ON THE USE OF HIDDEN MARKOV MODEL TO PREDICT THE TIME TO FIX BUGS

by
Mayy Habayeb
Bachelor of Electrical Engineering
Jordan University of Science and Technology, 1989

A thesis

presented to Ryerson University
in partial fulfillment of the
requirements for the degree of Master of Industrial Engineering
in the Program of Mechanical and Industrial Engineering

Toronto, Ontario, Canada, 2015

© Mayy Habayeb 2015

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners. I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my thesis may be made electronically available to the public.

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 prioritize their work. Several studies of this problem have been conducted during 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 thesis, a novel approach using Hidden Markov Models and temporal sequences of developer activities is introduced. The approach is empirically demonstrated using eight years of bug reports collected from the Firefox project. The model correctly identifies bug reports with expected bug fix times. The approach is also compared against the frequency based classification approaches. The results indicate around 10% higher accuracy .

Acknowledgements

I would like to thank my supervisor Professor Ayse Bener for her continuous support, encouragement and valuable advice. Also, Assistant Professor Andriy Miranskyy, and Postdoctoral fellow Syed Shariyar Murtaza for their insightful guidance and amazing support during my period of study. I would like to thank my family especially my husband, Ammar Al-Taher, and my two children, Leen and Yahya, for supporting me and being patient with me throughout the past two years.

Contents

List of Tables

List of Figures

Chapter 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 prioritize their work and improve the development activities on bugs that would require more time [14],[17].

In 2011, Bhattacharya et al. [7] evaluated various univariate and multivariate regression models to predict the time 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%.

The time to fix bugs has been studied by several research groups throughout the past decade. Some studies focused on predicting bug fix time through classification, for example classifying bug fix time into requires 'fast'/'slow' [12],[31],[18] or classifying them into 'cheap' and 'expensive' [14] or employing multi-classification using an equal-frequency binning algorithm [24]. Other studies used regression techniques to predict bug fix time [2],[7]. Another group of studies focused on assisting the bug triage effort by introducing recommendations for developer/reviewer assignments [5],[4],[16] or automatically filtering out certain type of bugs [10],[26],[3],[15].

Most prior research has focused on the use of frequency of occurrence of activities in bug repositories to build their models for time to fix bugs. These attributes include the number of copied developers, comments exchanged, developers involved, etc. The frequency based models ignore the temporal characteristics of the activities in bug repositories. Let us consider this 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, another evidence of temporal activities is the bug life cycle in a bug management system. For example, Figure 1.1 shows the bug life cycle of the Bugzilla system [20]. 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 the reporter confirms it. Once confirmed, a bug is either assigned to a particular developer or made publicly available so that any developer may volunteer to fix it. Once 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 Figure 1.1 that a bug would pass through many stages before getting to the resolved state. However, by taking a closer look at the data it was noted that many bugs actually start as ’unconfirmed’ and move immediately into the state ’resolved’, without passing through assignment state. For example, in the case of Mozilla Firefox project in Bugzilla, when the status changes were tracked, it could be seen that 94.064% of bugs in unconfirmed state moved to resolved state, and 70.56% of bugs in 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 with Firefox developers, it was concluded that there are many other states 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.

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

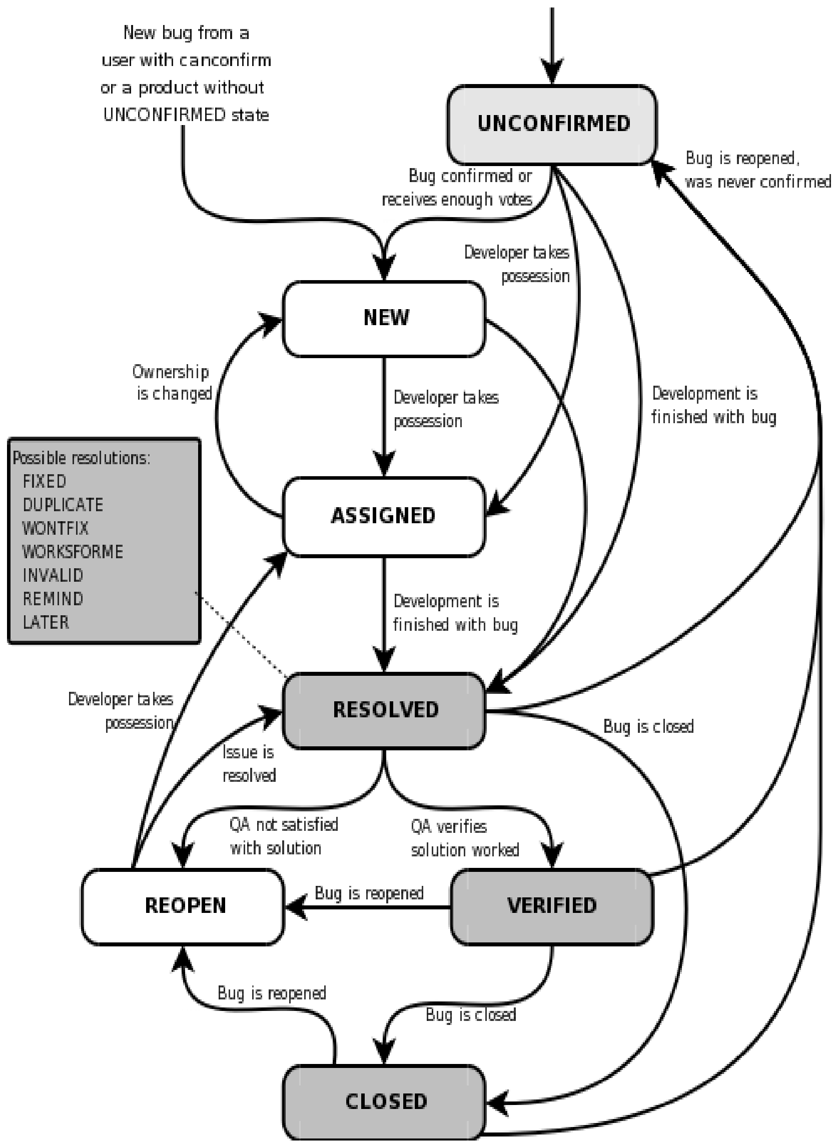

New bug from a user with canconfirm or a product without UNCONFIRMED state
UNCONFIRMED
Bug confirmed or receives enough votes
Bug is reopened, was never confirmed
Developer takes possession
NEW
Ownership is changed
Developer takes possession
Development is finished with bug
Possible resolutions: FIXED DUPLICATE WONTFIX WORKSFORME INVALID REMIND LATER
ASSIGNED
Development is finished with bug
Developer takes possession
RESOLVED
Bug is closed
Issue is resolved
QA not satisfied with solution
QA verifies solution worked
REOPEN
Bug is reopened
VERIFIED
Bug is reopened
Bug is closed
CLOSED

Figure 1.1: The life cycle of a Bugzilla bug

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. Therefore, an approach to predict the time to fix bugs is proposed based on the Hidden Markov Model (HMM) [25]. HMM can model doubly stochastic processes in a system; that is, a visible stochastic process and a hidden stochastic process that exist in system-bug repositories in the case of this study [11]. HMM was applied on historical bug reports by first transforming bug report activities into temporal activities, and then training HMM on those temporal activities. The new (unknown) bug reports were classified into two categories: (a) bugs requiring short time *(fast)* to fix; and (b) bugs requiring long time to fix *(slow)*.

