Wiedenbeck 2001; Ko et al. 2006; Li et al. 2013; Robillard et al. 2004; Lawrance et al. 2008; Piorkowski et al. 2011; Fritz et al. 2014; Minelli et al. 2015), using unfamiliar APIs (Dekel and Herbsleb 2009; Duala-Ekoko and Robillard 2012), and testing (Beller et al. 2015). The data collection methods of these studies usually include screen-capture video, think-aloud, instrumentation, and survey. However, the data analysis in these studies typically requires a significant amount of human effort (Bao et al. 2017). For example, the researchers must encode and transcript the video and other observation data manually. Furthermore, for ease of understandability, the collected data must be abstracted in the data transcription process. For example, Coman and Sillitti (2009) use the code access location (e.g. methods, classes, files) to segment the development sessions into task-related subsections automatically; however, they only consider developers' interaction in IDE and no semantic labels are assigned to the subsections.

We represent an approach using the CRF model that can segment and label the low-level interaction data automatically. Although training the CRF model does require a certain amount of human annotation, our study shows that a small amount of training data can achieve effective classification results. Therefore, the human efforts required in the proposed approach can be considerably less than the efforts to transcribe all the behavior data manually. Furthermore, the proposed approach does not rely on any application-specific support to collect the interaction data. Therefore, the proposed approach provides a generic and less demanding solution to study developers' behavior in software engineering.

Sequential Data Analysis Exploratory Sequential Data Analysis (ESDA) is a popular technique that has been applied in many behavior analysis studies. ESDA is any empirical undertaking seeking to analyze systems, environment, and/or behavior data where the sequential integrity of events has been preserved (Sanderson and Fisher 1994). ESDA can be subdivided into two basic categories: (1) techniques sensitive to sequentially separated patterns of events, for example, Fisher's cycles (Fisher 1991) and Lag sequential analysis (LSA) (Sackett 1978); (2) techniques sensitive to strict transitions between events, for example, Maximal Repeating Pattern Analysis (MRP) (Siochi and Hix 1991), Log linear analysis.

There are also many sequential data segmentation and labeling techniques including Hidden Markov Model (HMM) (Rabiner 1989), Maximum Entropy Markov Model (MEMM) (McCallum et al. 2000), and Conditional Random Field (CRF) (Lafferty et al. 2001). These techniques have been applied in many fields, such as bioinformatics (Durbin et al. 1998), natural language processing (Berger et al. 1996) and speech recognition (Lawrence 2008).

Researchers have used these segmentation and labeling techniques to mine game strategies. For example, Dereszynski et al. (2011) assumed equal length segments of 30 s, and applied HMM to learn the segment labels. Gong et al. (2012) proposed a CRF-based method to segment and label action list of a real-time strategy game called *StarCraft*. Their segmentation method does not consider human annotated ground truths in training and evaluation.

Because CRFs have proven to provide superior performance compared to both HMM and MEMM on many real-world sequence segmentation and labeling tasks (Lafferty et al. 2001; Pinto et al. 2003) and have been successfully applied to game action list segmentation and labeling (Gong et al. 2012), we chose CRF to segment and label the developers' action list in our work.

7 Conclusion

In this paper, we presented an approach for labeling and segmenting the time series of developers' interaction with the complete working environment in real software projects. We conducted an experiment using ten developers' daily working data in a real software project, which included approximately 240 effective working hours and 50,000 action records. Our experiment confirmed the following: (1) Compared with the classic classifier SVM, the proposed CRF-based approach provided superior performance with approximately 75% prediction accuracy. This is because the CRF model considers the neighboring context in model training. (2) The proposed approach was robust over the size of the training data. It required only a small amount of training data to produce accurate classification results. (3) Cross-developer prediction using the proposed approach achieved acceptable accuracy (i.e., approximately 60%). Overall, our experiment confirmed that the proposed approach was more generic and less demanding compared with existing approaches on studying developers' behavior. Therefore, the proposed approach is considerably easier to be replicated in new studies, and provides a foundation for mining high-level behavior patterns in software development. In the future, we plan to collect more data from more developers and different projects to further validate the proposed approach.

