Fault Localization Using Textual Similarities

Zachary P. Fry
zpf5a@virginia.edu

University of Virginia

Abstract. Maintenance is a dominant component of software cost, and localizing reported defects so that they can be fixed is a significant component of maintenance. The size and complexity of contemporary systems makes such fault localization difficult, however. In addition, defect reports often contain incomplete information provided by users who may be unfamiliar with the code base.

We propose a lightweight and scalable approach that leverages the natural language present in both defect reports and source code to identify portions of the program that are potentially related to the bug in question. Our technique is language independent and does not require test cases. The approach represents defect reports and source files as separate structured document forms and ranks source files of interest based on a document similarity metric that leverages inter-document relationships.

We evaluate the fault-localization accuracy of our method against both lightweight baseline techniques and also reported results from state-of-the-art tools. Similar tools have been evaluated using a metric that quantifies the reduction of the overall search space when trying to locate faults. Given information from actual bug reports and their real-world fixes, we utilize a similar metric to gauge the effectiveness of our tool.

In an empirical evaluation of 5345 historical defects from three real-world programs totaling 6.5 million lines of code, our approach reduced the number of files inspected per defect by 88%. Additionally, we qualitatively and quantitatively examine the utility of the textual and surface features used by our approach and their implications on conventional defect reporting.

1 Introduction

Maintenance tasks can account for up to 90% of the overall cost of software projects [8, 13]. A significant portion of that cost is incurred while dealing with software defects [24]. Large software projects typically use bug reporting systems that allow users to submit reports directly; this has been shown to improve overall software quality [2, 25]. User-submitted bug reports vary widely in utility [16]; reports go through *triage* to allow developers to focus on those reports that are most likely to lead to a resolution. We propose a system to make the maintenance process more efficient by reducing the cost of localizing faults.

Fault localization is the process of mapping a fault (i.e., observed erroneous behavior) back to the code that may have caused it. Performing fault localization is relatively time consuming [30] and thus costly. For this reason, many existing techniques attempt to facilitate this process. In general, such techniques rely on test cases [1, 18, 10, 26], model checking [5, 6], or remote monitoring [22, 23]. These approaches may not be directly applicable to user-submitted bug reports, since reports rarely provide a full test case or program trace [16].

For this thesis, we address the cost of fault localization for user-submitted bug reports. We present a lightweight approach that maps defect reports to source code locations. Our approach relies primarily on textual features of both source code and bug report descriptions, although it takes advantage of certain additional information (e.g., stack traces, version control histories) when they are available. Notably, it does not require test cases, execution traces, or remote sampling, all of which can potentially limit the applicability of other fault localization strategies.

Our approach is based on several underlying assumptions about the textual features of both source code and bug reports. With respect to code, we assume that developers choose identifier and file names that are representative of observable program behavior. For bug reports, we assume that the reporters use a vocabulary based on their observations of program behavior — a vocabulary that will thus be in some ways similar to developers' in spite of the fact that reporters may not have access to the source. Finally, we hypothesize that a bug report and a code location are more likely to pertain to the same fault if they are similar in terms of word usage; we formally describe our distance metric in Section 3.

The Thesis Statement of this work comprises two main claims:

– *We can construct a static fault localization model that is at least as accurate as existing run-time approaches without requiring program executions.*
– *Our model's success is explained by primarily natural language information and not by additional features, as measured by its performance reduction as human-chosen words are replaced by random words in its input.*

The main contributions of this research project are thus:

– A lightweight, language-independent model that statically measures similarity between defect reports and source files for the purpose of locating faults. This comparison is based on a structured textual analysis of the natural language in both documents.
– A large empirical evaluation of our technique including 5345 real-world defects from three large programs totaling 6.5 million lines of code — over an order of magnitude larger than the evaluations in previous work [10, 18, 26].

The structure of this document is as follows. In Section 2, we motivate our approach by presenting an example fault with its associated bug report and source code. Section 3 outlines our approach and formally defines how we measure the relative similarity between code and bug report text. Next, Section 4 presents a detailed empirical evaluation of our approach. Section 5 places our work in context. Finally, Section 6 concludes.

2 Motivating Example

In this section, we present an example bug report taken from the Eclipse project. This example illustrates the potential benefit of matching the natural language in a bug description with keywords from the source code for the purpose of identifying the bug's location.

User-submitted bug reports typically consist of a free-form textual description of the fault. When presented with such a bug report, it is up to the developer to derive and locate the cause of the undesirable behavior. This requires thorough familiarity with the code base; for large projects, an important part of the triage process is finding which developer is most likely to be able to resolve a given bug report [3]. There are significant differences among developers in terms of how quickly they can locate a given fault [30]. Our goal is to significantly narrow the source code search space that the developer needs to consider, thereby decreasing the software maintenance cost overall.