The experiments were performed on Mozilla Firefox project, which is a free and open source web browser, developed for Windows, OS X, and Linux, with a mobile version for Android. It has high popularity and a wide spread user base of over 450 million users around the world [21]. The bug tracking system for Mozilla Firefox is Bugzilla, which also has a wide user base. These systems are also widely used in many other studies [12, 9] allowing for a comparison to be provided with this approach on a well known dataset. This HMM based approach was compared against the existing approaches that do not consider temporal characteristics of bug life cycle; and it showed that the predictions of this approach are approximately 10% more accurate.

1.1 Research Problem

The research problem is to develop a method to identify at an early stage the bug reports that would require a long time to fix. The 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, in order for them to assess the need to re-prioritize.

1.2 Contribution

There are three main contributions of this work:

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 on-line at: https://googledrive.com/host/0B5TJwohpS9LzcHc3NldtdjZQRFk.

3. An approach to predict the time to fix bugs using temporal activities by using the Hidden Markov Model.

The framework and concepts presented in this thesis can easily be applied to other industries. The framework is very flexible and can be applied by first , identifying the key processes related to the problem and then extracting the temporal activities related to these processes.

1.3 Thesis Outline

Chapter 2 contains related work and summaries of other research. Chapter 3 illustrates this approach and model and provides an introduction to the concepts of HMM theory used in this thesis. In Chapter 4, three experiment scenarios are presented to illustrate the possibilities of using temporal sequences in the predictive domain of time to fix a bug. In Chapter 5, comparisons are made utilizing the same attributes as temporal sequences versus frequency counts; in addition, this algorithm is compared with other algorithms. This is followed by highlighting threats to validity in Chapter 6. Conclusions and intended future work are summarized in Chapter 7.

Throughout the course of this thesis the terms "status" and "state" will be used interchangeably, also HMM will be used to denote Hidden Markov Models.

Chapter 2

Related Work

In 2006 Kim et al. [17] 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 efforts have been conducted to predict the estimated time needed to fix bugs. The efforts can be grouped into two categories. The first group focuses on predicting the overall estimated time required to fix a bug report. 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. 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 the machine learning algorithms to predict time to fix the bug report.

2.1 Bug fix time estimation and prediction

There are several studies in the literature related to bug fix time estimation. [2, 24, 14, 12, 18, 31].

In 2007 Panjer [24] 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 an 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% .

In 2007, Hooimeijer et al. [14] presented a descriptive model of the 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”.

The work in this thesis is similar to the above two studies in terms of setting thresholds for classification and targeting to identify bug reports that would require longer time (slow). It differs in terms of size and scope of the dataset used, approach, and algorithm.

In 2009, Anbalagan et al. [2] 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 number of participants involved in the bug fix. They developed a prediction model using ’linear regression’, and one of the key findings in the work 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 it took to correct it. The work in this thesis follows the same recommendations mentioned in Anbalagan et al. research by focusing on the human factor, but it differs by focusing on the levels of involvement and interactions between the participants rather than than just using the number of participants or unique participants. In particular, this work focuses on the temporal characteristics.

In 2010 Giger et al. [12] explored 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 2784 days. The experiments in this thesis are conducted on 86,444 bug reports of the Firefox project and use a fixed time threshold of two months. In addition, bug reports marked as enhancements are filtered out, and most importantly the temporal sequencing of events are used as attributes rather than counts.

In 2012 Lamkanfi et al. [18] 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 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. For their experiments they applied 'Naive Bayes' classifiers and used a median threshold of 2.7 days. In this thesis work the recommendation of focusing on the human factor is adopted. Yet this work differs in the threshold calculation, as bug reports closed on the same date of submission (day 0 bugs from) are filtered out which effects the threshold calculation. We also differ in temporal methodology of classification.

In 2013, Zhang et al. [31] 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 required for fix time. 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. The work presented in this thesis 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 similar fix time, yet it differs in terms of the overall approach and algorithm as the algorithm used is HMMs instead of 'K-Nearest Neighbour', also 'textual similarity' are not employed as they are computationally expensive.

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 [10, 5, 15, 3, 26, 27].

Cubranic et al. [10] and Anvik et al. [5] investigated the possibility of automating bug assignments through text categorization. In 2004 Cubranic et al. [10], attempted to cut out the role of the person responsible for initial triage, by automatically assigning bugs to developers based on text similarity. The team experimented on Eclipse project and used a Naive Bayes classifier and reported 30% correctly predicted assignments. Similarly, in 2006 Anvik et al. [4] used SVM to recommend the best developer. In 2011, Anvik et al. [5] further extended this previous work to cover five open source projects and experimented with six machine learning algorithms. They reported a 70%-98% precision once the recommender is implemented within the bug triage process and the triage team are presented with 3 recommendations. The work in this thesis differs from these studies in many aspects as the focus is on bug fix time prediction after the reports have been triaged.

In 2008, Wang et al. [26] 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% in comparison to 43%-72% using natural language alone. Similarly, Jalbert et al. [15] 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% of duplicate reports. The work in this thesis differs in many aspects. First the work does not focus on filtering out a certain type of report, rather on the required time-to-fix. Second the attributes used in this work as input to the classification algorithm are temporal sequences reflecting the level of interactions that happen at an early stage of the bug report submission versus text similarity attributes used in Jalbert et al. [15] work.

Weiss et al. [27] mined the bug database of the JBoss project, stored in the JIRA issue tracking system [6]. They used Lucene, a text similarity measuring engine, and kNN clustering to identify similarities of effort spent on bug fixing. They estimated bug fixing effort for a new bug report. They reported results with one hour error rate from the actual effort. In our study we do not carry out any similarity analysis but we focus on the temporal activities of the maintainers. Antoniol et al. [3] distinguished between requests for defect fixing and other requests based on the text of the issues posted in bug tracking systems.

The work presented in this thesis differs from the above researchers work in many ways. First, the proposed prediction model is based on non-textual attributes. Second, the use of the temporal sequences of the activities happening during the triage process as input to the classifier algorithm versus frequency counts which has not been accounted for in previous work. The goal of this work is to provide an early indication of troublesome bug reports rather than automatic filtering of certain types of bug reports.

The approach and algorithm presented in this thesis builds on the key findings of prior research in terms of the importance of the human factor in the bug report fixing process. However, a novel approach of building temporal sequences captured from the history logs reflective of key activities happening at an early stage of the bug report submission. Further, using these temporal sequences to learn HMM models as classifiers to predict the bug reports that are expected to take a long time to fix has been introduced in this research for the first time.

