A Static Technique for Bug Localization Using Character N-Gram Based Information Retrieval Model

by

Sangeeta

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Master of Technology

in

Computer Science

in the

Graduate Division
of the
Indraprastha Institute of Information Technology, Delhi

Committee in charge:
Dr. Ashish Sureka, Chair
Dr. Anjaneyulu Pasala
Dr. Srikanta Bedathur

September 2011

The dissertation of Sangeeta, titled *A Static Technique for Bug Localization Using Character N-Gram Based Information Retrieval Model*, is approved:

Chair ____________________________________ Date ______________

____________________________________ Date ______________

____________________________________ Date ______________

____________________________________ Date ______________

Indraprastha Institute of Information Technology, Delhi

Acknowledgements

I would like to express my deepest gratitude to my advisor Dr. Ashish for his guidance and support. His extreme energy, creativity and excellent coding skills have always been a constant source of motivation for me. The perfection that he brings to each and every piece of work that he does always inspired me to do things right at first time. He is a great person and one of the best mentors, I always be thankful to him.

I would also like to thank Amit for devoting his time in discussing ideas with me and giving his invaluable feedback. I would like to thank all PhD scholars in IIITD for making our lab such a great place to work. Thanks to Lucky and Ramjot for teaching me lots of system related things. Special thanks to Kevan and Rex from apache project who helped me in pointing to correct links related to apache project.

I would like to dedicate this thesis to my amazingly loving and supportive parents who have always been with me, no matter where I am.

Abstract

Bug or Fault localization is a process of identifying the specific location(s) or region(s) of source code (at various granularity levels such as the directory path, file, method or statement) that is faulty and needs to be modified to repair the defect. Bug localization is a routine task in software maintenance (corrective maintenance). Due to the increasing size and complexity of current software applications, automated solutions for bug localization can significantly reduce human effort and software maintenance cost.

We presented a technique (which falls into the class of static techniques for bug localization) for bug localization using a character N-gram based Information Retrieval (IR) model. We framed the problem of bug localization as a relevant document(s) search task for a given query and investigated the application of character-level N-gram based textual features derived from bug reports and source-code file attributes. We implemented the proposed IR model and evaluated its performance on dataset downloaded from two popular open-source projects (JBOSS and Apache).

We conducted a series of experiments to validate our hypothesis and presented evidences to demonstrate that the proposed approach is effective. The accuracy of the proposed approach is measured in terms of the standard and commonly used SCORE and MAP (Mean Average Precision) metrics for the task of bug localization. Experimental results reveal that the median value for the SCORE metric for JBOSS and Apache dataset is 99.03% and 93.70% respectively. We observed that for 16.16% of the bug reports in the JBOSS dataset and for 10.67% of the bug reports in the Apache dataset, the average precision value (computed at all recall levels) is between 0.9 and 1.0.
Keywords- *Bug Localization, Mining Software repositories (MSR), Information Retrieval (IR), Automated Software engineering (ASE)*

Contents

List of Figures

Chapter 1

Introduction

"Let he who has a bug free software cast the first stone" -by Assaad Chalhoub.

This is another way of saying that no matter how much time and effort goes into software testing it is hard to build a bug free software [15]. Software organizations spend huge amount of resources in software testing, Bill Gates [1] (chairman Microsoft) once said (in 1995):

"We have as many testers as we have developers. And developers spend half their time testing. We're more of a testing organization than we're a software organization."

Software testing is one of the most resource consuming task in software development life-cycle [24]. This resource consumption increases with software size and complexity. Even though software organizations do rigorous software testing it is practically infeasible for them to do exhaustive testing of software (testing for each possible input). Hence every large software have a maintenance phase after software delivered to end user. Corrective maintenance which involves bug fixing (reported by end-users) is a part of overall software maintenance process. It is estimated that corrective maintenance can consume upto 21% of all software maintenance resources, this amount can be huge for large software systems [16].
Large software projects receive huge number of bugs every day, for example in year 2005 Mozilla project[2] received on an average 300 bug per-day, and total of 51,154 bugs during 2002-2006. In Eclipse[3] nearly 13,016 bugs were reported in span of one year (jun-2004-jun-2005), having average of 37 reports per day, with a maximum of 220 reports in a single day[17]. Fixing such huge number of bugs will take considerable human effort and time. Bug

[1] `http://www.microsoft.com/presspass/exec/billg/`
[2] `http://www.mozilla.org/projects/`
[3] `http://www.eclipse.org/eclipse/`

fixing is a complex task as it requires understanding of bug and source code. In large and complex software systems software aging, poor-documentation and developer mobility makes software projects hard to understand for software developers[2]. This may slow down software project progress and may increase overall software maintenance cost. In order to bring down the overall resource consumption of corrective software maintenance it is required to empower software developers with tools and techniques that can facilitate them in bug fixing.

Bug fixing task consists of various sub-task such as understanding the bug, locating the cause of bug and finally fixing it. In most of the bug fixing cases locating cause of bug (bug localization) consumes most of the developer's time [25]. *Bug localization* is an important task in bug fixing process and can be formally defined as:

"Bug localization is a process of identifying the specific location(s) or region(s) of source code (at various granularity levels such as the directory path, file, method or statement) that is faulty and needs to be modified to repair the defect"[4, 3, 2].

Automated bug localization techniques take information about software system (source code and bug reports) as an input and produces a ranked list of documents (classes, files, or methods) that might require modification to fix the bug [2]. Automated bug localization domain have attracted considerable attention from the software engineering research community and various techniques for bug localization have been proposed in the literature [4, 2, 13, 1]. These techniques can be categorized as: Dynamic and Static. Dynamic bug localization techniques uses execution information of software to locate the bug[18, 19]. These techniques compare passing execution traces with erroneous execution traces to identify faulty program location. Dynamic bug localization approaches may not be appropriate for real world software project because of unavailability of execution information with the reported bug. In contrast static bug localization techniques use software source code (or other documents related with software) information to locate the bug. For example Hovemeyer et al. proposed a static technique for bug localization in which they used to evaluate classes against set of predefined rules to predict the bug. Static bug localization techniques have following advantages over dynamic bug localization techniques [2]:

- Static techniques do not require execution information of software.
- Static techniques can be applied at any stage of software development.

Some recent static techniques of bug localization have attempted to apply concepts from traditional Information Retrieval (IR) models such as LSI, LDA for bug localization. IR models frame bug localization as a search problem: retrieve relevant documents in a document collection for a given query. In this study we will keep our focus on IR based static techniques for bug localization, we will discuss in detail about IR based bug localization model in background (chapter 2) and detailed literature of IR based bug localization techniques in related work (chapter 8).

The aim of the research study presented in this thesis is to investigate novel text mining based approaches to analyze bug databases and version archives to uncover interesting patterns and knowledge which can be used to support developers in bug localization. We proposed an IR based approach to compute the semantic and lexical similarity between title and description in bug reports with file-name and path-name of source-code files by performing a low-level *character N-gram* based analysis.

Character N-gram based analysis techniques are found to be useful in the domain of information retrieval, text clustering, text classification etc. But the performance of character-level representation of bug reports and source code for the task of bug localization is an open research question, as all the earlier IR based techniques proposed in the literature (see Related Work 8) for bug localization are word based. We hypothesized that the inherent advantages of character-level analysis (language independence, robust towards noisy data etc.) is suitable for the task of automated bug localization as some of the key linguistic features present in bug report attributes and source-code attributes can be better captured using character N-grams in contrast to word-level analysis. The study presented in this thesis attempts to advance the state-of-the-art on IR based bug localization techniques and in context to the related work makes the following unique contributions:

1. A novel method for automated bug localization (based on the information contained in bug title and description expressed in free-form text and source code file-name and path-name) using a character-level N-gram model for textual similarity matching. While the method presented in this paper is not the first approach on IR based bug localization, this study is the first to investigate the performance of character-level N-gram language models for the bug localization problem. This study is the first to investigate sub-word features (slices of N characters within a word) and a bag-of-characters document representation model (rather than the bag-of-words model in previous approaches) for the task of static analysis and IR based bug localization.

2. An empirical evaluation of the proposed approach on real-world and publicly available dataset from two popular open-source projects (JBOSS and Apache). We experimented with different configuration parameters (such as weights for title-filename, title-pathname, description-filename and description-pathname comparisons and length normalization) and presented results throwing light on the performance of the approach as well as insights on the predictive power of different variables (present in the proposed textual similarity function).

Chapter 2

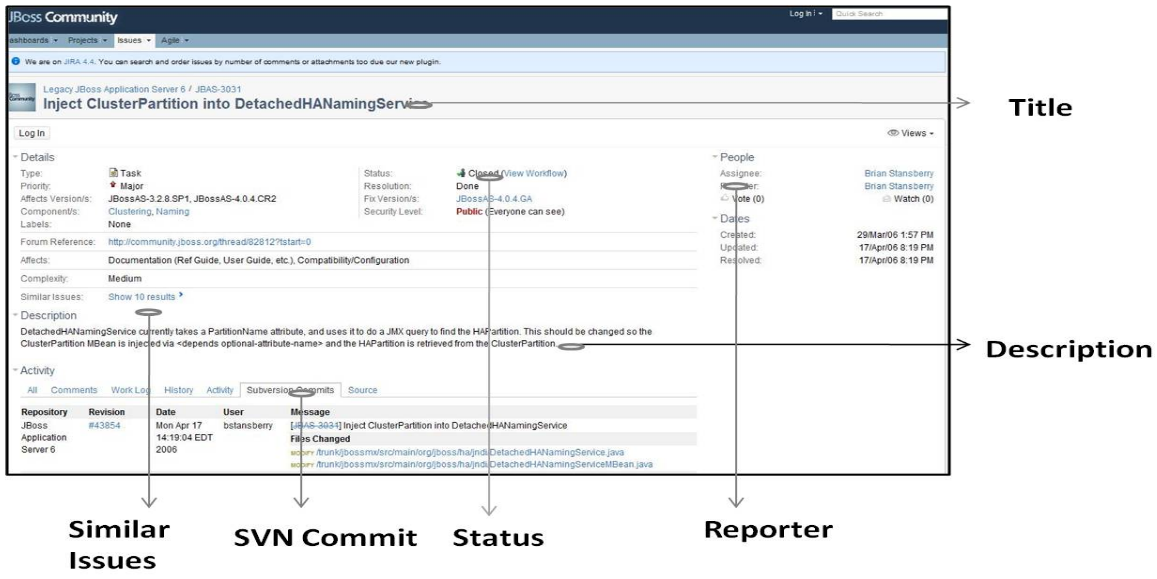
Background

This chapter provides background information about defect tracking system and IR models. We discuss bug report format (fields), life cycle of a bug, and structure of IR model for bug localization. In real world software projects software developers are expected to fix bug by understanding the problem from a given bug report. Hence it is necessary to understand bug report format (various fields present in the bug report) in detail in order to understand formation of automated bug localization tools using information present in bug report. Figure 2.1 displays a snapshot of a bug report taken from JBOSS issue tracking system. This consists of various different fields such as title, description, version, reporter, assignee, platform and many other fields which helps software developer in understanding the bug. Bug reporters can give snapshots, patches or stack traces related to the bug occurred which helps in further improving software developer's understanding about the bug. In addition to these traditional fields JIRA issue tracking system provides fields like similar issues and SVN commits. Similar issues fields can be useful in pruning the search space for current bug locations using locations of similar bugs locations. SVN commits provide the ground truth for bug report, which are helpful in identifying effectiveness of the automated bug localization system. This shows that bug reports consists of enormous amount of information. Automated bug localization techniques attempt to leverage this information for predicting bug locations. Next, we discuss bug life cycle and IR model for bug localization.

