On the Use of Hidden Markov Model to Predict the Time to Fix Bugs

Mayy Habayeb, Syed Shariyar Murtaza, Andriy Miranskyy, *Member, IEEE*, and Ayse Basar Bener, *Senior Member, IEEE*

Abstract—A significant amount of time is spent by software developers in investigating bug reports. It is useful to indicate when a bug report will be closed, since it would help software teams to prioritise their work. Several studies have been conducted to address this problem in the past decade. Most of these studies have used the frequency of occurrence of certain developer activities as input attributes in building their prediction models. However, these approaches tend to ignore the temporal nature of the occurrence of these activities. In this paper, a novel approach using Hidden Markov Models and temporal sequences of developer activities is proposed. The approach is empirically demonstrated in a case study using eight years of bug reports collected from the Firefox project. Our proposed model correctly identifies bug reports with expected bug fix times. We also compared our proposed approach with the state of the art technique in the literature in the context of our case study. Our approach results in approximately 33 percent higher F-measure than the contemporary technique based on the Firefox project data.

Index Terms—Bug repositories, temporal activities, time to fix a bug, hidden markov model

1 Introduction

Estimating bug fix time and identifying bugs that would require a long fix time at early stages of the bug life cycle is useful in several areas of the software quality process. Having advance knowledge of the time to fix bugs would allow software quality teams to prioritise their work and improve the development activities on bugs that would require more time [1], [2].

There can be multiple root causes for the bugs that require long time to fix; e.g., difficult to fix, requiring large number of changes in the code, or no one cares. Managers and developers can review backlog of bug reports from the perspective of such known causes and assign them to appropriate person for fixing. For example, 'difficult to fix' bugs can be assigned to senior developers and 'no-one cares' ones can be removed from the backlog to avoid repetitive review in the next iteration. However, manual process of identifying bug reports that can take shorter or longer time to fix can be daunting, especially when the number of bug reports are large. Automated solutions can help alleviate this problem.

- *M. Habayeb, S.S. Murtaza, and A.B. Bener are with the Department of Mechanical and Industrial Engineering, Data Science Lab, Ryerson University, Toronto, Ontario M5B 2K3, Canada. E-mail: {mayy.habayeb, syed.shariyar, ayse.bener}@ryerson.ca.*
- *A. Miranskyy is with the Department of Computer Science, Data Science Lab, Ryerson University, Toronto, Ontario M5B 2K3, Canada. E-mail: avm@ryerson.ca.*

Manuscript received 11 Jan. 2017; revised 14 June 2017; accepted 27 Aug. 2017. Date of publication 27 Sept. 2017; date of current version 18 Dec. 2018. (Corresponding author: Andriy Miranskyy.)
Recommended for acceptance by T. Bultan.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2017.2757480

In 2011, Bhattacharya et al. [3] evaluated various univariate and multivariate regression models to predict the time required to fix bugs. They employed various attributes in bug repositories used by prior researchers. They concluded that the predictive power of existing models for bug fix time predictions is only between 30 to 49 percent.

The time to fix bugs has been studied by several research groups over the last decade [3], [4], [5], [6], [7], [8], [9], [10]. Some studies have focused on predicting bug fix time through classification, for example classifying bug fix time into fast/slow [4], [5], [6], classifying them into cheap and expensive [1], or by employing multi-classification using the equal-frequency binning algorithm [7]. Other studies have used regression techniques to predict bug fix time [3], [8]. Another group of studies has focused on assisting the bug triage effort by introducing recommendations for developer/reviewer assignments [11], [12], [13] or by automatically filtering out certain type of bugs [14], [15], [16], [17].

Most of the previous researchers have focused on the use of frequency of occurrence of activities in bug repositories to build their models for the required time to fix bugs. These attributes include the number of copied developers, the comments exchanged, the developers involved, etc. The frequency based models ignore the temporal characteristics of the activities in bug repositories. Let us consider the example of a chain of activities in a bug repository. A junior developer fixes a bug and sends the code patch to a senior developer for review. The senior developer can approve the request, assign the bug to another developer, or write a comment on the "whiteboard". This sequence of events continues until the bug is resolved.

Similarly, the bug life cycle in a bug management system is a further evidence of temporal activity. For example, Fig. 1 shows the bug life cycle of the Bugzilla system [18].

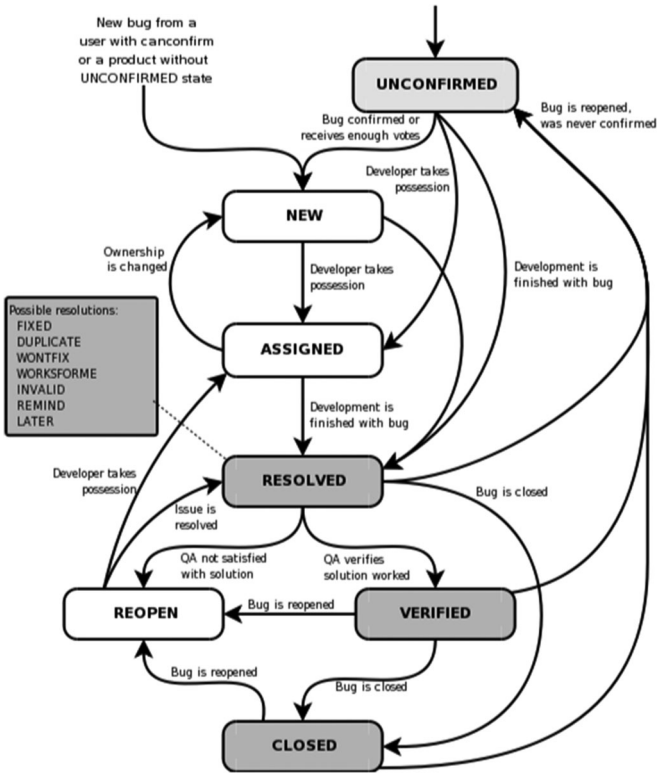

Fig. 1. The life cycle of a Bugzilla bug (acquired from [18]).

The bug report usually remains in an unconfirmed state until an initial assessment (triage[1]) effort confirms that it requires investigation, or it can start from a new state provided that the reporter confirms it. Once confirmed, a bug is either assigned to a particular developer or it is made publicly available so that any developer may volunteer to fix it. Once the status is reported as "resolved", "verified", or "closed" the bug is considered to be closed and an appropriate value is populated in the resolution attribute (e.g., fixed, duplicate, incomplete). One would assume from Fig. 1 that a bug would pass through many stages before getting to the resolved state. However, by taking a closer look at the data we noted that many bugs actually start as unconfirmed and move immediately into the resolved state without passing through an assignment state. For example, in the case of Mozilla Firefox project in Bugzilla, we tracked the status changes and determined that 94.064 percent of bugs in the unconfirmed state moved to the resolved state, and 70.56 percent of bugs in the new state moved to resolved state. This indicates that these bugs passed only through two states and that the status field is not indicative of the true bug state. After discussing this point with Firefox developers, we concluded that there are many other states that the bug report could be in, such as: waiting on reporter feedback, waiting on code review approval, development activities being carried out, idle state, etc.

Thus, there are temporal state activities of a bug, activities of developers for a bug, and at the same time there are some hidden temporal activities in bug repositories during each bug's life cycle. We, therefore, propose an approach to predict the time to fix bugs based on the Hidden Markov Model (HMM) [19]. HMM can model doubly stochastic processes in a system; that is, a visible stochastic process and, in our case, a hidden stochastic process that exists in system-bug repositories [19]. We applied HMM on historical bug reports by first transforming bug report activities into temporal activities, and then training HMM on those temporal activities. We classified the new (unknown) bug reports into two categories: (a) bugs requiring a short time *(fast)* to fix; and, (b) bugs requiring a long time to fix *(slow)*.

We performed our experiments on the Mozilla Firefox project, which is a Free and Open Source web browser that is developed for Windows, OS X, and Linux, with a mobile version for Android. It is highly popular and has a widespread user base of over 450 million users around the world [20]. Bugzilla is the bug tracking system for Mozilla Firefox, which also has a widespread user base. These systems are also widely used in many other studies [1], [4] allowing us to provide a comparison of our approach on a well known dataset. We compared our HMM based approach against the existing approaches that do not consider the temporal characteristics of the bug life cycle, and in this particular case study we found that our predictions have higher accuracy.

1.1 Research Problem

Our research problem is to identify at an early stage the bug reports that would require a long time to fix. Our focus is to provide a quality team with an early indication that there is a high probability that a bug report will remain in their system for more than a certain time threshold, for example two months, so that they can assess the need to re-prioritise.

1.2 Contribution

Our work has three main contributions:

1) A technique to create a temporal dataset of activities that occurred during the life cycle of bugs in a bug repository.
2) A Firefox dataset of temporal activities, available online at https://drive.google.com/drive/u/1/folders/0B5TJwohpS9LzcHc3NldtdjZQRFk
3) An approach to predict the time to fix bugs using temporal activities by using the HMM.

1.3 Research Outline

Section 2 contains related work and summaries of other research. Section 3 illustrates our approach and model, and provides an introduction to the concepts of HMM theory used in this research. In Section 4 we present three experiment scenarios to illustrate the possibilities of using temporal sequences in the predictive domain of time to fix a bug. In Section 5, we distinguish the use of the same attributes as temporal sequences versus frequency counts. In addition, we compare our approach with the approach of Zhang et al. [5]. In Section 6, we show further experiments regarding the optimal size of training data and change in severity. We follow this by highlighting threats to validity in Section 7. We conclude and mention intended future work in Section 8.

Throughout the course of this research we will use the terms "status" and "state" interchangeably.

1. Triage: The process of determining the most important people or things from amongst a large number that require attention: Oxford Dictionary.

2 Related Work

In 2006 Kim et al. [2] studied the life span of bugs in ArgoUML and PostgreSQL projects. They reported that the bug-fixing time had an approximate median of 200 days. Due to the high cost of maintenance activities and high volume of bug reports submitted, especially for many popular Open Source systems, several research have been conducted to predict the estimated time needed to fix a bug [1], [4], [5], [6], [7], [8], [11], [14], [15], [16], [17], [21]. These efforts can be grouped into two categories. The first group focuses on predicting the overall estimated time required to fix a bug report [1], [4], [5], [6], [7], [8]. Mostly, the studies in this group employ classification or regression techniques. This group uses initial and post submission data as features. The second group focuses on reducing the estimated time required to fix the bug during the initial bug triage process [11], [14], [15], [16], [17], [21]. The studies in this group focus on identifying and filtering certain bug reports or automatically assigning the bug report to the best candidate developer in an effort to reduce the overall bug report fix time. These studies mainly use the initial attributes available with the bug report (e.g., the description and summary of the bug report, the report submitter details, and the time of submission) with machine learning algorithms to predict the time required to fix the bug.

2.1 Bug Fix Time Estimation and Prediction

In 2007, Panjer [7] explored the viability of using data mining tools on the Eclipse Bugzilla database to model and predict the life cycle of Eclipse bugs from October 2001 to March 2006. He employed multi-class classification using the equal-frequency binning algorithm to set the threshold bins. Afterwards, he used various machine learning algorithms, such as 0-R and 1-R, Naive Bayes, Logistic regression, and Decision Trees to achieve a prediction performance of 34.5 percent.

In 2007, Hooimeijer et al. [1] presented a descriptive model of bug report quality. Their main focus was to investigate the triage time (inspection time) of a bug report. They applied linear regression on 27,000 bug reports from the Firefox project in an attempt to identify an optimal threshold value by which the bug report may be classified as either "cheap" or "expensive".

Our work is similar to the above two studies in terms of setting thresholds for classification and targeting to identify bug reports that would require a longer time (slow). However, we differ in terms of the size and scope of the dataset used, the approach, and the algorithm.

In 2009, Anbalagan et al. [8] investigated the relationship between bug fix time and number of participants in the bug fixing process. They carried out their tests on 72,482 bug reports from the Ubuntu project. They found a linear relationship between bug fix time and the number of participants involved in the bug fix. They developed a prediction model using linear regression, and one of their key findings was that there is a strong linear relationship (the correlation coefficient value is ≈ 0.92) between the number of people participating in a bug report and the time that it took to correct it. We follow their recommendations by considering the human factor, but we differ by focusing on the levels of involvement and interactions between the participants rather than just using the number of participants or unique participants. In particular, we focus on the temporal characteristics.