Chapter 3

The Approach

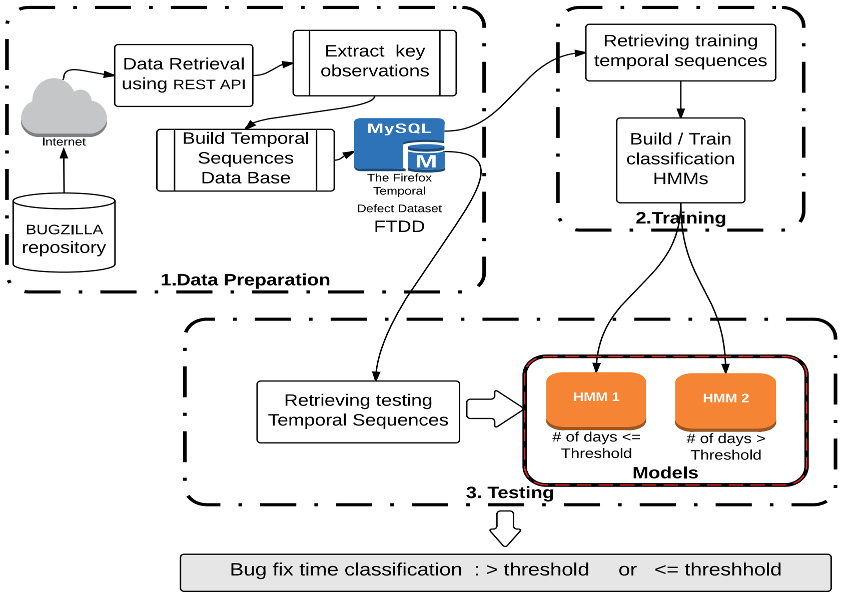

Data Retrieval using REST API
Extract key observations
Internet
Build Temporal Sequences Data Base
MySQL
The Firefox Temporal Defect Dataset
FTDD
BUGZILLA repository
1.Data Preparation
Retrieving training temporal sequences
Build / Train classification HMMs
2.Training
Retrieving testing Temporal Sequences
HMM 1
# of days <= Threshold
HMM 2
# of days > Threshold
Models
3. Testing
Bug fix time classification : > threshold or <= threshhold

Figure 3.1: The Approach

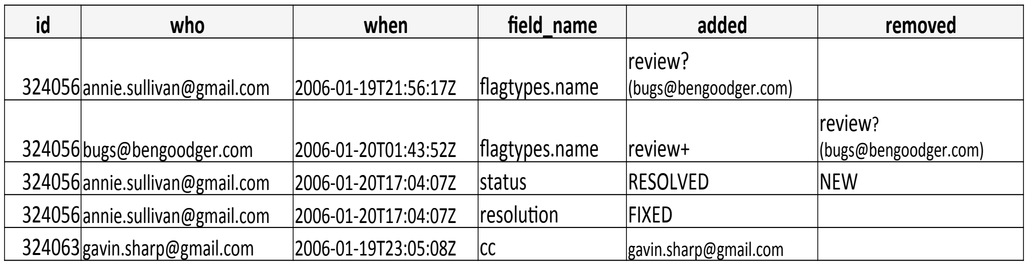
The approach adopted to address the research problem can be summarized in three steps as illustrated in Figure 3.1. First, temporal sequences of developer activities are extracted from a bug repository, such as Bugzilla repository[1]. This step is described in detail in Section 3.1. Second, the temporal sequences of activities of the resolved bugs are used to train Hidden Markov Models (HMMs). Two HMMs are trained by separating resolved bugs into two categories: (a) bugs with long time to fix (slow); and (b) bugs with short time to fix (fast). Two types of bugs are separated by measuring the median number of days taken by them for resolution. Bugs with the resolution time more than the median number of days are assigned to the long-time (slow) category and bugs with lesser resolution time than the median number of days are assigned to the short-time (fast) category: one HMM is then trained on each category. This step is further explained in Section 3.2. Third, the first few temporal sequences of activities for the latest bugs are extracted and pass them to the trained HMMs. The trained HMMs classify each bug into two categories: slow and fast time to fix. The intuition is to identify the bugs that are going to take longer time to fix in early stages of the life cycle of the bug, and help managers in allocating resources to bugs. This step is explained in detail in Section 3.3.

3.1 Extracting Temporal Sequences of Developer Activities

The approach that was used to extract developer activities from a bug repository depends on the software quality process setup within an organization 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. For this study a period of eight years (2006 - 2014) of Firefox project in the Bugzilla repository is covered. Firefox project was chosen because of its widespread customer base, diversity of end users, and popularity among the research community [26, 14, 19, 7, 16].

[1]https://www.bugzilla.org/

New feature requests were excluded because they are not bugs and they tend to follow a different development cycle. A sample history log is shown in Figure 3.2. In this history log, the temporal activities that take place in the bug repository while developers are fixing bugs can be observed. Figure 3.2 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).



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	

Figure 3.2: Sample history log

The criterion for selecting the temporal activities (as shown in Figure 3.2) from the bug history logs is based on the level of involvement by developers and bug reporters. All the activities that occur in a Bugzilla Firefox repositiroy are listed in Table 3.1. These activities were divided into three categories: involvement, communications, and bug condition. These are further explained below.

Involvement

Involvement activities reflect the types of people involved in the life cycle of the bug report. According to this rationale, focus was placed 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. [14] called this variable "Reporter Reputation", while Zanetti et al. [30] confirmed this finding. The reporter's experience was captured by indicating their status as novice (N), intermediate (M) or experienced (E) at the time of report creation (for details, see the description of N, M, E in Table 3.1).

Table 3.1: 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.

Developers: Giger et al. [12] and Jeong et al. [16] used the developer assigned to a bug as a key feature in their predictive models. 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 can volunteer to fix it. These two cases are distinguished in this thesis using the symbols A and R, respectively (see Table 3.1 for details).

Expert involvement: Experts can be copied or referred to in order to approve development changes. These observations are tracked by mining and extracting information about notification activities and code reviews.

Carbon Copies (CC): Both Giger et al.[12] and Panjer et al.[24] used the CC count generated at various time intervals as a key feature in their prediction studies. Zanetti et al. [30] also employed CC activities to capture the social network interactions between developers and reporters. In this work any CC (i.e., notification activity) is denoted using the symbol C. If more than one C occurs in an hour, this case is labeled 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. [16] suggested an algorithm for recommending suitable reviewers and attempted to predict levels of code acceptance. In this work, information about code review requests is extracted to classify them as *normal* and *super* reviews denoted by V and S, respectively. Normal code reviews usually are assigned to the module owner, and super code reviews to a set of senior developers who oversee significant architectural refactoring and API changes (see [1] 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 (accept or reject). Responses to *normal* and *super* reviews are also captured, denoted by Y and H, respectively, which indicate that the waiting period is over. Super review requests and responses have been captured at this level of detail for the first time in this thesis.

Communications

Communication between developers indicates active efforts and progress toward resolving the bug. In this work, two communication observations are captured: 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[2], denoted by W.

[2]These have been used in previous studies but as frequencies at certain time snapshot [12, 24, 14].

Bug Condition

In this work, four key events throughout the bug's lifetime, which affect its overall status and possibly indicate a change in the expected sizing or resolution type, are captured:

1) Change in severity: Severity usually is set by the reporter and is changed infrequently by the triage or development team. Severity was changed for only 7% of bugs in the Firefox Bugzilla repository. Increased or decreased severity influences the overall standing of the bug report. Changes in severity are denoted by Q.

2) Change in priority: The triage or development team sets the priority. Again, the priority was changed for only 7% of 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. A change in priority is denoted with the symbol P.

3) Milestone set : This indicates identifying a target release version for deployment of the bug fix code. This observation is denoted with the symbol L.

4) Resolution reached: This final activity in the sequence of observations is denoted by Z.