2.1 Bug Life Cycle

A bug report passes through various phases before it is declared as fixed (or closed). Figure 2.2 shows simplified picture of *bug life cycle*.[1] A bug can be found by tester or user who can report this bug to defect/issue tracking system. When a new bugs comes into the system

[1]`http://www.bugzilla.org/docs/2.18/html/lifecycle.html`



We are on JIRA 4.4. You can search and order issues by number of comments or attachments too due our new plugin.

Legacy JBoss Application Server 6 / JBAS-3031

Inject ClusterPartition into DetachedHANamingService

Details

Type:	Task
Priority:	Major
Affects Version/s:	JBossAS-3.2.8.SP1, JBossAS-4.0.4.CR2
Component/s:	Clustering, Naming
Labels:	None
Status:	Closed (View Workflow)
Resolution:	Done
Fix Version/s:	JBossAS-4.0.4.GA
Security Level:	Public (Everyone can see)
Forum Reference:	http://community.jboss.org/thread/82812?tstart=0
Affects:	Documentation (Ref Guide, User Guide, etc.), Compatibility/Configuration
Complexity:	Medium
Similar Issues:	Show 10 results

People

Assignee:	Brian Stansberry
Reporter:	Brian Stansberry
Vote (0)	Watch (0)

Dates

Created:	29/Mar/06 1:57 PM
Updated:	17/Apr/06 8:19 PM
Resolved:	17/Apr/06 8:19 PM

Description

DetachedHANamingService currently takes a PartitionName attribute, and uses it to do a JMX query to find the HAPartition. This should be changed so the ClusterPartition MBean is injected via <depends optional-attribute-name> and the HAPartition is retrieved from the ClusterPartition.

Activity

All Comments Work Log History Activity Subversion Commits Source

Repository	Revision	Date	User	Message
JBoss Application Server 6	#43854	Mon Apr 17 14:19:04 EDT 2006	bstansberry	[JBAS-3031] Inject ClusterPartition into DetachedHANamingService
				Files Changed
				MODIFY /trunk/jbossmx/src/main/org/jboss/ha/jndi/DetachedHANamingService.java
				MODIFY /trunk/jbossmx/src/main/org/jboss/ha/jndi/DetachedHANamingServiceMBean.java

Figure 2.1: A typical bug report from JBOSS issue tracker system

it has to pass through various quality checks before it is actually considered as a valid bug (bug that needs fixing). Quality check process identifies duplicate bugs, bug which does not consists of enough information, bug which are invalid etc. After a bug is found to be valid it is triaged to some expert who is expected to have knowledge to fix the bug, this expert is known as "Assignee" of bug. Now assignee's job is to fix the bug and commit its related changes to version control system. These changes get reviewed and verified by quality assurance team before final release. After final verification by quality assurance team a bug can be declared as closed. Although bug life cycle is shown to be sequential (for simplicity) in Figure 2.2 in reality it have pointers to previous stages. For example a closed bug can be reopened or assignee of the bug report can be changed (current assignee can assign bug to other expert). Complete detail for bug life cycle can be found in [21, 22].

2.2 Information Retrieval Model for Bug Localization

Information retrieval (IR) can be defined as: *"Retrieving relevant documents (or documents that satisfy user information need) from large and unstructured collection of documents"* [23]. Information retrieval is an art and science of searching (or retrieving) relevant documents

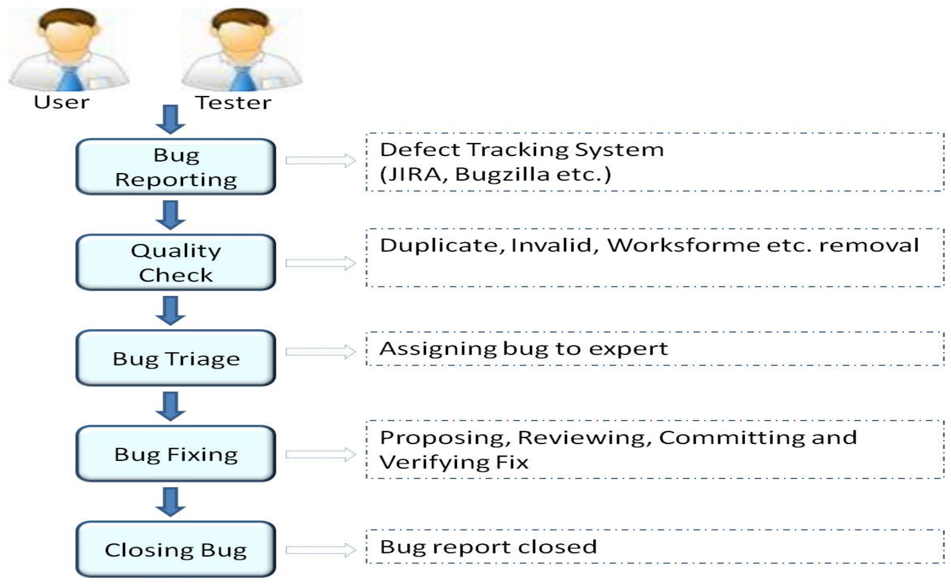

User
Tester
Bug Reporting
Defect Tracking System (JIRA, Bugzilla etc.)
Quality Check
Duplicate, Invalid, Worksforme etc. removal
Bug Triage
Assigning bug to expert
Bug Fixing
Proposing, Reviewing, Committing and Verifying Fix
Closing Bug
Bug report closed

Figure 2.2: Bug life cycle

from the large collection of documents[2], for example:

- Searching for articles on image processing.
- Retrieving web pages relevant to endangered species.
- Retrieving advertisement on latest laptops brands present in the market.

All these are real world examples that we encounter in our daily life. Web search engines such as Google, Yahoo, Bing etc. are the biggest applications of IR system. These search engines indexes millions of documents (unstructured or semi-structured nature) which are used for IR model building. When a user input's a query this IR model is used to provide user ranked list of document which are ordered according to their relevance to the given query.

IR models are gaining popularity in bug localization domain mainly because of two reasons: 1) scalability, and 2) language independence [3]. These features of IR model allow automated bug localization tools to remain applicable as software grows in size and complexity. Figure 2.3 is showing general IR system for bug localization problem. This system consists of four parts: 1) IR model formation, 2) document collection formation, 3) query

[2]`http://www.dsoergel.com/NewPublications/HCIEncyclopediaIRShortEForDS.pdf`

formation and 4) query execution. For bug localization problem IR models are build using software source code information. In addition to source code other information present with the software system such as software documentation, software specification or previous bug locations can be used for IR model formation. Document collection represents at which level of granularity bug localization system need to locate the bug, it can be at statement, method, class, or package level. Document collection is formed from source code by breaking it into desired level of granularity. Any new bug report is considered as a query for the system for which relevant documents need to be retrieved. New bug report are converted to query using query formation module. All this information (IR model, document collection, query) is used by query engine module to produce ranked list of documents from document collection. Documents are ranked in order of their relevance with respect to current query. These ranked documents can used by software developers to predict bug location during bug fixing. Various IR based techniques for bug localization have been proposed in the literature, some of these techniques we will discuss in related work (see chapter 8).

