Retrieval from Software Libraries for Bug Localization: A Comparative Study of Generic and Composite Text Models

Shivani Rao
Purdue University
West Lafayette, IN, USA.
sgrao@purdue.edu

Avinash Kak
Purdue University
West Lafayette, IN, USA.
kak@purdue.edu

ABSTRACT

From the standpoint of retrieval from large software libraries for the purpose of bug localization, we compare five generic text models and certain composite variations thereof. The generic models are: the Unigram Model (UM), the Vector Space Model (VSM), the Latent Semantic Analysis Model (LSA), the Latent Dirichlet Allocation Model (LDA), and the Cluster Based Document Model (CBDM). The task is to locate the files that are relevant to a bug reported in the form of a textual description by a software developer. We use for our study iBUGS, a benchmarked bug localization dataset with 75 KLOC and a large number of bugs (291). A major conclusion of our comparative study is that simple text models such as UM and VSM are more effective at correctly retrieving the relevant files from a library as compared to the more sophisticated models such as LDA. The retrieval effectiveness for the various models was measured using the following two metrics: (1) Mean Average Precision; and (2) Rank-based metrics. Using the SCORE metric, we also compare the retrieval effectiveness of the models in our study with some other bug localization tools.

Categories and Subject Descriptors

D.2.13 [**Software**]: Software EngineeringReusable Software; H.3.1 [**Content Analysis and Indexing**]: [Dictionaries, Indexing methods]; H.3.3 [**Information Search and Retrieval**]: [Retrieval Models]

General Terms

Algorithms

Keywords

Latent Dirichlet Allocation, Latent Semantic Analysis, Information Retrieval, Bug Localization, Software Engineering

1. INTRODUCTION

Information retrieval (IR) based bug localization means to locate a bug from its textual description. That is, we want to find the files, methods, classes, etc., that are directly related to the problem that is causing abnormal execution behavior of the software.

Bug localization techniques developed in the past can be placed in two categories: those that carry out dynamic analysis of program execution behavior in order to locate a bug, and those that are based on a static analysis of the code. Dynamic analysis of a program relies on finding differences between the control flows of its passing and failing runs under certain input conditions [3, 22]. On the other hand, static analysis of a software system works typically at the granularity level of a class. Each class is examined against a set of rules that predict if the class has bugs. These rules typically refer to the language syntax and the programming conventions appropriate to the language used for the source code. The failing rules are then used to detect and categorize the bugs [14, 11].

Dynamic bug localization techniques suffer from the drawback that they are based on the availability of two control flows — the passing control flow and the failing control flow. This may not be satisfiable in real-world scenarios. The static methods, on the other hand, are usually customized to detect irregularities in a *particular* programming language following a *particular* coding convention, which makes them rather restrictive in scope. A bug localization tool built for C/C++ may not be useful for source code written in Java. In addition, static tools are bug-detection tools that make prognoses of possible locations of the bugs [3, 11, 7].

In light of the shortcomings of the traditional static and dynamic approaches to bug localization, researchers have attempted to use statistical methods based on information retrieval (IR) concepts. Information retrieval is the process of engaging in a question-answer (Q/A) session in which the answers are derived from a model learned from the a collection of documents. For the problem at hand, the model would be learned from the source code files and one may expect a subset of these files/methods/classes to be retrieved in answer to the information requested in a query (bug report). IR based bug localization approaches are not only independent of the programming language and the business concepts of the software system, they are also scalable and extensible to large software systems. IR based approaches also tend to be more general and can be used for post-diagnosis of bugs especially when the passing and failing execution behavior of the program is not available. It is therefore not surprising that IR based methods are being applied in other areas of software maintenance and program comprehension [19, 6, 21, 9, 13, 18]. Table 1 summarizes equivalences between the terminology commonly used in IR and that in bug localization in software engineering.

Since the IR based approaches were originally developed for natural languages, there exist certain challenges when one tries to adapt them to retrieval from software libraries. The two key challenges are: vocabulary mismatch and the lack of availability

Terminology in IR	Terminology in Software Engineering
Document	Source files of the software library
Query	Bug report and/or its textual description
Terms	Identifier names
Retrieval	Bug localization
Index	Source library

Table 1: Parallels between IR terminology and the more traditional SE terminology

of good evaluation datasets. Vocabulary mismatch occurs when a query contains a word that was not seen before in the documents used for model construction. For the case of software libraries, the vocabulary mismatch problem arises from the use of abbreviations and concatenations of variable names and identifiers by the developers at the time of code development. Such words are called hard-words. The words used in a query may carry the same semantic intent as portions of the hard-words, but may not match them structurally. To get around this problem, some researchers have explored using vocabulary transformations [15] and identifier splitting [10, 8] to make source code look more like natural language text. To address the second problem, researchers have taken to building their customized datasets for open source software like JEdit, JBoss, Eclipse, Rhino etc. However most of these datasets have a very small number of bugs, of the order of 5 to 15, and are not made available for other researchers for the purpose of comparison.

In this paper, we compare five basic IR models, and some variants thereof, for retrieval from software libraries for the purpose of bug localization. The five models in our study are: the Vector Space Model, the Unigram model, the Latent Semantic Analysis model, the Latent Dirichlet Allocation model, and the Cluster Based Document Model. We believe that our comparative study is made significant by our use of a standard dataset — the iBugs dataset [5] — that can be used for both static and dynamic bug localization. This dataset, based on the ASPECTJ software, has 291 bugs from 6546 software files. This dataset allows researchers to compare static and dynamic bug detection techniques with IR based bug localization techniques with many real bugs.

In the rest of this paper, in Section 2 we review the previously published work. Then, in Section 3, we review the retrieval algorithms using the five models mentioned above and a couple of variations thereof. In Section 4 we describe how the source files must be preprocessed so that they can be used for IR-based algorithms. Subsequently, we compare the performance obtained with these models using the iBugs repository in Section 5. This is followed by discussion and conclusion in Sections 6 and 7, respectively.

2. RELATED WORK ON BUG LOCALIZATION

While most existing static and dynamic bug localization techniques may be thought of as bug detection or prognosis tools, an IR based bug localization technique is best considered as a post-diagnosis (debugging) tool. For IR based bug localization, the bug logs (including the bug titles and descriptions) and their corresponding patches are treated as entities to be used in Q/A sessions vis-a-vis the software. Figure 1 shows a typical bug report and the associated patch file names for the ASPECTJ software. Most datasets used for IR based bug localization are open-source software projects that have the bugs and their resolutions reported at their respective development websites.

Marcus et al. [20] have investigated IR-based bug localization using the VSM model to represent the source code. That contribution used the JEdit software to demonstrate that the patch files that were posted on the bug tracking system were a subset of the files extracted by the IR tool. Cleary et al. [2] extended these experiments using two more models: the Latent Semantic Indexing model and the Cognitive Assignment model. They also provide a comparison of performance for these models. Use of the Latent Dirichlet Allocation model for source code retrieval for bug localization was explored in [17]. These authors have reported retrieval results on the following software projects: Mozilla, Eclipse, Rhino and JEdit.