Acknowledgments The authors thank to all the developers in Hengtian who participated in this study. This work was partially supported by NSFC Program (No. 61602403 and 61572426).

References

Anvik J, Hiew L, Murphy GC (2006) Who should fix this bug? In: Proceeding of the 28th international conference on software engineering (ICSE), pp 361–371

Bao L, Ye D, Xing Z, Xia X (2015a) ActivitySpace: a remembrance framework to support interapplication information needs. In: Proceedings of 30th IEEE/ACM international conference on automated software engineering (ASE), pp 864–869

Bao L, Xing Z, Wang X, Zhou B (2015b) Tracking and analyzing cross-cutting activities in developers' daily work. In: Proceedings of 30th IEEE/ACM international conference on automated software engineering (ASE), pp 277–282

Bao L, Li J, Xing Z, Wang X, Xia X, Zhou B (2017) Extracting and analyzing time-series HCI data from screen-captured task videos. Empir Softw Eng 22(1):134–174

Beller M, Gousios G, Panichella A, Zaidman A (2015) When, how, and why developers (do not) test in their IDEs. In: Proceedings of the 2015 10th joint meeting on foundations of software engineering (FSE), pp 179–190

Berger AL, Pietra VJD, Pietra SAD (1996) A maximum entropy approach to natural language processing. Comput Linguist 22(1):39–71

Chang T-H, Yeh T, Miller R (2011) Associating the visual representation of user interfaces with their internal structures and metadata. In: Proceedings of the 24th annual ACM symposium on user interface software and technology (UIST), pp 245–256

Coman ID, Sillitti A (2009) Automated segmentation of development sessions into task-related subsections. Int J Comput Appl 31(3):159–166

Corritore CL, Wiedenbeck S (2001) An exploratory study of program comprehension strategies of procedural and object-oriented programmers. Int J Hum Comput Stud 54(1):1–23

Dekel U, Herbsleb JD (2009) Reading the documentation of invoked API functions in program comprehension. In: Proceedings of 17th IEEE international conference on program comprehension (ICPC), pp 168–177

Dereszynski EW, Hostetler J, Fern A, Dietterich TG, Hoang T-T, Udarbe M (2011) Learning probabilistic behavior models in real-time strategy games. In: AAAI conference on artificial intelligence and interactive digital entertainment

Dewan P, Agarwal P, Shroff G, Hegde R (2009) Distributed side-by-side programming. In: Proceedings of the 2009 ICSE workshop on cooperative and human aspects on software engineering, pp 48–55

Duala-Ekoko E, Robillard MP (2012) Asking and answering questions about unfamiliar APIs: an exploratory study. In: Proceedings of 34th international conference on software engineering (ICSE), pp 266–276

Durbin R, Eddy SR, Krogh A, Mitchison G (1998) Biological sequence analysis: probabilistic models of proteins and nucleic acids. Cambridge University Press, Cambridge

Fisher C (1991) Protocol analyst's workbench: design and evaluation of computer-aided protocol analysis, PhD thesis, Carnegie-Mellon University, Pittsburgh

Fleiss JL (1971) Measuring nominal scale agreement among many raters. Psychol Bull 76(5):378

Fritz T, Shepherd DC, Kevic K, Snipes W, Bräunlich C (2014) Developers' code context models for change tasks. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering (FSE), pp 7–18

Gong W, Lim E-P, Achananuparp P, Zhu F, Lo D, Chua FCT (2012) In-game action list segmentation and labeling in real-time strategy games. In: Proceedings of IEEE conference on computational intelligence and games (CIG), pp 147–154

Hundhausen CD, Brown JL, Farley S, Skarpas D (2006) A methodology for analyzing the temporal evolution of novice programs based on semantic components. In: Proceedings of the ACM international computing education research workshop, pp 59–71

Hurst A, Hudson SE, Mankoff J (2010) Automatically identifying targets users interact with during real world tasks. In: Proceedings of the 15th international conference on intelligent user interfaces (IUI), pp 11–20

Ko AJ, Myers BA (2005) A framework and methodology for studying the causes of software errors in programming systems. J Vis Lang Comput 16(1):41–84