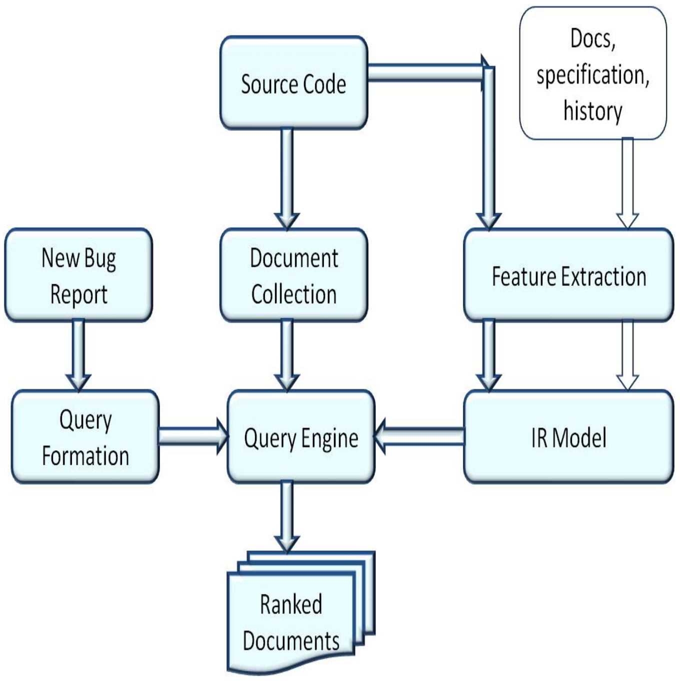


Figure 2.3: General IR model for bug localization

Chapter 3

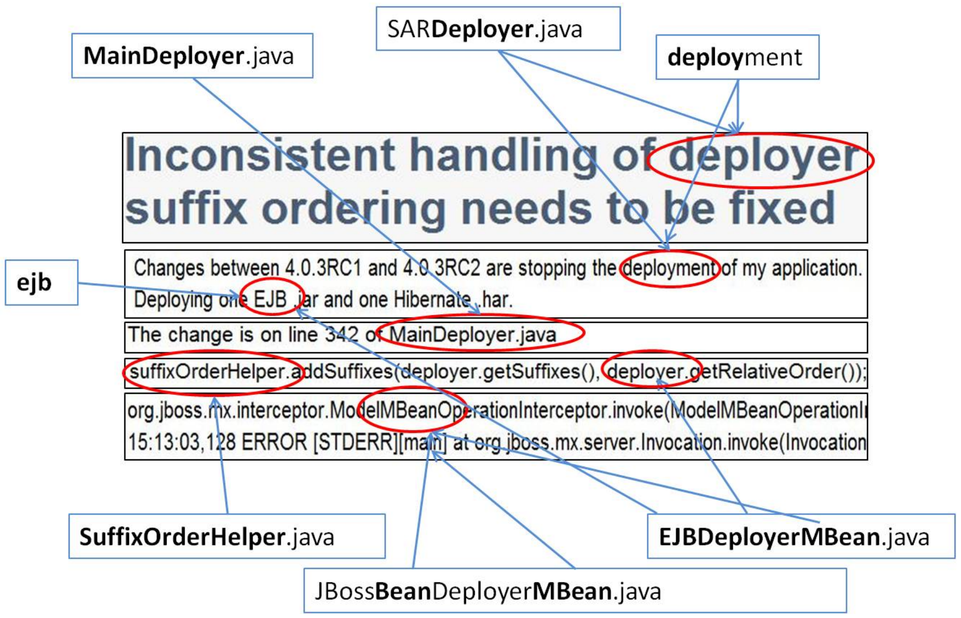
Character N-gram Model

N-gram means *"a subsequence of N contiguous items within a sequence of items"*. Word N-grams represent a subsequence of words and character N-grams represent a subsequence of characters. For example, various word-level bi-grams (N=2) and tri-grams (N=3) in the phrase *"Mining Software Repositories Research"* are: *Mining Software*, *Software Repositories*, *Repositories Research*, *Mining Software Repositories* and *Software Repositories Research*. Similarly, various character-level bi-grams and tri-grams in the word *"Software"* are: *So, of, ft, tw, wa, ar, re, Sof, oft, ftw, twa, war* and *are* respectively.

In contrast to word level N-gram textual features, character level N-gram feature extraction provides some unique advantages in context of information retrieval, which aligns with the task of bug localization. For example consider Figure 3.1, which is showing a snapshot of a bug report with name(s)/path(s) of file modified by this bug. This example is showing that there exists considerable overlap between character N-grams present in name(s)/path(s) of file modified by bug and character N-grams present in bug report text, such as *deploy*, *suffix*, *Bean* etc. Character N-gram based approaches can match these textual features between bug report and source code without requiring much preprocessing of text present in bug reports/source code. In contrast word level feature matching techniques will need preprocessing tools such as parser, stemmers to match concepts such as (*deployment, deployer*) or (*suffixOrderHelper, suffixOrderHelper.addSuffix()*).

Following are some of the advantages that character N-gram based approaches provides for the task of bug localization. Each advantage is supported by real world examples taken from JBOSS and Apache bug reports demonstrating effectiveness of proposed solution in bug localization domain:

1. Ability to Match Concepts Present in Source Code: Bug reports frequently consists of source code segments and system error messages which are not natural language text. Consider BugID JBAS-4649 which modifies the file *"JRMPInvokerProxyHA"*.



Inconsistent handling of deployer suffix ordering needs to be fixed

Changes between 4.0.3RC1 and 4.0.3RC2 are stopping the deployment of my application. Deploying one EJB .jar and one Hibernate .har.

The change is on line 342 of MainDeployer.java

```
suffixOrderHelper.addSuffixes(deployer.getSuffixes(), deployer.getRelativeOrder());
```

```
org.jboss.mx.interceptor.ModelMBeanOperationInterceptor.invoke(ModelMBeanOperationI
15:13:03,128 ERROR [STDERR][main] at org.jboss.mx.server.Invocation.invoke(Invocation
```

Figure 3.1: Example showing title and description containing N-grams of file name and file path modified by bug report

Bug report contains terms *"invoker HA proxies"*. Character N-gram based approach performs matching at character sequence level and concepts embedded in file names are matched with concepts present in bug reports. Although there are some lexical difference in these two terms they share several character N-grams which can be leveraged by character level matching techniques. In contrast a word level matching technique can not match concept present in these two strings (pre-processing of *camel case* terms is required).

2. Match Term Variation to Common Root: For example, BugID JBAS-1862 contains the term "*Interceptors*" and modifies a file with filename containing the term *"Interceptor"*. Terms *interceptor, interceptors, interception* are morphological variations of the term *intercept*. World level string matching algorithm will require a stemmer function to map all these morphological variation to common root for producing desired result. Character level analysis can detect concept match without a stemmer. Some more examples derived from the dataset: *(Call,Caller,Calling), (Manage, Manager, Managed), (Suffix, Suffixes), (Transaction, Transactions), (Proxy, Proxies), (Exception, Exceptions), (Enum, Enums), (Marshel,UnMarshel), (Unit, Units), (Connection, Connections, Connector), (Timer, Timers), (Authenticated, Unauthenticated), (Load, Loads, Loader), (Wrapper, Wrapped), (Starting, Startup), (Cluster, Clustering), (Pool,*

Pooling), (resend, resent), (Invoke, Invoker).

3. Ability to Match Misspelled Words: Consider BugID JBAS-1552 which consists of a misspelled word *"Managment"*. This bug modifies several files present in the *"Management"* directory. These two words share several character N-grams such as *"Man"*, *"ana"* , *"nag"* , *"men"*, *"ent"* , *"Mana"*, *"anag"*, *"ment"*, *"Manag"* etc and hence a character N-gram based algorithm will be able to give a good similarity score between these two words. Few examples derived from the dataset: *(behaiviour, behavior), (cosist, consist), (releated, related), (Releation, Relation), (chekc, check), (posible, possible), (Becouse, Because), (Managment, Management), (Princpal, Principal), (Periode, Period).*

4. Ability to Match Words and their Short-Forms: Bug Reporters frequently use short forms and abbreviation in the bug reports such as *spec* for *specification*, *app* for *application* etc. Character-level approaches are robust against use of such short forms, because a short-form shares several character N-grams with their expanded form. We found various such examples in our dataset such as: *(Config, Configuration), (Auth, Authentication), (repl, replication), (impl, implementing), (Spec, Specs, Specification), (Attr, Attribute), (Enum, Enumeration), (Sub, Subject).*

5. Ability to Match Words Joined Using Special Characters: Terms in bug reports are sometimes joined using various special characters. These special characters are used by bug reporters in different context, such as putting emphasis *"ScopedSet..."*, *[TimerImpl]*, or joining compound words such as *"module-option"*. Word-level matching requires knowledge of such facts and will require a domain specific tokenization (text pre-processing) tool. In contrast, character level analysis techniques are able to detect concept matching because of partial matching.

6. Ability to Match Substring Permutations: A words based approach which is able to get substrings (or splitting using camel case) of a string, generally can not detect orientation difference in the words. For example consider the file names *"JMSManagedConnection"* and *"JMSConnectionManager"*. After splitting followed by stemming of these two strings, a word level approach will get substrings *"jms"*, *"connection"*, *"manager"* for both the strings. As word based approaches are generally represented as "bag-of-words" they do not have any ordering information. Hence a word level approach can not detect simple permutations of substring present in the word and hence may not be able to produce the desired result. While a character level algorithms do not require such hard-word splitting, they can easily detect such permutation. Strings having same order of characters will get higher score in character N-gram based approach and algorithms will be able to produce the desired result. Table 3.1 lists the

advantages of the character N-gram based analysis over word-based analysis.

S. No.	**File/Directory Name**	**Title/Description**	**Description**
1	local,HALocalManagedConnectionFactory	<-local-tx-datasource>	Word based technique will require parsing of *"<-local-"* and camel case break to match file *"HALocalMangedConnectionFactory"* and directory *"local"*.
2.	cmp	cmp2	To match *"cmp"*(desired directory) with *"cmp2"*, a word based technique will require to remove digit from *"cmp2"*.
3.	EjbModule	EjbModule. addInterceptor	*"EjbModule.addInterceptor"* is a source code fragment which is giving information about file method that need to be fixed, but to match it with desired file (*EjbModule*) a word level analysis technique will require to split it on dot(.) to separate method (*addInterceptor*) from file name(EjbModule).
4.	JDBCEJBQLCompiler	EJBQLToSQL92Compiler	These two strings consists of so many matching character N-grams, but still its difficult for a word based technique to match these words, as most of the stemming or parsing algorithm wont be able to extract individual concepts.
5.	ScopedSetAttribute TestCase	"ScopedSet..." Unit tests	There exists considerable overlap between these two strings but still a word based technique will require careful parsing of the word *"ScopedSet"* and also need to apply camel casing break on string *"ScopedSetAttributeTestCase"* followed by stemming of word *"Tests"*, to match these two strings.
6.	TimerImpl	[TimerImpl]	To match these two strings word based technique require knowledge of "[", "]" and to do the parsing accordingly.

7.	ModuleOptionContainer	<jaas:module-option name = “unauthenticatedIdentity” > guest </jaas:module-option>	Column 3 is showing fragment of configuration file present in one of the bug reports. Configuration files requires a different parsing mechanism, in this case will require removal of “:”, “-” and other words concatenated with *“module-option”*, to match the similar concept present in source code file and bug report.
8.	class-loading	class loader	Word based technique will require to remove *hyphem(“-”)* from file name and also need to perform stemming on *loader* and *loading*, to match file name *(“class-loading”)* and string *(“class”, “loader”)*.
9.	UsernamePassword LoginModule	Password/User	In this case word based technique not only require to perform tokenization based on “/” but also need to match *“User”* and *“Username”*.
10.	InvokeCommand	’invoke’	Intelligent *parsing* and *camel case* break is required for word based technique to match these two strings.
11.	cmp, jdbc2, JDBCStoreManager2	/cmp/jdbc2/JDBCStoreManager2	Complete path for file that need to modified is given in a bug report(see column 3), but still it can not be matched unless a word based technique perform tokenization based on “/”.
12.	JDBCFindByPrimaryKeyQuery	Bean2Home.findByPrimaryKey(pk)	Column 3 is containing source code fragment which requires parsing of *“function calls”* to match concept present in source code file(column 2) with concept in bug report(column 3).
13.	JaasSecurityManagerService	(“jboss.security:service= JaasSecurityManager”)	Source code fragment in bug report which requires removal of ‘=’,‘ ” ’, etc. to match both the strings
14.	WebPermissionMapping	“WebPermissionMapping”	Even though complete file name *“WebPermissionMapping”* is present int the bug report a word based techniques can not match two string without removing double quotes(”).