Although bug localization using IR has been explored by researchers, none of the work reported has been evaluated on a standard dataset. In addition, the number of bugs used to evaluate these algorithms is very low, of the order of 5 to 15 bugs. Dallmeier et al [5] have created a dataset called iBUGS that contains a large number of real bugs with corresponding test-suites in order to generate failing and passing test runs using ASPECTJ software. The authors claim that iBUGS is the only dataset to use real-life programming projects having a large size with realistic bugs. There are two datasets of different sizes available with the iBUGS repository; our results here are based on the larger dataset.

3. TEXT MODELS AND RETRIEVAL ALGORITHMS

In this section we briefly review the following models and the corresponding retrieval functions:

VSM : Vector Space Model

LSA : Latent Semantic Analysis Model

UM : Unigram Model

LDA : Latent Dirichlet Allocation Model

CBDM : Cluster-Based Document Model

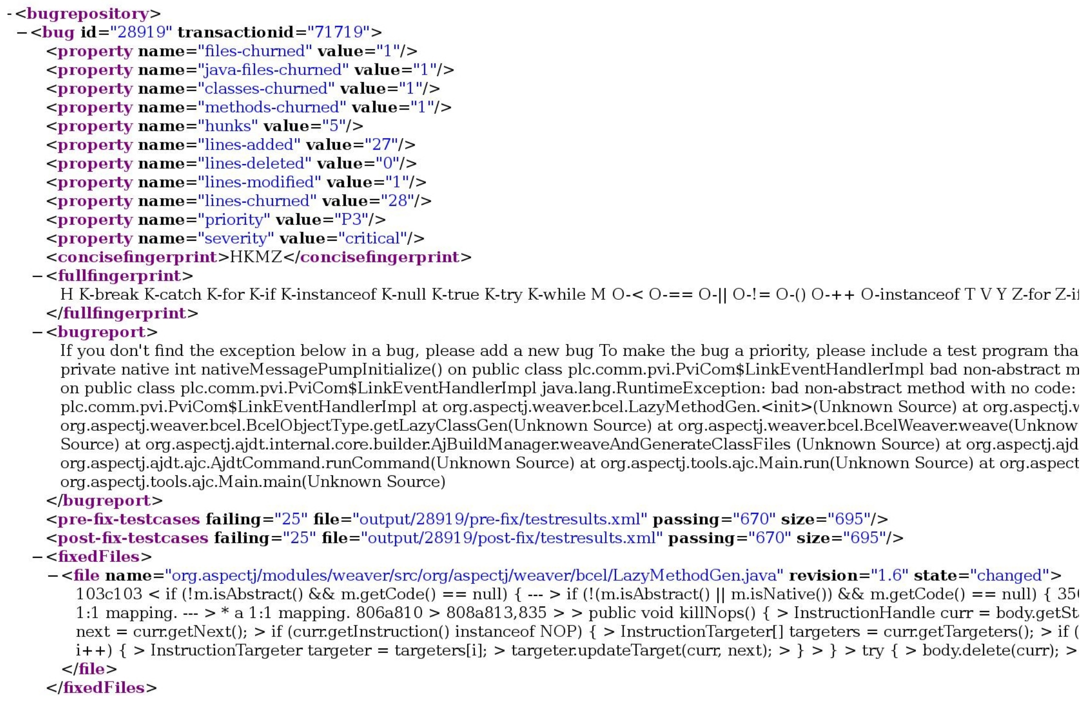
Of these five models, the first two, VSM and LSA, are purely deterministic models, whereas the other two, UM and LDA, are probabilistic. CBDM has both a deterministic and a probabilistic implementation. Among the deterministic approaches, VSM would be considered to be the simplest way to represent documents for the purpose of information retrieval. The same goes for UM among probabilistic approaches.

3.1 Generic Models

We will now provide a brief overview of the five generic text models used in our comparative study. In each case, we will also mention how a query is represented in the model. We will assume there are M source files in the library, and $\mathcal{V}$ is the set of the vocabulary terms/words occurring in the library.

As mentioned earlier, VSM (Vector Space Model) is the simplest way to represent a document for the purpose of information retrieval. With $\mathcal{V}$ denoting the vocabulary in a set D of documents, a document in VSM is represented by a $|\mathcal{V}|$-dimensional vector $\vec{w}$, with each element of the vector representing the frequency of occurrence of a word in the document. Each element of this vector is reserved for a unique word of the vocabulary[1] The advantage of

[1]Whereas $\vec{w}$ will represent any document in the library, we will use $\vec{w}_m$ to represent the m^{th} document in an indexed access to the documents. The n^{th} element of this vector will be denoted $w(n)$ or $w_m(n)$ as the case may be.



```
- <bugrepository>
 – <bug id="28919" transactionid="71719">
     <property name="files-churned" value="1"/>
     <property name="java-files-churned" value="1"/>
     <property name="classes-churned" value="1"/>
     <property name="methods-churned" value="1"/>
     <property name="hunks" value="5"/>
     <property name="lines-added" value="27"/>
     <property name="lines-deleted" value="0"/>
     <property name="lines-modified" value="1"/>
     <property name="lines-churned" value="28"/>
     <property name="priority" value="P3"/>
     <property name="severity" value="critical"/>
     <concisefingerprint>HKMZ</concisefingerprint>
   – <fullfingerprint>
       H K-break K-catch K-for K-if K-instanceof K-null K-true K-try K-while M O-< O-== O-|| O-!= O-() O-++ O-instanceof T V Y Z-for Z-i
     </fullfingerprint>
   – <bugreport>
       If you don't find the exception below in a bug, please add a new bug To make the bug a priority, please include a test program tha
       private native int nativeMessagePumpInitialize() on public class plc.comm.pvi.PviCom$LinkEventHandlerImpl bad non-abstract m
       on public class plc.comm.pvi.PviCom$LinkEventHandlerImpl java.lang.RuntimeException: bad non-abstract method with no code:
       plc.comm.pvi.PviCom$LinkEventHandlerImpl at org.aspectj.weaver.bcel.LazyMethodGen.<init>(Unknown Source) at org.aspectj.w
       org.aspectj.weaver.bcel.BcelObjectType.getLazyClassGen(Unknown Source) at org.aspectj.weaver.bcel.BcelWeaver.weave(Unknow
       Source) at org.aspectj.ajdt.internal.core.builder.AjBuildManager.weaveAndGenerateClassFiles (Unknown Source) at org.aspectj.ajd
       org.aspectj.ajdt.ajc.AjdtCommand.runCommand(Unknown Source) at org.aspectj.tools.ajc.Main.run(Unknown Source) at org.aspect
       org.aspectj.tools.ajc.Main.main(Unknown Source)
     </bugreport>
     <pre-fix-testcases failing="25" file="output/28919/pre-fix/testresults.xml" passing="670" size="695"/>
     <post-fix-testcases failing="25" file="output/28919/post-fix/testresults.xml" passing="670" size="695"/>
   – <fixedFiles>
     – <file name="org.aspectj/modules/weaver/src/org/aspectj/weaver/bcel/LazyMethodGen.java" revision="1.6" state="changed">
         103c103 < if (!m.isAbstract() && m.getCode() == null) { --- > if (!(m.isAbstract() || m.isNative()) && m.getCode() == null) { 35
         1:1 mapping. --- > * a 1:1 mapping. 806a810 > 808a813,835 > > public void killNops() { > InstructionHandle curr = body.getSt
         next = curr.getNext(); > if (curr.getInstruction() instanceof NOP) { > InstructionTargeter[] targeters = curr.getTargeters(); > if (
         i++) { > InstructionTargeter targeter = targeters[i]; > targeter.updateTarget(curr, next); > } > } > try { > body.delete(curr); >
       </file>
     </fixedFiles>
```

Figure 1: ***A typical bug report***

the VSM model is the simplicity of the computations that go into model construction and, as we will show later, the ease with which a query can be compared with the documents. Its main disadvantage is the generally large dimensionality and sparseness of the document vectors. Just like the documents, a query is also represented by a $|\mathcal{V}|$ dimensional vector. In this case, the different elements of this vector will represent the frequencies of the query words.