Consider the following defect report from the Eclipse project, Bug #91543, entitled "Exception when placing a breakpoint (double click on ruler)." The description is as follows:

```
With M6 and also with build I20050414-1107 i get the stacktrace
below now and then when wanting the place a breakpoint when double
clicking in the editor bar.  if i close the editor and reopen it
again it goes ok.

!MESSAGE Error within Debug UI:
!STACK 0
org.eclipse.jface.text.BadLocationException
      at
org.eclipse.jface.text.AbstractLineTracker.getLineInformation(
          AbstractLineTracker.java:251)
...
```

Initially, a developer might be inclined to inspect code implicated directly. In this case, one might check the *AbstractLineTracker* file and other files in the stack trace, or search the list of all files that reference a *BadLocationException*. Additionally, one might scan the files that were changed prior to either of the particular builds mentioned. Finally, based on basic searching one might uncover any of the following files: *Breakpoint.java*, *MethodBreakpointTypeChange.java*, *BreakpointsLocation.java*, and *TaskRulerAction.java* among hundreds of others. This example illustrates that the search space is large, even when a programmer uses the defect report's specific information.

In the actual patch for this bug, developers edited only two source files. *ToggleBreakpointAction.java* contained the majority of changes that addressed this defect report, with one minor change to a call-site in *RulerToggleBreakpointActionDelegate.java*. Some of the methods in those files include:

```
ToggleBreakpointAction(..., IVerticalRulerInfo rulerInfo)
ToggleBreakpointAction.reportException(Exception e)
```

```
    RulerToggleBreakpointActionDelegatecreateAction(ITextEditor editor,
       IVerticalRulerInfo rulerInfo)
```

The identifier names associated with these two files show clear language overlap with the report above. For example, even when only the report title and the method names are considered, key words such as *breakpoint*, *exception* and *ruler* occur in both sets. When examining the overall word similarity, the two files that were changed for the fix are among those files most similar to the text in the defect report. Using textual similarity not only avoids unrelated methods considered by traditional search techniques, but further limits the fault localization search space by trimming files with coincidental or narrow language overlap. Aggregating overall word similarity ensures that only documents with considerable and meaningful similarity are favored. We hypothesize that prioritizing the search space by ranking files of interest in this manner can greatly facilitate fault localization — using only the defect report in question and the current source code. In the next section, we present a model to take advantage of this intuition.

3 Methodology

Our goal is to reduce the cost of the fault localization process. Given only a bug report describing a fault and the project source code, we desire an ordered list of source files that are likely to contain the cause of that fault. To do this, we map both defect reports and source code text to respective structured document intermediate representations based on the conceptual parts of each unstructured document. We then build a model based on relationships between subparts of each document, and rank each source file accordingly.

3.1 Structured Document Representation

Both defect reports and source files are represented as structured documents. For our purposes, a structured document consists primarily of several term frequency vectors. A *term frequency vector* is a mapping from terms (i.e., words) to the frequencies with which they appear in a given document. Structured documents may also contain significant categorical or non-natural language data, such as stack traces, which we model as ordered sequences of strings.

We map defect reports to our intermediate form directly. Defect reports are structured natural language files containing multiple parts, such as title, description, optional stack trace, project versions affected, and operating systems affected [16]. We first focus on the natural language title and description. We break the text into a list of terms by splitting on whitespace and punctuation and converting each term to all lowercase characters; we then construct term frequency vectors from the resulting multiset of words. Additionally, we also parse and record categorical data, such as the operating system and software version, representing them as discrete features in the structured document. Finally, we parse any stack traces into ordered sequences of strings.

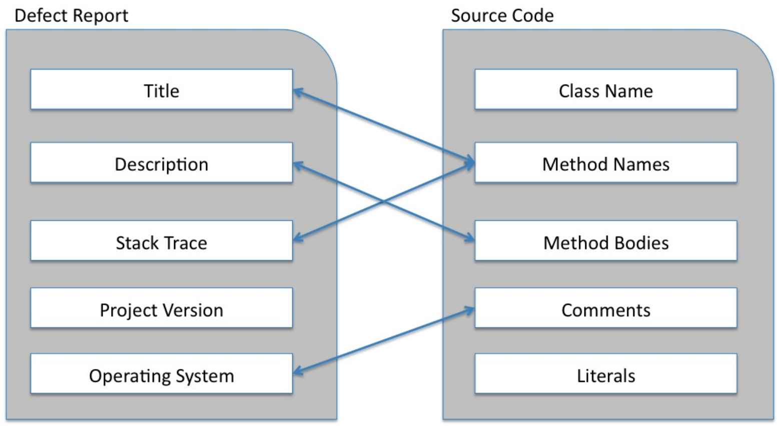

Fig. 1. Example Representation of our structured document comparison technique.