15.	AbstractDeploymentScanner	(jboss.deployment.scanner.AbstractDeploymentScanner$Scanner Thread)	Fragment of bug report containing stack trace, which needs parsing of dots(.) and character ($). In general word based technique require smart parsing of stack traces present in bug reports to match them with source code.

Table 3.1: Illustrative examples of concepts present in bug reports and file-name and path-name (derived by examining the experimental dataset) demonstrating the advantages of the character N-gram based analysis over word-based analysis

Chapter 4

Solution Approach

In this chapter we will describe our system architecture and its various building blocks. Figure 4.1 presents the high level architecture of the proposed solution. This system consists of three components: (1) Character N-gram based feature extractor (2) Similarity computation module and (3) Rank generator, detailed description of each module is as follows:

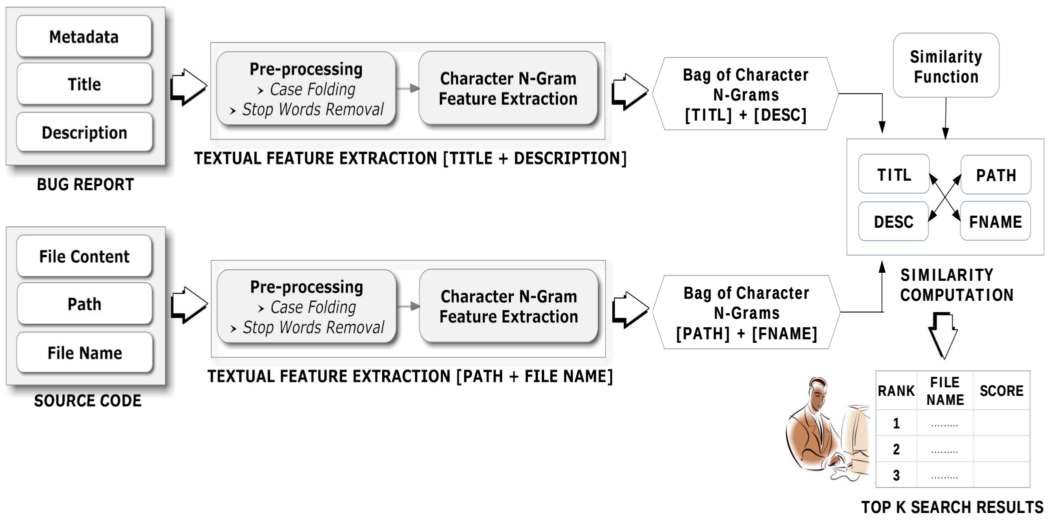


Figure 4.1: High-level solution architecture depicting the text processing pipeline, various modules and their connections

4.1 Feature extraction

Bug reports and source code files are structured documents consisting of various attributes such as title, description, reporter, version (for bug report) and file name, file path, method name (for source code). Feature extraction module extracts desired attributes from bug report (title and description) and source code (file name and file path). This module then pre-process (removes stop words and converts them into lowercase) these features and converts them into bag-of-character N-grams (of length 3 to 10) document representation. This module returns four bag-of-character (or vectors) N-grams, one for each feature which further used by other modules.

4.2 Similarity Function

We define a similarity function for computing the textual similarity or relatedness between a bug report (query represented as a bag-of-character N-grams) and a source-code file (document in a document collection represented as a bag of character N-gram). Equation 4.1 presents the formula for computing the similarity between two documents in the proposed character N-gram based IR model. Let U and V represent a vector of character N-grams. For example, U can be a bag-of-character N-gram derived from the title of the bug report and V can be a bag of character N-gram derived from the source code file-name. Similarly, character N-grams from bug report description and directory path can be assigned as U and V respectively.

$$\boxed{SIM(U,V) = \frac{\sum_{u \epsilon U} \sum_{v \epsilon V} Match(u,v) \times Length(u)}{|U| \times |V|}} \tag{4.1}$$

$$Match(u,v) = \begin{cases} 1 & if : u = v \\ 0 & Otherwise \end{cases} \tag{4.2}$$

$$|U| = \sqrt[2]{f_{u_1}^2 + f_{u_2}^2 + .. + f_{u_n}^2} \tag{4.3}$$

The numerator of $SIM(U,V)$ first compares every element in U with every element in V and measures the number of matches. The value of n is added to the cumulative sum in case of a match (character N-grams being exactly equal). The higher the number of matches, the higher is the similarity score. Matches containing longer strings contributes more towards

the sum (as the length of the string is added) in contrast to strings which are shorter.
The denominator of $SIM(U,V)$ is the length normalizing factor. The normalization factor in the denominator is used to remove biases related to the length of the title and description in bug reports as well as the file-name and path-name for source-code files. We perform experiments with and without normalization to study the influence of length normalization on the overall accuracy of the retrieval model. Variable f_{u_i} represents the frequency of an n-gram (u_i) in the bag of character N-grams in equation 4.3 .

4.3 Rank Generation

Equation 4.1 is used to calculate individual similarity score between feature pairs, T-F (title and file name), T-P (title and path), D-F (description and file name) and D-P (description and path). These individual similarity scores can be combined in several ways to compute final similarity score between source code file and bug report. As shown in Equation 4.4, the overall similarity is computed as a weighted average of these four individual scores, represented by $SIM_{SCORE}(BR,F)$. Rank of a file for a bug report is calculated using its SIM_{SCORE} value, high value of SIM_{SCORE} represents high similarity and hence better rank. This function consists of several tuning parameters (W_1, W_2, W_3, W_4) as a result of which multiple implementations of the similarity function are possible. We experimented with different values of the configuration parameters to learn the impact of various variables used for computing textual similarity and also to identify configuration which gives best results. We will discuss results obtained in chapter 7.

$$\boxed{\begin{aligned} SIM_{SCORE}(BR,F) = W_1 * SIM(T-F) + W_2 * SIM(T-P) + W_3 * SIM(D-F) + \\ W_4 * SIM(D-P) \end{aligned}} \tag{4.4}$$

Chapter 5

Performance and Evaluation Metrics

We measured the performance (predictive accuracy) of the proposed approach using two evaluation metrics: SCORE and MAP (Mean Average Precision). SCORE and MAP metric are employed as evaluation metric by previous studies on bug localization using Information retrieval based techniques [6][4][7].

5.1 SCORE

SCORE metric denotes the percentage of documents in the repository (search space) that need not be investigated by the bug-fixer for the task of bug localization. Accuracy and SCORE are directly proportional, higher SCORE value means higher accuracy. Assume file level granularity and consider a situation where the size of the search space is equal to 1000 files. A SCORE of 80% (or 0.80) means that the bug resides in top 20% (200 files in a search space of 1000 files) of the document (file) collection. For example consider a bug for which three files need to be fixed, let these file given rank 12, 40, and 60 respectively by some technique 'X'. If there are 200 files in the search space, SCORE value for this bug report (using technique X) is: (200-60)/200 =70%. Mathematical formulation of SCORE metric is presented in equation 5.1, where 'm' is number of files that a bug modifies, 'N' is the total number of files present in the search space, and fi_{Rank} is the rank assigned to i^{th} file. SCORE is a standard and common evaluation metrics for measuring the effectiveness of bug-localization approaches [6][4][7].

$$\boxed{SCORE = \left(1 - \frac{max(f1_{Rank}, f2_{Rank}, ..., fm_{Rank})}{N}\right) \times 100} \tag{5.1}$$

5.2 MAP

Mean average precision (computed for a set of queries i.e., for the set of bug reports in the evaluation dataset) is equal to the mean of the Average Precision (AP) scores for each query in the experimental dataset. The proposed IR model computes the similarity score between the query and each of the file in the document collection and returns a ranked list of files based on the computed numeric score. AP consists of computing the precision of the system at the rank of every relevant document retrieved. MAP is a well know metric to measure retrieval performance for IR systems [3][8].

Equation 5.2 presents the general formula for AP wherein P_i denotes the precision at i^{th} relevant file retrieved and M denotes the total number of relevant file for a bug. For example, consider a bug report (query) which modifies three files (number of relevant files is equal to three). Assume the system assign ranks 2, 4 and 7. In this example, the AP is calculated as shown in equation 5.3. MAP is calculated by taking the mean of AP values for all the bug reports in the dataset(see Equation 5.4 where N is total number of bugs in database).

$$\boxed{AP = \frac{(P_1 + P_2 + ... + P_M)}{M}} \tag{5.2}$$

$$\boxed{AP = \frac{\frac{1}{2} + \frac{2}{4} + \frac{3}{7}}{3}} \tag{5.3}$$

$$\boxed{MAP = \frac{(AP_1 + AP_2 + ... + AP_N)}{N}} \tag{5.4}$$

Chapter 6

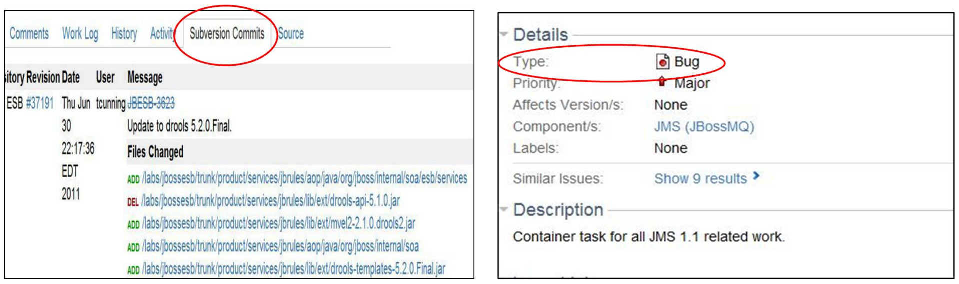
Experimental Dataset

In order to validate our hypothesis (character N-gram based technique is effective in bug localization), we performed an empirical evaluation on dataset downloaded from the issue tracking system of two popular open source projects: JBOSS[1] and APACHE GERONIMO[2]. The dataset is publicly available as a result of which the experiments performed in this work can be replicated in future for improving this technique and for comparing it with other techniques. We conducted experiments on dataset belonging to two open source projects to remove any project-specific bias and to investigate the generalization power of proposed solution. We implemented a crawler using JIRA API[3] and downloaded the relevant data (bug report metadata, title description, modified files etc).
Table 6.1 displays details regarding the experimental dataset downloaded from JBOSS and Apache open source projects. We extracted 569 bug reports from the JBOSS project and 637 bug reports from the Apache project satisfying certain conditions. As shown in the Table 6.1, one of the conditions is that the issue report should be CLOSED/RESOLVED and FIXED/DONE. Furthermore we extracted bug reports which are of type BUG (as the focus of the work is bug localization) and not other types of issues such as feature requests, task or sub-task. In JIRA issue tracking system, the issue tracker and the SVN commits are integrated and hence the ground-truth (all the files modified for a bug report) is available. The ground-truth (actual value) is compared with the predicted value to compute the effectiveness of the proposed IR model. JIRA issue tracking system provides information about TYPE of issue and provides implicit link between issues and its SVN commit, as shown in figure 6.1. Hence it is not required to explicitly identify type of issue or traceability link between bug report and its respective SVN commit.

