Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Inference of Development Activities from Interaction with Uninstrumented Applications

Lingfeng Bao · Zhenchang Xing · Xin Xia · David Lo · Ahmed E. Hassan

the date of receipt and acceptance should be inserted later

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 often use various different applications including IDEs, Web Browsers, documentation software (such as Office Word, Excel, PDF applications, etc), and many others to complete their tasks. This creates significant challenges in collecting and analyzing developers' behavior data. Researchers usually instrument the software tools to log the developers' behavior for further studies. This is feasible for studies on development activities using specific software tools. However, instrumenting all software tools commonly used in real work settings is very 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. The abstraction is often performed manually or based on simple heuristics. In this paper, we propose an approach to tackling the above two challenges in collecting and ana-

Lingfeng Bao
College of Computer Science, Zhejiang University, Hangzhou, China
E-mail: lingfengbao@zju.edu.cn

Zhenchang Xing
School of Computer Engineering, Nanyang Technological University, Singapore
E-mail: zcxing@ntu.edu.sg

Xin Xia (✉)
College of Computer Science and Technology, Zhejiang University, China
Department of Computer Science, University of British Columbia, Canada
E-mail: xxia@zju.edu.cn, xxia02@cs.ubc.ca

D. Lo
School of Information Systems, Singapore Management University, Singapore
E-mail: davidlo@smu.edu.sg

A. E. Hassan
School of Computing, Queen's University, Canada
E-mail: ahmed@cs.queensu.ca

lyzing developers' behavior data. First, we propose to use our ActivitySpace framework to improve the generalizability of data collection. ActivitySpace uses operating-system level instrumentation to track the developers' interactions with a wide range of applications in the real work settings. Second, we use a machine learning approach to reduce the human efforts to abstract low-level behavior data. Specially, considering the sequential nature of the interaction data, we propose a Condition Random Field (CRF) based approach to segment and label developers' low-level actions into a set of basic but meaningful development activities. To test the generalizability of our data collection approach, we deploy the ActivitySpace framework in our industry partner's company and collect the real working data from 10 professional developers' one-week work in three real software projects. The experiment with the collected data shows that with the 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 needed for behavior data analysis.

1 Introduction

Researchers in the software engineering community have great interests 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).

While previous researches often produce impressive results in their own environments, their data collection and analysis methods often cannot be easily transferred to a new study context.

- First, 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.
- Second, most studies require *application-specific support* for collecting the developers' behavior data. For example, Vakilian et al. (Vakilian et al. 2012) rely on the Eclipse IDE feature to record the refactorings that are applied by developers. In most 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.
- Third, existing studies usually 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 will render these activity-specific heuristics irrelevant.

In this paper, our objective is to investigate whether we can use the machine learning approach to infer a *set* of basic development activities in *real work settings* from a developer's interaction with applications in the *entire* working environment *without application-specific support*. Inference of the basic development activities from the developer's low-level actions would 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.

- 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 low enough level to obviate application-specific support while collecting detailed enough information for inferring the development activities at a higher-level of abstraction.
- Software development involves a wide range of development activities, such as coding, debugging, testing, searching, and documentation. These activities often 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. 2015b,a). 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 log a time series of action records. Each action record logs the mouse or keyboard action, and the relevant window information and the focused UI component information. For example, when a developer is viewing a Java source file *example.java* in Eclipse, he/she clicks *Open Resource* menu. ActivitySpace will generate 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 could infer the intentions and activities of 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 are being debugged. Given a time series of these action records, we could infer that the developer is debugging the program by 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 time periods when the developer debugs the software from those when a developer performs other development activities that may also involve low-level debugging, code editing, and web browsing actions? For example, for the time period during which a developer is coding, he/she may integrate a code snippet found on the Web into the project, and execute the code to ensure that the reused code works as expected. As another example, for the time 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 see it in action. Although these development activities have similar low-level actions, the developer's intention would be very different, i.e., debugging versus coding versus web browsing. Heuristics-based rules that only examine individual actions but not the surrounding behavior context would be unlikely to handle 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 will be much easier to reveal the developer's intention and activities by examining the surrounding actions before and after an individual action. For example, if coping-pasting a piece of code from web browse to IDE occurs in between many coding actions, we would consider this copy-paste action as part of a coding activity. In contrast, if the copy-paste action occurs in between many web page browsing actions, we would 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 which represents the collective meaning of the actions in this segment. Knowing these action segments and their semantic labels would 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 Models (HMMs) (Rabiner 1989), Maximum Entropy Markov Models (MEMMs) (McCallum et al. 2000), and Conditional Random Fields (CRFs) (Lafferty et al. 2001). CRFs outperform both HMMs and

MEMMs on many real-world sequence segmentation and labeling tasks (Lafferty et al. 2001; Pinto et al. 2003). CRFs have 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 our approach using week-long data from 10 professional developers in a real work setting.

The remainder of the paper is structured as follows. Section 2 describes our approach including data collection, feature selection and CRF model training. Section 3 reports our experiment result to verify our approach. Section 4 discusses some 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 shows 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 action records as *action list* in this work. Then, our 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 Conditional Random Fields (CRFs). In the prediction phase, we use the trained CRF classification model to automatically assign the development-activity labels to the actions in the unseen action lists.

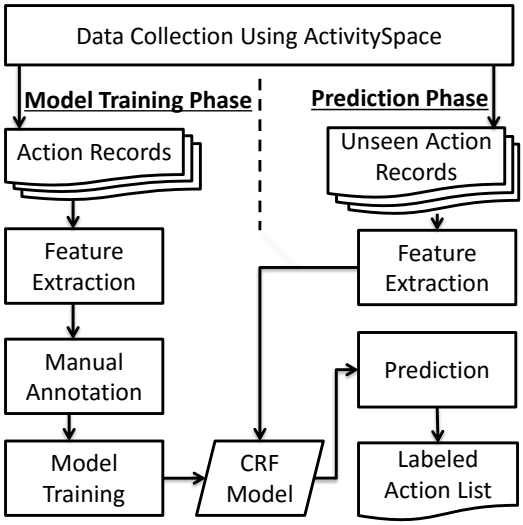


Fig. 1: The CRF-based framework

2.1 Data Collection

To improve the generalizability of our 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 application specific support, ActivitySpace uses operating-system (OS) window APIs and accessibility API 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 so that it does not interfere with one's normal work (See Section 3.3.2 for a pilot study of ActivitySpace performance).

ActivitySpace can monitor three types of mouse events (including mouse move, mouse wheel, and mouse click), and two types of keyboard events (including normal keystrokes like alphabetic and numeric keys, and shortcut keystrokes like "Ctrl+F" (Search or Find), "Ctrl+O" (Eclipse shortcut key for quick outline)). Basic window information includes the position of 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

Table 1: Examples of action records that are collected by ActivitySpace

Timestamp	T_1	...	T_n
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

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 help us infer the developers' 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 show that at time T_1 the developer selects the file "JSTreeDao.java" in "Project Explorer" in Eclipse, and at time T_n the developer searches *java calendar* on Google in Firefox.