In 2010, Giger et al. [4] explored the possibility of classifying bugs into "slow" and "fast" based on the median of the total fix time. They used decision trees on six Open Source datasets (including a small sample dataset of Firefox). First, they used the median of 359 days as a threshold for classification, and reported a precision of 0.609 and a recall of 0.732. Second, they followed this with a series of tests including post-submission data where they reported a precision of 0.751 and a recall of 0.748 against a median of 2,784 days. Our experiments use 86,444 Firefox bugs and a fixed threshold of two months. In addition, we filter out the reports that are reported as enhancements, and we employ the temporal sequencing of events rather than counts.

In 2012, Lamkanfi et al. [6] suggested filtering out bugs with very short life cycles, they referred to them as conspicuous reports. They experimented with ten different datasets, one of which was Firefox with 79,272 bug reports covering the period of July 1999 to July 2008. They used the median as a threshold and reported a slight improvement on the prediction results if the bugs with short life cycles are filtered out. In their experiments, they applied Naive Bayes classifiers and used a median threshold of 2.7 days. We apply their recommendation in our experiments. However, we differ in the threshold calculation because we filter out bug reports that are closed on the same date of submission (i.e., day 0 bugs) from the median calculation. We also differ in the temporal methodology of our classification.

In 2013, Zhang et al. [5] experimented on three commercial software projects from CA technologies. They explored the possibility of estimating the number of bugs that would be fixed in the future using Markov models and estimating the time required to fix the total number using Monte Carlo simulations. They also proposed a classification model to classify bugs as slow and fast using k-Nearest Neighbour against various thresholds related to the median of times required to fix bugs. In their study, the median of time required to fix is not revealed due to data sensitivity issues; thus, they use a unit scale where 1 unit is equal to the median. For feature space they use summary, priority, severity, submitter, owner, category, and reporting source. Our work is similar to their work in terms of classification using the median fix time as a threshold and the concept that similar bugs would require a similar fix time, yet we differ in terms of the overall approach and algorithm because we use HMM instead of K-Nearest Neighbour. We do not employ textual similarity either, since it is computationally expensive.

The main difference between our work and earlier work is the use of temporal sequences of bug activities happening during the triaging process. This temporality of activities is not considered by earlier approaches. In our approach, each feature carries a different weight based on the events that happened before it. For example, let us look at the carbon copy activity. In the case of other researchers [1], [4], [6], [7], [8], the total number of occurrences of carbon copies that happened for a given bug report is used as the weight for the carbon copy feature. However, in our approach, the weights of this feature are based on the temporal ordering

of activities in bug reports. For example, a carbon copy activity followed by an activity of bug fixing has a different weight than the carbon copy activity followed by the change in priority of this bug report. In this way, our approach captures the temporality of features. Through these temporal sequences of activities, we capture involvement of developers, reporters, and managers alongside their communications and give an early indication of whether the bug is going to be slow (troublesome). The researchers mentioned above have not done this.

2.2 Reducing Bug Triage Time

Some researchers have focused on reducing the bug triage effort, which usually happens during the first few days from the date of submission [11], [14], [15], [16], [17], [21].

Cubranic et al. [14] and Anvik et al. [11] investigated the possibility of automating bug assignments through text categorization. In 2004, Cubranic et al. [14], attempted to cut out "the triageman" by automatically assigning bugs to developers based on text similarity. The team experimented on the Eclipse project and used a Naive Bayes classifier, and reported 30 percent correctly predicted assignments. Similarly, in 2006 Anvik et al. [12] used SVM to recommend the best developer. In 2011, Anvik et al. [11] further extended this previous work to cover five Open Source projects and experimented with six machine learning algorithms. They reported a 70-98 percent precision once the recommender is implemented within the bug triage process and the triageman is presented with three recommendations. Our work differs from these studies in many aspects because we focus on bug fix time prediction after the reports have been triaged.

In 2008, Wang et al. [15] investigated both the natural language and the execution information of a bug report to determine the duplicate bug reports automatically. They reported a detection rate of 67-93 percent in comparison to 43-72 percent using natural language alone. Similarly, Jalbert et al. [17] attempted to identify duplicate reports, they experimented on a dataset of 29,000 bug reports from the Mozilla project. Their model used bug report features (self-reported severity, the relevant operating system, and the number of associated patches/screenshots), textual semantics, and graph clustering to predict duplicate status. They reported that their system was able to filter out 8 percent of duplicate reports.

Weiss et al. [21] mined the bug database of the JBoss project, stored in the JIRA issue tracking system [22]. They used Lucene, a text similarity measuring engine, and kNN clustering to identify the similarities of effort spent on bug fixing. They estimated bug fixing effort for a new bug report. They reported results with a one hour error rate from the actual effort. In our study we do not carry out a similarity analysis but instead we focus on the temporal activities of the maintainers. Antoniol et al. [16] distinguished between requests for defect fixing and other requests based on the text of the issues posted in bug tracking systems.

In 2013, Mani et al. [23] investigated the role of contributors in bug fixing process. They pointed out the importance of catalyst contributors as key features to the bug assignment process. They defined catalyst contributors as either bug tossers or commenters (not the bug reporter or the bug committer). They examined the effect of the catalyst contributors on 98,304 bug reports from 11 open source projects maintained by Apache Software Foundation (ASF) and five commercial projects developed by IBM. They found that 65 percent of bugs across projects require 2 to 5 non-committers on average. To identify the essential non-committers (catalyst), they propose and implement a network analysis based approach called Minimal Essential Graph (MEG). In this approach, they investigate how many catalysts retained in MEG are predominantly commenters or bug-tossers. They also suggest that future studies take into consideration the catalysts. Our work, takes into consideration the catalysts as we track key activities carried out by any contributors to the bug fixing process (shown in Section 3.1). However, our work focuses on the classification of time to fix bugs not on the investigation of catalysts in bug reports.

In 2013 Tian et al. [24] suggested an approach to automatically assign a priority to a bug report. They used several features, such as temporal, textual, author, related-report, severity, and product to predict the priority of bugs. They report relative improvement of F-measure by 58.61 percent (in comparison with baseline techniques). Our work is focused on classifying the time to fix bugs and not on priority assignment. We have also provided a comparison to the contemporary technique. In 2014 Garcia et al. [25] focused their efforts on predicting blocking bugs. Blocking bugs are those bugs that may increase maintenance costs, reduce overall quality, and delay the release of the software systems. They built decision trees for each project, predicting whether a bug will be a blocking bug or not. They also analysed these decision trees in order to determine which attributes are the best indicators of these blocking bugs. They reported that their prediction models achieved F-measures of 15-42 percent: four times better than the baseline random predictors. Our work does not focus on blocking bugs but on classification of the time to fix bugs.

Our work differs from the works of the researchers mentioned above. First, our proposed prediction model is based on non-textual attributes. Second, we use the temporal sequences of the activities happening during the triage process. Our goal is to provide an early indication of troublesome bug reports rather than automatic filtering of bug reports.

3 The Approach

Our approach can be summarised in three steps, as illustrated in Fig. 2. First, we extract temporal sequences of developer activities from a bug repository, such as Bugzilla repository.[2] We describe this step in detail in Section 3.1. Second, we use the temporal sequences of activities of the resolved bugs to train HMMs. We train two HMMs by separating resolved bugs into two categories: (a) bugs with a long time to fix (slow); and (b) bugs with a short time to fix (fast). We separate two types of bugs by measuring the median number of days taken by them for resolution. Bugs with a resolution time more than the median number of days are assigned to the long-time (slow) category and bugs with a lesser resolution time than the median number of days are assigned to the short-time (fast) category. One

2. https://www.bugzilla.org/

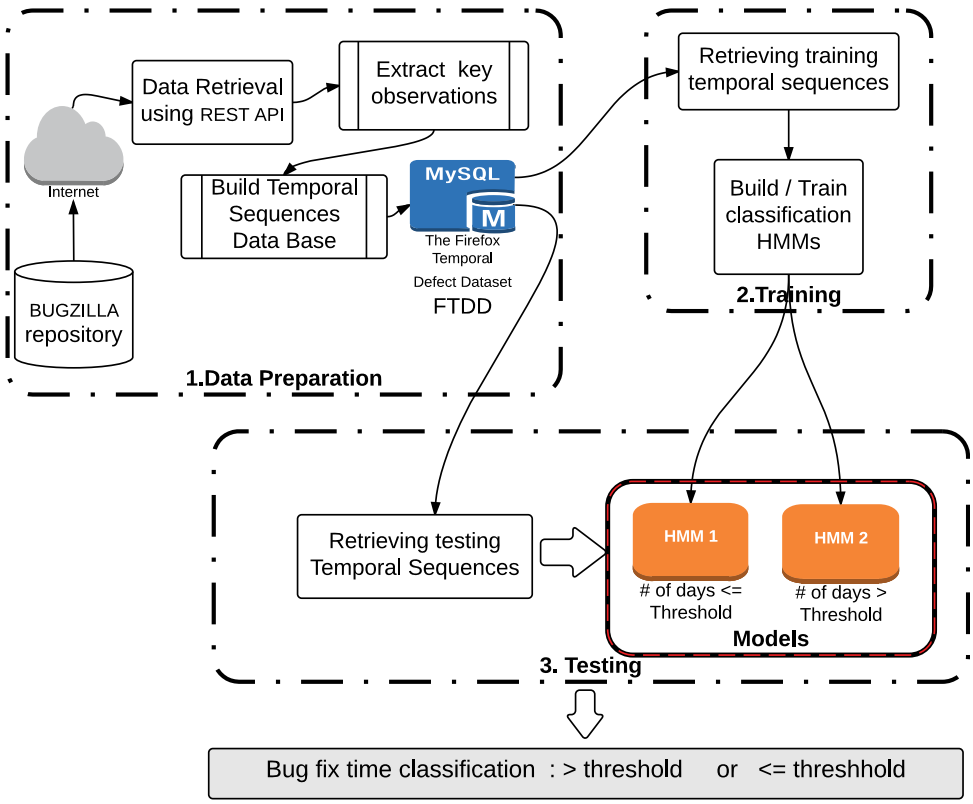

Fig. 2. The approach.

HMM is then trained on each category. This step is further explained in Section 3.2. Third, we extract the first few temporal sequences of activities for the latest bugs and pass them to the trained HMMs. The trained HMMs classify each bug into two categories: a slow time to fix and a fast time to fix. The intuition is to identify the bugs that are going to take a longer time to fix in the early stages of the life cycle of the bug and to help the managers in allocating resources to fix bugs. This step is explained in detail in Section 3.3.

3.1 Extracting Temporal Sequences of Developer Activities

The approach that we used to extract developer activities from a bug repository depended on the software quality process setup within an organisation and the features available in a bug repository (or bug tracking system). Nevertheless, the same concepts can be easily adapted and extended to different projects and bug tracking systems. In our study we cover a period of eight years (2006-2014) of the Firefox project in the Bugzilla repository. We chose the Firefox project because of its widespread customer base, the diversity of its end users, and its popularity among the research community [1], [3], [13], [15], [26].

We excluded new feature requests because they are not bugs and they tend to follow a different development cycle. A sample history log is shown in Fig. 3. In this history log, we can observe the temporal activities that take place in the bug repository while the developers are fixing bugs. Fig. 3 shows the id (the bug id), who (developers' emails), when (time of the activity), field name (type of activity), added (the new value of the type of activity), and removed (the old value of the activity).

Our criterion for selecting the temporal activities (as shown in Fig. 3) from the bug history logs is based on the level of involvement by developers and bug reporters. All of the activities that occur in a Bugzilla Firefox repository are listed in Table 1. We divided these activities into three categories: involvement, communications, and bug condition. These are further explained below.

3.1.1 *Involvement*

Involvement activities reflect the types of people involved in the life cycle of the bug report. According to this rationale, we focused on extracting certain key activities related to involvement, as described below.

Reporters. Several studies have indicated the importance of the reporter's experience. Hooimeijer et al. [1] called this

id	who	when	field_name	added	removed
324056	annie.sullivan@gmail.com	2006-01-19T21:56:17Z	flagtypes.name	review? (bugs@bengoodger.com)	
324056	bugs@bengoodger.com	2006-01-20T01:43:52Z	flagtypes.name	review+	review? (bugs@bengoodger.com)
324056	annie.sullivan@gmail.com	2006-01-20T17:04:07Z	status	RESOLVED	NEW
324056	annie.sullivan@gmail.com	2006-01-20T17:04:07Z	resolution	FIXED	
324063	gavin.sharp@gmail.com	2006-01-19T23:05:08Z	cc	gavin.sharp@gmail.com	

Fig. 3. Sample history log.

