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

Sangeeta Lal and Ashish Sureka
Indraprastha Institute of Information Technology (IIIT-Delhi), India
{sangeeta, ashish}@iiitd.ac.in

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. Fault localization is a routine task in software maintenance (corrective maintenance). Due to the increasing size and complexity of current software applications, automated solutions for fault localization can significantly reduce human effort and software maintenance cost.

We present a technique (which falls into the class of static techniques for bug localization) for fault localization based on a character n-gram based Information Retrieval (IR) model. We frame the problem of bug localization as a relevant document search task for a given query and investigate the application of character-level n-gram based textual features derived from bug reports and source-code file attributes. We implement the proposed IR model and evaluate its performance on dataset downloaded from two popular open-source projects (JBoss and Apache).

We conduct a series of experiments to validate our hypothesis and present 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 fault 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 observe 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.

Categories and Subject Descriptors
H.4 [Information Systems Applications]: Miscellaneous;
D.2.9 [Software Engineering]: Management—Productivity, Programming teams;
H.3.1 [Content Analysis and

Indexing]: [Dictionaries, Indexing methods]; H.3.3 [Information Search and Retrieval]: [Retrieval Models]

General Terms
Algorithms, Experimentation, Measurement

Keywords
Bug Localization, Mining Software Repositories (MSR), Information Retrieval (IR), Automated Software Engineering (ASE)

1. INTRODUCTION
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. Fault localization is a routine task in software maintenance (corrective maintenance). Due to the increasing size and complexity of current software applications, automated solutions for fault localization can significantly reduce human effort and software maintenance cost.

The aim of the research study presented in this paper 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 propose an IR based approach (framing the problem as a search problem: retrieve relevant documents in a document collection for a given query) 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-level n-gram based analysis. 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, re, Sof, oft, ftw, twa, war and are respectively. The paper by Peng et al. lists the advantages (such as natural-language independence, robustness towards noisy data and effective handling of domain specific term variations) of character-level n-gram language models for language independent text categorization tasks [11].
Automated solutions for the software engineering task of fault 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. [13] perform a survey and comparison of few IR based models for fault localization. We review traditional techniques and present some of the main points relevant to this work in Table 1. As displayed in Table 1, we arrange previous work in a chronological order and mention the key features of previous methods. We study 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. perform experiments on LEDA C++ Class library and work at Class level granularity whereas Lukins et al. perform experiments on Rhino, Eclipse and Mozilla dataset and work at Method level granularity (refer to Table 1).

The approach presented in this paper departs from previous approaches for bug localization as we make use of low-level character-level representation whereas all the previous approaches are based on word-level analysis. We hypothesize that the inherent advantages of character-level analysis is suitable to the task of automating 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 performance of character-level representation for the task of fault localization is an open research question. The study presented in this paper attempts to advance the state-of-the-art on IR based fault 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 filename and path-name) using a character-level n-gram model for text 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 fault 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 experiment with different configuration parameters (such as weights for title-filename, title-pathname, description-filename and description-pathname comparisons and length normalization) and present results throwing light on the performance of the approach as well as insights

<table>
<thead>
<tr>
<th>Study</th>
<th>Bug Reports</th>
<th>Source Code</th>
<th>Granularity</th>
<th>IR Model</th>
<th>Case Study</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Antoniol, 2000 [1]</td>
<td>Maintenance requests (excerpt from the change log files)</td>
<td>source code or higher level documents</td>
<td>Class level</td>
<td>vector space model and stochastic language model</td>
</tr>
<tr>
<td>3</td>
<td>Marcus, 2005 [8]</td>
<td>Specific keywords in change requests</td>
<td>Source code</td>
<td>Class level</td>
<td>latent semantic indexing, dependency search</td>
</tr>
<tr>
<td>4</td>
<td>Poshvyvanyk, 2006 [12]</td>
<td>Bug Description</td>
<td>Source code elements</td>
<td>Class Level and Method Level</td>
<td>SBP Ranking and LSI Based</td>
</tr>
<tr>
<td>5</td>
<td>Lukins, 2008 [6]</td>
<td>Manual extraction of keywords from Title and Description</td>
<td>Comments, Identifiers and String-Literals</td>
<td>Method Level</td>
<td>LDA Based</td>
</tr>
<tr>
<td>6</td>
<td>Fry, 2010 [4]</td>
<td>Title, Description, Stack Trace, Operating System</td>
<td>Class Name, Method Names, Method Bodies, Comments, Literals</td>
<td>File Level</td>
<td>Word-based cosine similarity vector comparison</td>
</tr>
<tr>
<td>7</td>
<td>Moin, 2010 [9]</td>
<td>Title &amp; description</td>
<td>Source File Hierarchy</td>
<td>Revision Path Level</td>
<td>SVM Classifier</td>
</tr>
<tr>
<td>8</td>
<td>Nichols, 2010 [10]</td>
<td>Bug description of old and new bugs</td>
<td>Identifier names, string literals</td>
<td>Method Level</td>
<td>LSI Based</td>
</tr>
</tbody>
</table>