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] show that a wide range of applications expose their internal HCI data to operating-system accessibility APIs. Therefore, operating-system level instrumentation like ActivitySpace allows our data collection method to be applied in working environments with different applications and data collection requirements.

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

2.2 Feature Selection

As seen in Table 1, the HCI data collected by ActivitySpace is rather detailed and fine-grained. To make the raw HCI data easier to understand and label manually, 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**: it can be directly inferred from process name, such as eclipse, firefox.
- **AppType**: we categorize common-used applications into 6 predefined application types: IDE, Web Browser, Office, Text Editor, PDF Reader and Others.
- **WindowName**: it 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 in which the developer is working. For example, in Eclipse, the window title contains the perspective information, such as "Java", "Java EE" or "Debug". Similar to Eclipse, 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 kept as separate keyboard input. For mouse events, this feature is set to "NA".
- **UIType, UIName, UIValue, ParentUI**: all 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 is editing the source file "pricingoptionSearch.xhtml" in Eclipse. Then, he switches to Chrome to access the modified web page and tries out some input on the web page. After that, he switches back to Eclipse. He uses shortcut key "Ctrl+Shift+R" to activate the "Open Resource" dialog in Eclipse to locate another file containing pricing option, then he uses shortcut key "Ctrl+F" to search the keyword "option" in file *pricingoptionSearch.xhtml*. Next, he reads

a web page about AJAX on StackOverflow. Finally, he debugs the "pricingoptionSearch.xhtml" file in Eclipse debug perspective.

2.3 Manual Annotation

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

- **Coding (C)**: the 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 IDEs or text editors.
- **Debugging (D)**: the developers' debugging activities refer to not only using debugging tools in IDEs, but also activities like viewing bug reports in documentation software or web browsers.
- **Testing (T)**: the 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)**: the developers' navigation activities refer to browsing, searching and locating some resources in the IDEs, such as browsing source files in the "Project Explorer", searching some words in a file, or searching some classes or methods.
- **Web Browsing (W)**: the 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 application, etc.). 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 developers press 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 for "navigation" activities.

Given an action list like 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 developer's actions in a fragment are associated with the same type of development activity. The principle of determining fragment boundaries is to look for distinct transitions between development activities. Once we have the initial list of action fragments, we go through the list again to see if adjacent action fragments can be merged into one action fragment or an action fragment can be further segmented into more fine-grained fragments.

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

Finally, we annotate each action in a fragment with the development-activity label that the majority of the actions in the fragment belong to.

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 actions and the involved applications in the actions individually. For example, as part of a debugging activity, the developer can read the bug report in documentation software, while as part of documentation activity, the developer can read some requirement specification in documentation software. Similarity, the developer can use the web browser to search the Web (as part of Web Browsing activity), read some documents (as part of documentation activity), or even test the web application (as part of testing activity). Or 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 like CRFs to infer development activities from low-level actions.

2.4 CRF Classifier for Development Activities

Conditional Random Field (CRF) is a type of discriminative undirected probabilistic graphical model which 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, a CRF can take context into account. CRFs relax the independence assumptions that are required by hidden Markov models (HMMs), which define a joint probability distribution over label sequences given a observation sequence, and also avoid the label bias problem, which is a weakness of Maximum-Entropy Markov model (MEMMs).

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 find 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 will produce 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. More details about linear-CRF could 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 the 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 positive, true negative, false positive, and false negative, respectively. We can calculate precision, recall and F1-score as bellow:

- **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 give a view on the general prediction performance. The macro-averaged metrics weight equally

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

all the classes, regardless of how many instances belong to them. The macro-averaged metrics can be calculated as bellow:

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

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 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 higher the Kappa metric is, the better the classifier is compared to the random guess classifier. The interpretations for Kappa values could be seen in Table 3.

3.2 Research Questions

In this work, we propose to use CRF-based approach to abstract low-level developers' actions into a set of basic but 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 our proposed CRF-based approach for segmenting and labeling the developers' action lists? And how much improvement could our proposed approach achieve over heuristic-rules based method and an ordinary classifier (i.e., SVM)?*

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

SVM is a very popular classification algorithm for analyzing data and has been used in many past 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

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

set of features as for the CRF classifier. Comparing with SVM, CRFs take the neighboring context into account so that they could usually achieve better performance on sequential data classification than SVM.

To investigate the benefits of machine learning approach over hand-crafted heuristic-rules based method, we design the following heuristic rules to infer development activities from 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 most of applications, stepping shortcut "F5", "F6" in Eclipse, etc.
- **Debugging**: all actions in the Debug perspective of IDE, key input actions using debugging keyboard shortcuts[3], such as "F5", "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", etc.
- **Navigation**: all other actions except *coding* and *debugging* actions.
- **Web Browsing**: all actions in the websites other than bugzilla web pages and web pages of tested application.
- **Documentation**: all actions in the documents other than bug report documents and testing documents.

We write a python script according to these heuristic rules and apply this script to our collected data to infer development activities.

Random guess classifier randomly predicts actions' label. The accuracy and recall for all the classes are the same as the probability of picking a certain class (i.e. 1/6 in our study), and the precision is equal to the proportion of instances that belong to a class.

In our experiment, we regard half a working day as a working session for each participant. So, we divide the collected action list into 10 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 are order preserved and the order of consecutive folds are also preserved. Then we use the first n folds as the training data and the remaining $10 - n$ folds as the testing data (n is from 1 to 9). An evaluation metric of a classifier is the average of 9 rounds of testing. To evaluate the performance of our CRF-based classifier and the three baseline classifiers, we perform three analyses: 1) We compare the accuracy, Kappa metric and macro-averaged precision/recall/F1-score of different classifiers on all the data as a whole. We also compare the precision, recall and F1-score of different classifiers on different labels of the whole data. 2) We compare the accuracy, Kappa metric and macro-averaged precision/recall/F1-score of different classifiers for each participant. 3) We

[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.

compare the precision, recall and F1-score of different classifiers on different labels of each participant's data.

RQ2 *How does the size of 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 may have a critical impact on the classification result. In general, the larger the training data is, the better performance a trained classifier. However, sometimes increasing training data may not always produce a better performing classifier. Therefore, we would like to investigate the impact of different sizes of training data on the classification performance. To answer this question, we conduct two experiments with different ways of controlling the size of training data. First, we use the accuracies of 9 rounds of testing for each participant obtained in the RQ1 to answer this question. That is, the size of training data ranges from 0.5 – 4.5 days of the action list of each participant. Second, to investigate the effort of manual annotation required for different participants, we control the size of training data by the number of action records to be labeled. We set the number of action records in training data from 100 to 1000 with a step size of 100 (i.e., 9 trials), then we calculate the accuracy of each trial and compare the results. In both experiments, the order of action records is preserved.

RQ3 *Can the CRF classifier be applied for 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 the developers' behavior and actions are often different, we hypothesize that if the two developers are in a similar working context, such as in the same project, or performing similar tasks, then the action lists of different developers may share some common characteristics. Therefore, cross-developer prediction may be possible. To answer this question, we use one participant's action list as the training data, and the other participants' action lists as the test data to evaluate the performance of the CRF. We also use all other participants' action lists as the training data, and the left participant's action list as the test data to evaluate the performance of the CRF. We use the accuracy as the evaluation metric in this experiment. We also report the Kappa metrics of cross-developer prediction.