[1] `https://issues.jboss.org`
[2] `https://issues.apache.org/jira/browse/GERONIMO`
[3]`http://docs.atlassian.com/software/jira/docs/api/latest/`

	Label	**JBOSS**	**APACHE**
A	Start Issue ID	1 (JBAS-1)	1 (GERONIMO-1)
B	Last Issue ID	8879 (JBAS-8879)	6143 (GERONIMO-6143)
C	Reporting Date Start Issue	2004-11-7	2003-08-20
D	Reporting Date Last Issue	2011-2-16	2011-09-06
E	Number of Issue Reports Downloaded	8072	4092
F	Number of Issue Reports Unable to Download	807	2051
G	Number of Issue Reports (Status = CLOSED/RESOLVED, Resolution = DONE and containing SVN commit)	2433	1915
H	Number of Issue Reports of Type = BUG AND in SET G	1114	1269
I	Number of Issue Reports in SET H AND containing at-least one modified Java file	1090	939
J	Number of Issue Reports in SET I AND a specific version	576 (VERSION = 4.x)	736 (VERSION = 1.x, 2.x, 3.x)
K	Number of Issue Reports in SET J AND for which MODIFIED file Found in specified version	569	637

Table 6.1: Details regarding the experimental dataset from JBOSS and Apache projects



```
itory Revision Date    User     Message
ESB #37191     Thu Jun tcunning JBESB-3623
               30               Update to drools 5.2.0.Final.
               22:17:36         Files Changed
               EDT              ADD /labs/jbossesb/trunk/product/services/jbrules/aop/java/org/jboss/internal/soa/esb/services
               2011             DEL /labs/jbossesb/trunk/product/services/jbrules/lib/ext/drools-api-5.1.0.jar
                                ADD /labs/jbossesb/trunk/product/services/jbrules/lib/ext/mvel2-2.1.0.drools2.jar
                                ADD /labs/jbossesb/trunk/product/services/jbrules/aop/java/org/jboss/internal/soa
                                ADD /labs/jbossesb/trunk/product/services/jbrules/lib/ext/drools-templates-5.2.0.Final.jar
```

Details

Type:	Bug
Priority:	Major
Affects Version/s:	None
Component/s:	JMS (JBossMQ)
Labels:	None
Similar Issues:	Show 9 results

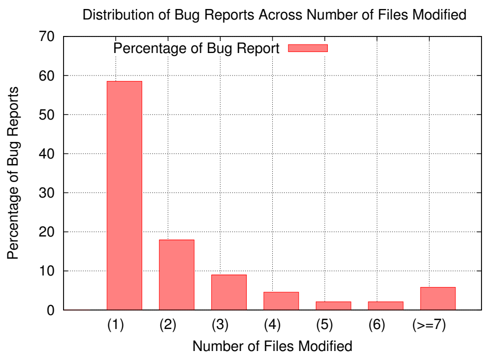
Description

Container task for all JMS 1.1 related work.

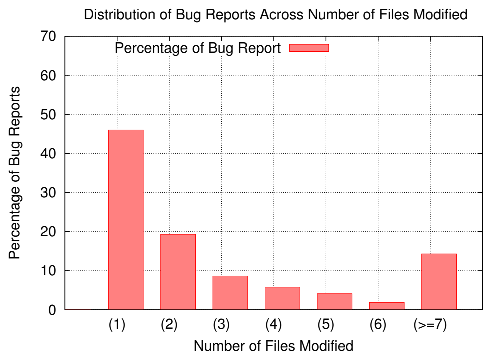
(a) Implicit Link between Bug Report and Related SVN Commit

(b) Implicit Distinction between Issue Types

Figure 6.1: Features of JIRA Issue Tracking System



(a) JBOSS Dataset



(b) Apache Dataset

Figure 6.2: Distribution of the number of files modified in JBOSS and Apache dataset

Bug reports have a field called as *"affected version"* in which the bug reporter specifies the version number of the application that caused the failure. We downloaded bug reports belonging to specific versions (for which sufficient bug reports were available for experimental proposes) for JBOSS and Apache (refer to Table 6.1 for details). We download the source code for various versions mentioned in the dataset as the bug localization is performed on the specific version on which the bug was reported. For example JBOSS dataset consists of a total of 20 versions (see appendix A for more details). In our case, the size of the search space is equal to the number of Java files in the software. For the 20 versions in JBOSS dataset, the minimum value for the number of files (size of the search space) is: 6212, maximum is: 9202 and average is: 7947.7. For the Apache dataset 28 versions(see appendix A) are downloaded (minimum size of search space is 1428, maximum size is 3182 and the average value is 2305). Figure 6.2 is showing distribution of bug reports depending on number of files they modified, for example 58% bug reports in JBOSS and 46% bug reports in Apache modified only one source code file. This distribution is calculated to understand dataset characteristics.

Chapter 7

Experimental Results

Multiple experiments are done to measure effectiveness of proposed solution for bug localization task and to get more insights. Proposed solution used to rank each source code file present in the system (version specified in the bug report) according to final score obtained by them (see equation 4.4). These ranked files are then compared with ground-truth (available through JIRA issue tracker system) to evaluate system effectiveness. Apart from measuring effectiveness of the proposed solution we did experiments to identify which one of the four features is most effective for bug localization and to identify effect of length normalization on bug localization accuracy. In the following sections we will discuss details of our experiments and results obtained.

7.1 Predictive Power of Each Predictor Variable

We experimented with five tuning parameters (SIM-EQ, SIM-TF, SIM-TP, SIM-DF and SIM-DP as described in Table 7.1) to throw light on the predictive power of the four predictor variables (title and description in bug report with filename and path in version control system). Weights W(T-F), W(T-P), W(D-F) and W(D-P) present in table 7.1 denotes the weights W_1, W_2, W_3 and W_4 present in equation 4.4 respectively. Figure 7.1 and 7.2 are displaying descriptive statistics using a box-plot (five-number summary: minimum value, lower-quartile (Q1), median (Q2), upper quartile (Q3) and maximum value) of SCORE values (under five different tuning parameters) for the JBOSS and APACHE experimental dataset respectively. In addition to the 25^{th}, 50^{th} and 75^{th} percentile, Figure 7.1 and 7.2 also reveals the degree of dispersion or spread in the SCORE values. We found that ***SIM-EQ*** (the weight for all the four predictor variables being equal or 0.25 each) configuration *outperforms all other configuration.*

The box-and-whisker diagram of figure 7.1 reveals that the textual similarity between the bug report description and the file-name (JBOSS dataset) is relatively better predictor (Q1:70.38%, Q2:97.03%, Q3:99.92%) in contrast to the other three predictors. Figure 7.1 and 7.2 are showing that bug report *description* is a better predictor than bug report title.

	Label	Norm.	W(T-F)	W(T-P)	W(D-F)	W(D-P)
1	SIM-EQ	N	0.25	0.25	0.25	0.25
2	SIM-TF	N	1	0	0	0
3	SIM-TP	N	0	1	0	0
4	SIM-DF	N	0	0	1	0
5	SIM-DP	N	0	0	0	1

Table 7.1: Five different configurations of weights (for each of the four comparisons: T-F, T-P, D-F and D-P) to study the predictive power and influence of each of the four possible comparisons between bug report attributes and source-code file attributes

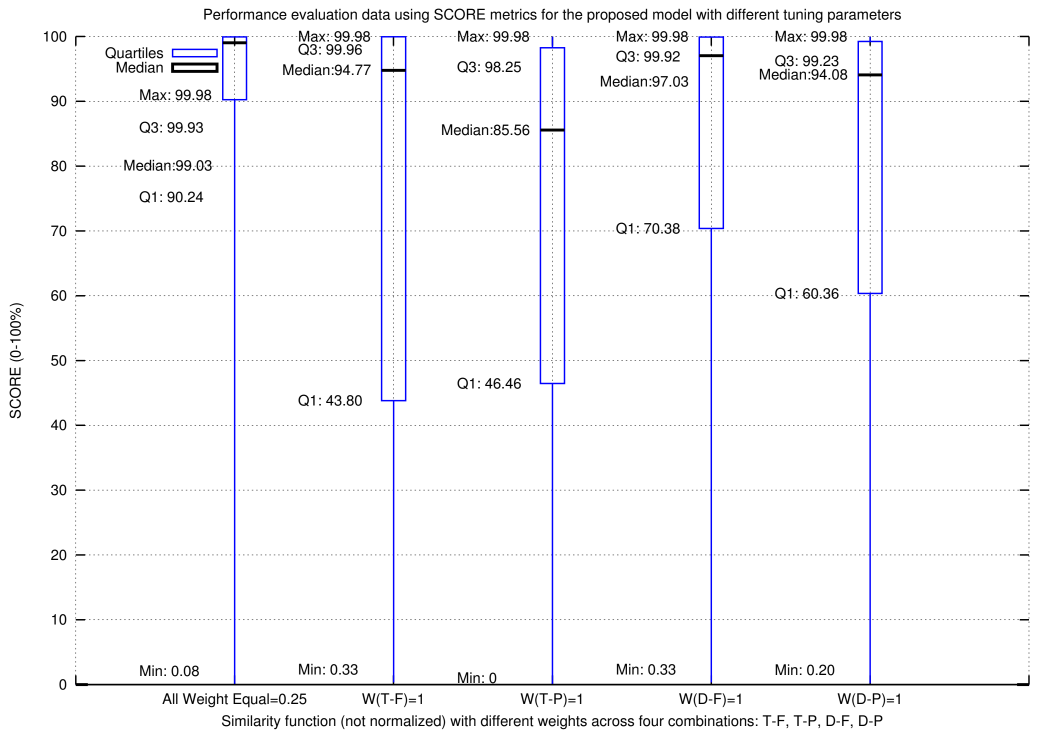


Figure 7.1: A box-and-whisker diagram displaying minimum, lower quartile (Q1), median (Q2), upper quartile (Q3), and maximum SCORE values for the proposed IR model with five different tuning parameters (similarity function is not normalized) to study the predictive power of each of the four feature combinations (T-F, T-P, D-F and D-P)[JBOSS Dataset]