Source code, which is not expressly written in natural language, is handled similarly, but with a few extensions that have been shown to be effective in previous work involving textual analysis [28]. We obtain an initial list of terms by splitting on whitespace and punctuation. However, we obtain further terms by taking advantage of paradigms such as Hungarian notation, camel case capitalization, and the use of underscores to separate terms in a single string. For example, given the string "nextAvailableToken" we increment frequencies for the following terms: "next", "available", "token", and "nextAvailableToken". Source files are also structured and can also be decomposed into substructures. Substructures include method signatures, method bodies, comments, and string literals, among others. In addition to the overall term frequency vector for the entire file, each substructure is processed separately into its own term frequency vector. Thus our intermediate representation for a source file will include an term frequency vector, a term frequency vector for words in comments, one for words in method bodies, and so on.

Intuitively, the similarity between a defect report and a source file is built up from the similarities between their intermediate representations (i.e., their term frequency vectors). We wish to empirically determine which vectors are the most predictive of fault localization when the two structured documents in question are compared. For instance, previous work has shown that defect titles are highly significant when searching for duplicate reports [17]; we hypothesize that they may be similarly significant when attempting to locate defects. Section 3.3 goes into more detail on this subject.

3.2 Textual Document Similarity

Intuitively, two documents are similar when they have a large fraction of their terms in common. The formal basis of the main similarity metric we employ is the standard cosine similarity between two vectors v_1 and v_2 (here $v_1 \bullet v_2$ denotes the dot product):

$$\cos(\theta) = \frac{v_1 \bullet v_2}{|v_1| \times |v_2|}$$

The more terms the two corresponding documents share, the closer the vectors are to collinear and, we assume, the more related concepts they both describe.

In practice, some terms are more indicative of underlying similarity than others. For example, terms such as "int", "class" or "the" may occur frequently in two unrelated documents. We wish to limit the impact of such terms on our similarity metric. However, since we desire a language-independent approach, rather than hand-crafting an *a priori* stop-list of common words to discount, we will derive that information from the set of available defect reports and source code. Intuitively, two documents that share a rarer term, such as "VerticalRuler" should be measured more similar than two documents that share a common term such as "int".

To formalize this intuition we use the TF-IDF measure [19], which is common in information retrieval tasks. We want to measure how strongly any given term describes a document with respect to a set of context documents. Given a document d and a term t, the TF-IDF weighting $\mathsf{weight}(d,t)$ is high if t occurs rarely in other documents, but relatively frequently in d. Conversely, a low weight corresponds to a term that is frequent globally and/or relatively infrequent in d. The weight for a document d and term t is computed as follows:

$$\mathsf{tf}(t,d) = \frac{\#\text{ occurrences of } t \text{ in } d}{\text{size of } d} \qquad \mathsf{idf}(t) = \frac{\#\text{ of documents}}{\#\text{ of docs that contain } t}$$

$$\mathsf{weight}(t,d) = \mathsf{tf}(t,d) \times \mathsf{idf}(t)$$

Here tf is "term frequency" and idf is the "inverse document frequency". With the background formalisms thus described, we now explain how we combine them to aid fault localization.

3.3 Our Technique

We combine portions of the cosine similarity metric and the TF-IDF weighting to form an overall similarity metric:

$$\mathsf{similarity}(v_1, v_2) = \sum_{t \in\ v_1 \cap v_2} v_1[t] \times v_2[t] \times \mathsf{idf}(t)$$

For each term contained in both documents, we multiply the product of its frequencies in both documents by that term's idf weight. The aggregate sum over all words' values then serves as the similarity measure for those two documents. The major distinction between this metric and standard approaches is that we do *not* normalize for the size of the documents. While normalization is natural in many information retrieval tasks, we claim that the special structure of source code and the fault localization task make it undesirable here. For example, consider a defect report that mentions the term “VerticalRuler” in a project where the only source code mention of that term occurs inside one very large source file. In such a case, we would like to report the single source file as very similar to the defect report. However, if the file’s size were normalized, it would appear to be less similar to the defect report than smaller files that share more common terms (e.g., “mouse”).

In general, other works apply size-based normalization when large documents increase false positive rates or otherwise degrade the accuracy of a given method. We claim that the loss of precision associated with normalization outweighed the benefits it provided. This is in line with previous claims [28] that traditional information retrieval search techniques used for documents do not map perfectly to code- based textual analysis.

While the above technique is intended for use with two term frequency vectors, we require certain adaptations for other types of structured data. Categorical data, such as operating system flavors or program versions, are treated as a vector with a single term and the metric can be used in the standard fashion. Stack trace vectors — sequences of strings representing method names — can be compared as word vectors by using the positional index of a method in the call trace name as its frequency.

Given a defect report D and set of source files $f_1 \ldots f_n$, our goal is to produce a rank-ordered list of the files, weighted such that files likely to contain the defect are at the top. Human developers then inspect the files on the ranked list in order until the fault has been localized. The rank of a file f_i is given as follows:

$$\mathsf{rank}(D, f_i) = \sum_{v_j \in D} \sum_{v_k \in f_i} c_{jk} \mathsf{similarity}(v_j, v_k)$$

where v_j ranges over all of the term frequency vectors in the defect report’s intermediate representation, v_k ranges over all of the term frequency vectors in the source file’s intermediate representation, and each c_{jk} is a weighting constant for that particular vector pair. The c_{jk} constants are the formal model: a high value indicates that similarity in the associated pair of sub-substructures (e.g., defect report title paired with source code comments) is relevant to fault localization.