Table 1: Previous approaches for bug localization using static analysis and information retrieval models. The papers are listed in a chronological order and the traditional methods are classified into various dimensions.
2. SOLUTION APPROACH

Figure 1 presents the high level architecture of the proposed solution. The system consists of three components: (1) Character N-gram Based Feature Extractor (2) Similarity Computation and (3) Rank Generator. The feature extractor module extracts all the character N-grams (of length 3 to 10) from bug reports (title and description) and source code files (file-name and file-path). The similarity computation module calculates the final score between bug report and source code described in the following Section. In contrast to word level n-gram textual features, character level n-gram feature extraction provides some unique advantages which aligns with the task of bug localization. Each advantage (in the following list) is supported by real world examples taken (derived from manual analysis and visual inspections) from JBoss and Apache bug reports demonstrating effectiveness of proposed solution in bug localization domain. Table 2 presents examples demonstrating the advantages of the character n-gram based analysis over word-based analysis.

1. Ability to match concepts present in source code and string literals: Bug reports frequently contain source code segments and system error messages which is not natural language text. Consider bugid JBAS-4649 which modifies the file "JRMPInvokerProxHyHA". 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.

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 such concept matches without requiring a word stemmer. Some more examples derived from the dataset are: (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 bug JBAS-1552 which consists of a misspelled word "Management". 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", "anaa", "inent", "Manag" etc and hence a character n-gram based algorithm will be able to give a good similarity score between these two words (without requiring a language-specific spell-checker). Few more motivating examples derived from evaluation dataset are: (behaviour, behavior), (consist, consist), (related, related), (Relation, Relation), (check, check), (possible, possible), (Become, Because), (Management, Management), (Principal, Principal), (Period, Period).

4. Ability to match words and their short-forms: Bug reporters frequently use short-forms and abbreviation in 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 in-

Figure 1: High-level solution architecture depicting the text processing pipeline, various modules and their connections on the predictive power of different variables in the proposed textual similarity function.
Table 2: 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

serting emphasis "Scoped..., [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 preprocessing) tool. In contrast, character level analysis techniques are able to detect concept matching because of partial matching.

\[
SIM(U, V) = \sum_{u \in U} \sum_{v \in V} Match(u, v) \times \text{Length}(u) / |U| \times |V| 
\]  

\[
Match(u, v) = \begin{cases} 
1 & \text{if } u = v \\
0 & \text{Otherwise} 
\end{cases} 
\]

\[
|U| = \sqrt{f_2^2 + f_2^2 + \ldots + f_n^2} 
\]

\[
SIM_{SCORE}(BR, F) = W_1 \times SIM(T - F) + W_2 \times SIM(T - P) + W_3 \times SIM(D - F) + W_4 \times SIM(D - P) 
\]
Table 3: Details regarding the experimental dataset from JBoss and Apache projects

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

Figure 2: Distribution of the number of files modified in JBOSS dataset

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 $f_i$ represents the frequency of an n-gram ($u_i$) in the bag of character n-grams. $SIM_{SCORE}(BR,F)$ represents the similarity score between a bug report and the source-code file. As shown in Equation 4, the overall similarity is computed as a weighted average of the similarity between title and description of the bug report with the path-name and file-name of the source-code file. $T$ represents title of the bug report, $D$ represents description of bug report, $F$ represents source-code file-name and $P$ represents file path-name. $SIM(T - F)$ represents character n-gram textual similarity between the title (bug reports) and file-name (source code file) using Equation 1. $W_1, W_2, W_3$ and $W_4$ are weights (tuning parameters to increase or decrease the importance of each of the four similarity comparisons) such that their sum is equal to 1.

3. Evaluation Metrics

We measure 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 fault localization using Information Retrieval Based techniques [3][4][5].

3.1 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 fault localization. Accuracy and SCORE are directly proportional. A 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 fault resides in 20% (200 files in a search space of 1000 files) of the document (file) collection. Mathematical formulation of SCORE metric is presented in equation 5, where 'm' is number of files that a bug modifies, 'N' is the total number of files present in the search space, and f<sub>i</sub><sup>Rank</sup> is the rank assigned to i<sup>th</sup> file. SCORE is a standard and common evaluation metrics for measuring the effectiveness of bug-localization approaches [5][4][5].

\[
SCORE = \left(1 - \frac{\max(f_1^{Rank}, f_2^{Rank}, ..., f_m^{Rank})}{N}\right) \times 100
\]

3.2 MAP

Mean Average Precision or MAP (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 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. Average precision consists of computing the precision of the system at the rank of every relevant document retrieved. MAP is a well known metric to measure retrieval performance for IR systems [13][7].