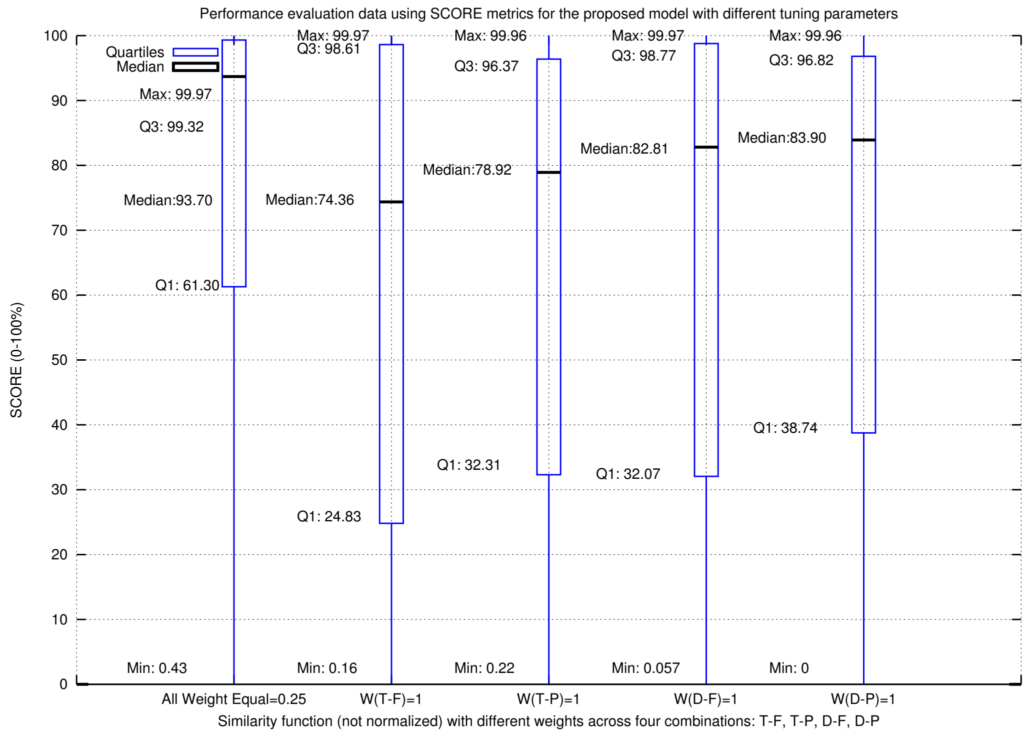


Figure 7.2: A box-and-whisker diagram displaying minimum, lower quartile (Q1), median (Q2), upper quartile (Q3), and maximum SCORE values for the proposed IR model with five different tuning parameters (similarity function is not normalized) to study the predictive power of each of the four feature combinations (T-F, T-P, D-F and D-P)[APACHE Dataset]

7.2 Effectiveness of Character N-gram Model

We measured effectiveness of our approach using multiple metrics to provide different perspectives as explained below. In rest of the experiments we will use SIM-EQ configuration, as it was found to be best in above experiments. Figure 7.3 is showing box-plot (described above) for **SCORE** values obtained by each bug report (using SIM-EQ configuration) for both JBOSS and Apache dataset. This figure is showing that for the SIM-EQ (the weight for all the four predictor variables being equal or 0.25 each) experimental setting, the median value for the SCORE metric is 99.03% for the JBOSS dataset (refer to Figure 7.3). This indicates that for 50% of the bug reports in the JBOSS dataset, the proposed solution was able to prune the 99.03% search space (which means that the bug fixer needs to investigate about 1% of the source files to localize the bug). The median value for the SCORE metric (SIM-EQ experimental setting) is 93.70% for the APACHE GERONIMO project.

We used **Average Precision** as another metric to evaluate effectiveness of our system. The histograms in Figure 7.4a and 7.4b are displaying the distribution of average precision values for the set of bug reports in the JBOSS and APACHE evaluation dataset respectively. The average precision value is computed at all recall levels (rank at which each relevant document is retrieved). Consider a situation where a bug report modifies three files. Assume that the search result rank for each of the files are: 5, 10 and 30 respectively. The average precision (at each recall point) in this case will be: $[1/5 + 2/10 + 3/30]/3 = 0.167$. We have used non-normalized similarity function and SIM-EQ configuration to calculate average precision in figure 7.4a and 7.4b . The X-axis in figure 7.4a and 7.4b denotes the range (0.3 means from 0.2 to 0.3). Figure 7.4a reveals that for 16.16% of the bug reports in the JBOSS dataset, the average precision value is between 0.9 to 1.0. Figure 7.4b shows that for 10.67% of the bug reports in the APACHE dataset, the average precision value is between 0.9 to 1.0.

Figure 7.5 presents another complementary perspective to examine the effectiveness of the proposed bug localization method. Figure 7.5 displays a histogram in which the X-axis represents document rank 1 to 5 (in the search results) and the Y-axis represents percentage of bug reports in the JBOSS and APACHE dataset. We observed that the percentage of bug reports in the JBOSS and APACHE dataset in which at-least one of the modified file(s) is retrieved as ***rank 1*** is 40% and 26% respectively. This shows that there is a significant (above 25%) percentage of bug reports for which the solution was able to retrieve at-least one of the files in the impact-set as the first search result (rank 1).

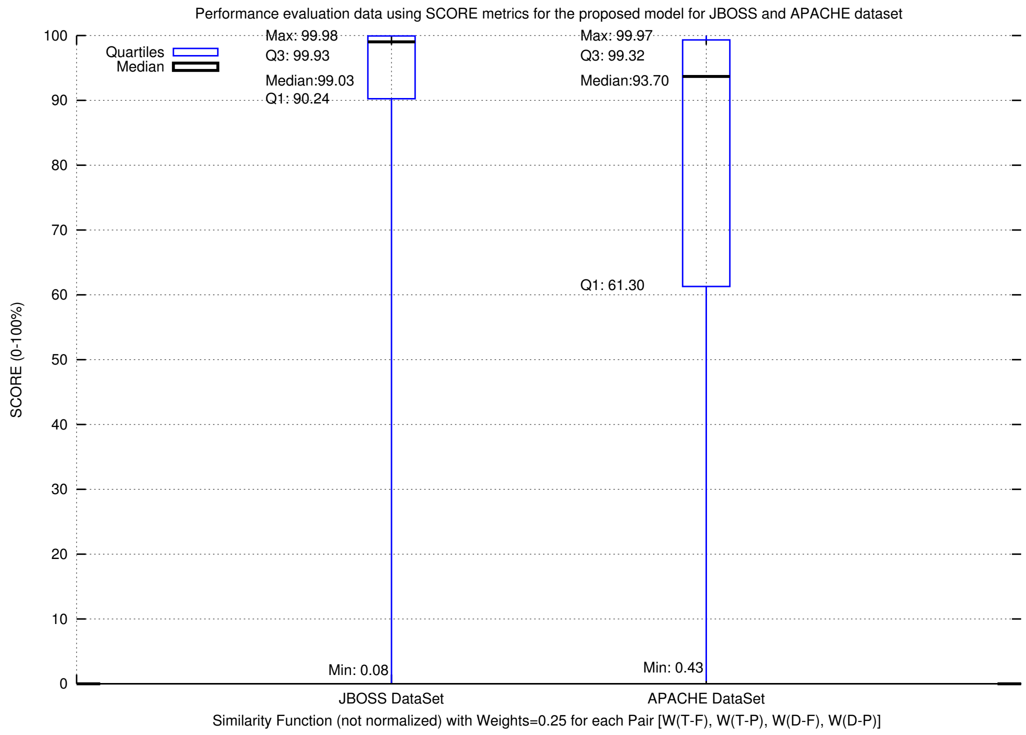
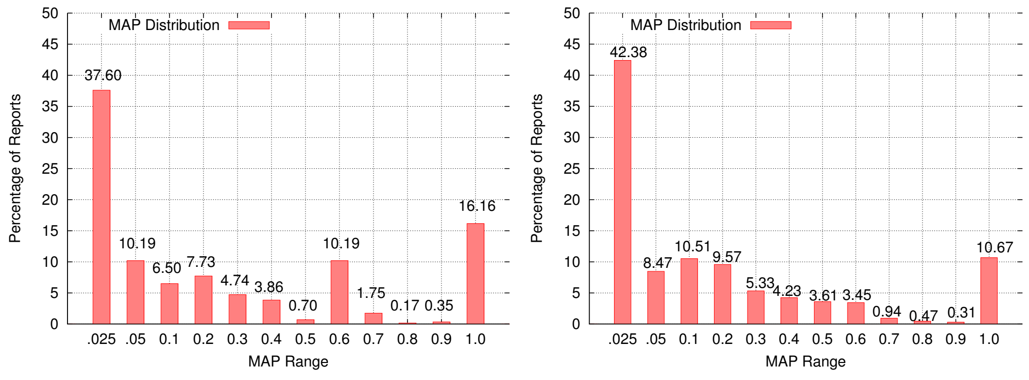


Figure 7.3: A box-and-whisker diagram displaying minimum, lower quartile (Q1), median (Q2), upper quartile (Q3), and maximum SCORE values for the proposed IR with equal weights given to each tuning parameter (similarity function is not normalized) [JBOSS and APACHE Dataset]



(a) JBOSS Dataset (b) Apache Dataset

Figure 7.4: Distribution of average precision values for the set of bug reports in the JBOSS and APACHE evaluation dataset. The average precision value is computed at all recall levels (ranks at which each relevant document is retrieved).