One approach would be to use machine learning or regression to determine the values for the c_{jk} weightings. The size of our dataset, which includes tens of millions of datapoints, precludes such a direct approach, however. Attempts to apply linear regression to the dataset exhausted memory on a 36 GB, 64-bit machine. We instead use several common statistics as a starting point for a parameter space optimization to obtain a precise model (see Section 4.2.

4 Evaluation

We conducted two main experiments to evaluate our approach. The first directly compares the accuracy of our technique to other lightweight baselines at file-level localization and indirectly compares to state-of-the-art techniques. The

Program	Defects Used	Files Used	Lines of Code Used	Language(s)	Avg. report length (lines)	Avg. report title (words)
Eclipse	1,272	23,601	3,476,794	Java	172.535	8.642
Mozilla	3,033	14,651	2,262,877	Java, C++	316.811	9.428
OpenOffice	1,040	9,992	815,473	Java, C++	60.547	5.623
Total	5,365	48,244	6,555,144	-	-	-

Fig. 2. Subject programs used in our evaluations. “Defects” counts reports that could be linked to a particular set of changes. “Files” counts retrieved source files in the project branch, including those not involved in defect reports. “Lines of Code” measures the size of those source files, while “Languages” lists their programming languages. The last two columns measure aspects of the defect reports used.

second experiment quantitatively verifies our hypothesis that fault localization via textual analysis depends significantly on human word choice.

4.1 Subject applications and defects

The experiments were conducted using 3 large, mature open source programs and 5345 total bug reports, shown in Figure 2.

We chose these projects for several reasons. First, they are relatively indicative of substantial, long-term real-world development in terms of size (6.5 million lines of code total) and maturity (each is 8 to 11 years old). Additionally, each project has both bug report and source code repositories.

For each program, we obtained the subset of the available defect reports for which we could establish a definitive link between the report and a corresponding set of changes to source files. We thus restricted attention to those defect reports that were mentioned by number in source control log messages. We additionally restricted attention to reports of actual faults, omitting feature requests and other invalid reports filed using the bug report system. Also, we only considered defects for which all corresponding changes took place in source files in the main branch of each project (e.g., omitting changes to minor branches, testing branches, or data files).[1] Finally, we excluded files or reports that could not be processed (e.g., from CVS or parsing errors).

4.2 Model Coefficients

Our first step is to build a model relating similarity comparisons between defect report and source code structures to fault localization. In the terminology of Section 3.3, this involves determining values for the 28 distinct c_{jk} weights.

To build such a model we first performed an analysis of variance (ANOVA) on a subset of the data to estimate the predictive power of each possible document comparison. For each defect report we consider all of the files that were eventually fixed by the developers and also 150 files, chosen at random, that were

[1] Eclipse’s `/cvsroot/eclipse`; Mozilla’s `/cvsroot`; OpenOffice’s `/trunk`.

Report Substructure	Code Substructure	Relative Weight (c_{jk}) in Model
Report title	Method bodies	23.06
Report body	Method signatures	21.48
Report title	Comments	10.53
Report body	Class name	9.46
Report body	Comments	7.89
Stack trace	Class name	7.79
Report body	Method bodies	5.72
Component	Method bodies	4.30
Operating system	Comments	3.48
Component	Comments	3.03
Product	String literals	1.94
Report title	Method signatures	1.32

Fig. 3. The coefficients associated with our model. Since we are interested in relative rankings rather than an absolute value, the coefficients have been normalized so that their aggregate sum is 100.

not.[2] We pair each such file f_i with the original defect report D to produce one datapoint. Each datapoint has multiple associated features (i.e., the explanatory variables): there is one feature for each each of the 28 $\langle v_j, v_k \rangle$ vector pairs, with the measured similarity serving as the feature value. The response variable for a given datapoint is set to 1 if the file was modified by developers and 0 otherwise.

The ANOVA measures the ratio of the variance explained by each feature (i.e., each $\langle v_j, v_k \rangle$ similarity) over the variance not explained. We use this ratio as a starting point for c_{jk}. These values may not be optimal because our final model goal is to rank order the final and not to minimize the error with the artificial 0 and 1 response variables.

Our second step was to perform a principle component analysis (PCA) to determine the number of components that were relevant to the task of detecting the location of a fault in source code. Given our 28 possible document substructure comparisons, this analysis showed that a combination of 12 accounted for more than 99% of the overall variance in the data. The final c_{jk} values obtained via a gradient ascent parameter space optimization. In each iteration, the best model available was compared to similar models, each constructed by increasing or decreasing the value of a single c_{jk} by 10%. The comparison was conducted using the *score* metric detailed in Section 4.3. We terminated the process when the improvement between one iteration and the next was less than 0,01%; this took 5 iterations. We used the final c_{jk} values as our formal model. Figure 3 shows the final document substructures selected for the model and their respective coefficients.