TABLE 1
Observation Set

Observation	Symbol	*Description*
Reporting	N	Reporter has only reported one bug as of bug creation date.
	M	Reporter has reported more than one and less than ten bugs prior to bug creation date.
	E	Reporter has reported more than ten bug reports prior to bug creation date.
Assignment	A	Bug confirmed and assigned to a named developer.
	R	Bug confirmed and put to general mailbox for volunteers to work on.
Copy	C	A certain person has been copied on the bug report.
	D	More than one person has been copied within one hour on the bug report.
Review	V	Developer requested code review.
	Y	Response to code review.
	S	Developer requested super review.
	H	Response to super code review.
File Exchange	F	File exchanged between developers and reporters.
Comments exchange	W	Comment exchanged on whiteboard.
Milestone	L	A Milestone has been set for solution deployment.
Priority	P	Priority changed for bug report.
Severity	Q	Severity changed for bug report.
Resolution	Z	Bug reached status resolved.

variable "Reporter Reputation", while Zanetti et al. [27] confirmed this finding. We capture the reporters experience by indicating their status as novice (N), intermediate (M), or experienced (E) at the time of the report creation (for details, see the description of N, M, E in Table 1).

Developers. Giger et al. [4] and Jeong et al. [13] used the developer assigned to a bug as a key feature in their predictive models. Bicer et al. [28] used the social network analysis on issue repositories to predict defects. Caglayan et al. [29] and Miranskyy et al. [30] revealed interesting facts related to the developer networks. Once received, a bug report usually remains in an $unconfirmed$ state until an initial assessment (triage) effort confirms that it requires investigation, which changes its state to confirmed. Note that if a bug is reported by a person with authority $can-confirm$, its starting state is immediately set to new (i.e., confirmed). Once confirmed, a bug is either assigned to a particular developer or made publicly available so that any developer may volunteer to fix it. We distinguish these two cases using the symbols A and R, respectively (see Table 1 for details).

Experts involvement. Experts can be copied or referred to in order to approve development changes. We track these observations by mining and extracting information about notification activities and code reviews.

Carbon Copies (CC): Both Giger et al. [4] and Panjer et al. [7] used the CC count generated at various time intervals as a key feature in their prediction studies. Zanetti et al. [27] also employed CC activities to capture the social network interactions between developers and reporters. We denote any CC (i.e., notification activity) using the symbol C. If more than one C occurs in an hour, then we label this case with the symbol D. Such a high frequency of notifications typically indicates important observations.

Code Review: Many researchers have extensively investigated code reviews. For example, using the Firefox dataset, Jeong et al. [13] suggested an algorithm for recommending suitable reviewers and attempted to predict levels of code acceptance. We extract information about code review requests and classify them as $normal$ and $super$ reviews denoted by V and S, respectively. Normal code reviews are usually assigned to the module owner, and super code reviews are usually assigned to a set of senior developers who oversee significant architectural refactoring and API changes (see [31] for details of the review process). These observations indicate that bug fixing has neared completion and that the bug owner awaits the reviewer's decision (i.e., accept or reject). We also capture responses to $normal$ and $super$ reviews, denoted by Y and H, respectively. This indicates that the waiting period is over. To the best of our knowledge, our dataset is the first to capture super review requests and responses.

3.1.2 Communications

Communication between developers indicates active efforts and progress toward resolving the bug. We capture two communication observations: 1) file attachments exchanged among developers, reviewers, or reporters, denoted by F; and 2) comments exchanged between developers, indicating a discussion or exchange of information,[3] denoted by W.

3.1.3 Bug Condition

We capture key events throughout the bug's lifetime. These key events affect its overall status and possibly indicate a change in the expected sizing or resolution type. We capture four observations:

1) Change in severity: Severity is usually set by the reporter and is changed infrequently by the triage or development team. Severity was changed for only 7 percent of bugs in the Firefox Bugzilla repository. Increased or decreased severity influences the overall standing of the bug report. We denote changes in severity by Q.
2) Change in priority: The triage or development team sets the priority. Again, the priority was changed for only 7 percent of the bugs in the Firefox Bugzilla repository in order to more clearly indicate the assessment effort being carried out and the condition of the bug. Although most studies have found that this field has low predictive power and effect, it might produce different results when put in the context of a temporal sequence. We denote a change in priority with the symbol P. If both priority and severity change during the life cycle of the bug fix, our model captures all three cases: (a) when there is only a change in priority; (b) when there is only a

3. Although these have been used in previous studies, they have been used as frequencies at certain time snapshot [1], [4], [7].

TABLE 2
Changes in Severity and Priority of Firefox Bug Reports

	Number of reports	% Of Total
Bug reports with only change in severity	3,451	5.45%
Bug reports with only change in priority	3,790	5.98%
Bug reports with both change in priority and change in severity	588	0.93%
Bug reports with unchanged severity and priority	55,521	87.64%
Total	63,350	100%

change in severity; and (c) when there is both a change in priority and severity. We arrange these cases based on the timestamp of occurrence. In other words, activities of the users of the bug tracking system (that are recorded by the system) are ordered from the earliest to the latest (for a given bug) and they are fed into the model.

In addition, if activities happen in between a change in priority and a change in severity, such as a file transfer or an assignment, we pick them up too, and present them in the temporal sequence. Table 2 shows the percentage of occurrence of each of the abovementioned cases.

3) Milestone set: This indicates the identification of a target release for deployment of fix, we denote this observation with symbol L.
4) Resolution reached: This final activity in the sequence of observations is denoted by Z.

We store all of the key activities in a database, which we call the "Firefox Temporal Defect Dataset (FTDD)"; we have already made the FTDD publicly available [32]. Thus, we transform a bug report into a set of temporal activities, as shown in Fig. 4. The order of the symbols for a bug represents the sequential order in which they occur in the bug repository. The meaning of the symbols can be determined from the symbols in the Table 1.

We use the heuristic assumption that similar sequences will take similar periods of time. This concept has been previously used by Jeong et al. [13] in their research to reduce the tossing period (waiting time) of bugs before their assignments to developers. We have expanded on this concept by extracting the assignment sequences alongside the human involvement, human communication, and bug condition sequences to identify troublesome bugs earlier.

```
N,C,C
N,C,
N,C,C,C,
N,D,W,C,
E,C,W,P,W,P,C,A,A,W,V,A,Y,V,V,Y,V,Y,B,W,L
N,C,C,C,
N,C,W,C,W
```

Fig. 4. Sample of temporal sequences each row represents a bug report.

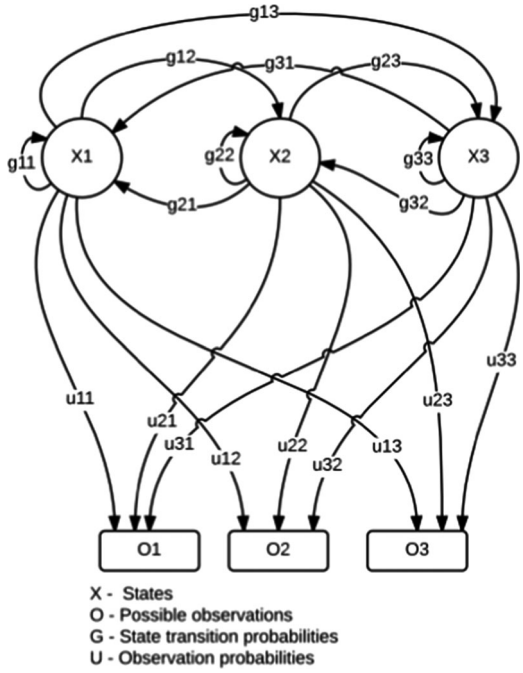

Fig. 5. A hidden Markov model.

3.2 Training

As mentioned earlier in Section 1, HMM can model doubly stochastic processes in a system; that is, a visible stochastic process and a hidden stochastic process that exist in a system-bug repositories [19]. HMM was introduced in the late 1960s and early 1970s [19]. HMM is a statistical model that is based on two stochastic processes. The first stochastic process is a Markov chain that is characterised by states and transition probabilities. The states of the Markov chain are usually hidden in an HMM. The second stochastic process represents observations that occur in a temporal order and are associated with the states of the first stochastic process. Fig. 5 illustrates a simple HMM. The components shown in Fig. 5 represent an HMM, and HMM can be summarised as follows [19]:

- Number of States (α): α represents the number of states, and the individual states are represented as $X = \{X_1, X_2, \ldots, X_\alpha\}$ and the state at time t as q_t. In Fig. 5, the number of states, α, is 3.
- Observations (β): β represents the number of observation symbols per state. Individual observations can be denoted by $O = \{O_1, O_2, \ldots, O_\beta\}$. The observation symbols correspond to the physical output of the system being modelled.
- State Transition Probability Distribution (G): The transitions between the α states are organised by a set of probabilities called the state transition probabilities. In our example, (g_{12}, g_{23}, g_{21}, etc.) denote the states transition probabilities. The state transition probability distribution is represented as $\{G\} = g_{ij}$, where g_{ij} is the probability when the state at time $t+1$ is X_j and the state at time t is X_i. Formally, g_{ij} is defined as

$$g_{ij} = \mathbb{P}[q_{t+1} = j | q_t = i], \quad 1 \leq i, \quad j \leq \alpha, \tag{1}$$

where q_t denotes the current state.

TABLE 3
Probability of Seeing an Umbrella

Weather condition	Probability
Sunny	0.1
Rainy	0.8
Foggy	0.3

TABLE 4
Probability of Weather Condition Changes

		Tomorrow's Weather		
		Sunny	Rainy	Foggy
Today's Weather	Sunny	0.8	0.05	0.15
	Rainy	0.2	0.6	0.2
	Foggy	0.2	0.3	0.5

- Observation Symbol Probability Distribution (U): The observation symbols' probability distribution in a state j is denoted by $U = \{u_j(\kappa)\}$, where $u_j(\kappa)$ is the probability that symbol O_κ is observed in state X_j, and is represented as

$$u_j(\kappa) = \mathbb{P}[O_\kappa \text{ at } t | q_t = X_j], \quad 1 \le j \le \alpha, \quad 1 \le \kappa \le \beta, \tag{2}$$

  where O_κ denotes the κth observation symbol.
- Initial State Distribution (π): HMM needs to be initialised at the beginning; the initial state distribution at $t = 1$ is given as

$$\pi = \{\pi_i\}, \quad \pi = \mathbb{P}[q_1 = X_i], \quad \text{and} \quad 1 \le i \le \alpha, \tag{3}$$

  where q_1 represents the probability at time 1.

Thus, an HMM model λ, can be represented with the following compact notation

$$\lambda = (G, U, \pi).$$

The HMM models can be discrete or continuous. If the observations are recorded at certain points in time, then we consider the model to be discrete. If the observations are continuously measured, then a continuous probability density function is used instead of a set of discrete probabilities [19].

To illustrate we will use the following example, adopted from [33]: suppose someone is locked in a room for several days, and he/she is asked about the weather outside. Then another person enters the room. The only piece of evidence the person has is whether the new person who comes into the room carrying an umbrella or not. Let us suppose that the probabilities of seeing an umbrella, based on a weather condition, are provided in Table 3.

The hidden states in this example would be three states: Sunny, Rainy, and Foggy. The observation sequence is developed by recording daily if the person coming in holds an umbrella or not. Thus, the emission probabilities can be derived from Table 3 and the state transition probabilities can be derived from Table 4. By using these instruments, we can infer the weather states based on the actions of the new person coming into the room.

In our case, we use the discrete model. An example of an HMM model in our case can be seen in Fig. 6.