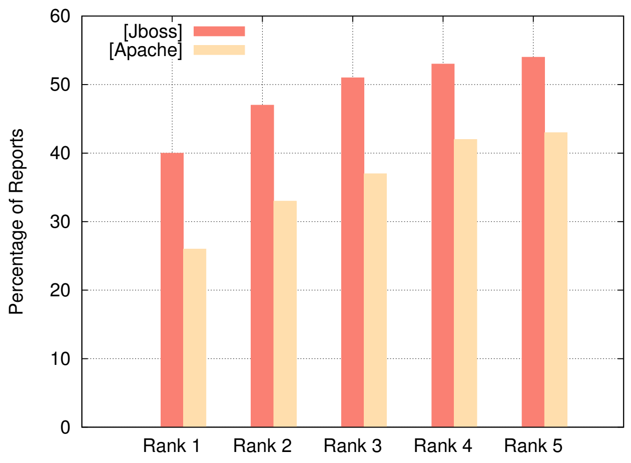
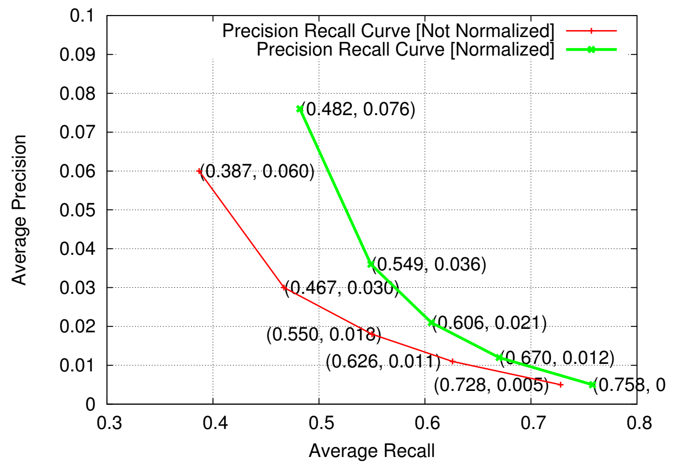


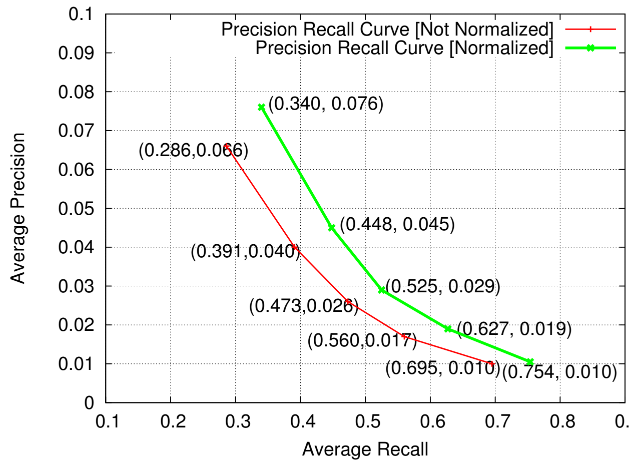
Figure 7.5: Percentage of bug reports in the JBOSS and APACHE dataset in which at-least one of the modified file(s) is retrieved as Rank 1-5

7.3 Normalization Effect

Normalization is considered as an important pre-processing step in most of the information retrieval applications. Fry et. al mentioned in their work that normalization in bug localization task may result in loss of precision, and hence not required. In our work we tried to throw light on this thought by comparing precision achieved by normalized and non-normalized settings. Figure 7.6a and 7.6b shows the precision-recall curve (for predefined values of Nr or Top K rank) for the JBOSS and Apache dataset respectively. The precision values in Figure 7.4a and 7.4b is computed at each recall point (the average precision value is reported) whereas in Figure 7.6a and 7.6b the precision and recall values are computed for pre-defined values of Top K (where K = 10, 25, 50, 100 and 250). For example, if 2 out of 5 relevant documents (impacted source code files) are retrieved amongst the Top 50 search results (Nr=50), then the precision at Nr=50 is 2/50 = 0.04 and the recall is 2/5 = 0.4. We calculated the precision and recall value for each bug report at all the five pre-defined Nr values (for both JBOSS and Apache dataset, normalized and not-normalized setting) and then computed the average precision and recall for various Nr values (SIM-EQ setting). As illustrated in Figure 7.6a and 7.6b, the observed trend is that the recall value increases as Nr increases and the average precision declines. Figure 7.6a and 7.6b reveals that the effectiveness of the system (in terms of precision and recall) is more in length normalized settings as compared to non-normalized settings.



(a) JBOSS Dataset

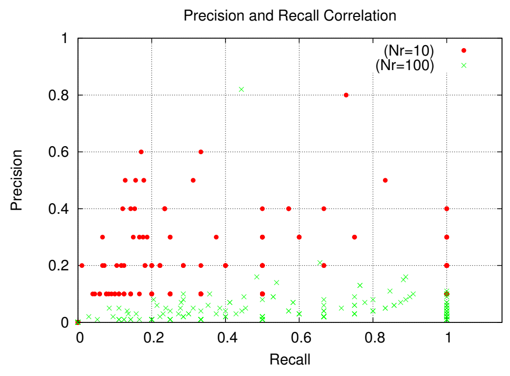


(b) Apache Dataset

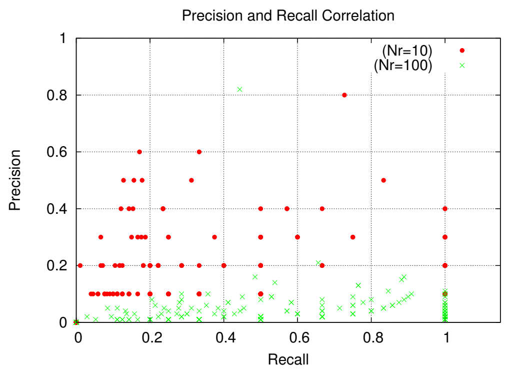
Figure 7.6: Precision-Recall Curve (average precision at various average recall values) for the normalized as well as not-normalized setting

7.4 Scatter Plots

Figure 7.6a and 7.6b are plots of the average values of precision and recall for entire bug dataset whereas figure 7.7 and 7.8 are scatter plots in which each point is representing precision and recall value for a bug report. Precision and recall values are calculated for fixed Nr values for normalized and SIM-EQ configuration. Figure 7.7 and 7.8 are plotted for Nr=(10,100) and Nr=(10,25,50,100,250) respectively.

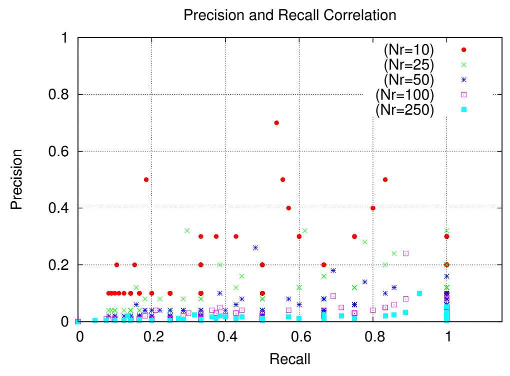


(a) JBOSS Dataset

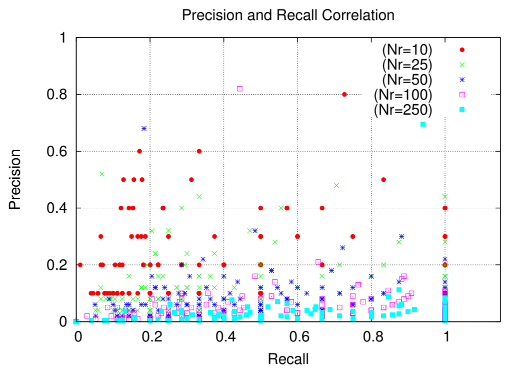


(b) Apache Dataset

Figure 7.7: Scatter plot displaying the precision and recall value for each bug report (fixed and pre-defined values of Nr=10 and Nr=100)



(a) JBOSS Dataset



(b) Apache Dataset

Figure 7.8: Scatter plot displaying the precision and recall value for each bug report (fixed and pre-defined values of Nr=10,25,50,100,250)

Chapter 8

Related Work

Automated solutions for the software engineering task of bug localization using information retrieval models (static analysis based technique in contrast to dynamic analysis which requires program execution) is an area that has attracted several researcher's attention. Rao et al. performed a survey and comparison of few IR based models for bug localization. In their study they found that simple text models such as Unigram Model (UM) and Vector Space Model (VSM) were found to be more effective compared to more complex models such as LDA. We reviewed traditional techniques and presented some of the main points relevant to this work in Table 8.1. As displayed in Table 8.1, we arranged previous work in a chronological order and mentioned the key features of previous methods. We studied previous methods for IR based bug localization in terms of the IR model, granularity level, experimental dataset (case study) and the bug reports and source code elements used for comparisons. For example, Antoniol et al. performed experiments on LEDA C++ Class library and work at Class level granularity whereas Lukins et al. performed experiments on Rhino, Eclipse and Mozilla dataset and worked at Method level granularity (refer to Table 8.1). Table 8.1 is also presenting some work on concept location (or feature location), which is closely related to bug localization task. For rest of the section we focused our discussion only on bug localization techniques.

Table 8.2 is showing comparison of some bug localization techniques([1, 2, 4, 13]) which are closely related to proposed solution. Chen et al. proposed a social network based technique between co-cited bug locations to predict faults. They used PageRank[1] to calculate rank of bug locations. Location of a new bug is predicted using similar bug reports and their fixed locations. This model is able to evolve over a period of time using self-learning mechanism (using new bug reports and their bug locations). They reported accuracy

[1]`http://www.google.com/about/corporate/company/tech.html`

	Study	Bug Reports	Source Code	Granularity	IR Model	Case Study
1	Antoniol' 2000 [9]	Maintenance requests (excerpt from the change log files)	source code or higher level documents	Class level	vector space model and stochastic language model	LEDA C++ Class library
2	Canfora' 2005 [10]	Title & description of new and past change requests	Revision comments	File Level	Probabilistic model described in [Jones, 2000]	Mozilla Firefox and KDE (kcalc, kpdf, kspread, koffice)
3	Marcus' 2005 [11]	Specific keywords in change requests	Source code	Class level	latent semantic indexing, dependency search	AOI 3D modeling studio, Doxygen
4	Poshyvanyk' 2006 [14]	Bug Description	Source code elements	Class Level and Method Level	SBP Ranking and LSI Based	Mozilla SBP Ranking and LSI Based
5	Chen'2008 [1]	Meta Information and Description	Bug report and source file links	—	PageRank, Vector Space Model	SVN, AgroUML
6	Lukins' 2008 [2]	Manual extraction of keywords from Title and Description	Comments, Identifiers and String-Literals	Method Level	LDA Based	Rhino, Eclipse and Mozilla
7	Fry' 2010 [4]	Title, Description, Stack Trace, Operating System	Class Name, Method Names, Method Bodies, Comments, Literals	File Level	Word-based cosine similarity vector comparison	Eclipse, Mozilla, OpenOffice
8	Moin' 2010 [13]	Title & description	Source File Hierarchy	Revision Path Level	SVM Classifier	'UI' component Eclipse Project
9	Nichols' 2010 [12]	Bug description of old and new bugs	Identifier names, string literals	Method Level	LSI Based	Rhino

Table 8.1: Previous approaches for bug localization using static analysis and information retrieval models. These papers are listed in a chronological order and the traditional methods are classified into various dimensions.

of 40.9% and 15-20% on SVN and AgroUML projects respectively. Fry et al. proposed a bug localization technique which leverages similarity between textual features of bug report and source code. They did comprehensive analysis of 28 feature pairs between bug report and source code and identified 12 most suited pairs for bug localization task, using ANOVA technique.[2] Bug report title and method bodies (in source files) was found to be most effective pair for bug localization. Experiments on three projects showed that their technique is effective and able to prune nearly 88% of search space. Lukins et al.experiment showed that LDA outperforms LSI based bug localization tecnique. In their work comments, string literals and other source code elements was used to build LDA module. Manual queries were generated from bug report which were given as an input to LDA model to generate ranked files as output. Moin et al. proposed a bug localization technique based on Support Vector Machine (SVM) classifier. Their model used title and description of bug report as feature and path of the file modified as class label for classifier training. First time they used file path level granularity to localize bug which can be useful in some applications such as bug triage [13]. Their system was able to achieve 98% accuracy(precision and recall). All these techniques are based on textual similarity between bug report and source code files, which shows that textual features are useful in bug localization.