3.3 Experimental Setup

In this work, we would like to investigate the effectiveness and applicability of our approach for analyzing professional developers' behavior data in real software development settings. Therefore, we deploy our ActivitySpace framework (Bao et al. 2015a) in our industry partner company and collect and annotate 10 professional developers' one-week working data in three software projects. Next, we describe the participant developers, the raw behavior data collected, and the manual annotation process to generate the ground-truth

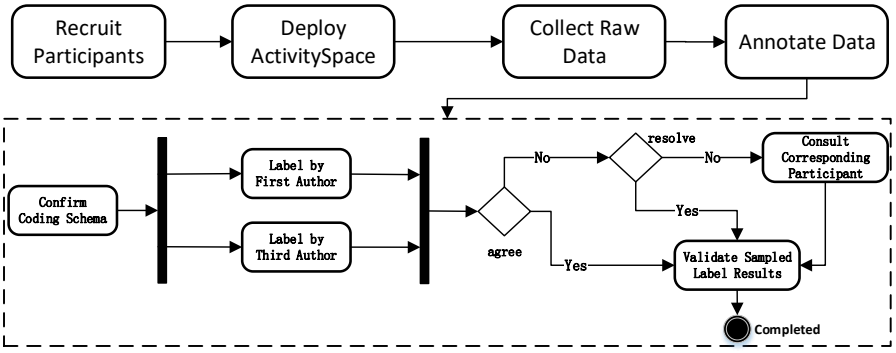


Fig. 2: The experiment procedure

development activity labels for answering the above research questions. The whole experiment procedure is shown in Figure 2.

3.3.1 Participants

We recruit 10 professional developers from Hengtian, an IT company in China. Hengtian is an outsourcing company which has more than 2,000 professional developers. It mainly does outsourcing projects for US and European corporations (e.g., StateStreet Bank, Cisco, and DST). Table 4 summarizes the demographics information of the 10 participant developers. Each of the 10 recruited developers (S1, S2, ..., S10) has at least 2 years of working experience. All of them use Java as a programming language in their daily work. They are from three different projects: participant developers S1-S5 are from a finance system re-engineering project, S6-S8 are from an e-commerce website project, while S9 and S10 are from a project which develop and maintenance an internal information management system for the company. These 10 developers' activities could be different according to the tasks of which they are in charge, such as bug fixes, front web development, database development, business analysis, etc. This makes our dataset more diverse. In addition to providing working data for the study, these 10 participants are also consulted to confirm the coding schema of development activities, resolve the disagreement between annotators, and validate the label results (See Section 3.3.4).

3.3.2 ActivitySpace *Deployment Settings*

We deploy our ActivitySpace tool on the participants' computers and collect their working data throughout one week. In this study, we configure ActivitySpace to track the developers' interactions with the applications that are 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++*),

Table 4: The demographics information of the 10 participant developers

	Project	Project Description	Experience	Responsibility
S1	P1	a financesystem re-engineering project	2 years	Web Development
S2	P1		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 website project	4 years	Web Development
S7	P2		3 years	Web Development
S8	P2		3 years	Quality Assurance, Business Anlysis
S9	P3	an internal management system of company	5 years	Database Development
S10	P3		2 years	Deployment

and IDEs (e.g., *Eclipse, Visual Studio*). We configure ActivitySpace to collect keyboard events and mouse-click events for focused UI components. ActivitySpace is configured to ignore mouse-move events and mouse-wheel events. The developers have the option to turn off the ActivitySpace's data tracking when they work on proprietary data and/or applications from the outsourcing company. This will only affect the amount of the data collected, but not the developers' working behavior, because they use the same working environments for proprietary or non-proprietary data and applications.

We conduct a pilot study to verify the runtime performance of ActivitySpace. We modify the code of ActivitySpace to log the time spent on recording one action record. We deploy the modified ActivitySpace on the computer of one developer that participates in our study and collect one hour of his working data. ActivitySpace collects 426 action records in total. The average and standard deviation time is 0.08 ± 0.04 second per action record. Note that this time statistic includes the time spent on logging the time, which is an overhead of the modified ActivitySpace. We also ask the developer to monitor the memory usage during the data collection. The developer reports that the memory usage of ActivitySpace is stable (about 40M RAM). The configuration of the developer's computer is Windows 7, 8GB RAM and Intel(R) Core(TM) i5-4570 CPU. This pilot study shows that ActivitySpace does not interfere with developers' normal work.

3.3.3 Raw Interaction data

For the efficiency of action logging, ActivitySpace does not perform any analysis of the logged actions during the logging process. We post-process the logged raw data to determine the effective working hours for each developer. Effective working hours refer to the time when a developer continuously performs some actions on his/her computer. If a developer does not perform any actions on the computer for a long period of time (30 minutes in this study),

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**

we will not regard such time period as effective working time. Furthermore, we only collect the developers' actions which are related to the project. Actions such as reading news on news websites (e.g., weibo.com) are filtered out. All developers run ActivitySpace on their computer for 5 working days. As summarized in Table 5, after the data preprocessing, we collect in total about 240 hours effective working data which contain more than 50,000 action records. We regard half a working day as a working session for each developer, and compute the average and standard deviation of action records per working session (see the last column in Table 5). For different developers, since their tasks and behavior are different, the number of action records varies from about 3,600 to about 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 10 participants in our study and the 10 additional developers. In the document, we first show the definition of activity labels (as defined in Section 2.3) to the developers. Then we ask two questions: 1) what are your major activities in your working time? 2) do you think our coding schema cover all your activities in your working time? Eighteen out of 20 developers think our coding schema covers their major activities in working time. Three example responses are given follow:

- *My job is in charge of Java development, most of my work is coding and fix bugs. I will also search for solutions when I have technical problem. Sometimes I will maintain the documents in my project.*
- *I am a database administrator. I write a lot of SQL scripts and also need test many scripts submitted by other developers. To solve some exception, I usually look for answers from internet or refer to official documentation.*
- *I am a QA and response for testing three projects in the company. I think my tasks belong to "Testing", "Navigation", "Web Browsing" and "Documentation", because I don't need write much code and only write very simple scripts sometimes.*

Two developers point out some limitations of our coding schema, which is as follow:

- *I think this coding schema is for developers, but I am not only a developer but also a project manager. I spend a lot of 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. But If I use some other applications, e.g. email client, message sender, etc., to communicate with other people. Which label do these activities belong to?*

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 the purpose of this study. We leave the coding-schema limitations raised by the two developers as future work.

To ensure a high level of rigor in our manual annotation process, the first author and the third author who both have more than 6 years programming experience manually label these collected action records. The manual annotation process is divided into 10 annotation sessions, i.e., labeling the action list of one participant per session. The entire manual annotation process took about 80 hours in total, including manual annotation and validation of the annotation results with participant developers.