The report title and body, as well as the method bodies and comments, are involved in many of the most useful relationships in our fault localization model.

[2] The inclusion of 150 files was chosen to be as large as possible while allowing the problem to be tractable on available hardware.

Test Set of Defects	Defects Used	Our approach	Stack trace Baseline	Code churn Baseline
OpenOffice only	1040	67.918%	57.979%	72.755%
Eclipse only	1272	86.909%	56.295%	73.131%
Mozilla only	3033	92.159%	50.152%	93.860%
Stack traces only	325	86.175%	65.060%	76.442%
Complete set	**5345**	**88.193%**	**53.137%**	**84.820%**

Fig. 4. *Score* values for selected techniques. The "Test Set" column lists examined subsets of the 5345 defects from three programs. "Our Approach" measures the *score* obtained by our technique. The "Stack trace" baseline favors files mentioned in user-provided stack traces, and the "Code churn" baseline favors frequently-changed files.

With respect to defect reports, the titles and bodies contain the majority of the natural language information chosen by the reporter and, as such, are more helpful than extraneous categorical data and stack traces. Comparatively, we believe that code comments are effective when matching terms from bug reports because they are written explicitly in natural language and often encapsulate code specifications in a manner complementary to the language inherent in the code's identifiers. Method bodies contain most of the text associated within code files and thus also serve as effective predictors. Notably, more obscure categorical information and string literals found in code were less useful to the model.

4.3 Experiment 1 — Ability to localize faults

Our first experiment measures the accuracy of our technique when localizing faults. We compare two versions of our technique against two baseline approaches directly. We also indirectly compare against the published results of three state-of-the-art tools using a common metric.

Score metric We adopt the *score* metric for measuring the accuracy of a fault localization technique. The *score* metric is commonly used in fault localization research [10, 18, 26]. A *score* value represents how much of a given code base one would **not** have to examine to find a fault. For example, a ranking for an OpenOffice defect report that requires the user to inspect 2,000 of the 9992 files before finding the right file has a *score* of $(9992 - 2000)/9992 = 80\%$. Higher *score* values indicate better accuracy. We apply the *score* metric at the file level of granularity. We report the average *score* over all defects available.

Figure 4 shows the results. A lower baseline of 50% represents inspecting files in random order. Our approach outperforms all baselines over the entire test set (highlighted in boldface in Figure 4) and is generally better than other approaches in most subsets. The "Stack traces only" subset includes all defect reports that featured stack traces. Note that of these 5345 defect reports, only 325 (6%) contained stack traces.

Over the entire test set, we outperform the stack trace and code churn baselines by around 35% and 3% respectively. While the performance gain over a stack trace baseline is immediate, the lower performance gain over code churn requires more of an explanation. First, note that code churn is particularly effective on Mozilla defect reports: if they are excluded, our approach outperforms it by 10%. Second, since code churn obtains a *score* of over 84% overall, only a 16-point *score* increase is possible. In that regard, our 4-point increase constitutes 20% of the remaining room for improvement. Finally, on large projects, even small gains are significant: a 3% *score* increase prevents an average of 1,929 source files (or 262,205 lines of code) from being considered during the fault localization search for an average subject bug.

Our technique performed most poorly on OpenOffice defects: if only Eclipse and Mozilla are considered, our performance is 90%. This can be explained by a particular quirk of the OpenOffice project: their bug reports contain less-descriptive titles, thus reducing our primary source of textual similarity (see Section 3.1).

The results presented in Figure 4 show that our tool outperforms lightweight baselines. We also suggest that our technique may perform better than more heavyweight techniques. Several state-of-the-art fault localization techniques report accuracy values for their tools in terms of the distribution of subject faults over the scale of possible *score* measures. For comparison purposes we use a weighted average of each *score* interval to calculate an overall accuracy measure for each approach. The tools of Jones *et al.* [18], Cleve *et al.* [10], and Renieris *et al.* [26] achieved aggregate *score* measures of 77.797%, 63.415%, and 56% respectively. The largest of these projects evaluated on 132 defects over seven files containing at most 560 lines of code each. While these results are measured on different test sets and are therefore not directly comparable, we note that our technique obtains a *score* result 9 points higher than previous work and is evaluated on an order-of-magnitude more defects and files.

Finally, our technique is lightweight in terms of execution time. Assuming code files are kept indexed as word vectors, our tool always runs in under 10 seconds per defect report and generally takes less than 1 second.

4.4 Experiment 2 — Evaluation of human word choice

Our second experiment tests our hypothesis that our *score* accuracy is mainly due to correctly extracting and comparing the natural language chosen by humans in defect reports and source files. We first demonstrate that our technique's accuracy is not dominated by other features, such as length, bug priority, or bug lifespan. Secondly we alter the natural language of the subject reports systematically, showing that performance degrades in a proportional manner.