In Fig. 6, the actual states are hidden from us, the only thing truly visible are the observations (comments exchanged, code reviewed, etc.) being emitted from the bug fixing processes (assessment, development, code review, deployment, etc.). The processes (assessment, development, code review, etc.) are comprised of the set of tasks that need to be coordinated between the developers and managers in order to fix (resolve) the bug. This task co-ordination is visible through a set of observations (activities) in the bug repository, such as comments exchanged, code reviewed, etc. The task co-ordination can also result in the back and

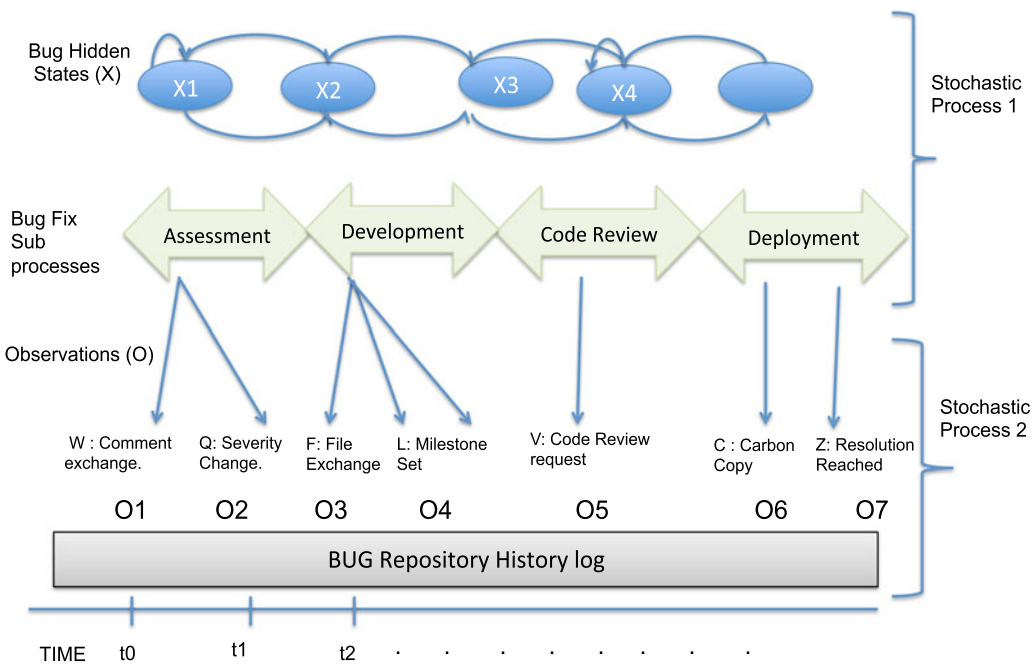

Fig. 6. HMM model for bug fix time classification.

forth movement between processes, such as development and code review, deployment and assessment, etc. This is also visible in the bug repository through observations. Thus, there are hidden states in the bug fixing processes, and the observations can occur on each hidden state, causing the transitions from one state to another. Therefore, this process can be modelled using HMM.

To build an HMM model, $\lambda = (G, U, \pi)$, automatically from a sequence of observations (activities in our case) $O = \{O_1, O_2, \ldots, O_\beta\}$, we need to train it on the sequences of observations [19]. In other words, we need to learn the parameters G and U from the sequences of observations, such that $\mathbb{P}\{O|\lambda\}$ is maximised. The Baum-Welch (BW) algorithm is one of the most commonly used algorithms for learning (or training) the parameters G and U [19]. It follows an iterative process using the forward and backward algorithms to determine the probabilities of parameters G and U from a sequence of observations. We also used the BW algorithm to train HMM.

We retrieve the sequences of observations for bug reports from the bug repository for training HMM. For example, a sequence of observations for a bug report could look like: new reporter, carbon copy, whiteboard comment exchanged, carbon copy, whiteboard comment exchanged. According to our shorter representation, this would be a sequence of symbols N, C, W, C, W. These symbols are also shown in Table 1. We extract such sequences of observations for each bug report until their final resolution. We actually train two HMMs on these sequences of observations of bug reports. The first HMM is trained on these observation sequences related to bug reports that require a total number of days to fix the bug (reach a resolution) below a certain threshold. The second HMM is trained on observation sequences related to bug reports that require the total time to fix the bug beyond a certain threshold. The threshold is calculated based on the median number of days required to reach a resolution. The approach of partitioning bugs has also been adopted by earlier researchers in training a machine learning algorithm [4]. The two HMMs are then used to determine the type of the HMM of the new observation sequence of the latest bug report. The type of HMM determines the number of days that the bug will take to resolve; that is, above the median or below the median. This is further discussed in the next f.

3.3 Testing

Our proposed approach focuses on classifying the new unresolved bug report into one of two types: the report will take long time (slow) to fix, or the report will take short time (fast) to fix. In order to achieve this, we retrieve observation sequences, $O = \{O1, O2, O3, \ldots, On\}$, of the latest bug reports, and pass them to two HMM models, $\lambda 1$ and $\lambda 2$, which were trained earlier on the resolved bugs. We then determine the probability that the observations are generated by one of the two models; that is, we determine P $\{O|\lambda 1\}$ and P$\{O|\lambda 2\}$ using the forward algorithm [19]. We select the maximum of P$\{O|\lambda 1$ $\}$ or P$\{O|\lambda 2$ $\}$ and assign the new unresolved bug report to the type of bug reports that the selected HMM represents. We only select the first few observations, O, for a new bug report. For example, we select the number of observations occurring a) only on day 1 of the bug report or b) the first five observations, and pass them to HMMs to identify the time that a bug report is going to consume; that is, greater than the median number of days or less than the median number of days.

TABLE 5
Confusion Matrix

		Predicted	
		slow	fast
Actual	slow	TP	FP
	fast	FN	TN

3.4 Evaluation Criteria

To evaluate our proposed approach, we divide the bug repository dataset into two parts: (a) bug reports for training; and, (b) bug reports for testing. We use approximately 60 percent of the resolved bug reports in the training set and 40 percent of the resolved bug reports in the test set. The known resolution time of the bug reports in the test set allows us to evaluate the performance of the HMMs.

We measure the performance of our classifier by first using a confusion matrix and then determining the precision, recall, F-measure, and accuracy of the classifiers. The confusion matrix stores correct and incorrect predictions made by classifiers [34]. For example, if a bug is classified by HMMs as requiring a long time to fix (slow) and it is truly taking a long time to fix (slow), then the classification is a true positive (TP). If a bug is classified as slow and it is not actually slow, then the classification is a false positive (FP). If a bug is classified as requiring a short time to fix (fast) and it is actually in the slow class, then the classification is false negative (FN). If it is classified as fast and it is actually in the fast class, then the classification is true negative (TN). Table 5 summarises these four possible outcomes.

Using the values stored in the confusion matrix, we calculate the precision, recall, F-measure, and accuracy metrics for the bugs to evaluate the performance of our HMM models [34].

Precision. Precision is the fraction of retrieved instances that are relevant, and it is calculated as [34]

$$Precision = \frac{TP}{TP + FP}. \tag{4}$$

For our model, high precision would mean that the percentage of correctly identified troublesome (slow) bug reports out of the total bug reports that our model predicted as slow would be high. Similarly, the low precision values for our model would imply lower percentage of correctly identified slow bug reports out of the total predicted slow bug reports.

Recall. Recall, also known as sensitivity, is the fraction of relevant instances that are retrieved, and it is calculated as [34]

$$Recall = \frac{TP}{TP + FN}. \tag{5}$$

For our model, recall values would show the percentage of correctly predicted slow bug reports out of total slow bug

TABLE 6
Number of Bugs Resolved by Year

Year	Total bug reports submitted	No. of new feature requests	No.of bug reports	No.of resolved bugs	No. of bugs resolved excluding bugs resolved on creation date (day 0)	Median No. of days till resolution	% of resolved bug reports
2006	11,571	809	10,762	10,531	7,125	179	97.85%
2007	9,962	662	9,300	8,918	6,526	194	95.89%
2008	16,070	1,012	15,058	14,114	10,557	230	93.73%
2009	11,967	659	11,308	10,290	7,538	203	91.00%
2010	14,402	861	13,541	10,633	7,854	97	78.52%
2011	12,381	720	11,661	8,362	5,834	28	71.71%
2012	11,609	441	11,168	7,024	4,659	10	62.89%
2013	12,485	449	12,036	8,508	6,974	17	70.69%
2014	15,652	373	15,279	8,064	6,283	7	52.78%
Totals	116,099	5,986	110,113	86,444	63,350		78.50%

reports present in the test set. Higher recall values would mean that higher percentage of the slow bug reports in the test set are correctly predicted.

Accuracy. Accuracy reflects the percentage of correctly classified bugs to the total number of bugs. It is computed as follows [34]

$$Accuracy = \frac{TP + TN}{TP + TN + FN + FP}. \quad (6)$$

F-measure. F-measure, also known as F-score or F1 score, can be interpreted as a weighted average of the precision and recall. It actually measures the weighted harmonic mean of the precision and recall, where F-measure reaches its best value at 1 and worst score at 0. It is defined as [34]

$$F\text{-}measure = \frac{2 \times Precision \times Recall}{Precision + Recall}. \quad (7)$$

4 Experiments

In this section we aim to demonstrate the application of our approach in the form of three different experiments. In the first experiment we focus on demonstrating the ability of classification on retrieving various fixed lengths temporal sequences of activities for bugs in the test set. In the second experiment we focus on classification of temporal sequences of bugs from the test set within the first week of their submissions. In the third experiment we train on bugs of previous years and test on bugs of future years. We begin by providing some initial statistics for the dataset and then explain all of the experiments.

4.1 Dataset: Firefox Bugzilla Project

As we mentioned in Section 3.1, we use bugs that are opened within a period of eight years (2006-2014). These bugs are stored in Firefox Bugzilla project. During this eight year period, 116,099 bug reports related to the Firefox project were submitted to the Bugzilla repository system. Of these reports, 5,986 were submitted as new feature requests and they were not considered in our experiments. We extracted the histories of all bug reports that had reached a status of resolved. In total, 86,444 reports of the 110,113 bug reports have been resolved. This indicates that there are 23,669 reports that have not reached a resolution, some dating back to 2006. These figures are illustrated in the first five columns of Table 6. The sixth column in Table 6 indicates the number of reports resolved per year, excluding all reports closed on submission date. We note that 26.7 percent of the total submitted bug reports are closed on the date of submission. In the seventh column in Table 6, we show the median of numbers of days required for resolution in each year. We note the median for the number of days required to resolve decreases as we move in time. This might give an initial impression that the bug process fix time is more efficient. However, we note that the percent of total resolved bugs, as illustrated in the last column of Table 6, decreases over time. This means that many of the reported bugs are resolved in subsequent years. In addition, when the remaining bug reports are resolved, the median will also increase. On further investigation of columns seven and eight from Table 6, we note that caution is required in selecting the dataset range because the last two or three years might not give a full picture of the remaining bugs in the systems. For example, the median number of days for resolution required for year 2014 is 7 days, which seems very optimistic. However, it only covers 47.22 percent of the bug reports because the rest of the bug reports tend to remain open for future years.

In Table 7 we take a closer look at the bug reports closed on date of submission; that is, total resolution time is 0 days. We note that only 6.98 percent are closed with a resolution value of *fixed*, while the rest are marked with a resolution of *duplicate*, *invalid*, *worksforme*, *wontfix*, or *incomplete*.

This is a reflection of the bug triage effort required during the first week of submission, and, as stated earlier, several studies have worked on addressing this area by suggesting machine learning algorithms that would automatically filter out duplicate reports [15], [17], [35], [36] or would automatically assign developers to bug reports [11], [12], [14]. This is also a reflection of the nature of the reporters, because most of these reports are submitted by novice reporters. We observed that 15,116 out of the 23,094 bug reports, closed on the day of reporting, were initiated by novice reporters. This is illustrated in Table 7.

Table 8 reflects the median number of days required to reach a resolution. The first row shows the full dataset and the second row shows the statistics in case we exclude the reports closed on the date of submission; that is, *day 0*

TABLE 7
Resolution of Bugs Closed on Submission Date versus Reporter Expertise

Resolution	Reporter			Total bug reports	Total bug reports %
	Novice	Moderate	Experienced		
DUPLICATE	7,961	3,238	1,354	12,553	54.36%
FIXED	717	262	633	1,612	6.98%
INCOMPLETE	467	134	24	625	2.71%
INVALID	4,670	1,349	414	6,433	27.86%
WONTFIX	322	110	101	533	2.31%
WORKSFORME	979	261	98	1,338	5.79%
Bug reports count	15,116	5,354	2,624	23,094	
Bug reports %	65.45%	23.18%	11.36%	100%	

TABLE 8
Statistics for Bug Resolutions

Resolution	Min	1st Qu.	Median	Mean	3rd Qu.	Max
Resolution of all bug reports	0	0	9	166.7	191	2,497
Resolution of bug reports excluding day 0	1	5	53	227.4	346	2,497

reports. We notice that the median of the number of days increases from nine days to 53 days around two months, if the bug reports closed on the submission date are excluded. This makes it more viable for time estimates. We, therefore, focus on the bug reports that have recorded actions one day after submission. This fact is also in line with Lamkanfi et al.s [6] findings that the fix times, as reported in Open Source systems, are heavily skewed with a significant amount of reports registering fix times less than a few minutes; that is, *day 0*. In addition, after excluding the bug reports that are closed on the date of submission, the median number of days (53) required for fixing is similar to the (55) median days used by Hooimeijer et al. [1] for the classification of bugs as cheap and expensive for the Firefox dataset. Accordingly, a two month time interval, which is closest to the 53 days of median resolution, is a good threshold to separate bug reports into two categories for training the HMMs.