The annotators follow the segmentation-based annotation process and principles described in Section 2.3. During an annotation session, the two annotators independently annotate 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 is 0.61 which indicates substantial agreement between the annotators. After completing the manual annotation process, the two annotators discussed their disagreements to reach a common decision. We have two types of disagreements. First, most of disagreements are 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 contain a mix of similar numbers of different types of actions. The two annotators may disagree on action labels in such segments. If they cannot 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, as well as the screenshots that ActivitySpace takes when logging the action actions. This helps 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 the final decision on action labels.

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

Recall that we showed the definition of 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 again as well as their feedbacks to our coding schema to refresh their memory. Then, for each sampled segment, we provide the corresponding developer with code files, web pages and/or other documents, as well as the screenshots that ActivitySpace takes in that segment. We asked the developer to confirm whether the collected data is consist with their work at the time the data was collected. Reviewing the documents and screenshots logged in a segment helps the developer recall his/her activities in the segment (Safer and Murphy 2007). After that, we asked the developer to explain what activities they did in the segment and why they did these activities. Following on 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 lasted about 6 hours. 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 agree 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 across the 10 participants. We can see that different developers have different distributions of development activities in their work. The number of action records labeled in three types of development activities, namely coding (25.93%), testing (26.75%) and navigation (25.28%), is more than the total number of action records labeled in the other three development activities. 5%-10% action records are labeled in debugging for some participants, such as S1, S2, S3, S5, S6, S7, S10. The participants S4, S8 and S9 have few or no action records which are labeled in debugging. We find this is because S4 is mainly responsible for front-end

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%)

development, S8's main work is testing and business analysis, and S9 is in charge of database development during the week of our data collection. We further confirm with S4 that he usually uses log to observe and verify code change, but he rarely uses IDE debug features. Therefore, there are only few debugging action records for the participant S4.

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

3.5 Results

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

Table 7, Table 8 and Table 9 show the confusion matrixes of CRF, the heuristic-rules method, and the SVM baseline for the whole data of all the 10 participants, respectively. For each approach, the confusion matrix is constructed by accumulating the confusion matrixes of all rounds of validation. Based on these confusion matrixes, we compute the overall accuracy and Kappa metric of CRF, the heuristic-rules method, and the SVM baseline for the whole data, as shown Table 10. Note that the accuracy of the random guess classifier is 0.167. Table 11 shows the precision, recall and F1-score for each label using CRF, the heuristic-rules method, the SVM baseline and the random guess classifier for the whole data. The last column of Table 11 is the corresponding macro-averaged metrics.

The accuracies and the macro-averaged metrics show that our CRF classifier achieves the best performance on the whole data, the performance of the

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

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)

heuristic-rules method is in the middle, and the SVM baseline performs the worst. Except for the macro-averaged recall where our CRF classifier and the heuristic-rules method are almost identical, our CRF classifier achieves much better results for all other metrics. Furthermore, the classification results of our CRF classifier have substantial agreement with the actual labels, while the heuristic-rules method has moderate agreement, and the SVM baseline has only slight agreement.

Table 11: Precisions, recalls, f1-scores and the corresponding macro-averaged metrics for each label using CRF, heuristic-rules, SVM and random guess 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	0.167	0.167	0.167	0.167	0.167	0.167	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	0.202	0.111	0.207	0.200	0.122	0.065	0.151

Table 12: Accuracies of CRF, heuristic-rules, SVM and random guess for each participants

	CRF	**RULE**	**SVM**	**Random**
S1	0.808±0.071	0.386±0.029	0.420±0.342	0.167
S2	0.756±0.066	0.691±0.023	0.100±0.148	0.167
S3	0.720±0.054	0.524±0.049	0.515±0.170	0.167
S4	0.808±0.089	0.637±0.046	0.355±0.266	0.167
S5	0.863±0.024	0.419±0.019	0.470±0.349	0.167
S6	0.772±0.057	0.542±0.028	0.434±0.316	0.167
S7	0.730±0.100	0.644±0.033	0.287±0.282	0.167
S8	0.616±0.128	0.740±0.051	0.140±0.184	0.167
S9	0.724±0.197	0.437±0.042	0.414±0.235	0.167
S10	0.809±0.052	0.616±0.035	0.584±0.279	0.167
Mean	0.761±0.084	0.564±0.036	0.372±0.257	0.167

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)

Regarding the precision, recall and F1-score for each label in the whole data, CRF has the highest precision on all labels except for the precision of the heuristic-rules method for the label “Debugging”. CRF has the highest recall on three labels: “Coding”, “Testing” and “Navigation” while the heuristic-rules method has the highest recall on the other three labels. CRF achieves 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 most of debugging activities are performed in debugging perspective, and thus can be easily inferred by heuristic rules. Similar reason can be extended to the better performance of the heuristic-rules method for the label“Documentation”. That is, developers perform most of documentation activities in office or PDF software, which can be easily inferred.

3.5.2 RQ1: Effectiveness of CRF - Performance for Each Participant

Table 12 shows the accuracy of our CRF classifier, the heuristic-rules method, and the SVM baseline for each participant. The reported accuracy of a classifier

is the average accuracy of nine rounds of testing. We also show the standard deviation of the accuracies of nine rounds of testing. The accuracy of the random guess classifier is 0.167 by construction.

The results show that our CRF classifier has the best accuracies for all the participants, except the participant S8. For the heuristic-rules method, the accuracies of all the participants are not very stable: the lowest is 0.386 for the participant S1, and the highest is 0.740 for the participant S8. For the participant S8, we find that the participant performs a lot of documentation activities. Most of documentation activities are performed in Office or PDF software, which can be easily identified by heuristic rules. The heuristic-rules method also has a good accuracy result for the participant S2, because this participant performs many navigation activities which can be easily identified by heuristic rules, such as "Find and Search" window. But the accuracies of other participants in general are not very good. The average accuracy of the heuristic-rules method across all the participants is 0.560, while the average accuracy of our CRF classifier is 0.761.

For the SVM baseline, the accuracies are all below 0.6 (only 0.325 on average). Except the participant S1, SVM is worse than the heuristic-rules method. This could be because SVM does not consider the behavior context and the local features alone cannot reliably classify actions. Furthermore, the results show that the standard deviation of the heuristic-rules method is the smallest, since its performance depends on only the defined heuristic rules but not the size of training data. The standard deviation of CRF is much smaller than that of the SVM baseline, which indicates that CRF is more robust than the SVM baseline. Our results show that machine learning approach may not always outperform heuristic rules, unless machine learning approach is well designed to take into account the characteristics of the data.