The LSA (Latent Semantic Analysis) model attempts to reduce the large space spanned by the vectors of the VSM model by carrying out an SVD decomposition of the term-document matrix and retaining the top K eigenvectors. The result is a reduced K dimensional representation for the documents. To elaborate, let A be the $V \times M$ term-document matrix, where M is the number of documents in the library, each column of this matrix representing the term-frequency vector for a document. SVD decomposition implies expressing A in the following form:

$$A \;\; = \;\; U \Sigma V^T \tag{1}$$

where U is $|\mathcal{V}| \times M$ orthogonal matrix, Σ a $M \times M$ matrix of singular values, and $|\mathcal{V}|$ an $M \times M$ orthonormal matrix. Dimensionality reduction is achieved by retaining just K highest singular values from the matrix Σ. The resulting truncated matrix are denoted by U_K ($|\mathcal{V}| \times K$ matrix), Σ_K ($K \times K$ diagonal matrix with the most significant singular values values) and V_K ($K \times M$ matrix with the m^{th} column representing the topic distribution of the m^{th} document). The reduced K-dimensionality representation of the original document vector $\vec{w}$ may now be expressed as

$$\vec{w}_K \;\; = \;\; \Sigma_K^{-1} U_K^T \vec{w} \tag{2}$$

where Σ_K and V_K are the truncation-to-K-dimensions versions of Σ and V in the SVD decomposition, and where $\vec{w}_K$ is the reduced dimensionality version of the document $\vec{w}$.

Given a query q, itself a $|\mathcal{V}|$ dimensional vector of word frequencies, the LSA model constructed by SVD decomposition and truncation is used to create a K-dimensional version of this query by:

$$q_K \;\; = \;\; \Sigma_K^{-1} U_K^T q \tag{3}$$

While we are on the subject of LSA modeling, we must also quickly mention a variation on such models that we denote by "LSA2". To explain the difference between LSA and LSA2, note that in LSA each document is first mapped into its reduced dimensionality form by using Equation (2). For LSA2, a document vector $\vec{w}_K$ is re-projected into the original $|\mathcal{V}|$ dimensional space by

$$\tilde{w} = U_K \Sigma_K \vec{w}_K^T \tag{4}$$

The remapped $\tilde{w}$ is thought of as a "smoothed" version of the original document vector $\vec{w}$. The remapped $|\mathcal{V}|$-dimensional vectors can be pulled into a $|\mathcal{V}| \times M$ matrix that we will denote $\tilde{A}$:

$$\tilde{A} = [\tilde{w}_1, \tilde{w}_2, \ldots, \tilde{w}_M] \tag{5}$$

Since the documents are now back in a $|\mathcal{V}|$-dimensional term space, they can be compared directly with the query vectors.

The Unigram Model (UM) fits a single multinomial distribution to the word count frequencies in each document[2] The Unigram representation of each document is a $|\mathcal{V}|$- dimensional probability vector (the elements of the vector sum to 1) $p(w|\vec{w})$. Unigram models suffer from the following drawback: If a word has never occurred in a document, it gets a 0 probability of occurrence in the probability vector for that document. This can create problems in probability update formulas. To avoid such singularities, the probabilistic representation of the documents is "smoothed" using the collection model that measures the probability of occurrence of a word in the

[2] A generalization of the UM is LM (the Language Model) which employs a mixture of multinomial distributions over n-grams (n consecutive words that are considered to form a single phrase) to represent the documents.

entire collection of the documents. The collection model is denoted by $p_c(w)$ and calculated simply on the basis of the frequency of occurrence of a word in the entire collection. The smoothed Unigram representation of the m^{th} document is thus given by

$$p_{uni}(w|\vec{w}_m) = (1-\mu)\frac{w_m(n)}{\sum_{n=1}^{n=|\mathcal{V}|} w_m(n)} + \mu\frac{\sum_{m=1}^{M} w_m(n)}{\sum_{m=1}^{M}\sum_{n=1}^{n=|\mathcal{V}|} w_m(n)} \quad (6)$$

where $w_m(n)$ is the frequency of occurrence of the n^{th} vocabulary term in document m. The second term in the above equation is the collection model.

That brings us to the fourth modeling approach: the LDA. This is a topic-based model that represents each document with a K dimensional probability vector given by $\vec{\theta}_m = p(\vec{z}_m|\vec{w}_m)$, and one for the entire library: $\vec{\phi}_t = p(\vec{w}_t|z=t)$. This allows for two possible representations of a document: a topic based representation $\vec{\theta}_m$, which is a K-dimensional probability vector, and a maximum-likelihood representation (MLE-LDA) that we denote by $p_{lda}(w|\vec{w}_m)$). The latter is a $|\mathcal{V}|$-dimensional probability vector that is calculated using the following equation:

$$p_{lda}(w|\vec{w}_m) = \sum_{t=1}^{t=k} p(w|z=t)p(z=t|\vec{w}_m) \quad (7)$$

$$= \sum_{t=1}^{t=k} \phi(t,w)\theta_m(t) \quad (8)$$

For the two aforementioned document representations that are allowed by the LDA model, we have two corresponding query representations. When a query is represented using its topic distribution, it is denoted $\vec{\theta}_q$. Such a query representation can be compared directly with the topic representations of the documents $\vec{\theta}_m$. On the other hand, when using the MLE-LDA version of the LDA model, we can represent a query directly by its first order distribution. Such a query representation is compared with $p_{lda}(w|\vec{w})$.