Ko AJ, Myers B, Coblenz MJ, Aung HH et al (2006) An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. IEEE Trans Softw Eng 32(12):971–987

Koru AG, Ozok A, Norcio AF (2005) The effect of human memory organization on code reviews under different single and pair code reviewing scenarios. In: ACM SIGSOFT software engineering notes, vol 30, pp 1–3

Lafferty J, McCallum A, Pereira F et al (2001) Conditional random fields: probabilistic models for segmenting and labeling sequence data. In: Proceedings of the eighteenth international conference on machine learning (ICML), vol 1, pp 282–289

Lawrence R (2008) Fundamentals of speech recognition. Pearson Education, India

Lawrance J, Bellamy R, Burnett M, Rector K (2008) Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks. In: Proceedings of the SIGCHI conference on human factors in computing systems (CHI), pp 1323–1332

Lawrance J, Bogart C, Burnett M, Bellamy R, Rector K, Fleming SD (2013) How programmers debug, revisited: an information foraging theory perspective. IEEE Trans Softw Eng 39(2):197–215

Le T-DB, Lo D (2013) Will fault localization work for these failures? An automated approach to predict effectiveness of fault localization tools. In: Proceedings of IEEE international conference on software maintenance (ICSM), pp 310–319

Li H, Xing Z, Peng X, Zhao W (2013) What help do developers seek, when and how? In: Proceedings of 20th working conference on reverse engineering (WCRE), pp 142–151

Maiga A, Ali N, Bhattacharya N, Sabané A, Guéhéneuc Y-G, Antoniol G, Aïmeur E (2012) Support vector machines for anti-pattern detection. In: Proceedings of the 27th IEEE/ACM international conference on automated software engineering (ASE), pp 278–281

McCallum A, Freitag D, Pereira FC (2000) Maximum entropy Markov models for information extraction and segmentation. In: Proceedings of the seventeenth international conference on machine learning, vol 17, pp 591–598

Minelli R, Mocci A, Lanza M (2015) I know what you did last summer: an investigation of how developers spend their time. In: Proceedings of IEEE international conference on program comprehension (ICPC), pp 25–35

Nguyen AT, Nguyen TT, Nguyen HA, Nguyen TN (2012) Multi-layered approach for recovering links between bug reports and fixes. In: Proceedings of the 20th ACM SIGSOFT international symposium on foundations of software engineering (FSE), pp 63–73

Pinto D, McCallum A, Wei X, Croft WB (2003) Table extraction using conditional random fields. In: Proceedings of the 26th annual international ACM SIGIR conference on research and development in information retrieval, pp 235–242

Piorkowski D, Fleming SD, Scaffidi C, John L, Bogart C, John BE, Burnett M, Bellamy R (2011) Modeling programmer navigation: a head-to-head empirical evaluation of predictive models. In: Proceedings of IEEE symposium on visual languages and human-centric computing (VL/HCC), pp 109–116

Rabiner LR (1989) A tutorial on hidden markov models and selected applications in speech recognition. Proc IEEE 77(2):257–286

Robillard MP, Coelho W, Murphy GC (2004) How effective developers investigate source code: an exploratory study. IEEE Trans Softw Eng 30(12):889–903

Sackett GP (1978) Observing behavior: theory and applications in mental retardation. University Park Press, Baltimore

Safer I, Murphy GC (2007) Comparing episodic and semantic interfaces for task boundary identification. In: Proceedings of conference of the centre for advanced studies on collaborative research, pp 229–243

Sanderson PM, Fisher C (1994) Exploratory sequential data analysis: foundations. Hum Comput Interact 9(3–4):251–317

Sillito J, De Voider K, Fisher B, Murphy G (2005) Managing software change tasks: an exploratory study. In: International symposium on empirical software engineering, p 10

Siochi AC, Hix D (1991) A study of computer-supported user interface evaluation using maximal repeating pattern analysis. In: Proceedings of the SIGCHI conference on human factors in computing systems (CHI), pp 301–305

Sun C, Lo D, Wang X, Jiang J, Khoo S-C (2010) A discriminative model approach for accurate duplicate bug report retrieval. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering (ICSE), pp 45–54