Finally all the key activities are stored in a database; it is called "The Firefox Temporal Defect Dataset (FTDD)" and it has been made publicly available [13]. Thus, a bug report is transformed into a set of temporal activities as shown in Figure 3.3. The order of the symbols for a bug represent the sequential order in which they occur in the bug repository and the meaning of the symbols can be determined from the symbols in the Table 3.1.

```
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
```

Figure 3.3: Sample of temporal sequences each row represents a bug report

3.2 Training

As mentioned earlier in chapter 1, HMM can model doubly stochastic processes in a system; that is, a visible stochastic process and a hidden stochastic process that exist in system-bug repositories. Hidden Markov Model (HMM) was introduced in the late 1960s and early 1970s [25]. HMM is a statistical model that is based on two stochastic processes. The first stochastic process is a Markov chain that is characterized 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. Figure 3.4 illustrates a simple (HMM). The components shown in Figure 3.4 represent an HMM, and HMM can be summarized as follows [25]:

- Number of States (α) : α represents the number of states, and the individual states are represented as $X = \{X_1 , \ldots ,X_\alpha \}$and the state at time t as q_t. In Figure 3.4, the number of states, α, is 3.
- Observations (β): β represents the number of observation symbols per state. Individual observations can be denoted by $O = \{O_1 , \ldots ,O_\beta \}$. The observation symbols correspond to the physical output of the system being modeled.
- State Transition Probability Distribution (G) : The transitions between the α states are organized by a set of probabilities called the state tran-

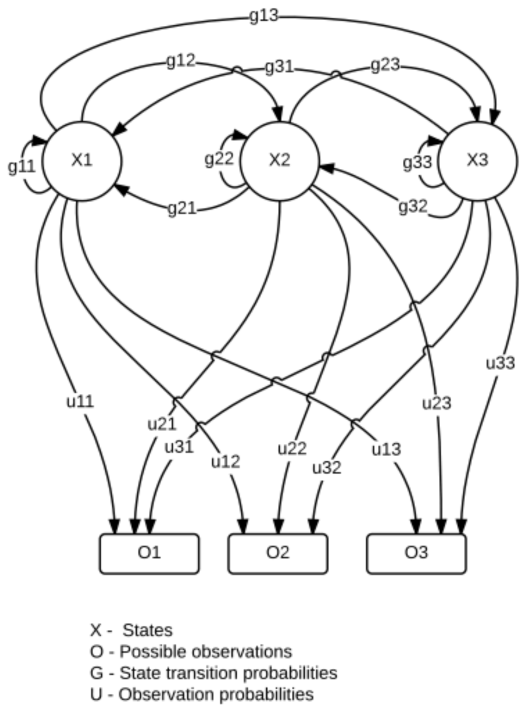

g13
g12
g31
g23
g11
X1
g22
X2
g33
X3
g21
g32
u11
u21
u31
u12
u22
u32
u13
u23
u33
O1
O2
O3
X - States
O - Possible observations
G - State transition probabilities
U - Observation probabilities

Figure 3.4: A Hidden Markov Model

sition probabilities. In this example g_{12}, g_{23}, g_{21} etc. denote the state 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

$$g_{ij} = P[q_{t+1} = j | q_t = i], \quad 1 \leq i, \quad j \leq \alpha, \tag{3.1}$$

where q_t denotes the current state.

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

$$u_j(k) = P[O_k \quad at\ t | q_t = X_j], \quad 1 \leq j \leq \alpha, \quad 1 \leq k \leq \beta, \tag{3.2}$$

where O_k denotes the kth observation symbol.

- Initial State Distribution (π): HMM needs to be initialized in the beginning; the initial state distribution at $t = 1$ is given as

$$\pi = \{\pi_i\}, \quad \pi = P[q_1 = X_i], \quad \text{and} \quad 1 \leq i \leq \alpha,$$

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. In our case, we use the discrete model.
An example of an HMM model in our case could be comprehended from Figure 3.5.

In Figure 3.5, the actual states are hidden and the only things which are 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.) 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 forth movement between processes, such as development and code review, deployment and assessment, etc. This task co-ordination is also visible in the bug repository through observations. Thus, there exist hidden states in the bug fixing processes, and the observations can occur on each hidden states, causing the transitions from one state to

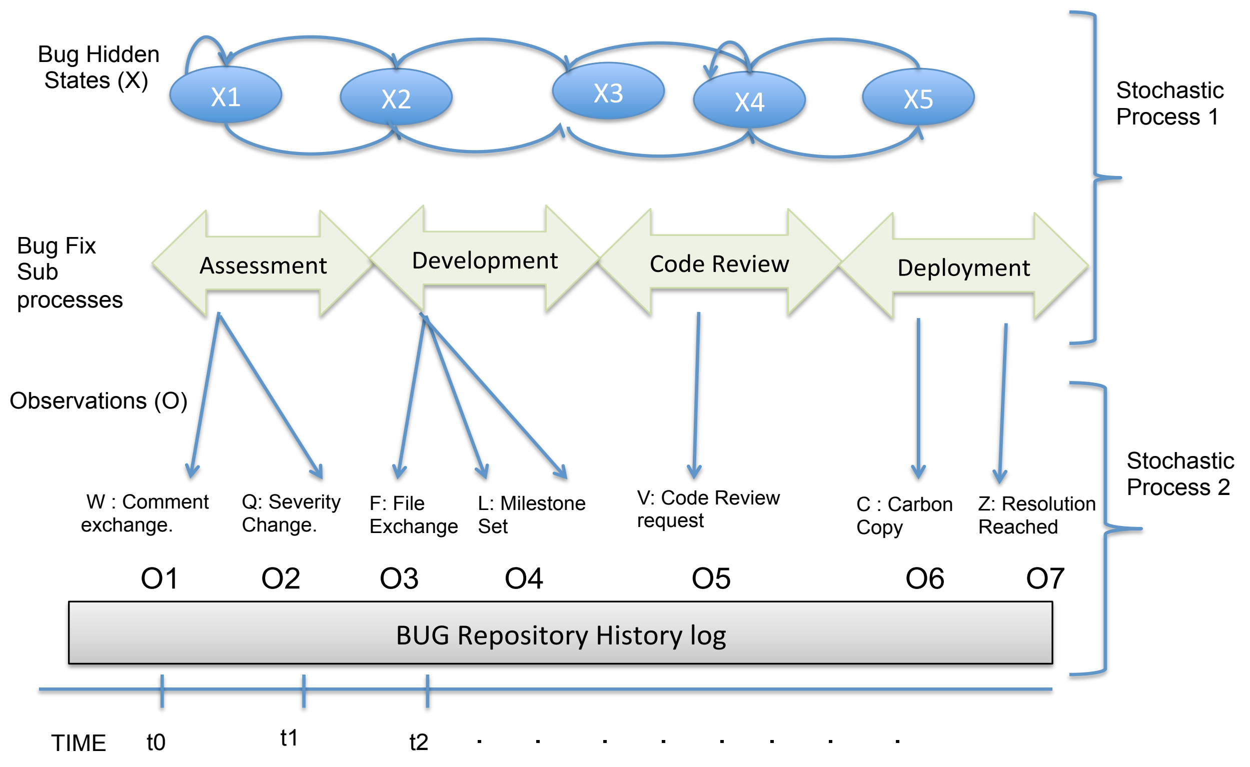


Figure 3.5: HMM model for bug fix time classification

another. Therefore, this process can be modeled using Hidden Markov Model.

To build an HMM model, $\lambda = (G, U, \pi)$, automatically from a sequence of observations (activities in this case) $O = \{O_1$,...,$O_\beta\}$, it needs to be trained on the sequences of observations [25]. In other words, the parameters G and U must be learned from the sequences of observations, such that P$[O|\lambda]$ gets maximized. Baum-Welch algorithm (BW) is one of the most commonly used algorithms for learning (or training) the parameters G and U [25]. It follows an iterative procedure using the forward and backward algorithms to determine the probabilities of parameters G and U from a sequence of observations. In this work, BW algorithm has also been used to train HMM.
The sequences of observations for bug reports are retrieved from the bug repository for training HMM. For example, a sequence of observations for a bug report could be like: New Reporter, Carbon copy, Whiteboard com-

ment exchanged, Carbon Copy, Whiteboard comment exchanged. According to the shorter representation used here it would be a sequence of symbols N,C,W,C,W. They are also shown in Table 3.1. Such sequences of observations for each bug report are extracted untill their final resolution. Two HMMs are actually trained 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 [12]. 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 the bug will take to resolve; i.e., above the median or below the median. This is further discussed in the next section.