Finally, we review the Cluster Based Document Model (CBDM), another relatively popular approach for text modeling [16]. In this approach, we start by clustering the documents into **K** clusters using deterministic algorithms like K-means, hierarchical, agglomerative clustering, and so on. The rest of what is done in this approach falls into two categories: deterministic and probabilistic. In the deterministic version of the algorithm, we find the closest cluster to a query and retrieve all the documents in that cluster. The relative ranking of the documents within a retrieved cluster is calculated in order to subsequently retrieve the most relevant documents for the query at hand. Note that the documents, the cluster centers, and the queries are all represented in the $|\mathcal{V}|$ dimensional vector space, where, as before, $\mathcal{V}$ is the vocabulary set for the library, and a correlation based similarity measure is used to compare a query with the cluster centers. The probabilistic version of CBDM represents the clusters using a multinomial distribution over the terms in the vocabulary. This distribution is commonly denoted by $p_{ML}(w|Cluster_j)$ where ML denotes the maximum likelihood estimate. Using this multinomial, we can express the probabilistic distribution for a word in a document in cluster $Cluster_j$ by:

$$p_{cbdm}(w|\vec{w}_m) = \lambda_1\frac{w_m(n)}{\sum_{n=1}^{n=|\mathcal{V}|} w_m(n)} + \lambda_2 p_c(w) + -(\lambda_3)p_{ML}(w|Cluster_j) \quad (9)$$

Note that w represents a single word in a document whose term frequency vector is given by $\vec{w}$ and that belongs to cluster $Cluster_j$. In this formula, λ_1, λ_2 and λ_3 are the mixture proportions that add up to 1 and specify the relative involvement of each of the components: the Unigram model, the collection model, and the cluster model. For low values of λ_3, the CBDM behaves like a Unigram model. For low values of λ_1, the CBDM behaves like a deterministic cluster based retrieval model described above.

3.2 Composite Text Models

It is also possible to combine some of the generic models to form what are generally more powerful composite text models. A strong rationale for combining two or more retrieval algorithms is that every retrieval algorithm can be expected to suffer from retrieval noise and that this noise can be expected to be different for the different retrieval algorithms. So when we combine models and thus retrieval algorithms, we can hope to smooth out some of the retrieval noise.

Table 4 shows the two such composite models we have included in our comparative study. The component generic models that go into these composites are all based on the original $|\mathcal{V}|$-dimensional term space.

When we combine the MLE-LDA representation of a document with its Unigram representation, we can write the following for the overall probabilistic distribution for a document:

$$p_{combined}(w|\vec{w}_m) = (1-\lambda)p_{uni}(w|\vec{w}_m) + \lambda(p_{lda}(w|\vec{w}_m) \quad (10)$$

Similarly, when we combine the the LSA2 representation with the term-document matrix (A) to yield a powerful representation:

$$A_{combined} = \lambda\tilde{A} + (1-\lambda)A \quad (11)$$

where is the mixture parameter, $0 \leq \lambda \leq 1$.

3.3 Similarity Metrics

We will now briefly review how a query is compared with the documents in a model for the purpose of retrieval. As mentioned earlier, each model has its own way of representing a query. So each similarity measure shown below will use the query representation that is specific to that model.

The deterministic models use the cosine similarity measure on the normalized representation of the documents and the queries [23]. For the VSM model the similarity is computed using the following equation:

$$sim(\vec{w}_q, \vec{w}_m) = \frac{\vec{w}_q.\vec{w}_m}{|\vec{w}_q||\vec{w}_m|} \quad (12)$$

The scoring function for the LSA model compares the q_K and $\vec{w}_K$ vectors using the above equation.

With regard to the probabilistic models UM and LDA, the following two key similarity measures have been explored by researchers: the probability of likelihood and the divergence of distribution. When using the probability of likelihood as a similarity measure, one calculates the probability of the query given a document. To elaborate, if $\vec{w}_m$ denotes the vector representation of the m^{th} document and $\vec{w}_q$ is the vector representation of the query, then the probability of likelihood is denoted by $P(\vec{w}_q|\vec{w}_m)$.

$$sim(\vec{w}_q, \vec{w}_m) = P(\vec{w}_q|\vec{w}_m) \quad (13)$$

The other similarity measure used is the KL divergence between the probability distributions for the query and the documents. For LDA, if $\vec{\theta}_m$ and $\vec{\theta}_q$ represent the parameters of the multinomial distributions for the document and the query respectively, then these

Model	Representation	Similarity Metric
VSM	frequency vector	Cosine similarity
LSA	K dimensional vector in the eigen space	Cosine similarity
Unigram	$\|\mathcal{V}\|$ dimensional probability vector (smoothed)	KL divergence
LDA	K dimensional probability vector	KL divergence
CBDM	$\|\mathcal{V}\|$ dimensional combined probability vector	KL divergence or likelihood

Table 2: Generic models used in the comparative evaluation

Model	Representation	Similarity Metric
LSA2	$\|\mathcal{V}\|$ dimensional representation in term-space	Cosine similarity
MLE-LDA	$\|\mathcal{V}\|$ dimensional MLE-LDA probability vector	KL divergence or likelihood

Table 3: These represent variations on the generic models

are treated as one-dimensional distributions and the similarity measure is written as $-KL(\vec{\theta}_m, \vec{\theta}_q)$. The higher the divergence the lower the similarity. This similarity metric may be expressed in the following form:

$$sim(\vec{w_q}, \vec{w}_m) = -KL(\vec{\theta_m}, \vec{\theta_q}) \tag{14}$$

One can write a similar expression for the divergence based similarity metric for the case of UM.

Table 2 summarizes the text models and the corresponding similarity metrics. Table 3 presents the same information on the variations on the LSA and the LDA models we presented earlier. Finally, Table 4 shows the same for the composite models we use in this comparative study.

4. PREPROCESSING OF SOURCE FILES

In order to use any dataset for evaluation of IR based bug localization, three things are required: (1) Bugs with textual descriptions that can be used as queries; (2) Names of patch files that correspond to the bugs; and (3) The source files that form the entire library.

The iBUGS dataset is a benchmarked dataset that was created to evaluate automated bug detection and localization tools [5]. The creators of iBUGS used snapshots of the software before and after a bug fix to build test suites to create passing and failing runs of the software. Originally, iBUGS was used to evaluate two bug localization tools, AMPLE and FINDBUGS, the former a static tool and the latter dynamic. There are in all 369 bugs in iBUGS that were extracted from ASPECTJ bugzilla over a period of five years. Of these, 223 bugs contain at least one test suite. These bugs are documented together with the related patches and other very useful metadata in an XML file called *repository.xml*.

Model	Representation	Similarity Metric
Unigram + LDA	$\|\mathcal{V}\|$ dimensional combined probability vector	KL divergence or likelihood
VSM + LSA	$\|\mathcal{V}\|$ dimensional combined VSM and LSA representation	Cosine similarity

Table 4: The two composite models used

With that introduction to iBUGS, we will now describe the preprocessing steps that must be applied to the ASPECTJ source code files before they can be used for retrieval by any of the IR-based techniques. These steps are as follows:

1. As mentioned, iBUGS is a compilation of the bug related information over a span of five years. During this time, certain patch files that were relevant to the older bugs have become unavailable. These files were searched for in other branches and versions of the ASPECTJ software and added to the source library.

2. Second, iBUGS includes changes made to the XML configuration, build, and README files. From all the file that resulted from the previous step, we retain only the software files, that is, the files with the extension ".java". After this step, our library ended up with 6546 Java files.