	Study	**Metric Used**	**Project**	**Dataset Size (No. of Bug Reports)**	**Performance**
1	Chen'2008 [1]	Hit accuracy	SVN AgroUML	211 1024	40.9% 15%-20%
2	Lukins'2008 [2]	Rank of retrieved files	Rhino	35	77%
3	Fry'2010 [4]	SCORE	OpenOffice Eclipse Mozilla	1040 1272 3033	67.91% 86.9% 92.15%
4	Moin'2010 [13]	Accuracy	'UI' component of Eclipse project	2000(approx.)	98.51%

Table 8.2: Performance of some closely related *bug localization* techniques.

[2]http://www.experiment-resources.com/anova-test.html

Chapter 9

Conclusions

We conclude that degree of textual similarity between title and description of bug reports with source code file-names and directory paths can be used for the task of bug localization. Experimental evidences demonstrate that low-level character N-gram based textual features are effective and have some unique advantages (robustness towards noisy text, natural language independence, ability to match concepts present in programming languages and string literals) in comparison to world-level features.

We performed experiments on publicly available real-world dataset from two popular open-source projects (JBOSS and Apache) and investigated the overall effectiveness of the proposed model. In addition we investigated impact of length normalization and effectiveness of each of the four predictor variables (title-filename, description-filename, title-path, and description-path). Empirical evidences revealed that length normalization improves average precision and recall and description has more predictor power than title. The results (measured in terms of SCORE and MAP metrics) of static analysis technique for the problem of fault localization using character N-gram based information retrieval model are encouraging. Experimental results revealed that the median value for the SCORE metric for JBOSS and Apache dataset is 99.03% and 93.70% respectively. We observed that for 16.16% of the bug reports in the JBOSS dataset and for 10.67% of the bug reports in the Apache dataset, the average precision value (computed at all recall levels) is between 0.9 and 1.0. We observed that the percentage of bug reports in the JBOSS and APACHE dataset in which at-least one of the modified file(s) is retrieved as rank 1 is 40% and 26% respectively.

Appendix A

Dataset Version details

A.1 Selected Source-code Versions and Distribution of Issue Reports among them.

A.1.1 JBOSS Dataset

S.No.	Version	No.of Java Files	No. of Issues Reported(Out of 576)
1	JBossAS-4.0.2RC1	6212	10
2	JBossAS-4.0.3RC1	7977	6
3	JBossAS-4.0.3RC2	8080	8
4	JBossAS-4.0.0 Final	6398	27
5	JBossAS-4.0.1 Final	6729	72
6	JBossAS-4.0.1 SP1	6755	27
7	JBossAS-4.0.2 Final	6307	53
8	JBossAS-4.0.3 Final	7996	24
9	JBossAS-4.0.3 SP1	7999	69
10	JBossAS-4.0.4.CR2	9042	12
11	JBossAS-4.0.4.GA	9202	65
12	JBossAS-4.0.4RC1	8597	19
13	JBossAS-4.0.5.CR1	8490	12
14	JBossAS-4.0.5.GA	8730	48
15	JBossAS-4.2.0.CR1	8201	5
16	JBossAS-4.2.0.CR2	8315	6
17	JBossAS-4.2.0.GA	8322	17
18	JBossAS-4.2.1.GA	8366	16

19	JBossAS-4.2.2.GA	8524	42
20	JBossAS-4.2.3.GA	8712	38

A.1.2 Apache(Geronimo Project) Dataset

S.No.	Version	No.of Java Files	No. of Issues Reported(Out of 736)
1	1.0	1911	51
2	1.0-M1	1428	4
3	1.0-M2	1730	19
4	1.0-M3	1877	54
5	1.0-M4	1850	31
6	1.0-M5	2318	54
7	1.1	1711	81
8	1.1.1	1718	13
9	1.2	1923	58
10	2.0	2487	25
11	2.0-M1	2486	7
12	2.0-M2	2492	8
13	2.0-M3	1985	17
14	2.0-M5	2187	37
15	2.0-M6	2472	25
16	2.0.1	2486	16
17	2.0.2	2552	24
18	2.1	2563	48
19	2.1.1	2576	20
20	2.1.2	2598	5
21	2.1.3	2630	11
22	2.1.4	2616	24
23	2.1.5	2641	8
24	2.1.6	2641	5
25	2.1.7	2655	2
26	2.2	3149	68
27	2.2.1	3182	10
28	3.0-M1	2598	11

References

[1] Ing-Xiang Chen, Cheng-Zen Yang, Ting-Kun Lu, and Hojun Jaygarl. Implicit Social Network Model for Predicting and Tracking the Location of Faults. 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2008): 136-143,Turku, Finland, Jul. 28 -Aug. 1, 2008.

[2] Lukins Stacy K., Kraft Nicholas A., and Etzkorn Letha H. Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation. Proceedings of the 2008 15th Working Conference on Reverse Engineering, 155–164, 2008.

[3] Shivani Rao and Avinash Kak. Retrieval from Software Libraries for Bug Localization: A Comparative Study of Generic and Composite Text Models. 8th working conference on Mining software repositories (MSR), 2011.

[4] Fry Zachary P. Fault Localization Using Textual Similarities. MCS Thesis, University of Virginia, 2012.

[5] Adrian Bachmann and Abraham Bernstein. Data Retrieval, Processing and Linking for Software Process Data Analysis. Technical report, University of Zurich, Department of Informatics, 2009.

[6] Cleve Holger, Andreas Zeller. Locating causes of program failures. Proceedings of the 27th international conference on Software engineering(ICSE '05), 342–351, 2005.

[7] Jones James A., Mary Jean Harrold., Empirical evaluation of the tarantula automatic fault-localization technique. Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering(ASE '05), 273–282, 2005.

[8] Voorhees Ellen M., Variations in relevance judgments and the measurement of retrieval effectiveness. Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval(SIGIR '98), Melbourne, Australia,315–323, 1998.

[9] Antoniol G., Canfora G., Casazza G., de Lucia A. Identifying the Starting Impact Set of a Maintenance Request: A Case Study, Proceedings of the Conference on Software Maintenance and Reengineering(CSMR '00), 227, 2000.

[10] Canfora Gerardo and Cerulo Luigi. Impact Analysis by Mining Software and Change Request Repositories, Proceedings of the 11th IEEE International Software Metrics Symposium, 29, 2005.

[11] Marcus Andrian, Rajlich Vaclav, Buchta Joseph, Petrenko Maksym, and Sergeyev Andrey. Static Techniques for Concept Location in Object-Oriented Code, Proceedings of the 13th International Workshop on Program Comprehension, 33–42, 2005.

[12] Nichols Brent D., Augmented bug localization using past bug information. Proceedings of the 48th Annual Southeast Regional Conference(ACM SE '10), 61:1–61:6, 2010.

[13] Moin Amir and Khansari Mohammad. Bug Localization Using Revision Log Analysis and Open Bug Repository Text Categorization. Open Source Software: New Horizons,IFIP Advances in Information and Communication Technology, Agerfalk Par, Boldyreff Cornelia, Gonzalez-Barahona, Jesus, Madey Gregory, Noll John, Springer Boston, 188-199,319, 2010.

[14] Poshyvanyk Denys, Marcus Andrian, Rajlich Vaclav, Gueheneuc Yann-Gael, Antoniol, Giuliano. Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification. Proceedings of the 14th IEEE International Conference on Program Comprehension, 137–148, 2006.

[15] Pankaj Jalote. An Integrated Approach to Software Engineering. Third Edition, Narosa Publishing House, 2005.

[16] Bennat P. Lientz AND E. Buton Swanson. Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations. Addison-Wesley, Reading, MA, 1980.

[17] Anvik John. Automating bug report assignment. Proceedings of the 28th international conference on Software engineering (ICSE '06), Shanghai, China, 937–940, 2006.

[18] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight bug localization with AMPLE. Proceedings of the sixth international symposium on Automated analysis-driven debugging (AADEBUG'05), Monterey, California, USA, 99–104, 2005.

[19] Hiralal Agrawal, Joseph R. Horgan, Saul London, and W. Eric Wong. Fault Localization using Execution Slices and Dataflow Tests. Proceedings Sixth International Symposium on Software Reliability Engineering, Toulouse, France, 143–151, 1995.

[20] David Hovemeyer and William Pugh. Finding bugs is easy. SIGPLAN Not., 39:92–106, New York, NY, USA, 2004.

[21] Bug Life Cycle & Guidelines, http://www.exforsys.com/tutorials/testing/bug-life-cycle-guidelines.html.

[22] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? Proceedings of the 28th international conference on Software engineering (ICSE '06), Shanghai, China, 361–370, 2006.

[23] Christopher D. Manning, Prabhakar Raghavan and Hinrich Schtze. Introduction to Information Retrieval. Cambridge University Press, 2008.

[24] Joachim Wegener, Klaus Grimm, Matthias Grochtmann, Harmen Sthamer, and Bryan Jones, Systematic testing of real-time systems. In Proceedings of the 4th European Conference on Software Testing, Analysis and Review (EuroSTAR 1996), Amsterdam, Netherlands, 1996.

[25] Kai-hui Chang, V. Bertacco, and I. L. Markov. Simulation-based bug trace minimization with BMC-based refinement. Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design (ICCAD '05), San Jose, CA, 1045–1051, 2005.