4.2 Experiment 1: Using Fixed Length Temporal Sequence of Activities for Classification

In this experiment, we start by partitioning the dataset into two parts of 60 percent (training set) and 40 percent (test set). We train two HMMs' on the training set by partitioning it further into bugs with resolution time less than the median number of days and greater than the median number of days. This is already explained in Section 2.

TABLE 9
Median of Activities versus Median of Resolution Per Bug Report

Activity within the observation sequence	Median of No. of days for activity to occur	Median for No. of days till bug resolution
Second activity	2	60
Third activity	6	63
Fourth activity	8	44
Fifth activity	9	35
Sixth activity	12	33

After training HMMs, we use the 40 percent test set to simulate the scenario of having two, three, four, five, and six activities of an unresolved bug report in a bug repository and using HMMs to predict the (slow or fast) time to fix bugs. In Table 9 we illustrate the median number of days required for two to six activities to occur for bug reports. Table 9 also shows the median number of days required to actually resolve these bug reports. For example, the median number of days for the second activity to occur is two days, implying that 50 percent of the bug reports by the second day of submission have at least two activities. On the other hand, the median number of days required to resolve these bug reports was 60 days, implying that 50 percent of these defect reports required more than two months to resolve. Thus, there is room for our proposed predictive model to predict the time to fix bug reports.

Table 11 illustrates the length of sequence activity versus the median number of days required to fix the bug report. As can be seen from the second column, 62,443 of the bug reports have a total sequence length between two and twenty-five activities, which is around 98.5 percent of the total number of bug reports. Comparing column number one "Length of sequence" to column number five "Median number of days", we could not draw any direct relationship between the number of activities in a bug report versus the time to fix a bug. For example, the median number of days required to resolve bug reports with eight activities in their bug fixing process required 33 days, while the median number of days required to fix bug reports with seven activities in their bug fixing process was 78 days. Columns three and four in Table 11 illustrate the percentage of bug reports resolved above and below the 60-day cut-off time used in our experiments. For the remaining 1.5 percent of the bug reports we grouped them by length in ranges (with the range increment of 25) and averaged the median number of days.

We further analyze individual activities based on the time required to fix the bug. Table 12 shows frequencies of the activities for bugs resolved in less than or more than 60 days (median of the time to fix). The table highlights

TABLE 10
Test Results for Fixed Length Temporal Sequences

No. of activities	No. of hidden states	Precision	Recall	F-measure	Accuracy
2	5	90.39%	61.89%	73.47%	67.37%
	10	91.38%	61.35%	73.41%	66.91%
	15	91.17%	61.43%	73.40%	66.96%
3	5	87.58%	67.52%	76.25%	72.72%
	10	86.91%	67.72%	76.12%	72.74%
	15	86.84%	67.98%	76.26%	72.97%
4	5	76.92%	71.53%	74.12%	73.15%
	10	75.62%	71.74%	73.63%	72.92%
	15	77.12%	71.58%	74.25%	73.25%
5	5	69.24%	74.83%	71.92%	72.97%
	10	68.89%	75.15%	71.88%	73.06%
	15	69.15%	75.10%	72.00%	73.11%
6	5	61.52%	76.12%	68.04%	71.11%
	10	61.25%	76.34%	67.97%	71.13%
	15	61.03%	76.55%	67.91%	71.17%

relations between the number of activities and the time to fix a bug. For example, the table suggests that the bugs requiring long time to fix involve exchanging significantly higher number of whiteboard comments (symbol "W"). Confirming and putting a bug to general mailbox for volunteers to work on (symbol "R") also leads to long time to fix. In contrast, bugs confirmed and assigned to a particular developer (symbol "A") were resolved within 60 days, indicating that not assigning ownership to bugs could cause bug resolution time to exceed two months.

TABLE 11
Median Number of Days Required for a Fix versus Number of Activities in Temporal Sequence

Length of sequence	Total count	% of bug reports resolved below 60 days	% of bug reports resolved above 60 days	Median No. of days
2	2,857	68.39%	31.61%	7
3	12,369	69.68%	30.32%	6
4	14,196	42.70%	57.30%	127
5	9,678	33.17%	66.83%	244
6	5,579	38.66%	61.34%	164
7	3,591	47.15%	52.85%	78
8	2,543	57.65%	42.35%	33
9	2,241	65.02%	34.98%	20
10	1,796	67.71%	32.29%	17
11	1,491	69.62%	30.38%	20
12	1,194	67.84%	32.16%	22
13	943	64.16%	35.84%	29
14	820	61.10%	38.90%	36
15	606	60.56%	39.44%	40
16	498	56.43%	43.57%	47
17	441	56.69%	43.31%	45
18	339	53.10%	46.90%	54
19	290	53.45%	46.55%	57
20	218	43.58%	56.42%	79
21	207	41.06%	58.94%	81
22	160	43.13%	56.88%	76
23	132	40.91%	59.09%	82
24	138	39.86%	60.14%	78
25	116	33.62%	66.38%	100
26-51	802	24.02%	75.98%	183
52-77	80	17.32%	82.68%	281
78-103	16	3.33%	96.67%	557
104-129	4	0.00%	100.00%	267
130-155	3	0.00%	100.00%	770
156-181	1	0.00%	100.00%	2,460
182-207	1	0.00%	100.00%	1,042

In Table 13, we show top 20 most common activity sequences for bugs resolved in less than or more than 60 days (median of the time to fix). The table shows sequences containing three or more activities. The table suggests that the amount of repeated sequences containing an exchange of comments on the whiteboard (symbol "W") is greater for bug reports that take longer time to fix.

In HMM, the number of hidden states is user defined. This means that we need to determine the best number of hidden states when training HMM. We, therefore, trained two HMMs with the different number of hidden states (5, 10, and 15) to determine the best number of hidden states. In other words, we repeated the experiments for 5-states HMM, 10-states HMM, and 15-states HMM.

In Table 10, we show precision, recall, F-measure, and accuracy for our results, when HMMs are trained with different number of hidden states and tested with different number of activities. First, we note that the overall accuracy and other measures are not significantly affected by the number of hidden states. Second, we note that, as the number of activities (observations) increase, the precision decreases and the recall increase. This means that the model can easily predict the time to fix bugs (as slow or fast) with more activities. The recall improvement is due to the fact that the bugs requiring short-time (fast) are closed.

From this case experiment, we conclude that when our approach is applied to the first three to four initial activities happening on the bug report, it achieves good prediction results. This implies that implementing our model in a real life scenario would result in providing the quality teams with an early indication of an expected longer time report.

We further carried out the experiments by looking at the reports submitted by novice reporters versus reports submitted by moderate and experienced reporters (see Section 3.1). For the threshold, we calculated the median number of days required to resolve for each reporter segment. The median number of days to resolve for bugs reported by novice, first timers, was 74 days; while the median number of days till

TABLE 12
Activities and Their Frequencies

Activity/Observation	Symbol	Frequency of activities for bug reports resolved below 60 days		Frequency of activities for bug reports resolved above 60 days	
		Count	Percentage	Count	Percentage
Assignment to named developer	A	9,189	6.58%	4,410	3.06%
Assignment to volunteers	R	268	0.19%	721	0.50%
More than one person has been copied within one hour on the bug report	D	7,856	5.63%	5,698	3.95%
A certain person has been copied on the bug report	C	62,837	44.99%	85,266	59.17%
Developer requested code review	V	18,033	12.91%	9,747	6.76%
Response to code review	Y	12,687	9.08%	5,047	3.50%
Developer requested super review	S	157	0.11%	87	0.06%
Response to super code review	H	210	0.15%	82	0.06%
File exchanged between developers and reporters	F	1,000	0.72%	794	0.55%
Comment exchanged on whiteboard	W	13,699	9.81%	22,269	15.45%
Milestone has been set for solution deployment	L	8,827	6.32%	4,804	3.33%
Priority changed for bug report	P	2,817	2.02%	2,558	1.78%
Severity changed for bug report	Q	2,079	1.49%	2,609	1.81%

resolution for bugs initiated by moderate and experienced reporters was 34 days. Our results for novice reporters were in line with the overall results presented in this experiment, due to the fact that 63 percent of bug reports were initiated by novice reporters. For experienced reporters, the overall accuracy dropped by 3 percent due to the low threshold of 34 days used for this test.

We continue our experiments and comparisons using a threshold of two months, which is closest to the median number of days in the dataset.

4.3 Experiment 2: Using the First Week's Activities to Predict the Time to Fix Bugs

In this experiment, we simulated a scenario of extracting temporal sequences of activities at certain points in time of the first week of submission of bug reports. In particular, we extracted the temporal activities up to the first day (*day 0*) of submission, up to the third day (*day 3*), and up to the seventh day of the first week (*day 7*) from the bugs in the bug repository. This is illustrated in Fig. 7. Day 0 seems to be the most important day; for this dataset the triage efforts actually happen on day 0. In this case, the length of the temporal sequences varies for each bug because they depend on the progress on each bug report. We also had to exclude those bug reports which do not have any sequences on the specific day or which have been resolved up until that day.

For example, consider the bug report with identifier number 405,296 opened on the 24th of November 2007. Table 14 shows all the key activities and their respective timestamps for this bug report. From this table, we can observe that M is the only activity up to day zero. M, C and

TABLE 13
Top 20 Most Common Activity Sequences of Length Three or Greater

For bug reports resolved below 60 days			For bug reports resolved above 60 days		
Sequence of activities	No. of occurrences	Percentage	Sequence of activities	No. of occurrences	Percentage
NCZ	5,679	37.34%	NWCZ	2,712	15.95%
NCCZ	3,131	20.59%	NCCZ	2,520	14.82%
MCZ	1,465	9.63%	NCWCZ	2,145	12.62%
NCCCZ	918	6.04%	NCZ	2,105	12.38%
MCCZ	799	5.25%	NCCCZ	1,190	7.00%
ECZ	594	3.91%	NCCWCZ	739	4.35%
NDZ	449	2.95%	MCZ	629	3.70%
ECCZ	380	2.50%	MCCZ	585	3.44%
NDCZ	271	1.78%	MWCZ	569	3.35%
MCCCZ	241	1.58%	NCWZ	531	3.12%
NCCCCZ	233	1.53%	NCCCCZ	431	2.54%
NQCZ	159	1.05%	NWCCZ	398	2.34%
ECCCZ	137	0.90%	NWZ	388	2.28%
NCDZ	134	0.88%	MCWCZ	365	2.15%
MDZ	126	0.83%	NCCWZ	351	2.06%
NCWZ	110	0.72%	ECZ	318	1.87%
NQCCZ	107	0.70%	MCCCZ	301	1.77%
NCWCZ	98	0.64%	ECCZ	286	1.68%
NDCCZ	93	0.61%	NCWCCZ	219	1.29%
NCQCZ	86	0.57%	NCCCWCZ	217	1.28%

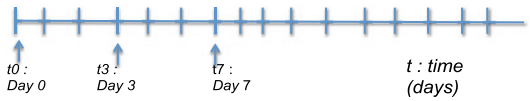

Fig. 7. Time points at which we extract test samples.

V are the activities up to day three. M, C, V, P, C, L, and L are the activities up to day seven.

We performed our experiment by dividing these bug reports with activities up to the first day into 60 percent training set and 40 percent testing set. We built HMM models for the first day of activities using this training set for different number of states (5, 10, and 15) on two types of bugs (bugs with slow time and fast time to fix). We then evaluated the HMM models on the testing set of the first day of activities. The length of activities of bug reports were variable in both training and testing set. Similarly, we trained HMM models on bug reports with activities up to third day and seventh day, and used those trained models to evaluate on bug reports with activities up to third day and seventh day, respectively.

In Table 15 we illustrate the results of this experiment with only the best number of states for HMM (i.e., 5 states) for the sake of brevity. We note that the overall F-measure for day 0 is 66.11 percent, and it only slightly improves for day 3 and day 7. All of the measures tend to follow the same pattern, except for precision, which increase dramatically as we move forward in time to reach a value of 81.07 percent on day 7. The improvement in precision rate can be attributed to the fact that some reports requiring a short-time are closed because the number of reports in the test sample drops by 659 reports which are closed between day 0 and day 7.

We conclude the viability of our approach from this experiment when applied as early as the date of submission. The initial activities happening during the first week of submission give a good indication of the overall expected time required to resolve the bug report.

The main purpose of experiments 1 and 2 was to illustrate the use of the activity sequences with HMM algorithm and their effect on the prediction accuracy. These experiments in fact show that the use of HMM on activity sequences is not affected by the change of Firefox's release process. The reason is that they only changed the priority of bug reports for resolution in the rapid releases [10]. These experiments actually are useful in asserting the use of HMM during the change of release process.