In many applications, the supply of data for training and testing will be limited [8]. In order to build good models, using as much as possible of the available data for training is required to avoid over-fitting [8] [28] for such cases. As the data size used in this research is fairly large 63,000 bug reports, the problem of over-fitting does not apply. Thus the choice of %60 of the data for training and 40% for testing is reasonable, and will avoid any variance changes. This is applied to all the experiments carried out in this thesis.

3.3 Testing

This approach focuses on classifying the the new unresolved bug report into one of two types: the report will take a long time (slow) to fix; or the report will take a short time (fast) to fix. In order to achieve this observation sequences, $O = \{O1, O2, O3, ...On\}$, of the latest bug reports are retrieved, and passed onto two HMM models, $\lambda 1$ and $\lambda 2$, trained earlier on the resolved bugs. Then the probability that the observations are generated by one of the two models is determined; i.e., P$\{O|\lambda1\}$ and P$\{O|\lambda2\}$ are determined using the forward algorithm [25]. The maximum of P$\{O|\lambda1\ \}$ or P$\{O|\lambda2\ \}$ is determined and the new unresolved bug report is assigned the type of bug reports that the selected HMM represents. Only the first few observations are selected, 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) first five observations and pass them to HMMs to identify the time a bug report is going to consume – i.e., greater than the median number of days or less than the median number of days.

3.4 Evaluation criteria

In order to evaluate this approach, the bug repository dataset is divided into two parts: (a) bug reports for training; and (b) bug reports for testing. Approximately 60% of the resolved bug reports are used in the training set, and 40% of the resolved bug reports in the test set. The known resolution time of the bug reports in the test set allows for the performance of the HMMs to be evaluated.

The performance of the classifier is measured 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 [28]. 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). Finally, if it is classified as fast and it is actually in the fast class, then the classification is true negative (TN). Table 3.2 summarizes these four possible outcomes.

Using the values stored in the confusion matrix, the precision, recall, F-measure, and accuracy metrics are calculated for the bugs reports to evaluate the performance of these HMM models [28].

Table 3.2: Confusion Matrix

		Predicted	
		slow	fast
Actual	slow	TP	FP
	fast	FN	TN

Precision: Precision is the fraction of retrieved instances that are relevant, and it is calculated as [28]:

$$Precision = TP/(TP + FP). \tag{3.3}$$

Recall : Recall, also known as sensitivity, is the fraction of relevant instances that are retrieved, and it is calculated as [28]:

$$Recall = TP/(TP + FN). \tag{3.4}$$

Accuracy: Accuracy reflects the percentage of correctly classified bugs to the total number of bugs. It is computed as follows [28]:

$$Accuracy = (TP + TN)/(TP + TN + FN + FP). \tag{3.5}$$

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 [28]:

$$F - measure = 2 * Precision * Recall/Precision + Recall. \tag{3.6}$$

Chapter 4

Experiments

In this chapter, the application of the approach explained earlier in Chapter 3 is demonstrated in the form of three different experiments. In the first experiment, demonstration on the ability of the classification model to identify slow bug reports based on retrieving various fixed-length temporal sequences of activities for bugs from the test set is executed. In the second experiment, demonstration on the ability of the classification model to identify slow bug reports based on retrieving various temporal sequences within the first week of the bug report submission from the test set, is executed. In the third experiment, temporal sequences related to bugs of previous years are used for training and temporal sequences of bugs of future years are used for testing. The third experiment is the most likely one to be used in a real life scenario. At the begging some initial statistics for the dataset are provided, and then all the experiments and their respective results are explained .

4.1 Dataset: Firefox Bugzilla Project

As mentioned previously in Section 3.1, bugs opened within the eight year period (2006-2014), stored in the Firefox Bugzilla project were used. 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 were not considered in these experiments. The histories of all bug reports that had reached a status of "resolved" were extracted. In total, 86,444 of the 110,113 bug reports have been resolved. This analysis 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 4.1. The sixth column in Table 4.1 indicates the number of reports resolved per year excluding all reports closed on submission date, 26.7% of the total submitted bug reports get closed on date the of submission. In the seventh column in Table 4.1, the median of numbers of days required for resolution in each year are shown. The median for the number of days required to resolve decreases as time passes. This observation might give an initial impression that the bug process fix time is more efficient than previous years. However the percent of total resolved bugs, illustrated in the last column of Table 4.1, decreases over time. This observation means that many of the reported bugs are resolved in subsequent years, and when the remaining bug reports will be resolved the median will increase too. On further investigation of columns seven and eight from Table 4.1, it can be seen that caution is required in selecting the dataset range, as the last two or three years might not give the full picture of 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% of the bug reports, because the rest of the bug reports tend to remain open for future years.

Taking a closer look at the bug reports closed on date of submission, i.e., total resolution time is 0 days, this result is displayed In Table 4.2. It is noted that only 6.98% are closed with a resolution value of *fixed*, while the rest are marked with a resolution of *duplicate*, *invalid*, *worksforme*, *wontfix*, or *incomplete*. This result 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, 26, 22, 23] or automatically assign developers to bug reports [5, 10, 4]. This result is also a reflection of the nature

Table 4.1: 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%

of the reporters as most of these reports are reported by novice reporters see Section 3.1.

Table 4.2: Resolution of bugs closed on submission date

Resolution	No. of bugs	%
DUPLICATE	12,553	54.36%
FIXED	1,612	6.98%
INCOMPLETE	625	2.71%
INVALID	6,433	27.86%
WONTFIX	533	2.31%
WORKSFORME	1,338	5.79%
Total	23,094	

Table 4.3 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 the reports closed on the date of submission are excluded – i.e., *day 0* reports. The median of the number of days increases from 9 days to 53 days around 2 months, if the bug reports closed on the submission date are excluded. This result also makes the classification more viable for time

estimates. Therefore, focus is placed on the bug reports that have recorded actions one day after submission. This fact is also in line with Lamkanfi et al. [18] 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; i.e., *day 0*. Also 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. [14] for the classification of bugs as cheap and expensive for Firefox dataset. Accordingly, two month time interval, closest to the 53 days of median resolution, is a good threshold to separate bug reports into two categories for training the Hidden Markov Models.

Table 4.3: 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

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

This experiment begins with partitioning the dataset into two parts of 60% (training set) and 40% (test set). Two HMMs are trained on the training set by partitioning it further into bugs with resolution times less than the median number of days and greater than the median number of days. This approach is already explained in Section 3.1.

After training HMMs, the 40% test set is used 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 4.4 the median number of days required for two to six activities to occur for bug reports is illustrated. Table 4.4 also shows the median number of days required to actually resolve those bug reports. For example, the median number of days for the second activity to occur is two days, implying that 50% 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% of these defect reports required more than two months to resolve. Thus, there is room for the predictive model to predict the time to fix bug reports.