3. The *repository.xml* file documents all the information related to a bug. This includes the BugID, the bug description, the relevant source files, and so on. This information was used to form the ground-truth information needed to evaluate the retrievals in our comparative analysis. The source files retrieved by any of the methods were compared with the source files listed in *repository.xml* to measure retrieval performance. We refer to this ground-truth information as constituting *relevance judgements*

4. For a quantitative evaluation of the different retrieval methods, it is necessary that every bug used for retrieval of the applicable software files has at least one relevant file in the library. The bugs that are documented in iBUGS but that did not have any relevant files in the source library that resulted from the previous step were eliminated. After this step, we were left with 291 bugs.

5. The final step consisted of processing the source files in a manner that is standard to retrieval from text corpora in IR. In what follows, we will list these steps and the underlying rationale:

   It is common for programmers to use juxtaposed words for identifiers in a computer program. For example, the words "handle" and "exception" will commonly be combined to form "handlexception". While such words present no difficulty to a human reader of the code (since the mind can effortlessly separate out their constituents and the associated meanings), if not separated during preprocessing, they can limit the scope of the retrievals. Such juxtaposed words, called hard-words, are split using an identifier-splitting method like the one's reported in [10, 8].

   We drop from the vocabulary the most common of the programming language keywords that we can expect to possess no discriminatory power with regard to any retrieval. These include keywords such as "for," "else," "while," "int,", "double," "long," "public," "void," etc. There are 375 such words in iBUGS ASPECTJ software. We also drop from the vocabulary all unicode strings, these being of the form $uXXXX$ where 'X' is a hex digit. Such strings, frequently denoting international characters, are used as values for some of the string variables.

   Even after the clean-up steps described above, we are left with a vocabulary of size that is approximately 40,000. For

Software Library Size (Number of files)	6546
Lines of Code	75 KLOC
Vocabulary Size	7553
Number of bugs	291
Avg No of relevant files/ bug	1

Table 5: The iBUGS dataset after preprocessing

further compression of the vocabulary, we calculated the library wide inverse-document-frequency (*idf*) and the term-frequency (tf) associated with each word. As used in our preprocessing, these are defined as

$$idf(n) \quad = \quad log(\frac{|D|}{|D_{w(m,n)>0}|}) \qquad (15)$$

$$tf(n) \quad = \quad \sum_{i=1}^{i=|D|} w(i,n) \qquad (16)$$

where D denotes the set of all source files. If a word occurred in every document/source file, for such a word we would have $idf(n) \approx 0$. Words with a very low *idf* are non-discriminative. Words with low *idf* and low tf occur rarely and can be eliminated for a more efficient vocabulary for retrieval. Words with low *idf* but high tf are discriminative and are not eliminated. Our experimental evaluation shows that this method of pruning the vocabulary does not affect the retrieval performance. On the other hand, it improves the retrieval efficiency.

Using the preprocessing steps listed above, we ended up with a "corpus" of 6546 source files containing roughly 75,000 lines of code, a vocabulary of 7553 unique words, and 291 bugs (Table 5).

5. COMPARATIVE EVALUATION

As mentioned before, retrieval consists of selecting the top N_r documents from a ranked list of documents retrieved vis-a-vis the query. The relevance judgements available from *repository.xml* is compared with the retrieved results to measure retrieval performance. We discuss two commonly used measures as follows:

Mean Average Precision (MAP): We extract from the iBUGS document *repository.xml* the names of the files relevant to a bug. For calculating MAP, we refer to this set as the $relevant$ set. The set of files that is actually retrieved for a given bug description is referred to as the $retrieved$ set. The Precision (P) and the Recall (R) values for a bug may now be calculated by:

$$Precision(P) = \frac{|\{relevant\} \bigcap \{retrieved\}|}{|\{retrieved\}|} \qquad (17)$$

$$Recall(R) = \frac{|\{relevant\} \bigcap \{retrieved\}|}{|\{relevant\}|} \qquad (18)$$

If the value chosen for N_r is so large as to include all the documents, then Recall will equal 1 because the intersection of the relevant set and the entire corpus will the relevant set itself. By the same token, if N_r is made too small, the result would be poor values for Recall and a possibly high value for Precision. This makes it imperative that both Precision and Recall be calculated for different choices for the N_r parameter. Precision and Recall values obtained in this manner are usually represented by the notation $P@N_r$ and $R@N_r$, respectively. If we were to plot Precision versus Recall values from the values obtained for $P@N_r$ and $R@N_r$, we would get a monotonically decreasing curve that has high values of Precision for low values of Recall and vice versa. MAP computes the area under such a Precision-Recall (P-R) curve and is a common metric for measuring the overall retrieval performance on text-collections. For each Recall value (obtained for a different N_r) one calculates $P@N_r$ and then takes an average of these precision values. When we average out the values of MAP over all the queries (bugs), we get an averaged MAP that is also simply referred to by MAP, the further averaging over the queries remaining implicit. Such averaged values for MAP signify the proportion of the relevant documents that will be retrieved, on the average, for any given query. The higher this MAP, the better the retrieval performance.

Model	Unigram	VSM
MAP	0.1454	0.0796

Table 6: MAP performance obtained for VSM and for UM

Rank of Retrieved Files: IR based bug localization researchers have employed this metric to analyze the performance of their retrieval algorithms[17]. A similar metric for fault detection, known as SCORE [12], can be used to indicate the proportion of a program that needs to be examined in order to locate a faulty statement. For each range of this proportion (example, $10 - 20\%$) the number of test runs that result in bugs is determined. For our work reported here, we have used the former metric — the rank-based metric — since it a more commonly used metric for IR-based bug localization research. With this metric, the number of queries/bugs for which the relevant source files are retrieved with ranks $r_{low} \leq R \leq r_{high}$ is reported. A ranked list of source files returned by the bug localization tool in response to a query/bug is used to compute the rank-based metrics. For the retrieval performance reported in [17], ranks used were $R = 1$, $2 \leq R \leq 5$, $6 \leq R \leq 10$ and $R > 10$.

We will now present the results of our comparative evaluation. Our presentation of the results is made somewhat complicated by the fact that the simplest of the models require no tuning parameters, whereas others require that we measure the performance for different values of the parameters that go into the models.

Our primary metric for reporting on the performance, especially the performance measures for different values of the model parameters, will be the MAP metric. However, subsequently, we will compare the models, each with its parameters set to the values that yielded the best MAP numbers, on the basis of the rank metric.

We will start with the VSM and UM models. Since there are no parameters to be tuned for both, we can report the retrieval performance summarily in the form of A-MAP results as shown in Table 6. The UM model was based on the representation shown in Equation (6) with $\lambda = 0.5$.

With regard to the performance numbers for the LDA model, we must first mention that the LDA model was created using a standard collapsed Gibbs sampler implementation of the learning algorithm [1]. The main experimental parameters that affect the results obtained with the LDA model are: (1) The number of topics (K); (2)

α	LDA	MLE-LDA	Unigram+ LDA
0.000667	0.015	0.05484667	0.1251741
0.006667	0.014	0.0657811	0.1232067
0.033333	0.013	0.08716058	0.119226
0.333333	0.0125	0.07196478	0.1210101

Table 7: MAP for different values of α while keeping the other parameters fixed at K=150 and $\beta = 0.01$.