TABLE 14
Activities of the Bug Report # 405,296

Activity Symbol	Date-Time
M	2007-11-24 22:02
C	2007-11-25 12:05
V	2007-11-25 12:47
P	2007-11-27 01:53
C	2007-11-27 01:53
L	2007-11-27 01:53
L	2007-12-10 03:45
C	2008-01-10 11:27
C	2008-01-11 08:32
L	2008-01-30 18:19
F	2008-02-01 19:14
Y	2008-02-01 19:48
C	2008-02-03 22:47
A	2008-02-04 00:15
C	2008-02-04 00:15
Z	2008-02-06 21:20

TABLE 15
Test Results for First Week Temporal Sequences

No. of reports in test	day	precision	recall	F-measure	accuracy
10,551	0	76.17%	58.40%	66.11%	60.95%
10,535	3	80.80%	58.44%	67.83%	61.67%
9,892	7	81.07%	57.65%	67.38%	60.75%

4.4 Experiment 3: Using Bug Reports of Prior Years to Predict the Time to Fix for Bug Reports of Future Years

In a real life scenario, earlier data should be used for training and later data data should be used for testing. Accordingly, we carry out this experiment where we train the model on the previous years' data, and test the model on the next year's data. In this experiment, we train HMM using data from the previous year and test using the data from the current year, starting from 2006 and ending in 2014. Equation (11) illustrates the train-test set pairs of datasets that we use for training and testing. Assume that we are trying to predict the current year t, then the training would be done on the previous year data $t-1$. We actually performed eight experiments to cover the eight year period: first we train HMMs on the bugs of 2006 and tested them on 2007, second we trained on 2007 and then tested on 2008, and we continued in this manner until 2014. During testing, we followed the same approach as in Experiment 1 (Section 4.2) by using the first four activities of bug reports for prediction.

$$(training_{\,year\,t-1},\quad testing_{\,year\,t}) \tag{8}$$

Table 16 illustrates the details of training and testing datasets for all the eight experiments.

Table 17 illustrate the results of this experiment. Our model achieves an overall accuracy of 75.23 percent and F-measure of 77.11 percent with a precision of 83.43 percent and a recall of 75.44 percent for year 2007. In this

TABLE 16
Training and Testing by Year

Experiment	Training		Testing		
	Year	No. of reports	Year	No. of reports	No. of reports used for testing
1	2006	7,125	2007	6,526	2,527
2	2007	6,526	2008	10,557	3,075
3	2008	10,557	2009	7,538	2,166
4	2009	7,538	2010	7,854	2,721
5	2010	7,854	2011	5,834	2,255
6	2011	5,834	2012	4,659	2,288
7	2012	4,659	2013	6,974	3'494
8	2013	6,974	2014	6,283	2,761

TABLE 17
Results of Year on Year Prediction

Iteration	Year	Presion	Recall	F-measure	Accuracy
1	2007	71.97%	75.44%	73.66%	74.27%
2	2008	83.43%	71.68%	77.11%	75.23%
3	2009	66.41%	79.66%	72.44%	74.73%
4	2010	55.11%	72.75%	62.71%	67.23%
5	2011	55.84%	64.14%	59.70%	62.31%
6	2012	69.62%	66.37%	67.95%	67.17%
7	2013	70.01%	68.33%	69.16%	68.78%
8	2014	82.98%	66.65%	73.93%	70.73%

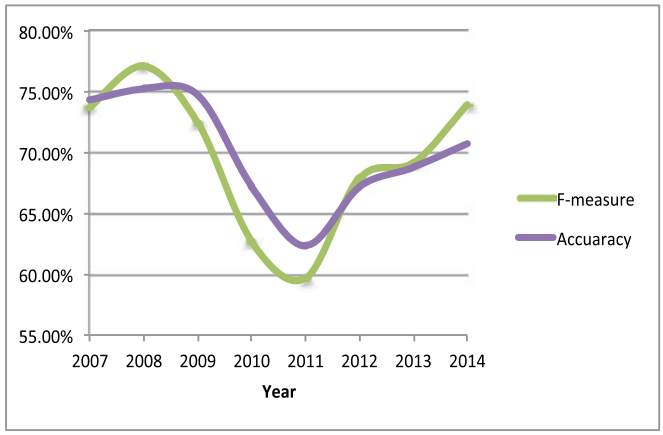

Fig. 8. Accuracy and F-measure year on year.

experiment, we experimented with the best hidden states (i.e., 5) HMMs. We also note that both the accuracy and F-measure for 2010 and 2011 suddenly drops. This drop is due to a change in the maintenance processes that occurred in Firefox during those years and resulted in the variations in developer activities for bug reports. To clarify the process change further, as of 2011 Mozilla decided to transition to a more agile (faster and less-conservative) approach to release management [37]. This new approach eliminated lengthy development cycle and prolonged beta testing phase. The purpose of this transition was to provide releases in a more rapid manner [38] and gain user confidence. Recently, da Costa et al. [10] have carried out a special case study on Firefoxs rapid release and concluded, surprisingly, that the median time needed to integrate a fix of a defect into Firefox source code is 50 days longer for rapid Firefox releases in comparison with the traditional Firefox releases. They explained this by noting that the traditional releases prioritize the integration of backlog issues, while rapid releases prioritize issues that were addressed during the current release cycle. Our proposed algorithm takes into consideration the temporality of this data set, and captures the process change very well. As shown in Fig. 8 and Table 17, once the process stabilised—the accuracy improved again.

As the datasets change frequently, software teams need to re-calibrate our models in order to use them effectively.

5 Comparison

Previous researchers have used a variety of classification algorithms to classify bugs into slow-fast resolution times. For example, Lamkanfi et al. [6] used Naive Bayes; Giger et al. [4]—decision trees; Panjer et al. [7]—Naive Bayes, decision trees, logistic regression, 0-R and 1-R; Hooimeijer et al. [1]—linear regression; and Zhang et al. [5]—k-Nearest Neighbour on the bug repositories data to classify bug reports into 'slow to fix' and 'fast to fix'.

However, there are several differences, as follows.

- The thresholds, nature, and scope of the datasets used in each study are different from ours (see Section 2 for details).
- The amount of post submission data, used in each study, varies.
- Previous studies have used frequencies of activities that occur in bug reports for classification rather than the temporal characteristics of activities. To clarify further, consider that we have the following temporal sequence of observations: *"NCCCWYDCVDY"*. In our approach, HMM would learn the temporal relationships between these activities by determining their conditional probabilities. Meanwhile, in other approaches the classification algorithms would use the frequencies of these activities for every bug report to train the model. In the case of the above example, the variety of classification algorithms, mentioned above, would use the following activities and their frequencies for training without considering their order: *"N(1), C(4), W(1), Y(2), D(1), V(1)"*.

In order to compare our model with the existing ones, we need to take into account all of the above. We select the work of Zhang et al. [5]. They experimented on three commercial proprietary software projects from CA technologies. They proposed a classification model to classify bugs as slow and fast using k-Nearest Neighbour (kNN) against various time thresholds (namely, 0.1, 0.2, 0.4, 1, and 2 time units, where one time unit represented the median of the distribution of times needed to fix a bug). The reasons we have chosen this work to compare against our work are as follows. First, they have reported very good results: an average weighted F-measure $= 72.45\%$. Second, the work is very recent and covers a commercial software scenario, which gives us an opportunity to test their algorithm on an open source data set and further compare the results of our algorithm versus their one. Third, the majority of features used in their work are available in the Firefox project.

In [5], they study three proprietary projects of the CA technologies, named as A, B, and C. The project are IT management products for enterprise customers. All the projects have been actively maintained for at least 5 years. The authors analyse the bugs reported over a 45 month interval for Project A, over a 30 month interval for Project B, and over a 29 month interval for Project C. Quality assurance and technical support teams report the bugs into a bug tracking system in response to customers complaints. The authors do not release the number of bug reports they use for confidentiality reasons but state that all these projects are large-scale projects and involve 40-100 developers, QA engineers, and support engineers. Also, the authors do not reveal the actual time required to fix a bug due to data sensitivity issues. Thus, they use a unit scale, where one unit is equal to the median of the distribution of times needed to fix a bug.

TABLE 18
Correspondence of Features Between the Products of CA Technologies and Firefox

Original study for CA projects			Corresponding features and values in Firefox		
Feature	Description	Values	Feature	Description	Values
Submitter	the bug report submitter		Reporter	The bug reporter	
Owner	The developer who is responsible for resolving the bug.		Developer	The person assigned to fix the bug	
Severity	The severity of a bug report	(Blocking, Functional, Enhancement, Cosmetic, or request for information)	Severity	The severity of a bug report	Blocker, critical, major, normal, minor, trivial, enhancement
Priority	the priority of a bug report	(Critical, Serious, Medium, Minor)	Priority	The priority of a bug report	P1,P2,P3,P4,P5 where P1 is the highest priority and P5 is the lowest
ESC	Indicating whether the bug is an externally discovered bug or an internally discovered bug		None	Not applicable	
Category	The category of the problem, such as, Account Management, Documentation, Configuration	A predefined lit of categories exists in the CA bug tracking system.	Component	Component field as a corresponding field	
Summary	A short description of the problem		Summary	Summary of the bug report	

In [5], they classify bugs as requiring either slow fixes or quick fixes; and they use a time threshold of 1 unit representing the median, similar to our work. The authors suggest kNN-based classification model for predicting the fix of a bug report as slow or as quick. They also made an assumption that the similar bugs could require similar bug-fixing effort.

They use summary, priority, severity, submitter, owner, category, and reporting source as features. The features are illustrated in column one in Table 18. Column two of this table shows the corresponding features and values, that we extracted from the Firefox project.

We gathered features that correspond to the features studied in [5], with the exception of the ESC feature and the Category feature. The ESC feature is not available in the Firefox bug tracking system. We could not find a suitable substitute for the ESC feature as Firefox does not track this information. Category feature is not available either; therefore, we chose Firefox's component feature as a suitable alternative. The component feature contains information about a particular architectural area of Firefox. For example "Translation" is a component that manages instant translation and language detection in Firefox, accordingly any bug report related to translation should have a component value of "Translation".

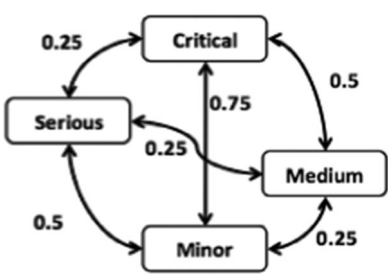

Fig. 9. Zhang et al. assumption for measuring distances between bug report priorities (acquired from [5]).

Another difference is the values of the severity and priority features between the CA products and Firefox. There are five values for severity in CA products; whereas, there are seven values in Firefox. Similarly, the priority feature in the CA project has four levels of priority; whereas, there are five levels of priority in Firefox. Zhang et al. [5] also define the distances between four different levels of priority as shown in Fig. 9. The levels, from highest to lowest are: Critical, Serious, Medium, and Minor. They assume that the distance between two adjacent priorities is equal to 0.25. They also suggest a similar approach to establish the distance between the levels of severity.

We adopt the same approach as Zhang et al. [5] by creating a matrix for the Firefox severity and priority fields as illustrated in Tables 19 and 20. The priority fields (ordered from the highest to the lowest priority) are: P1, P2, P3, P4, and P5. P1 indicates an important bug. P2 indicates a bug that has to be fixed (albeit not urgently). P3 suggests that it is a good idea to fix a bug in the future. P4 implies that the bug is not important. P5 means that the bug should not be fixed. For bugs that did not have priority assigned to them we used the value of P0. Given that we have six levels of priority, we chose to set the distance between adjacent levels to $1/6 \approx 0.17$. There are seven severity levels in the Firefox project, ordered from the most severe to the least severe:

TABLE 19
Priority Distance Matrix for Firefox

	P0	P1	P2	P3	P4	P5
P0	0	0.17	0.33	0.5	0.67	0.83
P1	0.17	0	0.17	0.33	0.5	0.67
P2	0.33	0.17	0	0.17	0.33	0.5
P3	0.5	0.33	0.17	0	0.17	0.33
P4	0.67	0.5	0.33	0.17	0	0.17
P5	0.83	0.67	0.5	0.33	0.17	0

Blocker, Critical, Major, Normal, Minor, Trivial, and Enhancement. Analogously to the priority field, we set the distance between adjacent severity levels to $1/7 \approx 0.14$.