Table 4.4: Median of activities versus median of resolution per bug report

No. of activities	Median of No. of days for activity to occur	Median for No. of days till bug resolution
2	2	60
3	6	63
4	8	44
5	9	35
6	12	33

In HMM, the number of hidden states are user defined. This means that the best number of hidden states when training HMM must be determined. Therefore, two HMMs were trained with the different number of hidden states (5, 10, and 15) to determine the best number of hidden states.

In Table 4.5, precision, recall, F-measure, and accuracy for the results are shown, when HMMs are trained with different number of hidden states and tested with different number of activities. First, it can be seen that the overall accuracy and other measures are not effected significantly by the number of hidden states. Second, it can be seen note that as the number of activities (observations) increase, the precision decreases and the recall increases. 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) get closed.

From this case experiment, it can be concluded that when this 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 this model in a real life scenario would result in providing the quality teams with an early indication of an expected slow time report.

The experiments were further carried out by looking at the reports submitted by novice reporters as previously explained in Table 3.1 versus reports submitted by moderate and experienced reporters as previously explained in Table 3.1(see Section 3.1). For the threshold, the median number of days required to resolve for each reporter segment were calculated. The median number of days to resolve for bugs reported by novice, first timers, was 74 days; while the median number of days till resolution for bugs initiated by moderate and experienced reporters was 34 days. The results for novice reporters were in line with overall results presented in this experiment, due to the fact that 63% of bug reports were initialed by novice reporters. As for experienced reporters, the overall accuracy dropped by 3% due to the low threshold of 34 days used for this test.
The experiments and comparisons are continued using a threshold of two months, which is closest to the median number of days of the dataset.

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

In this experiment, several HMMs were trained again, as in Experiment 1 (Section 4.2), for different numbers of states (5, 10, and 15) on two types of bugs (bugs with slow time and fast time to fix). In the case of testing, a scenario of extracting temporal sequences of activities at certain points in time in the first week of submission of bug reports was simulated. In particular, the temporal activities on the first day (*day 0*) of submission, on the third day (*day 3*), and on the seventh day of the first week (*day 7*) were extracted from

Table 4.5: 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%

the bugs in the test set. This is illustrated in Figure 4.1. The day 0 seemed to be the most important day and for this experiment as the triage efforts actually happen on day 0. In this case, the length of the temporal sequences vary for each bugs, because they depend on the progress on each bug report. Also, those bug reports from the test set which did not have any sequences recorded on the specific day chosen for the experiment or which have been resolved up until that day had to be excluded.

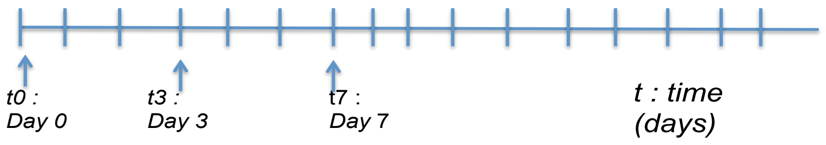


Figure 4.1: Time points at which we extract test samples

In Table 4.6 the results of this experiment are illustrated with only the best number of states for HMM (i.e., 5 states) for the sake of brevity. It can be seen that the overall F-measure for day 0 is 66.11%, and it only slightly improves on day 3 and day 7. All the measures tend to follow the same pattern, except for precision, which increase dramatically as time passes to reach a value of

81.07% on day 7. The improvement in precision rate can be attributed to the fact that some reports requiring 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.

Table 4.6: 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%

It can be concluded from the results of this experiment the viability of this approach when applied as early as date of submission. The initial activities happening during the first week of submission give a good indicator of the overall expected time required to resolve the bug report.
The results of *(day 3)*, 80.8% precision and 67.83% F-measure are better than random guessing and expert opinion.

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

Table 4.7 illustrates the details of training and testing datasets for all the eight experiments.

Table 4.7: Training and Testing by Year

	Training		Testing		
Experiment	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

In this experiment, the two HMMs are trained using data from the previous year and tested using the data from the next year, starting from 2006 and ending in 2014. Equation 4.1 illustrates the train-test set pairs of datasets that are used for training and testing. Assuming that the current year is being predicted at time t, then the training would be done on the previous year data time $t-1$. Eight experiments were actually performed to cover the eight year period: first HMMs are trained on the temporal sequences of bug reports submitted during year 2006 and tested using temporal sequences of bug reports from year 2007, second they are trained on the temporal sequences of the bug reports of 2007 and then tested on the temporal sequences of the bug reports 2008, and the experiments are continued in this manner until year 2014. During testing, the same approach as in Experiment 1 (Section 4.2) was followed by using the first four activities of bug reports for prediction, the reason for

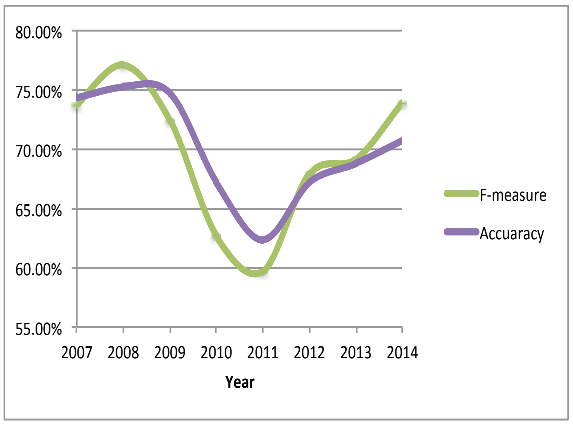


Figure 4.2: Accuracy and F-measure year on year

this is the best results were achieved using four activities (observations).

$$(training_{yaer\ t-1} \quad , \quad testing_{year\ t}) \tag{4.1}$$

Table 4.8 illustrate the results of this experiment. Our model achieves an overall accuracy of 75.23% and F-measure of 77.11% with a precision of 83.43% and a recall of 75.44% for year 2007. In this experiment, we experimented with the best hidden states (i.e., 5) HMMs.

Table 4.8: Results of year on year prediction

Iteration	Year	Precision	Re-call	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%

It can also be noted that both the accuracy and F-measure for 2010 and 2011 suddenly drops. The drops are due to a change in the maintenance processes that occurred in the Firefox organization during those years and resulted in the variations in developer activities for bug reports. However, once the processes stabilized, the accuracy improved again. This result is shown in Figure 4.2 and Table 4.8.

Examining Table 4.8 (excluding iterations 4,5,6 during which the processes changed) the results clearly indicate high levels of precision (70.01% - 83.43%), re-call (66.65% - 79.66%), F-measure (69.16% - 73.93%) and accuracy (68.78% - 74.73%) being achieved. From an implementation point of view, most likely the this scenario of (train,test) pairs would be implemented as most companies would assess previous years performance and consider it as base for planning for next years targets.