We also calculate the accuracy improvement using our CRF classifier compared with the heuristic-rules method or the SVM baseline, i.e., $improvement = \frac{A_{crf} - A_m}{A_m}$, where A means accuracy and m could be heuristic-rules method or the SVM baseline. The average accuracy improvement over the heuristic-rules method, the SVM baseline and the random guess classifier is 34.93%, 104.52% and 355.43%, respectively. For the heuristic-rules method, the accuracy improvement for the participant S1 is the biggest (109.32%), and the accuracy improvement for the participant S8 is the smallest (-16.76%). This is because the participant S1 is responsible for web development, he/she has to switch between different tasks frequently, such as coding, testing, navigation. In such complex working behavior, heuristic rules cannot work very well. In contrast, the participant S8 performs many documentation activities that can be easily inferred by heuristic rules, for example, by software applications used. For the SVM baseline, the accuracy improvement for the participant S2 is the biggest (655.38%), and the accuracy improvement for the participant S10 is the smallest (38.57%). To measure whether the improvement is 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

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

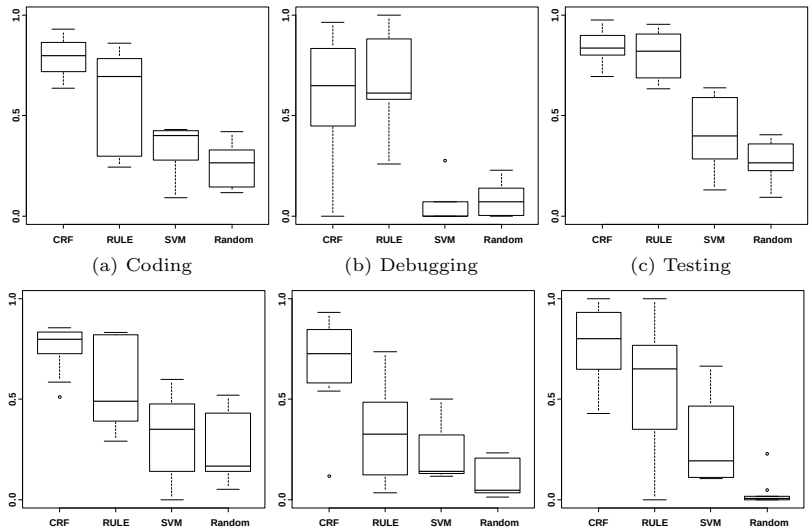
respectively, which indicates the improvement is statistically significant at the confidence level of 99%.

Table 13 shows the Kappa values of CRF, the heuristic-rules method, and the SVM baseline for each participant. We find that the Kappa values of CRF are better than those of the heuristic-rules method and the SVM baseline for all participants, except the participant S8. The agreements between the predictions using CRF and the actual labels for all participants are at least moderate. Eight out of ten Kappa values using CRF are above 0.60, which indicates substantial agreement. However, the agreement levels using the heuristic-rules method are fair and moderate for all the participants, except substantial for S8. The agreement levels using the SVM baseline are the worst. For the participants S2 and S8, the Kappa values of the SVM baseline are even less than zero, which indicates poor agreement.

To comparing the performance of CRF, the heuristic-rules method, the SVM baseline and random guess classifier for each participant, we also calculate the macro-averaged precisions, recalls and F1-scores of different classifiers for each participant (see Table 14). We find that the macro-averaged precisions of CRF are larger than those of the heuristic-rules method except for all participants except S8. For participant S8, the macro-averaged precision of CRF is very close to 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) are larger than those of CRF, but most of the macro-averaged F1-scores of CRF are larger than those of the heuristic-rules method except for the participant S2 and S8. For the participant S2 and S8, the macro-averaged F1-scores of CRF are very close to 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 the random guess classifier are all smaller than those of CRF and the heuristic-rules method. Overall, our CRF achieves the best performance for most of the participants.

3.5.3 RQ1: Effectiveness of CRF - Performance for Each Label

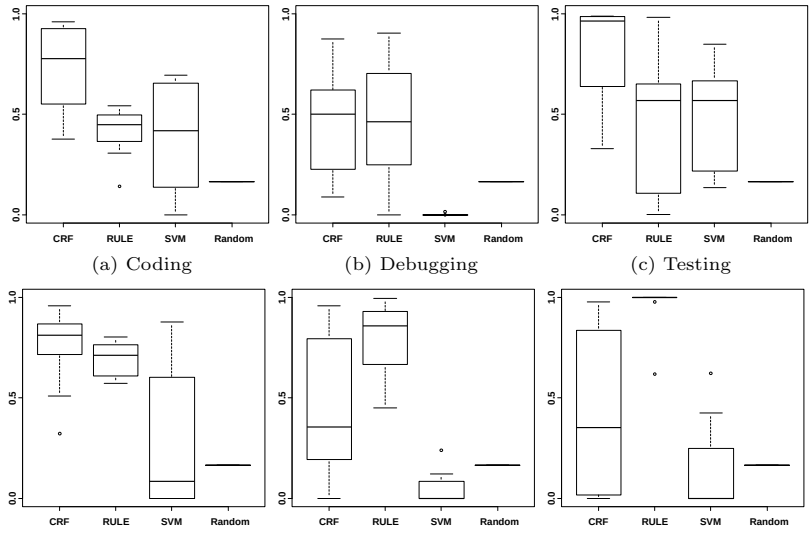
We use box plot to compare the results of precision, recall, and F1-score for each label using CRF, the heuristic-rules method, SVM and the random guess



(a) Coding (b) Debugging (c) Testing

(d) Navigation (e) Web Browsing (f) Documentation

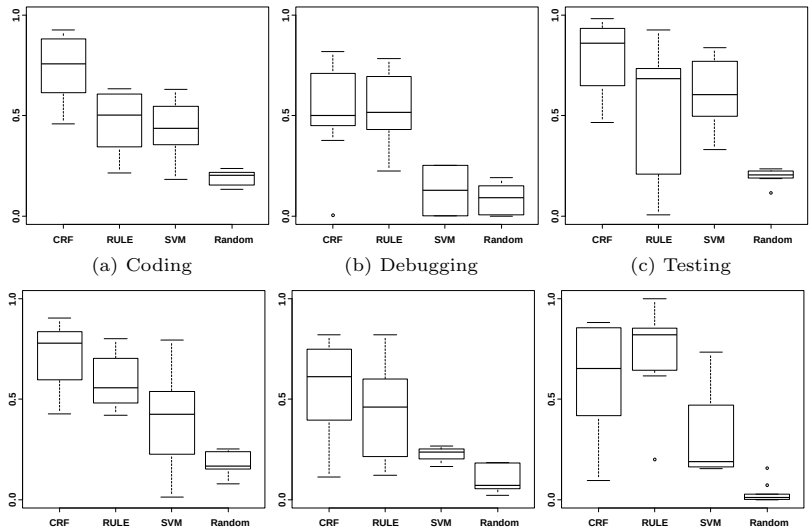
Fig. 3: Precisions for each label using CRF, heuristic-rules, SVM and random guess



(a) Coding (b) Debugging (c) Testing

(d) Navigation (e) Web Browsing (f) Documentation

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



(a) Coding (b) Debugging (c) Testing

(d) Navigation (e) Web Browsing (f) Documentation

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

classifier on each participant's data – see Figure 3, 4, 5 respectively. To plot the boxplot, we use the average metrics of 9 rounds of testing for each participant. But in some rounds of testing, there may exist very few or no action records labeled by one label in the test data of a participant. For example, when we use the first 7 working sessions of action records to predict the left 3 working sessions of action records for the participant S2, there are only 6 "Documentation" action records in the S2's testing data. For such rarely-occurred labels, the performance metrics of the CRF, the heuristic-rules method, and the SVM baseline will be invalid, Therefore, we ignore these kinds of data when plotting the boxplot.