Thung F, Lo D, Jiang L (2012) Automatic defect categorization. In: Proceedings of 19th working conference on reverse engineering (WCRE), pp 205–214

Tian Y, Sun C, Lo D (2012) Improved duplicate bug report identification. In: Proceedings of 16th European conference on software maintenance and reengineering (CSMR), pp 385–390

Vakilian M, Chen N, Negara S, Rajkumar BA, Bailey BP, Johnson RE (2012) Use, disuse, and misuse of automated refactorings. In: Proceedings of 34th international conference on software engineering (ICSE), pp 233–243

von Mayrhauser A, Vans AM (1997) Program understanding behavior during debugging of large scale software. In: Proceedings of the seventh workshop on empirical studies of programmers, pp 157–179

Wang J, Peng X, Xing Z, Zhao W (2011) An exploratory study of feature location process: distinct phases, recurring patterns, and elementary actions. In: Proceedings of 27th IEEE international conference on software maintenance (ICSM), pp 213–222

Wilcoxon F (1945) Individual comparisons by ranking methods. Biom Bull 1(6):80–83

Wu R, Zhang H, Kim S, Cheung S-C (2011) Relink: recovering links between bugs and changes. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on foundations of software engineering (ESEC/FSE)

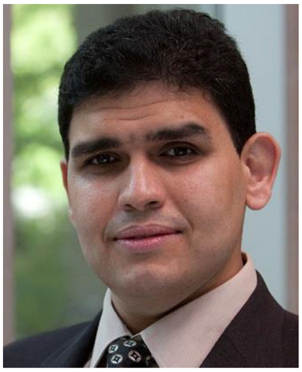
Xia X, Lo D, Wang X, Yang X, Li S, Sun J (2013) A comparative study of supervised learning algorithms for re-opened bug prediction. In: Proceedings of 17th European conference on software maintenance and reengineering (CSMR)

Lingfeng Bao is currently a postdoctoral research fellow in the College of Computer Science and Technology, Zhejiang University. He received his B.E. and PhD degrees from the College of Software Engineering, Zhejiang University in 2010 and 2016. His research interests include software analytics, behavioral research methods, data mining techniques, and human computer interaction.

Zhenchang Xing is the senior lecturer at the research school of computer science, Australian National University, Australia. Dr. Xing's research interests include software engineering and human-computer interaction. His work combines software analytics, behavioral research methods, data mining techniques, and interaction design to understand how developers work, and then build recommendation or exploratory search systems for the timely or serendipitous discovery of the needed information.

Xin Xia received his PhD degree in computer science from the College of Computer Science and Technology, Zhejiang University, China in 2014. He is currently a postdoctoral research fellow in the software practices lab at the University of British Columbia, Canada. His research interests include software analytic, empirical study, and mining software repository.

David Lo received his PhD degree from the School of Computing, National University of Singapore in 2008. He is currently an Associate Professor in the School of Information Systems, Singapore Management University. He has close to 10 years of experience in software engineering and data mining research and has more than 200 publications in these areas. He received the Lee Foundation Fellow for Research Excellence from the Singapore Management University in 2009, and a number of international research awards including several ACM distinguished paper awards for his work on software analytics. He has served as general and program co-chair of several prestigious international conferences (e.g., IEEE/ACM International Conference on Automated Software Engineering), and editorial board member of a number of high-quality journals (e.g., Empirical Software Engineering).



Ahmed E. Hassan is the Canada Research Chair (CRC) in Software Analytics, and the NSERC/BlackBerry Software Engineering Chair at the School of Computing at Queen's University, Canada. His research interests include mining software repositories, empirical software engineering, load testing, and log mining. He received a PhD in Computer Science from the University of Waterloo. He spearheaded the creation of the Mining Software Repositories (MSR) conference and its research community. He also serves on the editorial boards of IEEE Transactions on Software Engineering, Springer Journal of Empirical Software Engineering, and PeerJ Computer Science. Contact ahmed@cs.queensu.ca. More information at: http://sail.cs.queensu.ca/.