Zhang et al.[5] used the 'bag of words' approach, introduced by Ko et al. [39] to measure the distance between bug summaries. They calculated the distance between two bug report summaries by considering how different the two sets of words are. Before doing the comparison, they eliminated "useless"[4] words from the bags of words as follows:

> *"To eliminate useless words like "or", "that", "in", or "contains", we first generate a set of standard words extracted from the CA pre-defined categories. When comparing the difference between two sets of words from bug's reports, we only count those contained in the standard word set." [5]*

This approach for elimination of the useless word did not work for our dataset, because we found that the intersection between the words of the Component and the Summary field was nil. Therefore, we had to resort to the following approach for producing bags of words from bug summaries. First, we convert all words to lower case. Second, we remove special character and numbers. Third, we remove stop words. Fourth we stem text. The approach we adopted has increased the number of words in the bag and thus effected the computing time required for calculation of the distance between the summaries of bugs, as described further below.

Zhang et al. [5] did not specify the number of nearest neighbours k that they used in their study. Therefore, we pick a range: $k = 2, 3, 4, \ldots, 50$. As in [5], we adopt Euclidean distance measure to conduct our experiment.

To perform our comparison, we choose the experiment in Section 4.4, where we use data from year 2013 as our training set and data from 2014 as our test set. In Table 21, we present the results of the 49 runs using different values of the number of neighbours k against the same evaluation criteria as in our study. We then select two kNN models that yield the best results ($k = 2$ and $k = 47$) and compare them with our HMM-based approach in Table 22.

Table 22 shows that our HMM-based approach outperforms the kNN-based approach in all measures: e.g., compare F-measure of 42.5 percent with F-measure of 73.93 percent. It is also worth mentioning that in order to run the kNN we faced computational challenges—the kNN required approximately 24 hours[5] on a machine with 8 Xeon CPUs (the computations were performed in parallel); whereas, training and testing of the five state HMM for the same data took 10 minutes on a single CPU.

This shows that the use of temporal activities provides better results than use of frequency based activities. Thus, our approach to transform activities in bug reports into temporal activities can facilitate managers in predicting time to fix bugs in a better manner than other techniques.

4. Here we use terminology of Zhang et al. [5].

5. This is due to the time required to calculate the distance between the summary fields in the Firefox project, as we could not create a set of standard words extracted from a predefined Component field similar to [5]. Therefore, we had to create it at the level of the bug report.

6 Discussion

6.1 Amount of Training Data Needed

Literature recommends allocating more than 50 percent of data for training and the remaining data for testing [40]. In most of our experiments we partition the dataset into two parts of 60 percent (training set) and 40 percent (test set), except for experiment number 3 (see Section 4.4), where the amount of training and testing data varied based on the number of bugs opened during the year.

In order to determine the optimal amount of training data needed for our approach, we selected Experiment number eight (illustrated in Table 16) from Section 4.4 and changed the percentage of training data in order to plot a learning curve. To determine the learning curve of our HMM-based approach, we fix the size of the test set to 20 percent (4,889) records, and vary the size of training set by drawing the data from the remaining 50,680 records. Specifically, we vary the size of the training set from 10 to 80 percent in the increments of 10 percent. The results are shown in Table 23.

It can be observed from Table 23 that precision, recall, F-measure, and accuracy fluctuate as the training set size changes. However, the variation is minor; for example, accuracy is 73.02 percent when the training set size is 10 percent and it is 73.50 percent when the training set size is 70 percent. Similarly, precision is 69.26 percent when the training set size is 10 percent and it is 70.63 percent when the training set size is 70 percent. We notice that the best accuracy (73.50 percent) and F-measure (72.81 percent) are achieved when we use the 70 percent of bug records for training. We also note that using 10 percent (6,335) of bug records still gives accuracy of 73.02 percent and F-measure of 71.97 percent. This shows that our proposed HMM-based approach can be trained effectively even with 10 percent of the data in this particular case study.

In order to examine further the effect of the amount of training data and test data, we varied both training dataset and test dataset during another experiment. We selected experiment number eight, illustrated in Table 16 from Section 4.4 and changed the percentages of both the training and testing datasets. We started by partitioning the data set into two parts of 10 percent (training set) and 90 percent (testing set) and run the experiment. We then repeated the experiments with increments of 10 percent for the training set and decrements of 10 percent for the testing set. We stopped when the number of bug records in the training set reached 80 percent and the bug records in the testing set reached 20 percent. The results of these eight experiments are summarised in Table 24. It can be seen that recall, accuracy, and F-measure remain above 70 percent irrespective

TABLE 20
Severity Distance Matrix for Firefox

	Blocker	Critical	Major	Normal	Minor	Trivial	Enhancement
Blocker	0	0.14	0.29	0.43	0.57	0.71	0.86
Critical	0.14	0	0.14	0.29	0.43	0.57	0.71
Major	0.29	0.14	0	0.14	0.29	0.43	0.57
Normal	0.43	0.29	0.14	0	0.14	0.29	0.43
Minor	0.57	0.43	0.29	0.14	0	0.14	0.29
Trivial	0.71	0.57	0.43	0.29	0.14	0	0.14
Enhancement	0.86	0.71	0.57	0.43	0.29	0.14	0

TABLE 21
Results of Zhang et al. [5] Approach on Firefox with Different Value of k for kNN Algorithm

k	Precision	Recall	F-measure	Accuracy	k	Precision	Recall	F-measure	Accuracy
2	35.59%	52.84%	42.53%	55.43%	27	40.57%	41.37%	40.96%	62.78%
3	36.60%	49.18%	41.97%	57.55%	28	40.56%	41.20%	40.88%	62.80%
4	36.77%	49.73%	42.28%	57.62%	29	40.84%	40.82%	40.83%	63.07%
5	37.26%	47.49%	41.76%	58.66%	30	40.54%	40.77%	40.65%	62.85%
6	37.44%	47.38%	41.82%	58.86%	31	41.53%	40.22%	40.87%	63.67%
7	38.05%	46.45%	41.83%	59.68%	32	41.69%	40.16%	40.91%	63.79%
8	37.57%	45.68%	41.23%	59.36%	33	42.16%	39.84%	40.97%	64.17%
9	37.72%	44.75%	40.94%	59.70%	34	42.06%	39.23%	40.60%	64.17%
10	38.24%	45.30%	41.47%	60.09%	35	42.30%	39.34%	40.77%	64.32%
11	37.99%	45.03%	41.21%	59.90%	36	42.37%	39.29%	40.77%	64.37%
12	38.51%	45.25%	41.61%	60.36%	37	42.66%	39.07%	40.79%	64.59%
13	38.49%	43.93%	41.03%	60.58%	38	42.41%	38.80%	40.53%	64.46%
14	38.84%	44.32%	41.40%	60.84%	39	42.39%	38.20%	40.18%	64.51%
15	39.31%	43.61%	41.35%	61.38%	40	42.46%	38.74%	40.51%	64.49%
16	40.15%	44.21%	42.08%	62.02%	41	42.29%	38.09%	40.08%	64.46%
17	40.27%	43.55%	41.85%	62.22%	42	42.20%	37.81%	39.88%	64.42%
18	40.32%	43.93%	42.05%	62.20%	43	42.44%	37.87%	40.02%	64.57%
19	40.39%	43.50%	41.88%	62.32%	44	42.83%	38.36%	40.47%	64.78%
20	40.34%	43.22%	41.73%	62.32%	45	42.98%	37.65%	40.14%	64.95%
21	40.24%	42.68%	41.42%	62.32%	46	43.11%	37.43%	40.07%	65.05%
22	40.66%	42.57%	41.59%	62.68%	47	43.18%	37.38%	40.07%	65.10%
23	40.37%	42.02%	41.18%	62.53%	48	42.74%	37.49%	39.94%	64.81%
24	39.99%	41.80%	40.88%	62.25%	49	42.34%	37.16%	39.58%	64.59%
25	40.16%	41.69%	40.91%	62.41%	50	42.41%	37.38%	39.73%	64.61%
26	39.96%	41.09%	40.52%	62.34%					

of the size of the training and test dataset (precision dips below 70 percent, with the minimum of 68.34 percent). This shows that our approach requires minimal amount of data to effectively classify the bug reports as fast and slow (from the perspective of fixing activities).

6.2 Severity Magnitude

As explained earlier in Section 3.1.3, severity is originally set by a bug reporter. However, during the triage and bug fixing process the triage and development teams can increase or decrease the severity, depending on the nature of the error. We track this change using the activity symbol Q. There are seven levels of severity used in the Firefox project: Blocker, Critical, Major, Normal, Minor, Trivial, and Enhancement. Thus, any change would indicate either an increase in severity (e.g. the change from Normal to Blocker) or a decrease in severity (e.g. the change is from Normal to Minor). In order to assess the effect of severity increase or severity decrease versus the general severity change, we repeat Experiment 3 (described in Section 4.4).

TABLE 22
Comparison of Results Between our HMM Based Approach and Zhang et al. [5] kNN Approach for the Firefox Project

	k	Precision	Recall	F-measure	Accuracy
kNN-based approach	2	35.59%	52.84%	42.53%	55.43%
	47	43.18%	37.38%	40.07%	65.10%
HMM-based approach		82.98%	66.65%	73.93%	70.73%

TABLE 23
Understanding the Learning Curve for Our Approach

Training%	Precision	Recall	F-measure	Accuracy	Training size
10%	69.26%	74.90%	71.97%	73.02%	6,335
20%	70.17%	74.50%	72.27%	73.08%	12,670
30%	69.17%	74.59%	71.78%	72.80%	19,005
40%	67.97%	75.11%	71.36%	72.72%	25,340
50%	70.46%	74.81%	72.57%	73.37%	31,675
60%	71.00%	74.54%	72.73%	73.37%	38,010
70%	70.63%	75.14%	72.81%	73.50%	44,345
80%	71.29%	73.90%	72.57%	73.06%	50,680

TABLE 24
Changing the Amount of Training and Testing Data Sets

Training%	Precision	Recall	F-measure	Accuracy	Training size	Testing size
10%	68.49%	73.62%	70.96%	71.97%	6,335	21,952
20%	72.26%	72.30%	72.28%	72.29%	12,670	19,408
30%	69.90%	73.16%	71.49%	72.13%	19,005	17,044
40%	70.74%	72.98%	71.84%	72.27%	25,340	14,662
50%	70.19%	73.10%	71.62%	72.18%	31,675	12,147
60%	76.92%	71.53%	74.12%	73.15%	38,010	7,627
70%	68.34%	74.51%	71.29%	72.48%	44,345	6,105
80%	71.29%	73.90%	72.57%	73.06%	50,680	4,889

In Section 4.4, we use the data from year 2013 as our training set and the data from 2014 as our test set. Accordingly, we extract all the severity changes that occurred during 2013 and 2014 and for any increase in severity we use the symbol X. For any decrease in severity we use the symbol J and repeat the experiment. Table 25 shows the results of our HMM-based approach in case of using temporal sequences reflective of severity changes at two levels, X and J, versus the severity change at one level Q.

Table 25 shows very similar results, indicating that the effect of the change of severity change is minimal. We conjecture that this is due to two reasons. First, there were only 7 percent of bugs that had their severity modified. Second, 25 percent of changes in severity occur on the date of bug submission, which are excluded from our experiments as described in Section 4.1.

7 Threats to Validity

In this section, we describe certain threats to the validity of our research process. We classify threats into four groups: conclusion validity, internal validity, construct validity, and external validity [41].

A threat to conclusion validity exists when activities are not present in a bug report and that bug report takes a long time to fix. To clarify this threat we are talking about those bug reports when the first activity is a reporter experience and the second activity is a closure of the bug report. The reporter experience are denoted by N (New reporter), M (Intermediate reporter), or E (Experienced reporter). The closure is denoted by Z. When there are no activities recorded between the first activity and closure, then the bug report will not be predicted by our approach as requiring slow resolution time. These are the bugs that require longer time to fix but unfortunately they don't have activities recorded in the repository. Therefore, they are not predicted. This threat is mitigated by the fact that only 1.4 percent of bug reports has such a situation. Another threat for conclusion validity may come from overfitting HMM. However, we mitigated this threat by repeating all of the experiments six to ten times with randomly selected training and test sets. We also selected different number of states for HMM since we had to repeat the experiments for every state that we selected. HMM usually suffers from overfitting when dataset is small; however, in our case the dataset is large and therefore the data set size is not an issue for overfitting.

A threat to internal validity may exist in the implementation because an incorrect implementation can influence the output. For example, we wrote Perl scripts to retrieve the data, Python scripts to build the activity symbols, and SQL scripts to extract the temporal sequences of activities. In our investigation, this threat is mitigated by manually investigating the outputs.