Figure 3 shows that the precisions of CRF are the best for all labels, and the precisions of heuristic-rules are better than that of SVM for all labels. There are some cases where heuristic-rules or SVM have better performance than CRF, for example, for the label "Web Browsing", the participant S7 using the heuristic-rules method and the participant S4 using SVM (i.e., the outliers in Figure 3e). But the median values (the band inside the box) of CRF are all higher than those of the heuristic-rules method and SVM. For the label "Debugging", the median value of CRF is a bit larger than that of the heuristic-rules method. But in some cases, the precision of CRF for the label "Debugging" is smaller than that of the heuristic-rules method (see Figure 3b). This is because developers perform most of debugging activities in debugging perspective of IDEs, which can be easily inferred by heuristic rules.

For the results of recall, CRF has better performance than heuristic-rules method on all labels except the label "Web Browsing" and "Documentation",

while the SVM baseline has the worst performance on all labels (see Figure 4). For the label "Web Browsing" and "Documentation", we use heuristic-rules to cover most of web pages and documents that exist in our collected data based on the data observation. Therefore, the heuristic-rules method achieves better performance than CRF. However, this may not be generalizable on new data. For the result of F1-score which combines precision and recall, CRF has better performance than the heuristic-rules method except for the label "Debugging" and "Documentation", while the SVM baseline has the worst F1-score on all labels (see Figure 5). In some cases, the performance of SVM is very poor. For example, for the label "Debugging", the F1-scores of SVM are much smaller than those of CRF and the heuristic-rules. This can be because the SVM baseline does not consider the 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 larger than those of the baseline classifiers. The minimum F1-scores of all classifiers are usually very small, because there might be no action records for a particular label when the size of training data is small. In such cases, the F1-score of random guess classifier is zero. The heuristic-rules method achieves much larger minimum F1-scores than other classifiers on label "Debugging", "Navigation" and "Documentation". This is because the heuristic rules are not directly affected by the size of the training data. The median and mean values of F1-scores of CRF are all larger than those of the baseline classifiers except for the label "Documentation" using the heuristic-rules method. The random guess classifier achieves the smallest standard deviation, since 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 larger 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 but not the size of training data. The standard deviations of CRF are smaller than those of the SVM baseline expect for the label "Debugging" and "Web Browsing".

To measure whether the F1-score improvement of our CRF classifier is significant over the heuristic-rules method and the SVM baseline, we apply Wilcoxon signed-rank test (Wilcoxon 1945) for each label on F1-score (see the last column in Table 15). We also calculate 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", are smaller than 0.05 and the Cliff's deltas are at least in small effectiveness level, which means that the improvement of our CRF classifier is statistically significant at the confidence level of 95%. But the improvement for the other two labels, i.e., "Web Browsing" and "Documentation" is not statistically significant, because there exist some participants for

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

	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	0.248	0.000	0.201	0.192	0.043	0.946	<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	0.203	0.000	0.111	0.088	0.073	0.138	<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	0.254	0.028	0.205	0.200	0.036	0.898	<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	0.256	0.036	0.170	0.182	0.054	0.969	<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	0.257	0.000	0.075	0.098	0.065	0.293	<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	0.240	0.000	0.010	0.032	0.057	0.022	<0.001

whom the F1-scores of the heuristic-rules method are larger than those of CRF. For the SVM baseline, the p-values are all smaller than 0.05 and the Cliff's deltas are all in large effectiveness level, which means that the improvement of our CRF classifier is 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 the different size of 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 size of training data ranging from 100 to 1000. For each participant, we also calculate the mean and standard deviation of the accuracies (see the last column in these two tables).

From these two tables, we find that the performance of the CRF classifier is very stable for some participants (i.e., S1, S2, S5 and S10). Their standard deviations are very small. Given a very small size of training data, their accuracies are very high. For example, when the size of training data is only 100 action records, the accuracy of participant S1 is even larger than 0.70. For other participants, when the size of training data increases, the accuracy will become stable. But the required sizes of training data are different. For participant S3, S4, S6 and S7, when the number of action records in training

Table 16: Accuracies of different sizes of 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

Table 17: Accuracies of different sizes of 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

data is 500 or above, the accuracies become stable with small standard deviation. However, for participant S8 and S9, it requires much more training data. For participant S8, when the size of training data is more than 2.5 days, the accuracy value becomes stable and equal to ∼0.7. The accuracy values of participant S9 are very high (i.e., more than 0.9) when the size of training data is more than 3 days. We find that the size of training data in which the accuracy become stable depends on the label distribution in the training data and testing data. For example, there is even no action record labeled by "Coding" in the previous 500 action records of participant S9, and the most of actions are labeled as "Testing" and "Web Browsing". Therefore, the accuracies of participant S9 is very low (i.e., smaller than 0.3) when the size of training data is less than 500 action records.

Overall, we find the CRF classifier is robust and effective with small training data. But to achieve good performance, the labels in the training data should be diverse. To decrease the effort of manual annotation in practice, instead of labeling behavior data by consecutive working sessions (which could be uniform), we may observe the action distribution in working sessions and try to prepare the training data with diverse labels from non-consecutive working sessions.

Table 18: Accuracies of 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**

Table 19: Kappa values of 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**

3.5.5 RQ3: The Cross-Developer Prediction

In the experiment, we use each participant's action list as the training data to train a CRF model to predict the label of the other 9 participants' actions. We also use all other participants' action lists as the training data to train a CRF model to predict the label of the left participant's actions. Table 18 and Table 19 show the accuracies and Kappa values of cross-developer prediction using our 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 the last row of these two tables, each cell is the accuracies (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 in which the accuracy results are close to or higher than 60%. The mean accuracy of using the model trained by the action lists of the participant S6 to predict other participants' actions is the highest (i.e., 70%). For using all other participants to predict the left one, all accuracy results are higher

than 0.70. This shows that cross-developer prediction using CRF is feasible. However, we would like to point out that there are only 10 developers in our experiment and they all use Java as the main programming language. More data from different developers and projects is needed to verify whether the cross-developer prediction is feasible in general. We consider to collect more data from different developers in the future.

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

From the Kappa metrics in Table 19, we can see that the results of our classifier trained using all other participants' data for predicting the left participant have substantial agreement with the actual labels for six participants, and have 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 our classifier trained using one participants' data for predicting others, 31.1% cases have moderate agreement, and 51.1% have substantial or almost perfect agreement. For 17.8% cases with below-moderate agreement, it is mainly because the training data of one participant does not include the labels in the other participant's data. For example, using S8's data to predict S7's data, the agreement is only 0.29 (fair). This is because S8 performs many "Documentation" actions, while S7 does have very few "Documentation" actions in his working data.