The mixture proportions controlled by λ and ν (See Equation (10)); and (3) The hyper-parameters (α) that control the spread of topics over the source files/documents.

Figure 2 compares the performances of the three models that use the LDA representation for bug localization: the basic LDA, MLE-LDA and Unigram+LDA. As the reader can see, the composite model Unigram+LDA outperforms the model that uses only the basic LDA representation and the variation MLE-LDA. However, note that the composite model never does better than the Unigram Model.

To further study the performance of the parameter-rich composite Unigram+LDA model, note that the effect of the mixture parameters λ and ν is to control the proportions of the Unigram model, the LDA model, and the collection model in Equation (10). We see that the value of μ is directly proportional to the weight given to the collection model and the value of λ is inversely proportional to the weight given to LDA model. In Figure 4, each plot corresponds to a different value of λ and each of the curves shows the MAP values verses μ. It is evident from this figure that the lower the value of , the higher the involvement of the LDA model, and poorer the performance. Typically for values of $\mu \leq 0.5$, the performance of the composite model improves regardless of the proportion of the LDA model in the composite. In all of the LDA based models, the parameter α controls the spread of topics over the source files/documents. Table 7 shows the MAP values obtained for different values of α. The higher the value of α, the greater the extent of smoothing, and better the performance.

The results for the three flavors of the LSA model, namely LSA, LSA2 and VSM+LSA, are plotted for different values of K in Figure 3. Like LDA, LSA underperforms in comparison with the VSM model. The composite model VSM+LSA performs as well as the VSM model, but not better.

We will now presents the results obtained with the CBDM model. Table 8 tabulates a summary of our findings. Rows 1 and 2 show that for the same value of λ_3 the performance of the CBDM remains more or less the same. Row 3 shows the best retrieval performance for CBDM where the dominating model is the Unigram model. Surprisingly, from the result shown in Row 4 where the collection model is dominant, we can see that the performance does not suffer too much. In both cases, $\lambda_3 = 0.1$, which means that for low values of λ_3 the CBDM behaves like UM. In most cases, it was found that increasing the number of clusters did not improve the retrieval performance. Row 5 shows the performance obtained with very little involvement of the cluster model and as can be seen the retrieval performance is not affected by the number of clusters. For high values of λ_3, the CBDM behaves similar to a deterministic cluster model in which all the source files/documents within the closest cluster are ranked and retrieved without any assistance from the Unigram model. As shown in Row 6 of Table 8, the cluster-model by itself under-performs in comparison with a combined model and the VSM model. Hence the deterministic version of CBDM does not outperform the VSM model. This result is similar to the one obtained for LDA and LSA models.

Model parameters			K			
λ_1	λ_2	λ_3	100	250	500	1000
0.25	0.25	0.5	0.0931	0.0914	0.0867	0.0766
0.15	0.35	0.5	0.0883	0.0897	0.0963	0.0932
0.81	0.09	0.1	**0.143**	0.102	0.108	0.0995
0.27	0.63	0.1	0.1306	0.117	0.111	0.0998
0.495	0.495	0.01	0.141	0.141	0.141	0.141
0.05	0.05	0.99	0.069	0.075	0.072	0.065

Table 8: Retrieval performance measured with MAP for CBDM. $\lambda_1 + \lambda_2 + \lambda_3 = 1$.

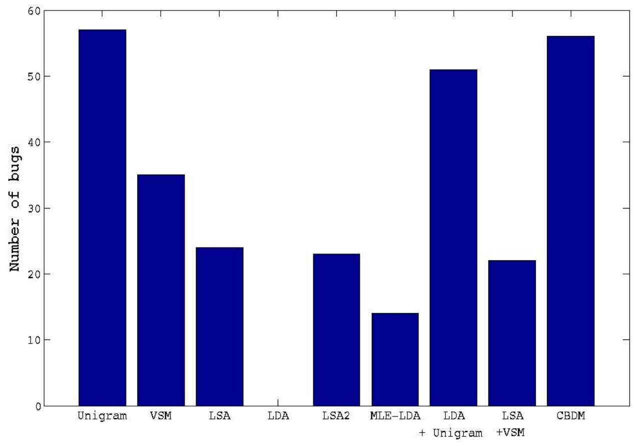


Figure 5: The height of the bars shows the number of queries (bugs) for which at least one relevant source file was retrieved at rank 1.

We will now use the rank-based metric to make a comparison across all the models. For this comparison, we use those parameters for each model that yielded the best MAP values in the results we have shown so far. This comparison is shown in Figure 5, which reports the number of queries (bugs) for which a relevant source file was retrieved at rank 1. As can be seen from the figure, UM and CBDM achieve comparable performance. With both these models, the retrieval engine retrieved at least one relevant file at rank 1 for 20% of the bugs. The LDA model gives worst performance with no queries/bugs retrieving any relevant source file at rank 1.

A more detailed look at the rankings of the retrieved source files is presented in Table 9 where we show the number of bugs for which the relevant source files were retrieved at rank 1, between ranks 2 and 5, between ranks 6 and 10, and for ranks greater than 10. These results demonstrate that, using the Unigram model, for over 50% of the bugs, the developers need not scan beyond 10 retrieved source files to locate the bug.

The results shown lead us to conclude that the models in our comparative evaluation can be ranked in the following decreasing order of retrieval performance (as measured by MAP and rank-based metrics): Unigram, Unigram+LDA, CBDM (probabilistic version), VSM, CBDM (deterministic version), VSM+LSA, LSA2, LSA, MLE-LDA and LDA.

6. DISCUSSION

Since the iBUGS dataset that we used in our comparative study has also been used by others in the evaluation of their own tools for bug localization (using both static and dynamic approaches),

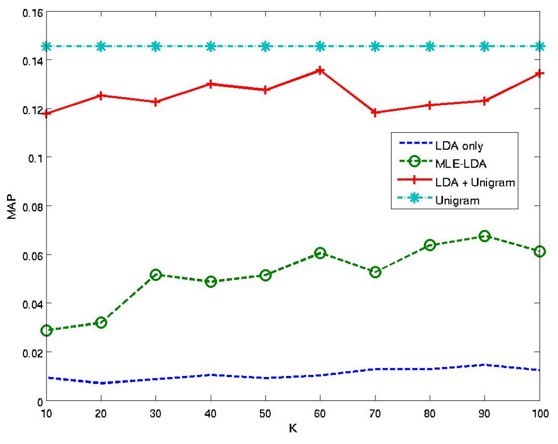


Figure 2: MAP for the three LDA models for different values of K. The parameters for the LDA+Unigram model are λ = 0.9 μ = 0.5, $\beta = 0.01$ and $\alpha = 50/K$.