A threat to construct validity exists because we depend on the bug repository data to retrieve and build a full picture of the bug life cycle. However, the bug repository does not capture all of the software engineering activities and communications. This is a common issue in bug tracking repositories. Another threat to construct validity relates to our heuristic assumption for reporter experience: we track and label the bug reporters based on the reporter id as it appears in the dataset. There might be cases where the same reporter has two reporter ids. We have noted some of these cases while looking at the e-mail addresses. The effect of this is minimal due to our overall approach of adding the number of bug reports reported to the unique reporter ids. We used a mitigation technique that manually inspected the list of e-mail addresses versus reporter id and noted an overall uniqueness between both.

One threat to external validity arises because the project that we chose is an Open Source project and the results might not be readily applicable to a commercial project. In commercial projects, the bug maintenance process is usually carried out by internal employees or outsourced to third parties under a service level agreement (SLA). In Open Source projects, the bug maintenance process is usually managed by the project core team, who rely heavily on the involvement of volunteers to carry out the maintenance activities. Despite minor differences, there are more similarities and the concept that we present can easily be mapped to any organizational process.

TABLE 25
Effect of Severity Change on the Results

Case	Precision	Recall	F-measure	Accuracy
When change in severity is captured at two levels as X and J	75.53%	69.48%	72.38%	71.18%
When change in severity captured at one level as Q	82.98%	66.65%	73.93%	70.73%

Another threat to external validity arises because our experiments only cover the bug reports from a single project, and so the same results may not apply to other Open Source projects. In this research our aim was to explore the effect of temporality in estimating the bug fixing time. As it is described in the paper by Wieringa and Daneva [42], software engineering studies suffer from the variability of the real world and the generalization problem cannot be solved completely. As they indicate, to build a theory we need to generalize to a theoretical population and have adequate knowledge of the architectural similarity relation that defines the theoretical population. In this study we do not aim to build a theory, rather we would like to have a deeper understating of temporality in a case study. However, the framework and concepts can be easily applied to other projects.

8 Conclusions & Future Work

In this research, we present a novel approach for identifying the time to fix bug reports. Our approach considers the temporal sequences of developer activities rather than frequency of developer activities used by previous approaches [1], [4], [7]. We provide a framework for the extraction of temporal sequences of developer activities from the bug repository that could be replicated as is or modified depending on the dataset and a particular context. We also train the HMM for classification of bug reports as: (a) bugs requiring a slow resolution time, and (b) bugs requiring a fast resolution time (see Section 3).

We compared our HMM-based approach against the state of the art approach. We noted that HMM achieved approximately 5 percent higher accuracy, 14 percent higher recall, 40 percent higher precision, and 33 percent higher F-measure. Therefore, HMM-based temporal classification of the time to fix bug report is better than the existing techniques. Our approach would enable the software quality teams to be able to have an early indication of anticipated troublesome bug reports and this will help them to prioritise their work.

In this case study we observed that temporal data is valuable in making defect prioritization and that the field needs further research and experiments to calibrate and create more accurate models. We have shared our dataset through the Mining Software Repositories (MSR) 2015 conference for other researchers in this field [32]. There are several ways to improve or extend our current research. First, our future plan includes conducting experiments on different Open Source projects and commercial datasets. Second, our research focused on using temporal sequences to classify the bug fix time, we believe it is worth using the same attributes for the purpose of predicting the expected outcome of the final resolution (such as invalid bug, wontfix bug, and duplicate bug).

Acknowledgments

This research is supported in part by NSERC DG no. 402003-2012.

References

[1] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *Proc. 22nd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2007, pp. 34–43.

[2] S. Kim and E. J. Whitehead Jr., "How long did it take to fix bugs?" in *Proc. Int. Workshop Mining Softw. Repositories.*, 2006, pp. 173–174.

[3] P. Bhattacharya and I. Neamtiu, "Bug-fix time prediction models: Can we do better?" in *Proc. 8th Working Conf. Mining Softw. Repositories.*, 2011, pp. 207–210.

[4] E. Giger, M. Pinzger, and H. Gall, "Predicting the fix time of bugs," in *Proc. 2nd Int. Workshop Recommendation Syst. Softw. Eng.*, 2010, pp. 52–56.

[5] H. Zhang, L. Gong, and S. Versteeg, "Predicting bug-fixing time: An empirical study of commercial software projects," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 1042–1051.

[6] A. Lamkanfi and S. Demeyer, "Filtering bug reports for fix-time analysis," in *Proc. 16th Eur. Conf. Softw. Maintenance Reengineering.*, 2012, pp. 379–384.

[7] L. D. Panjer, "Predicting eclipse bug lifetimes," in *Proc. 4th Int. Workshop Mining Softw. Repositories.*, 2007, Art. no. 29.

[8] P. Anbalagan and M. Vouk, "On predicting the time taken to correct bug reports in open source projects," in *Proc. IEEE Int. Conf. Softw. Maintenance.*, 2009, pp. 523–526.

[9] L. Marks, Y. Zou, and A. E. Hassan, "Studying the fix-time for bugs in large open source projects," in *Proc. 7th Int. Conf. Predictive Models Softw. Eng.*, 2011, Art. no. 11.

[10] D. A. da Costa, S. McIntosh, U. Kulesza, and A. E. Hassan, "The impact of switching to a rapid release cycle on the integration delay of addressed issues: An empirical study of the Mozilla Firefox project," in *Proc. 13th Int. Workshop Mining Softw. Repositories*, 2016, pp. 374–385.

[11] J. Anvik and G. C. Murphy, "Reducing the effort of bug report triage: Recommenders for development-oriented decisions," *ACM Trans. Softw. Eng. Methodology*, vol. 20, no. 3, pp. 10:1–10:35, 2011.

[12] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proc. 28th Int. Conf. Softw. Eng.*, 2006, pp. 361–370.

[13] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *Proc. 7th Joint Meetinge Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2009, pp. 111–120.

[14] D. Cubranc, "Automatic bug triage using text categorization," in *Proc. 16th Int. Conf. Softw. Eng. Knowl. Eng.*, 2004, pp. 1–6.

[15] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 461–470.

[16] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: A text-based approach to classify change requests," in *Proc. Conf. Center Adv. Studies Collaborative Res.: Meeting Minds*, 2008, pp. 1–15.

[17] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," in *Proc. IEEE Int. Conf. Dependable Syst. Netw. FTCS and DCC*, 2008, pp. 52–61.

[18] Mozilla, "Life cycle of a bug: Bugzilla online documentation." (2016). [Online]. Available: https://www.bugzilla.org/docs/2.18/html/lifecycle.html Accessed on: 27-Apr.-2016.

[19] L. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," *Proc. IEEE*, vol. 77, no. 2, pp. 257–286, Feb. 1989.

[20] Mozilla, "Mozilla." (2016). [Online]. Available: http://blog.mozilla.org/press/ataglance/ Accessed on: 27-Apr.-2016.

[21] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?" in *Proc. 4th Int. Workshop Mining Softw. Repositories*, 2007, Art. no. 1.

[22] Atlassian, "Jira issue tracking software." (2016). [Online]. Available: https://www.atlassian.com/software/jira

[23] S. Mani, S. Nagar, D. Mukherjee, R. Narayanam, V. S. Sinha, and A. A. Nanavati, "Bug resolution catalysts: Identifying essential non-committers from bug repositories," in *Proc. 10th Working Conf. Mining Softw. Repositories*, 2013, pp. 193–202.

[24] Y. Tian, D. Lo, and C. Sun, "Drone: Predicting priority of reported bugs by multi-factor analysis," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2013, pp. 200–209.

[25] H. V. Garcia and E. Shihab, "Characterizing and predicting blocking bugs in open source projects," in *Proc. 11th Working Conf. Mining Softw. Repositories*, 2014, pp. 72–81.

[26] A. Lamkanfi, J. Pérez, and S. Demeyer, "The Eclipse and Mozilla defect tracking dataset: A genuine dataset for mining bug information," in *Proc. 10th Working Conf. Mining Softw. Repositories*, 2013, pp. 203–206.

[27] M. S. Zanetti, I. Scholtes, C. J. Tessone, and F. Schweitzer, "Categorizing bugs with social networks," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 1032–1041.

[28] S. Biçer, A. B. Bener, and B. Çağlayan, "Defect prediction using social network analysis on issue repositories," in *Proc. Int. Conf. Softw. Syst. Process.*, 2011, pp. 63–71.
[29] B. Caglayan, A. B. Bener, and A. Miranskyy, "Emergence of developer teams in the collaboration network," in *Proc. 6th Int. Workshop Cooperative Human Aspects Softw. Eng.*, 2013, pp. 33–40.
[30] A. Miranskyy, B. Caglayan, A. Bener, and E. Cialini, "Effect of temporal collaboration network, maintenance activity, and experience on defect exposure," in *Proc. 8th ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas.*, 2014, pp. 27:1–27:8.
[31] Mozilla, "Super-Review Policy." (2016). [Online]. Available: https://www.mozilla.org/hacking/reviewers.html Accessed on: 27-Apr.-2016.
[32] M. Habayeb, A. Miranskyy, S. Murtaza, L. Buchanan, and A. Bener, "The Firefox temporal defect dataset," in *Proc. 12th Working Conf. Mining Softw. Repositories*, 2015, pp. 498–501.
[33] E. Fosler-Lussier, "Markov models and hidden Markov models: A brief tutorial," *Int. Comput. Sci. Ins.*, Berkeley, California, Tech. Rep. (TR-98-041), pp. 132–141, Dec. 1998.
[34] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*. San Mateo, CA, USA: Morgan Kaufmann, 2005.
[35] S. S. Murtaza, M. Gittens, Z. Li, and N. H. Madhavji, "F007: Finding rediscovered faults from the field using function-level failed traces of software in the field," in *Proc. Conf. Center Adv. Studies Collaborative Res.*, 2010, pp. 57–71.
[36] S. S. Murtaza, N. H. Madhavji, M. Gittens, and A. Hamou-Lhadj, "Identifying recurring faulty functions in field traces of a large industrial software system," *IEEE Trans. Rel.*, vol. 64, no. 1, pp. 269–283, Mar. 2015.
[37] R. Paul, "Ars Technica: Mozilla outlines 16-week Firefox development cycle," 2011. [Online]. Available: http://arstechnica.com/information-technology/2011/03/mozilla-outlines-16-week-firefox-development-cycle/ Accessed on: 27-Apr.-2016.
[38] A. Avram, "InfoQ: Firefox: Mozilla wants a new development process, firefox 4 and the Roadmap," 2011. [Online]. Available: http://www.infoq.com/news/2011/03/Firefox-4 Accessed on: 27-Apr.-2016.
[39] Y. Ko, "A study of term weighting schemes using class information for text classification," in *Proc. 35th Int. ACM SIGIR Conf. Res. Develop. Inf. Retrieval*, 2012, pp. 1029–1030.
[40] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *SIGKDD Explor. Newslett.*, vol. 11, no. 1, pp. 10–18, 2009. [Online]. Available: http://dx.doi.org/10.1145/1656274.1656278
[41] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Berlin, Germany: Springer Science & Business Media, 2012.
[42] R. Wieringa and M. Daneva, "Six strategies for generalizing software engineering theories," *Sci. Comput. Program.*, vol. 101, pp. 136–152, 2015.

Mayy Habayeb received the MASc degree in industrial engineering from Ryerson University, Canada, in 2015. She is currently a professor in the School of Engineering Technology and Applied Science, Centennial College. She specializes in the areas of systems analysis and design, software quality, big data, and artificial Intelligence.

Syed Shariyar Murtaza received the PhD degree from the University of Western Ontario, Canada, in 2011. He is currently associated with software security industry in Canada as a data scientist and with Ryerson University as an instructor and a collaborating researcher. He specializes in the applications of statistics and machine learning in software engineering, software security, and healthcare.

Andriy Miranskyy received the PhD degree in applied mathematics from the University of Western Ontario. He is an assistant professor with Ryerson University's Department of Computer Science. His research interests include mitigating risk in software engineering, focusing on software quality assurance, program comprehension, software requirements, big-data systems, and green IT. He has 17 years of software engineering experience in information management and pharmaceutical industries. Prior to joining Ryerson, he worked as a software developer in the IBM Information Management division, the IBM Toronto Software Laboratory. He is a member of the IEEE.

Ayse Basar Bener is a professor and the director of Data Science Laboratory (DSL), the Department of Mechanical and Industrial Engineering, Ryerson University. She is the director of Big Data in the Office of Provost and Vice President Academic with Ryerson University. She is a faculty research fellow of IBM Toronto Labs Centre for Advance Studies, and affiliate research scientist in St. Michael's Hospital in Toronto. Her current research focus is big data applications to tackle the problem of decision-making under uncertainty by using machine learning methods and graph theory to analyze complex structures in big data to build recommender systems and predictive models. She is a member of AAAI, INFORMS, AIS, and senior member of the IEEE.