Equation 6 presents the general formula for AP wherein \(P_i\) denotes the precision at i<sup>th</sup> 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 7. MAP is calculated by taking the mean of AP values for all the bug reports in the dataset.

\[
AP = \frac{P_1 + P_2 + ... + P_M}{M}
\]

\[
AP = \frac{1}{2} + \frac{2}{4} + \frac{3}{7}
\]

4. EXPERIMENTAL DATASET

In order to validate our hypothesis, we perform an empirical evaluation on dataset downloaded from the issue tracking system of two popular open source projects: JBOSS<sup>1</sup> and APACHE GERONIMO<sup>2</sup>. The dataset is publicly available as a result of which the experiments performed in this work can be replicated for further improvements in the technique and enable comparison with other approaches. We conduct experiments on dataset belonging to two open source projects to remove any project-specific bias and investigate

<sup>1</sup>https://issues.jboss.org

<sup>2</sup>https://issues.apache.org/jira/browse/GERONIMO
the generalization power of the solution. Table 3 displays the details of the experimental dataset. We implemented a crawler using JIRA API\(^3\) and downloaded the relevant data (bug report metadata, title description, modified files etc). In JIRA, the subversion commits (revision id, files changes, log messages, timestamp, author id) are integrated to the issue tracker and hence explicit traceability links between the changed files and associated bug report is available.

Table 3 shows details regarding the experimental dataset downloaded from JBoss and Apache open source projects. We extract 569 bug reports from the JBoss project and 637 bug reports from the Apache project satisfying certain conditions. As shown in the Table 3, one of the conditions is that the issue report should be of type BUG (as the focus of the work is fault localization) and not other types of issues such as feature requests, task or sub-task. Furthermore we extract bug reports which are CLOSED and fixed. 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. 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 download bug reports belonging to specific versions (for which sufficient bug reports were available for experimental proposes) for JBoss and Apache (refer to Table 3 for details). We download the source code for various versions mentioned in the dataset as the fault localization is performed on the specific version on which the bug was reported. For example for JBoss dataset consists of a total of 20 versions. 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. For the Apache dataset 28 versions are downloaded (minimum size of search space is 1428, maximum size is 3182 and the average value is 2305).

5. EXPERIMENTAL RESULTS

Figure 3 and 4 displays 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. In addition to the 25\(^{th}\), 50\(^{th}\) and 75\(^{th}\) percentile, Figure 3 and 4 also reveals the degree of dispersion or spread in the SCORE values. We experiment with five tuning parameters (SIM-EQ, SIM-TF, SIM-TP, SIM-DF and SIM-DP as described in Table 4) 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). Figure 3

\[^3\text{http://docs.atlassian.com/software/jira/docs/api/latest/}\]
and 4 reveals the five-number descriptive statistics and distribution of the SCORE values for each of the five tuning parameters.

We observe 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 3). This indicates that for 50% of the bug reports in the JBOSS dataset, the proposed solution was able to decrease the search space by 99.03% (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.

The box-and-whisker diagram of Figure 3 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 3 and 4 indicates that for 25% of the bug reports (using SIM-EQ experimental setting) the value of SCORE metric is above 99.93% and 99.32% for JBOSS and APACHE respectively.

While Figure 3 and 4 presents descriptive statistics on the predictive accuracy of the method using SCORE metrics, Figure 5 and 6 presents the performance results using the average precision metrics. We measure the effectiveness of our approach using both the metrics to provide multiple perspectives. The histogram in Figure 5 and 6 displays 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 (ranks 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...
Figure 9: Scatter plot displaying the precision and recall value for each bug report (fixed and pre-defined values of Nr or Top K search results) [JBOSS Dataset]

Figure 10: Scatter plot displaying the precision and recall value for each bug report (fixed and pre-defined values of Nr or Top K search results) [APACHE]

Figure 11: 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

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 fault 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 con-
cepts present in programming languages and string literals) in comparison to world-level features. We perform experiments on publicly available real-world dataset from two popular open-source projects (JBoss and Apache) and investigate the overall effectiveness of the proposed model as well as impact of length normalization and each of the four predictor variables (title-filename, description-filename, title-path, and description-path). Empirical evidences reveal that length normalization improves average precision and recall and description-filename pair has more predictor power in contrast to the other three pairs. 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.

7. ACKNOWLEDGEMENT

This research was supported by The Department of Science and Technology (DST) FAST Grant awarded to Dr. Ashish Sureka.

8. REFERENCES