Chapter 5

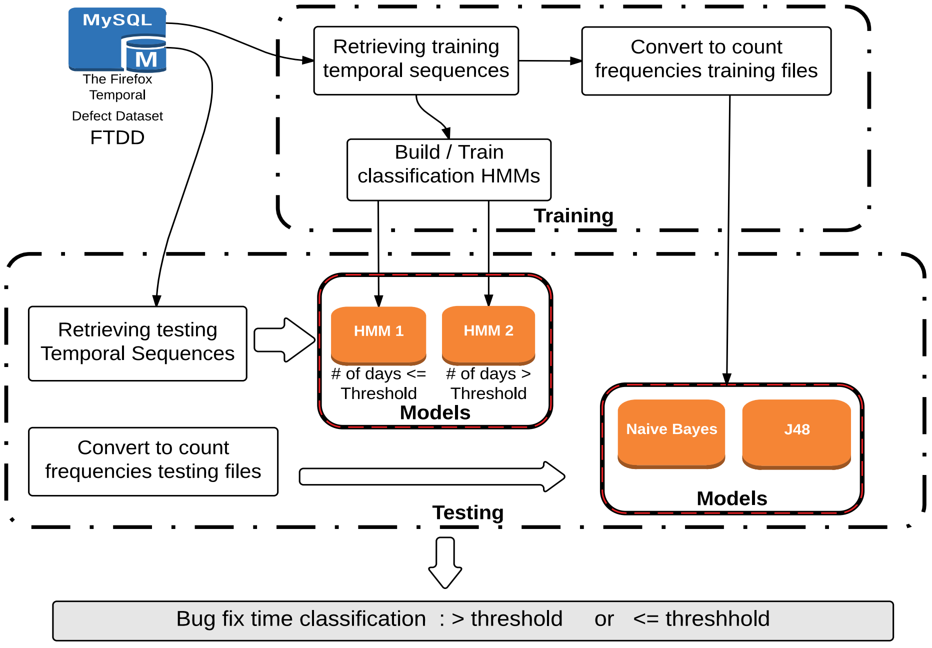
Comparisons and Discussion

In the past, researchers have used a variety of classification algorithms to classify bugs into slow-fast resolution times. For example, Lamkanfi et al. [18] used Naive Bayes, Giger et al. used decision trees [12], Panjer et al. [24] used Naive Bayes, decision trees, logistic regression, 0-R and 1-R, Hooimeijer et al. used linear regression [14] and Zhang et al. [31] used K nearest neighbour on the bug repositories to classify bug reports into “slow-to-fix” and “fast-to-fix”.

However, there are several differences between the approaches mentioned above and the approach presented in this thesis, as follows:

- The thresholds, nature, and scope of the datasets used in each study are different from ours (see Section 2).
- Each study has used a different set of attributes, such as date of bug submission, operating system, and text from the summary. These attributes are not used in our models.
- The amount of post submission data used also varies according to each study.
- The researchers have used frequencies of activities that occur in bug reports for classification rather than the temporal characteristics of activities. To clarify further, consider we have a temporal sequence of obser-

vations with the following value: *"NCCCWYDCVDY"*. In our approach, HMM would learn the temporal relationships between these activities by determining their conditional probabilities. 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 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)"*.



MySQL
The Firefox Temporal Defect Dataset
FTDD
Retrieving training temporal sequences
Convert to count frequencies training files
Build / Train classification HMMs
Training
Retrieving testing Temporal Sequences
HMM 1
# of days <= Threshold
HMM 2
# of days > Threshold
Models
Convert to count frequencies testing files
Naive Bayes
J48
Models
Testing
Bug fix time classification : > threshold or <= threshhold

Figure 5.1: Experimental design for comparison with other algorithms

It is not possible to directly compare the approach presented in this thesis with the ones in the literature due to the above reasons. However, for the sake of comparison, two of the most popular algorithms used by researchers are selected: Naive Bayes and C4.5 [28]. C4.5 and Naive Bayes are trained on the same dataset used in this study in a similar manner as HMM; i.e., training is done on 60% of the bug reports and testing is done on 40% of the bug reports. To compare this study's approach, two HMMs are trained with 5 hidden states one with temporal sequences related to bug reports that required more than 60 days and the other with temporal sequences related to bug reports that required equal or less than 60 days. All three algorithms, HMM, Naive Bayes and C4.5 (J48) are tested by using the activities that occurred on the first day (day 0) for the bug report in the test set. The intuition is to compare the accuracy on the first day of bug report submission. In particular, the intuition of the comparison is: (a) to compare temporal sequences of activities against the frequency counts of attributes; and (b) to compare HMM against the other frequency based algorithms on bug reports. The experimental design for comparison is illustrated in Figure 5.1.

The extracted bug reports for training are summarized in Table 5.1. As noted from the table, the training sample is large and balanced. Any resolution activity (observation) is filtered out; i.e., symbol Z is not included in our training samples.

Table 5.1: Bug Reports for Training

Period for resolution	Total No. of bug reports	No. of bug reports used for training	% of bug reports used for training to total
1 to 60 days	32,663	19,598	51.56%
greater than 60 days	30,687	18,412	48.44%
Total	63,350	38,010	

All of the remaining samples in the test dataset that have at least one activity are used for testing. Any reports that were closed on the day of submission are excluded. The experiment is repeated six times. Each time we use a different random sample for training. The number of bug reports used for training bug is maintained at 38,010 as indicated in Table 5.1. Table 5.2,

Table 5.2: No. of Bug Reports for Testing

Iteration	Fast (resolution <= 60 days)	Slow (resolution >60days)	Total
1	5377	5174	10,551
2	5,395	5,199	10,594
3	5,439	5,248	10,687
4	5,326	5,197	10,523
5	5,267	5,198	10,465
6	5,287	5,216	10,503

Table 5.3: Results

Iteration	Algorithm	Precision	Re-call	Accuracy	F-measure
1	HMM	76.169%	58.395%	60.951%	66.108%
	Naïve Bayes	56.60%	5.70%	51.62%	10.40%
	C 4.5	56.60%	5.70%	50.07%	10.40%
2	HMM	73.360%	59.733%	61.954%	65.849%
	Naïve Bayes	58.20%	6.20%	51.78%	11.20%
	C 4.5	42.50%	6.30%	49.82%	11.00%
3	HMM	76.467%	58.918%	61.574%	66.555%
	Naïve Bayes	53.50%	7.40%	51.37%	13.00%
	C 4.5	46.40%	8.00%	50.28%	13.60%
4	HMM	77.699%	58.357%	61.127%	66.653%
	Naïve Bayes	55.70%	7.00%	51.33%	12.50%
	C 4.5	44.30%	7.60%	49.64%	12.90%
5	HMM	64.275%	59.285%	60.066%	61.679%
	Naïve Bayes	53.10%	5.70%	50.71%	11.60%
	C 4.5	44.00%	8.00%	49.24%	13.00%
6	HMM	76.841%	58.500%	61.165%	66.428%
	Naïve Bayes	54.40%	6.20%	50.83%	11.10%
	C 4.5	44.70%	6.60%	49.56%	11.50%

illustrates our final test sample counts, used for each of the six experiments.

Table 5.3 summarizes the results of the comparison between Naive Bayes, C4.5 decision tree, and HMM. It can be seen that HMM outperforms both C4.5 and Naive Bayes. A Wilcoxon signed rank test to measure the significance of the difference in results between HMM and Naive Bayes, and HMM and C4.5) is further carried out. The choice of using the Wilcoxon signed rank test is because we are not certain that the data distribution is normal or not. The null hypothesis is that there is no significant difference in the results of HMM and Naive Bayes, and HMM and C4.5 decision tree. The significance level (alpha) was set to be 0.05. The test, in both the cases (HMM and Naive Bayes, and HMM and C4.5), revealed a Z-score of 2.201 and a two sided p value of 0.028. This indicates that the difference is statistically significant as p is less than 0.05 and accordingly null hypothesis is rejected.

This comparison experiment shows that the use of temporal activities provides better results than "use-of-frequency" based activities. Thus the approach to transform activities in bug reports into temporal activities can improve the prediction results. As a consequence the prediction outcome can facilitate managers in predicting time to fix bugs in a better manner than other techniques.