We hypothesize that human-chosen natural language in defect reports and source code is a critical factor in our fault localization approach. We first discount several potentially-prominent other features in terms of predictive power with respect to the *score* accuracy of our technique. The features examined cover both defect reports and source code: the Flesch-Kincaid readability level of the

Document feature	Correlation with *score*
Average report length	0.15
Rate of commenting in edited source	0.15
Maximum report length	0.13
Number of duplicate reports	0.12
Reported priority	0.12
Bug lifespan	0.11
Report readability	0.07
Number of edited source files	0.07

Fig. 5. Pearson correlation between surface features and our technique's *score*.

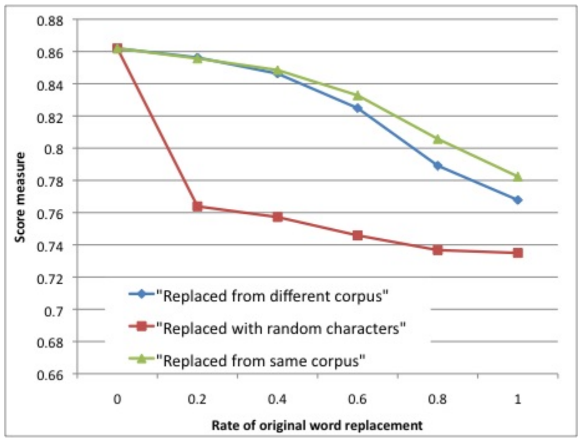

Fig. 6. The effect of replacing human-chosen words with various random words on our technique's *score* over all 5345 defects.

report in question [14], the assigned bug priority, the number of total reports for a bug when considering all duplicates, the maximum report length for a bug, the average report length for a bug, the overall lifespan of the bug from reported defect to reported patch, the number of source files edited as part of the patch, and the rate of commenting in the edited source code.

We calculated the Pearson correlation of all 5345 total bugs' *score* measures with these features. The correlations can be found in Figure 5. It is generally accepted that correlations below 0.3 are not statistically significant [15]. All observed correlations fell well within these bounds and therefore we conclude that these features do not significantly affect our model. However, of all correlations, report length and rate of commenting had the highest relative values. This supports our claim that natural language is key to our technique's success, since these features typically relate directly to the natural language present.

Next, we demonstrate that our model is greatly affected by the users' choice of language in defect reports and the developers' choice of language in source code. To evaluate this, we measure our *score* accuracy as more and more human-chosen words are replaced by random words. We used three different random techniques to replace human-chose words: replacing terms with words of comparable length from the same general set (e.g., the set of all report description words), replacing terms with words of comparable lengths from a different set (in this case, an English dictionary), and finally, replacing terms with strings of the same length made up of randomly selected characters (i.e., random noise). In each case, we altered the natural language in increments until the entire frequency vector had been changed, using the unaltered reports as a baseline. The results of this experiment can be found in Figure 6. Each datapoint represents the *score* of our algorithm running on the entire 5345-defect dataset with some fraction of each defect report's text altered.

As the natural language in defect reports is changed, and thus the useful information in the report is reduced, the performance of our technique degrades. The reduction in *score* is not strictly proportional, as is expected from the presence of common words and our use of the idf weighting: a few words account for much of the relevant document similarity in a given comparison and thus changing even a few of these important words to random characters degrades the performance of our tool significantly and immediately. However, when replacing the human-chosen words with terms from the same or different corpus, the performance degrades more slowly. Words from the same corpus are still related to the overall system and thus this result follows naturally. Comparatively, words from a different corpus, while not as distracting as random useless characters, aren't as related to the underlying natural language and thus degrade performance more than the previous technique. Ultimately, when all words in defect reports have been randomly replaced by other words from defect reports, the *score* of our technique drops to a level between the code churn and stack trace baselines. This supports our hypothesis that the natural language in reports and code is critical to our technique's *score* at fault localization.

4.5 Threats to validity

Although our experiments are designed to demonstrate that our technique performs well over a large number of defects and files, our results may not generalize to industrial practice. First, our benchmark programs may not be indicative. The programs we chose are all large, mature, open-source projects. While they span three individual domains, they may not generalize to all potential domains. Our results may not apply to younger, smaller projects, but we claim that fault localization becomes less interesting as the project shrinks (e.g., in the limit, fault localization is not a large concern for a project with only one or two source files). We view an evaluation on large datasets (e.g., ten times larger than previously-published evaluations [18, 10, 26]) as an advantage.

Bird *et al.* note that sampling bug reports for the purpose of experimentation may lead to biased results [7]. As a result, our technique may only be good at localizing certain types of faults (i.e., those that open source developers deign to mention in version control logs). Lacking a project with a linked version control and defect repository, we cannot mitigate this threat beyond our claim that manual inspection of the reports found the faults to be a relatively even cross-section of each project's repository over the history of that project (see Section 4.1).