4 Discussion

We introduce some implications based on our 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. In contrast, the behavior data in software development tasks are discarded, even though it has great potentials to improve software development practices. In fact, a developer study by Wang et al. (Wang et al. 2011) discovers 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. is conducted in controlled experiments and their behavior pattern discovery and analysis are done manually. Our study echoes their findings in real development tasks, and we show the potential of using machine learning approach to infer phases and behavior patterns in the process of software development tasks.

To verify this potential, we choose two participants (S1 and S2) and collect one more week's of interaction data in their work. Then, we use the trained CRF classifier to predict their activities in that week. We find that there are more action records labeled in "Coding" and "Web Browsing" (37% and 35% respectively) than other classes of actions for participant S1, while for participant S2, "Coding", "Testing" and "Debugging" are 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 agree that our analysis result is consistent with their real development activities. S1 says *"I need to implement a new function that requires a Java library I never use before. Therefore, in this week I read lots of online tutorial and develop a demo application to learn this library."* As a result, he spent a large amount of time on "Coding" and "Web Browsing". While S2's tasks in that week are mainly bug fixing since several bugs are reported in the module that he is responsible for. He spent most 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 could be as useful as artifact repository for software development. Our 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 will not only archive the behavior data, but also can 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 some new behavior-oriented aspects. It could help address some key challenges in software engineering knowledge management, such as the loss of the best practice of completing some tasks due to the leave of some key developers. Behavior patterns in the knowledge repository could enable 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. Our approach is an attempt to mitigate these difficulties in study developers' behavior. First, by operating-system level instrumentation, it could be relatively easier to adapt our approach in different study settings. Second,

using machine learning approach could reduce the amount of human efforts needed to manually examine the data.

We would like to highlight some technical prerequisites in deploying our approach in real work settings. First, although operating-system level instrumentation improves the generalizibility of data collection in different working environments, it is still 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 conduct 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. Second, although machine learning approach could potentially reduce the human efforts for analyzing behavior data in the long run, human labeling of behavior data is a must-do step in the beginning. The goal is to obtain high-quality training data with good behavior diversity and precise labels. These initial human efforts could still be significant. A way to mitigate this effort is to devise statistical methods to estimate how much data has to be annotated by observing the overall data in order to train a high-quality machine learning model.

We also experience some interesting ethical debates about deploying a behavior data analysis tool like our proposed approach in the company. The company managers appreciate the value of behavior analysis and behavior knowledge repository to improve the project management and the developers' practices in their daily work. Developers also see the potentials of behavior analysis for supporting their information needs in their work (Bao et al. 2015b), but they are reluctant to openly release their working data with others. Due to the growth of open source movement, developers have no problem to share their code. Traditionally, developers share how they code in video tutorials. Recently, experience developers share how they code in real time by 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 way to help developers grow.

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 collect more than 50 thousands action records from 10 developers' daily work. Due to the large size of action record to be labeled and human misunderstanding of developers' activities, there may exist some annotation errors. In order to decrease errors, two human annotators are involved in labeling all the records and spend about 80 hours to complete this task. The two annotators independently annotate the same data then the disagreements are solved by discussion and help from developers. Furthermore, we sample 10% of action records to let developers

check that whether our annotation results are right. In retrospect, we could improve the participants' validation step by intentionally planting false information (e.g., annotations that are obviously false), and then see whether the participants catch 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 consider only 6 basic development activities: *coding*, *debugging*, *testing*, *navigation*, *web search*, and *documentation*. However, the developers' behavior can be more complex, which could involve behaviors not covered by the defined 6 development activities. There could also potentially be some activities that are hardly recognizable, especially when the annotators lack some domain-specific knowledge. Furthermore, there could be certain level of ambiguities among the 6 development activities, which could lead to the erroneously labeling results. According to our experiment results, we believe that the 6 development activities in our schema are reasonable and can cover 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 about 240 effective working-hours behavior data from 10 developers' one-week work, and these 10 developers are from three different projects. Although the data is collected in real work settings, we acknowledge the limitation of data and the limitation of 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, F1-scores as the main evaluation metrics which 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 little 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), testing (Beller et al. 2015). The data collection methods of these studies usually include screen-capture video, think-aloud, instrumentation, survey. However, the data analysis in these studies usually requires significant amount of human effort (Bao et al. 2016). For example, the researchers have to encode and transcript the video and other observation data manually. Furthermore, in order

to make the data easy to understand, the collected data has to be abstracted in the data transcription process. For example, I.Coman and A.Sillitti (Coman and Sillitti 2009) use the code access location (e.g. methods, classes, files) to segment the development sessions into task-related subsections automatically, but 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 still requires certain amount of human annotation, our study shows that a small amount of training data can achieve good classification results. Therefore, the human efforts required in our proposed approach could be much less than the efforts to transcribe all the behavior data manually. Furthermore, our approach does not rely on any application-specific support to collect interaction data. Therefore, our 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 in which 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, Fishers 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 Models (HMMs) (Rabiner 1989), Maximum Entropy Markov Models (MEMMs) (McCallum et al. 2000), and Conditional Random Fields (CRFs) (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). Some researchers have used these segmentation and labeling techniques to mining game strategies. For example, Dereszynski et al. (Dereszynski et al. 2011) assumes equal length segments of 30 seconds, and applies HMMs to learn the segment labels. Gong et al (Gong et al. 2012) proposes 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.

Since CRFs are proved to have better performance than both HMMs and MEMMS 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 choose CRFs to segment and label the developers' action list in our work.

7 Conclusion

In this paper, we present an approach for labeling and segmenting time series of developers' interaction with the entire working environment in real software projects. We conduct an experiment using 10 developers' daily working data in a real software project, which includes about 240 effective working hours and 50 thousands action records. Our experiment finds that: (1) Compared with the classic classifier SVM, our CRF-based approach has better performance with about 75% prediction accuracy. This is because the CRF model takes into account the neighboring context in model training. (2) Our approach is also robust over the size of training data. It requires only a small amount of training data to produce accurate classification results. (3) Cross-developer prediction using our approach achieves acceptable accuracy (i.e., about 60%). Overall, our experiment shows that our proposed approach is more generic and less demanding, compared with existing approaches on studying developers' behavior. Therefore, our approach is much 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.