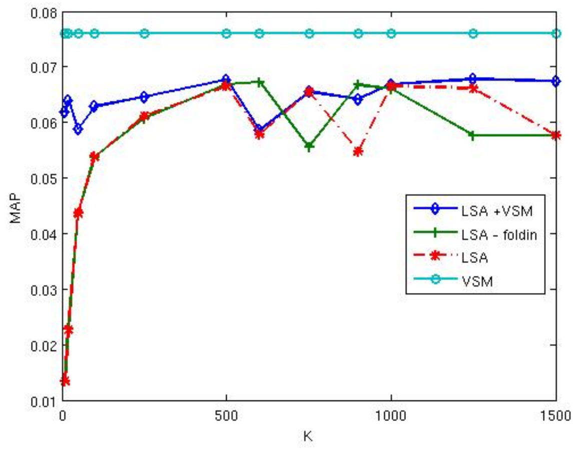


Figure 3: MAP values for the LSA model and its variations for different values of K. The experimental parameter for the LSA+VSM combined model is $\lambda = 0.5$.

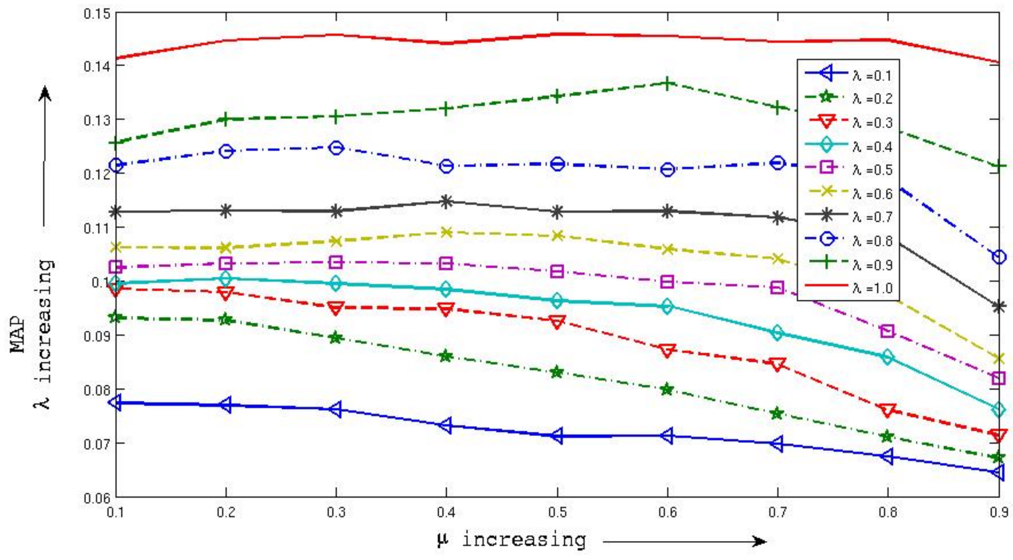


Figure 4: MAP plotted for different values of mixture proportions (λ and μ) for the LDA+Unigram composite model.

we are in a position to provide a performance comparison across the tools, including ours. The SCORE metric is convenient for this purpose as other researchers have used this metric to report their performance numbers. As mentioned earlier, the SCORE metric calculates the proportion of bugs versus the proportion of the code that must be examined for the localization of the bugs.

In Table 10 we plot the SCORE values for the different models in our own comparative study. As shown in the table, using the Unigram model (row 2 and column 2 of the table), 0.756 fraction of the bugs can be located by examining at most 0.01 fraction of the source library, and 0.966 fraction of bugs can be located by examining 0.1 fraction of the source library. Now compare these numbers with the SCORE values reported by [4], using AMPLE (a dynamic bug localization tool) — those results say that for 40% (0.4 fraction) of the bugs, one need not examine more than 10% (0.1) of the software files. See the column headed "0.1" for a comparison of AMPLE with the other text models in our study.

On the basis of the SCORE values obtained by us using the IR based models, we can claim that the IR based algorithms perform far better than AMPLE in localizing bugs. It is interesting to note that when a static bug detection tool like FINDBUGS was used to localize the bugs in the iBUGS repository, the authors [4] reported that none of the bugs could be localized correctly. This is possibly because static bug detection tools are good at detecting relatively simple coding errors made at the time of software development, but are not good at localizing complex bugs that involve run-time phenomena.

From these comparisons with a dynamic analysis based tool like AMPLE and a static analysis based tool like FINDBUGS, and on the basis of the SCORE values for statistical bug localization as

Ranks	Unigram	VSM	LSA	LDA	LSA2	MLE-LDA	Unigram+ LDA	VSM+ LSA	CBDM
$R = 1$	57	35	24	0	25	14	51	22	56
$2 \leq R \leq 5$	61	33	25	10	25	45	55	25	48
$6 \leq R \leq 10$	26	14	22	7	25	29	35	22	31
$R > 10$	147	209	220	274	218	203	150	220	156

Table 9: The number of bugs, out of 291, for which the source files relevant to the bugs were retrieved with ranks shown in the first column.

	0.01	0.03	0.05	0.07	0.09	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Unigram	0.756	0.887	0.928	0.959	0.966	0.966	0.983	0.986	0.997	0.997	0.997	0.997	0.997	0.997
VSM	0.536	0.705	0.804	0.849	0.890	0.911	0.962	0.979	0.986	0.997	0.997	0.997	0.997	0.997
LSA	0.529	0.663	0.735	0.759	0.770	0.770	0.856	0.900	0.921	0.938	0.955	0.966	0.972	0.972
LDA	0.364	0.588	0.704	0.766	0.787	0.794	0.842	0.856	0.873	0.907	0.938	0.962	0.973	0.983
LSA2	0.529	0.674	0.735	0.756	0.770	0.773	0.856	0.900	0.921	0.938	0.952	0.966	0.969	0.973
MLE-LDA	0.619	0.818	0.870	0.921	0.942	0.949	0.997	0.997	0.997	0.997	0.997	0.997	0.997	1
Unigram+LDA	0.739	0.890	0.942	0.967	0.969	0.973	0.990	0.993	0.997	0.997	1	1	1	1
VSM+LDA	0.526	0.680	0.739	0.753	0.7732	0.784	0.859	0.897	0.921	0.938	0.949	0.966	0.969	0.973
CBDM	0.742	0.880	0.935	0.966	0.969	0.969	0.986	0.993	0.997	0.997	0.997	0.997	0.997	0.997

Table 10: Each row shows the SCORE value for a different model. The column headings show the fraction of the source code that needs to be examined in order to locate the fraction of the bugs indicated by a cell entry.

shown in Table 10, we can draw two conclusions: (1) that dynamic bug localization tools, although useful in locating bugs that involve run-time phenomena, would need to examine a significantly greater fraction of the source code compared with IR-based statistical bug localization tools; and (2) the static analysis based bug localization tools may not turn out to be very useful, especially in relation to statistical tools.

It may seem counter-intuitive that the more sophisticated models, such as LDA, would perform poorly as compared to the simpler models like VSM and UM. We will now take a more careful look at the nature of the database vis-a-vis the models and attempt an explanation for this behavior.

The two topic based models, LDA and LSA, and the cluster model, CBDM, are based fundamentally on the term co-occurrence information. As to what extent the term co-occurrences can be used for a model that can discriminate between the source files can be measured with what is known as inverse-document-frequency (idf), which is calculated using Equation (15). The value of idf for a given word signifies the inverse ratio of the number of source files/documents that contain the word. In order to have a good clustering of the documents, we should have a uniform distribution of the idf over the words. The histogram of the idf for the source files in our dataset is plotted in Figure (6).