Our code churn baseline may not be indicative because it relies on eight to ten years of version control information. For example, it may perform particularly well on the larger and older Mozilla project, correctly giving low rankings to the many files that have been stable for years. In practice, a development organization may not have such rich version history information, or such stable files may be manually excluded by developers.

Finally, when comparing our results with that of established fault localization techniques using the *score* metric, we may be forced to estimate previous results to compare the approaches as directly as possible. Previous publications have reported *score* value *distributions* over intervals from 0% to 100%. We propose to estimate based on a weighted average of the medians of each interval. Ideally we will discover that even when we estimate previous *score* values using the upper end of each range (i.e., giving each previous approach the maximum value it could possibly have had), our technique's *score* values will still be higher overall.

5 Related Work

Related research to our work falls into two main categories: prior work in fault localization, and prior work in reverse engineering.

5.1 Fault Localization

Ashok *et al.* propose a similar natural language search technique in which users can match an incoming report to previous reports, programmers and source code [4]. By comparison, our technique is more lightweight and focuses only on searching the code and the defect report.

Jones *et al.* developed Tarantula, a technique that performs fault localization based on the insight that statements executed often during failed test cases likely account for potential fault locations [18]. Their approach is quite effective when a rich, indicative test suite is available and can be run as part of the fault localization process. It thus requires the fault-inducing input but not any natural language defect report. By contrast, our approach is lightweight, does not require an indicative test suite or fault-inducing input, but does require a natural language defect report. Both approaches will yield comparable performance, and could even be used in tandem.

Cleve and Zeller localize faults by finding differences between correct and failing program execution states, limiting the scope of their search to only variables and values of interest to the fault in question [10]. Notably, they focus on those variable and values that are relevant to the failure and to those program execution points where transitions occur and those variables become causes of failure. Their approach is in a strong sense finer-grained than ours: while nothing prevents our technique from being applied at the level of methods instead of files, their technique can give very precise information such as "the transition to failure happened when x became 2." Our approach is lighter-weight and does not require that the program be run, but it does require defect reports.

Renieris and Rice use a "nearest neighbor" technique in their Whither tool to identify faults based on exposing differences in faulty and non-faulty runs that take very similar executions paths [26]. They assume a large number of correct runs (e.g., normal test cases) and one failing run. Their approach uses a distance criterion to select the correct run that is closest to the failing run and produces a report of "suspicious" parts of the program. By comparison, we chose to limit the programmatic information used by our technique to only that which was reported by users: we do not use test case runs but do need natural language.

Liblit *et al.* use Cooperative Bug Isolation, a statistical approach to isolate multiple bugs within a program given a deployed user base. By analyzing large amounts of collected execution data from real users, they can successfully differentiate between different causes of faults in failing software [23]. Their technique produces a ranked list of very specific fault localizations (e.g., "the fault occurs when $i > arrayLen$ on line 57"). In general, their technique can produce more precise results than ours, but it requires a set of deployed users and works best on those bugs experienced by many users. By contrast, we do not require that the program be runnable, much less deployed, and use only natural language defect report text.

Jalbert *et al.* [17] and Runeson *et al.* [27] have successfully detected duplicate bug reports by utilizing natural language processing techniques. We share with these techniques a common natural language architecture (e.g., frequency vectors, TF-IDF, etc.). We differ from these approaches by adapting the overall idea of document similarity to work across document formats (i.e., both structured defect reports and also program source code) and by tackling fault localization.

5.2 Reverse Engineering

Syst *et al.* have developed an interactive reverse engineering tool called Shimba that allows users to explore different states of Objects within Java systems. The tool relies on both static and dynamic runtime information and stresses visualization capabilities for the goal of program understanding. In contrast, our approach uses strictly static information and contains no interface capabilities. We instead focus specifically on fault localization while keeping the overall process as lightweight as possible for the purpose of scalability.

D'Ambros *et al.* propose several analyses for mining artifacts from software repositories to aid program understanding especially related to long term evolution [11]. Our work also uses latent semantic information, but we focus specifically on the task of fault localization with overall program understanding as an implicit secondary goal.

Li *et al.* have examined the problem of extracting information from structured documents in addition to categorizing that information [21]. They focus on user queries in particular, which is similar to the bug reports we study in this thesis. They also note that tailoring analyses to specific corpora is particularly helpful, which we confirm with the use of inverse document frequency for weighting individual terms.

Shepherd *et al.* focused on both proving that the natural language in source code is meaningful and also on attempting to extract those language artifacts in a meaningful and useful manner [28]. They studied natural language use in code for the purpose of developing a specialized code-search technique specifically focused on identifying distributed concepts throughout a system. Similarly, Lawrie *et al.* have examined the quality of source code identifiers in terms of code comprehension [20]. They show that insightful and carefully chosen natural language identifiers make for more understandable and maintainable code. We build upon such work by leveraging these facts in the domain of fault localization.