References

J. Anvik, L. Hiew, G.C. Murphy, Who should fix this bug?, in *Proceeding of the 28th International Conference on Software Engineering (ICSE)*, 2006, pp. 361–371

L. Bao, D. Ye, Z. Xing, X. Xia, ActivitySpace: A Remembrance Framework to Support Interapplication Information Needs, in *Proceedings of 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015a, pp. 864–869

L. Bao, Z. Xing, X. Wang, B. Zhou, Tracking and Analyzing Cross-Cutting Activities in Developers' Daily Work, in *Proceedings of 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015b, pp. 277–282

L. Bao, J. Li, Z. Xing, X. Wang, X. Xia, B. Zhou, Extracting and analyzing time-series hci data from screen-captured task videos. Empirical Software Engineering, 1–41 (2016)

M. Beller, G. Gousios, A. Panichella, A. Zaidman, 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)*, 2015, pp. 179–190

A.L. Berger, V.J.D. Pietra, S.A.D. Pietra, A maximum entropy approach to natural language processing. Computational linguistics **22**(1), 39–71 (1996)

T.-H. Chang, T. Yeh, R. Miller, 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)*, 2011, pp. 245–256

I.D. Coman, A. Sillitti, Automated segmentation of development sessions into task-related subsections. International Journal of Computers and Applications **31**(3), 159–166 (2009)

C.L. Corritore, S. Wiedenbeck, An exploratory study of program comprehension strategies of procedural and object-oriented programmers. International Journal of Human-Computer Studies **54**(1), 1–23 (2001)

U. Dekel, J.D. Herbsleb, Reading the documentation of invoked API functions in program comprehension, in *Proceedings of 17th IEEE International Conference on Program Comprehension (ICPC)*, 2009, pp. 168–177

E.W. Dereszynski, J. Hostetler, A. Fern, T.G. Dietterich, T.-T. Hoang, M. Udarbe, Learning Probabilistic Behavior Models in Real-Time Strategy Games., in *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2011

P. Dewan, P. Agarwal, G. Shroff, R. Hegde, Distributed side-by-side programming, in *Proceedings of the 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*, 2009, pp. 48–55

E. Duala-Ekoko, M.P. Robillard, Asking and answering questions about unfamiliar APIs: An exploratory study, in *Proceedings of 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 266–276

R. Durbin, S.R. Eddy, A. Krogh, G. Mitchison, *Biological sequence analysis: probabilistic models of proteins and nucleic acids* (Cambridge university press, Cambridge UK, 1998)

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

J.L. Fleiss, Measuring nominal scale agreement among many raters. Psychological bulletin **76**(5), 378 (1971)

T. Fritz, D.C. Shepherd, K. Kevic, W. Snipes, C. Bräunlich, Developers' code context models for change tasks, in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014, pp. 7–18

W. Gong, E.-P. Lim, P. Achananuparp, F. Zhu, D. Lo, F.C.T. Chua, In-game action list segmentation and labeling in real-time strategy games, in *Proceedings of IEEE Conference on Computational Intelligence and Games (CIG)*, 2012, pp. 147–154

C.D. Hundhausen, J.L. Brown, S. Farley, D. Skarpas, A methodology for analyzing the temporal evolution of novice programs based on semantic components, in *Proceedings of the ACM International Computing Education Research Workshop*, 2006, pp. 59–71

A. Hurst, S.E. Hudson, J. Mankoff, Automatically identifying targets users interact with during real world tasks, in *Proceedings of the 15th international conference on Intelligent user interfaces (IUI)*, 2010, pp. 11–20

A.J. Ko, B.A. Myers, A framework and methodology for studying the causes of software errors in programming systems. Journal of Visual Languages & Computing **16**(1), 41–84 (2005)

A.J. Ko, B. Myers, M.J. Coblenz, H.H. Aung, et al., An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. IEEE Transactions on Software Engineering **32**(12), 971–987 (2006)

A.G. Koru, A. Ozok, A.F. Norcio, 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, 2005, pp. 1–3

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

J. Lawrance, R. Bellamy, M. Burnett, K. Rector, 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)*, 2008, pp. 1323–1332

J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, S.D. Fleming, How Programmers Debug, Revisited: An Information Foraging Theory Perspective. IEEE Transactions on Software Engineering **39**(2), 197–215 (2013)

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

T.-D.B. Le, D. Lo, 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)*, 2013, pp. 310–319

H. Li, Z. Xing, X. Peng, W. Zhao, What help do developers seek, when and how?, in *Proceedings of 20th Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 142–151

A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, G. Antoniol, E. Aïmeur, Support vector machines for anti-pattern detection, in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2012, pp. 278–281

A. McCallum, D. Freitag, F.C. Pereira, Maximum Entropy Markov Models for Information Extraction and Segmentation., in *Proceedings of the Seventeenth International Conference on Machine Learning*, vol. 17, 2000, pp. 591–598

R. Minelli, A. Mocci, M. Lanza, 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)*, 2015, pp. 25–35

A.T. Nguyen, T.T. Nguyen, H.A. Nguyen, T.N. Nguyen, 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)*, 2012, pp. 63–73

D. Pinto, A. McCallum, X. Wei, W.B. Croft, Table extraction using conditional random fields, in *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, 2003, pp. 235–242

D. Piorkowski, S.D. Fleming, C. Scaffidi, L. John, C. Bogart, B.E. John, M. Burnett, R. Bellamy, 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)*, 2011, pp. 109–116

L.R. Rabiner, A tutorial on hidden markov models and selected applications in speech recognition. Proceedings of the IEEE **77**(2), 257–286 (1989)

M.P. Robillard, W. Coelho, G.C. Murphy, How effective developers investigate source code: An exploratory study. IEEE Transactions on Software Engineering **30**(12), 889–903 (2004)

G.P. Sackett, *Observing Behavior: Theory and applications in mental retardation* (University Park Press, Baltimore, MD, 1978)

I. Safer, G.C. Murphy, Comparing episodic and semantic interfaces for task boundary identification, in *Proceedings of Conference of the Centre for Advanced Studies on Collaborative Research*, 2007, pp. 229–243

P.M. Sanderson, C. Fisher, Exploratory sequential data analysis: Foundations. Human–Computer Interaction **9**(3-4), 251–317 (1994)

J. Sillito, K. De Voider, B. Fisher, G. Murphy, Managing software change tasks: An exploratory study, in *International Symposium on Empirical Software Engineering*, 2005, p. 10

A.C. Siochi, D. Hix, 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)*, 1991, pp. 301–305

C. Sun, D. Lo, X. Wang, J. Jiang, S.-C. Khoo, A discriminative model approach for accurate duplicate bug report retrieval, in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, 2010, pp. 45–54

F. Thung, D. Lo, L. Jiang, Automatic defect categorization, in *Proceedings of 19th Working Conference on Reverse Engineering (WCRE)*, 2012, pp. 205–214

Y. Tian, C. Sun, D. Lo, Improved duplicate bug report identification, in *Proceedings of 16th European Conference on Software Maintenance and Reengineering (CSMR)*, 2012, pp. 385–390

M. Vakilian, N. Chen, S. Negara, B.A. Rajkumar, B.P. Bailey, R.E. Johnson, Use, disuse, and misuse of automated refactorings, in *Proceedings of 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 233–243

A. von Mayrhauser, A.M. Vans, Program understanding behavior during debugging of large scale software, in *Proceedings of the seventh workshop on Empirical studies of programmers*, 1997, pp. 157–179

J. Wang, X. Peng, Z. Xing, W. Zhao, 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)*, 2011, pp. 213–222

F. Wilcoxon, Individual comparisons by ranking methods. Biometrics bulletin **1**(6), 80–83 (1945)

R. Wu, H. Zhang, S. Kim, S.-C. Cheung, 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)*, 2011

X. Xia, D. Lo, X. Wang, X. Yang, S. Li, J. Sun, A comparative study of supervised learning algorithms for re-opened bug prediction, in *Proceedings of 17th European Conference on Software Maintenance and Reengineering (CSMR)*, 2013