As can be seen, the idf graph peaks at its two ends. This means that the 7553 words in the iBUGS repository have idf values that are either very high or very low. What that means is that most of the words in the vocabulary either occur in a large number of the source files or in very few. This reduces the relative discriminative power of the words and renders them useless for the LDA model and other complex models that rely on term co-occurrence.

From the above discussion it is clear that for large software libraries there is not much to be gained by employing complex models for bug localization. Note that our conclusions may not apply to a bug localization tool that must extract source code files from a large library that includes multiple projects that are diverse with regard to their vocabulary content and with regard to the statistical attributes associated with how the projects are populated with the vocabulary words. For such cases, a topic based approach, as represented by LDA type models, could prove more powerful. A good LDA model may be able to cluster together the files belonging to different projects under separate topics. For the same reasons, cluster based approaches may also yield superior performance for such libraries.

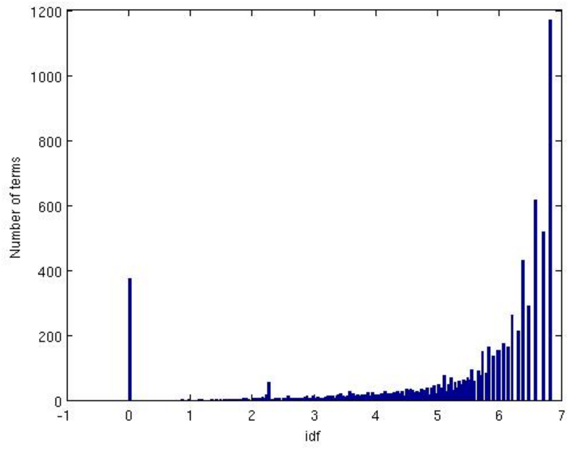


Figure 6: Histogram of the IDF values for the vocabulary words

7. CONCLUSIONS & FUTURE WORK

We have shown preliminary results comparing different text models as retrieval tools for the purpose of bug localization. The main conclusions of our comparative study are summarized here:

- IR based bug localization techniques are at least as effective as the static and the dynamic bug localization tools developed in the past.
- Sophisticated models like LDA, LSA, and CBDM do not outperform simpler models like Unigram or VSM for IR based bug localization for large software systems.
- An analysis of the spread of the word distributions over the source files with the help of measures such as tf and idf can give useful insights into the usability of topic and cluster based models for bug localization.

It would be wonderful if a comparative study such as ours could be carried out on even larger databases than iBUGS. Obviously, creation of the required ground-truth information would be a major challenge in that endeavor. Future work involves comparison of IR tools with the more standard tools such as the Source Navigator, grep and Doxygen. We also wish to explore using topic and cluster based models to perform retrieval for bug localization for software systems that involve multiple languages. Last but not the least, we believe that combining IR-based bug localization tools with those that carry out dynamic bug localization could significantly improve the state-of-the-art in bug localization.

8. ACKNOWLEDGMENTS

This work was supported by Infosys. Thanks are also owed to Bunyamin Sisman and Chad Aeschliman for their comments and feedback.

9. REFERENCES

[1] J. Chang. R-lda. `http://cran.r-project.org/web/packages/lda/`.

[2] B. Cleary, C. Exton, J. Buckley, and M. English. An Empirical Analysis of Information Retrieval based Concept Location Techniques in Software Comprehension. *Empirical Softw. Engg.*, 14(1):93–130, 2009.

[3] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight Bug Localization with AMPLE. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, AADEBUG'05, pages 99–104, New York, NY, USA, 2005. ACM.

[4] V. Dallmeier and T. Zimmermann. Automatic Extraction of Bug Localization Benchmarks from History. Technical report, Universiät des Saarlandes, Saarbrücken, Germany, June 2007.

[5] V. Dallmeier and T. Zimmermann. Extraction of Bug Localization Benchmarks from History. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 433–436, New York, NY, USA, 2007. ACM.

[6] P. T. Devanbu, R. J. Brachman, P. G. Selfridge, and B. W. Ballard. Lassie—A Knowledge-Based Software Information System. In *ICSE '90: Proceedings of the 12th international conference on Software engineering*, pages 249–261, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[7] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 57–72, New York, NY, USA, 2001. ACM.

[8] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker. Mining Source Code to Automatically Split Identifiers for Software Analysis. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, pages 71–80, Washington, DC, USA, 2009. IEEE Computer Society.

[9] W. B. Frakes and B. A. Nejmeh. Software Reuse through Information Retrieval. *SIGIR Forum*, 21(1-2):30–36, 1987.

[10] D. B. H. Field and D. Lawrie. An Empirical Comparison of Techniques for Extracting Concept Abbreviations from Identifiers. In *Proceedings of IASTED International Conference on Software Engineering and Applications*, 2006.

[11] D. Hovemeyer and W. Pugh. Finding Bugs is Easy. *SIGPLAN Not.*, 39:92–106, December 2004.

[12] J. A. Jones and M. J. Harrold. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. In *Automated Software Engineering*, 2005.

[13] A. Kuhn, S. Ducasse, and T. Gîrba. Semantic Clustering: Identifying Topics in Source Code. *Source Information and Software Technology archive*, 49:230–243, 2007.

[14] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: Statistical Model-Based Bug Localization. *SIGSOFT Softw. Eng. Notes*, 30:286–295, September 2005.

[15] H. Liu and T. C. Lethbridge. Intelligent Search Techniques for Large Software Systems. In *CASCON '01: Proceedings of the 2001 conference of the Centre for Advanced Studies on Collaborative research*, page 10. IBM Press, 2001.

[16] X. Liu and W. B. Croft. Cluster-Based Retrieval using Language Models. In *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '04, pages 186–193, New York, NY, USA, 2004. ACM.

[17] S. K. Lukins, N. A. Karft, and E. H. Letha. Source Code Retrieval for Bug Localization using Latent Dirichlet Allocation. In *15th Working Conference on Reverse Engineering*, 2008.

[18] Y. S. Maarek, D. M. Berry, and G. E. Kaiser. An Information Retrieval Approach for Automatically Constructing Software Libraries. *IEEE Trans. Softw. Eng.*, 17(8):800–813, 1991.

[19] A. Marcus and J. I. Maletic. Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 125–135, Washington, DC, USA, 2003. IEEE Computer Society.

[20] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An Information Retrieval Approach to Concept Location in Source code. In *In Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004*, pages 214–223. IEEE Computer Society, 2004.

[21] G. Mishne and M. D. Rijke. Source Code Retrieval using Conceptual Similarity. In *Proc. 2004 Conf. Computer Assisted Information Retrieval (RIAO âĂŹ04*, pages 539–554, 2004.

[22] M. Renieres and S. Reiss. Fault Localization with Nearest Neighbor Queries. In *Proceedings. 18th IEEE International Conference on Automated Software Engineering*, ASE'03, pages 30–39, 2003.

[23] I. Ruthven and M. Lalmas. A Survey on the Use of Relevance Feedback for Information Access Systems. *The Knowledge Engineering Review*, 2003.