Work has been done to measure the quality of natural language choices made by developers [9, 12, 20, 29]. Additionally, some of this work looks at restructuring or refactoring natural language artifacts in an attempt to reverse engineer the original developers' intentions and aid program understanding. We assert that measuring the quality of natural language is orthogonal to the work we present in this thesis. We are more concerned with the ability of the natural language in both defect reports and source code to localize faults, regardless of the language's *quality.* While higher quality information may allow our tool to compare documents more accurately, our tool currently achieves higher accuracy than state-of-the-art techniques without accounting for the quality of the underlying natural language.

6 Conclusion

We present a lightweight, scalable technique for localizing faults based on document similarities. We hypothesize that human-chosen natural language present in both defect reports and source code can be compared to identify potential fault locations based on natural-language descriptions. Our technique is entirely static and is language independent.

An empirical evaluation shows that our technique not only performs better than several baseline approaches, but is comparable to the state-of-the-art techniques without requiring significant overhead or a runnable program and a test suite. We also demonstrated that the word choice in natural language artifacts was truly the dominant factor in our approach.

A large empirical evaluation of our program on 5345 historical defects from three real-world programs totaling 6.5 million lines of code showed that we can reduce the search space for finding a fault by over 88% on average. We believe that this approach has the potential to significantly decrease the cost of fault localization, and thus software maintenance overall.

References

1. H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault localization using execution slices and dataflow tests. In *Software Reliability Engineering*, pages 143–151, 1995.
2. J. Anvik, L. Hiew, and G. C. Murphy. Coping with an open bug repository. In *OOPSLA workshop on Eclipse technology eXchange*, pages 35–39, 2005.
3. J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *International Conference on Software Engineering*, pages 361–370, 2006.
4. B. Ashok, J. Joy, H. Liang, S. K. Rajamani, G. Srinivasa, and V. Vangala. Debugadvisor: a recommender system for debugging. In *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 373–382, New York, NY, USA, 2009. ACM.
5. T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. *SIGPLAN Not.*, 38(1):97–105, 2003.
6. T. Ball and S. K. Rajamani. The slam project: debugging system software via static analysis. In *Principles of programming languages*, pages 1–3, 2002.
7. C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. T. Devanbu. Fair and balanced?: bias in bug-fix datasets. In *ESEC/SIGSOFT FSE*, pages 121–130, 2009.
8. B. Boehm and V. R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, 2001.
9. B. Caprile and P. Tonella. Restructuring program identifier names. In *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, page 97, Washington, DC, USA, 2000. IEEE Computer Society.
10. H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 342–351, New York, NY, USA, 2005. ACM.
11. M. D'Ambros, H. Gall, M. Lanza, and M. Pingzer. *Software Evolution*, chapter Analysing Software Repositories to Understand Software Evolution, pages 37–67. Springer Berlin Heidelberg, 2008.
12. F. Deissenboeck and M. Pizka. Concise and consistent naming. *Software Quality Control*, 14(3):261–282, 2006.
13. L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.
14. R. F. Flesch. A new readability yardstick. *Journal of Applied Psychology*, 32:221–233, 1948.
15. L. L. Giventer. *Statistical Analysis in Public Administration.* Jones and Bartlett Publishers, 2007.
16. P. Hooimeijer and W. Weimer. Modeling bug report quality. In *Automated software engineering*, pages 34–43, 2007.
17. N. Jalbert and W. Weimer. Automated duplicate detection for bug tracking systems. In *Dependable Systems and Networks*, pages 52–61, 2008.
18. J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Automated Software Engineering*, pages 273–282, 2005.
19. K. S. Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28:11–21, 1972.

20. D. Lawrie, H. Feild, and D. Binkley. Quantifying identifier quality: an analysis of trends. *Empirical Softw. Engg.*, 12(4):359–388, 2007.
21. X. Li, Y.-Y. Wang, and A. Acero. Extracting structured information from user queries with semi-supervised conditional random fields. In *SIGIR '09: Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, pages 572–579, New York, NY, USA, 2009. ACM.
22. B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Programming Language Design and Implementation*, June 9–11 2003.
23. B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Programming Language Design and Implementation*, pages 15–26, 2005.
24. C. V. Ramamoothy and W.-T. Tsai. Advances in software engineering. *IEEE Computer*, 29(10):47–58, 1996.
25. E. S. Raymond. The cathedral and the bazaar. In *Linux Kongress*, 1997.
26. M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. pages 30–39, 2003.
27. P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *International Conference on Software Engineering*, pages 499–510, 2007.
28. D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Aspect-oriented Software Development*, pages 212–224, 2007.
29. A. A. Takang, P. A. Grubb, and R. D. Macredie. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.*, 4(3):143–167, 1996.
30. I. Vessey. Expertise in debugging computer programs. *International Journal of Man-Machine Studies: A process analysis*, 23:459–494, 1985.