Chapter 6

Threats to Validity

In this chapter, certain threats to the validity of this research process are described. Threats are classified into four groups: conclusion validity, internal validity, construct validity, and external validity [29].

A threat to conclusion validity may arise due to random selection of the training and testing data. 60% of the data was randomly chosen for training the model and the remaining 40% was selected for testing the model. It is possible that with the selection of a different 60% the results might differ. However, this threat is mitigated by repeating the experiments through different experiments where in each experiment a different random sample is selected. In addition, 10-fold cross validation is used in the comparison experiment for Naive Bayes and C4.5. Also in Section 4.4, year to year training and testing is used, where the experiments are repeated eight times.
Another threat to conclusion validity may exist in terms of over fitting of HMM for a particular test set. However, this threat is mitigated by repeating the comparison experiments six times with different randomly selected training and testing sets. This threat is also mitigated by the selection of different number of states for HMM, which actually repeated the experiment for a different combination of states of HMM. Also the dataset used in this research is large and usually HMM suffers from over-fitting when dataset is small. Thus the chances of occurrence of over-fitting are negligible.

A threat to internal validity may exist in the implementation because an incorrect implementation can influence the output. For example, Perl scripts were written to retrieve the data, Python scripts were written to build the activity symbols and SQL scripts were written to extract the temporal sequences of activities. In this investigation, this threat is mitigated by manually investigating the outputs. Another threat exists when activities are not present in a bug report and that bug report takes a long time to fix. In such cases the bug report will not be predicted by this approach as requiring slow resolution time.

A threat to construct validity exists because the bug repository data is relied on to retrieve and build a full picture of the bug life cycle. However, the bug repository does not capture all software engineering activities and communications. This is a common issue in bug tracking repositories. Another threat to construct validity relates to the heuristic assumption for reporter experience in this study: the bug reporters are tracked and labeled based on the reporter ID as it appears in the dataset. There might be cases for which the same reporter has two reporter IDs. some of these cases were noted while observing the e-mail addresses of reporters. A mitigation technique that was used is manually inspecting the list of e-mail addresses versus reporter ID and noting an overall uniqueness between both.

A threat to external validity is that the project that was chosen 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 and they rely heavily on the involvement of volunteers to carry out the maintenance activities. Despite minor difference, there are more similarities and the concept that is presented here can be easily mapped to any organizational process. Another threat to external validity is that these experiments only cover the bug reports from a single project, the same results may not apply on other open source projects. However, the framework and concepts can easily be applied to other projects.

Chapter 7

Conclusions & Future Work

In this thesis, a novel approach for identifying the time to fix bug reports was presented. Our approach considers the temporal sequences of developer activities rather than frequency of developer activities used by previous approaches [14, 12, 24]. We provide a framework for extraction of temporal sequences of developer activities from the bug repository. We also train the Hidden Markov Model for classification of bug reports as: (a) bugs requiring slow resolution time, and (b) bugs requiring fast resolution time (see Section 3).

HMM was compared against Naive Bayes and C4.5 algorithms. HMM achieved around 10% higher accuracy and 55% higher F-measure. Therefore, HMM-based temporal classification of the time to fix bug report is better than the existing techniques. The software quality teams would be able to have an early indication of an anticipated troublesome bug reports by applying this approach and can prioritize their work.

These results are very promising and the field needs further research and experiments to calibrate and create more accurate models. This dataset has been shared through the Mining Software Repositories (MSR) 2015 conference for other researchers in this field to use [13]. There are several ways to improve or extend this current research: First, future plans include conducting the same experiments on different open source projects and commercial datasets.

Second, this research focused on using the temporal sequences for classifying the bug fix time, 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). Third, focus in this thesis was on the temporal sequences related to developers interactions and reporter experience. It might be worth investigating the effect of combing the other traditional attributes such as date of opening, size of summary filed, etc. and using the combined set as inputs to the classification algorithm. Fourth, incorporating human factors that affect the process into the algorithms. Such work would require understanding the factors and measuring these factors that affect decision making such as biases and then mathematically modeling them.

Bibliography

[1] Super-review policy. `https://www.mozilla.org/hacking/reviewers.html`. Accessed: 2015-02-14.

[2] P. Anbalagan and M. Vouk. On predicting the time taken to correct bug reports in open source projects. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 523–526. IEEE, 2009.

[3] 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 *Proceedings of the 2008 conference of the center for advanced studies on collaborative research: meeting of minds*, pages 1–15. ACM, 2008.

[4] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, pages 361–370. ACM, 2006.

[5] J. Anvik and G. C. Murphy. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(3):10:1–10:35, 2011.

[6] Atlassian. Jira issue tracking software. `https://www.atlassian.com/software/jira`.

[7] P. Bhattacharya and I. Neamtiu. Bug-fix time prediction models: can

we do better? In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 207–210. ACM, 2011.

[8] C. M. Bishop. *Pattern recognition and machine learning.* springer, 2006.

[9] G. Boetticher, T. Menzies, and T. Ostrand. Promise repository of empirical software engineering data. *West Virginia University, Department of Computer Science*, 2007.

[10] D. Cubranc. Automatic bug triage using text categorization. In SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering, 2004.

[11] P. Dymarski. Hidden markov models, theory and applications. *InTech Open Access Publishers*, 2011.

[12] E. Giger, M. Pinzger, and H. Gall. Predicting the fix time of bugs. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, pages 52–56. ACM, 2010.

[13] M. Habayeb, A. Miranskyy, S. Murtaza, L. Buchanan, and A. Bener. The firefox temporal defect dataset. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 498–501. IEEE, 2015.

[14] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 34–43. ACM, 2007.

[15] N. Jalbert and W. Weimer. Automated duplicate detection for bug tracking systems. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 52–61. IEEE, 2008.

[16] G. Jeong, S. Kim, and T. Zimmermann. Improving bug triage with bug tossing graphs. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 111–120. ACM, 2009.

[17] S. Kim and E. J. Whitehead Jr. How long did it take to fix bugs? In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 173–174. ACM, 2006.

[18] A. Lamkanfi and S. Demeyer. Filtering bug reports for fix-time analysis. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 379–384. IEEE, 2012.

[19] A. Lamkanfi, J. Pérez, and S. Demeyer. The eclipse and mozilla defect tracking dataset: a genuine dataset for mining bug information. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 203–206. IEEE Press, 2013.

[20] Mozilla. life cycle of a bug bugzilla online documentation. `http://www.bugzilla.org/docs/tip/html/lifecycle.html`. Accessed: 2014-05-18.

[21] Mozilla. Mozilla. `http://blog.mozilla.org/press/ataglance/`. Accessed: 2014-05-18.

[22] 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 *Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research*, pages 57–71. IBM Corp., 2010.

[23] 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. *Reliability, IEEE Transactions on*, 64(1):269–283, 2015.

[24] L. D. Panjer. Predicting eclipse bug lifetimes. In *Proceedings of the Fourth International Workshop on mining software repositories*, page 29. IEEE Computer Society, 2007.

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

[26] 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 *Proceedings of the 30th international conference on Software engineering*, pages 461–470. ACM, 2008.

[27] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 1. IEEE Computer Society, 2007.

[28] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[29] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[30] M. S. Zanetti, I. Scholtes, C. J. Tessone, and F. Schweitzer. Categorizing bugs with social networks. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1032–1041. IEEE, 2013.

[31] H. Zhang, L. Gong, and S. Versteeg. Predicting bug-fixing time: an empirical study of commercial software projects. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1042–1051. IEEE Press, 2013.