2012 9th IEEE Working Conference on Mining Software Repositories

Incorporating Version Histories in Information Retrieval Based Bug Localization

Bunyamin Sisman, Avinash C. Kak
Purdue University
West Lafayette, IN
{*bsisman, kak*}*@purdue.edu*

***Abstract*—Fast and accurate localization of software defects continues to be a difficult problem since defects can emanate from a large variety of sources and can often be intricate in nature. In this paper, we show how version histories of a software project can be used to estimate a prior probability distribution for defect proneness associated with the files in a given version of the project. Subsequently, these priors are used in an IR (Information Retrieval) framework to determine the posterior probability of a file being the cause of a bug. We first present two models to estimate the priors, one from the defect histories and the other from the modification histories, with both types of histories as stored in the versioning tools. Referring to these as the base models, we then extend them by incorporating a temporal decay into the estimation of the priors. We show that by just including the base models, the mean average precision (MAP) for bug localization improves by as much as 30%. And when we also factor in the time decay in the estimates of the priors, the improvements in MAP can be as large as 80%.**

***Keywords*-Bug Localization; Information Retrieval; Document Priors; Software Maintenance**

I. Introduction

A powerful search engine that can retrieve software artifacts with high precision is obviously of great importance to developers. Over the years, several IR (Information Retrieval) based approaches have been proposed towards that end [1], [2]. In the ongoing research that is based on using IR based tools for bug localization, Ashok et al. [3] have shown how relationship graphs can be used to retrieve source files and prior bugs in response to what they refer to as "fat queries" that consist of structured and unstructured data. Another IR based contribution, also aimed at the retrieval of software artifacts, is by Rao and Kak [4]. They have explored several generic and composite retrieval models for bug localization.

To remind the reader, in IR based bug localization, a query describing some defective behavior of the software is run against the code base in order to rank the software artifacts in the code base with the hope that the highly ranked retrieved artifacts will be those that are likely to have caused the defective behavior. In an IR based retrieval framework that leverages the prior evolutionary information concerning the development of the software, Kagdi et al. [5] have demonstrated how change impact analysis can be carried out by exploiting the conceptual and evolutionary couplings that exist between the different software entities. Along similar lines, Nguyen et al. [6] have proposed BugScout, an automated approach based on Latent Dirichlet Allocation to narrow down the search space while taking into account the defect proneness of the source files. Although much progress has been made through these and other similar contributions, a thorough and theoretically sound investigation of using version histories for the retrieval of defective software components is still missing. In this paper, we present a framework for the estimation of a prior probability distribution for the defect proneness of the files in a given version of a software project. Our work leverages the development history of the files while incorporating a temporal decay factor in order to place greater weight on recent maintenance efforts. We demonstrate that IR based bug localization accuracy can be significantly improved when such models for the priors are employed in retrieval.

In support of the predictive power of version and modification histories stored in software repositories, studies such as those reported in [7], [8], [9], [10] have demonstrated that the version histories store a wealth of information that could potentially be used to predict the future defect likelihoods of the software entities such as files, classes, methods, and so on. Along the same lines, Zimmermann et al. [11] have shown that prior modification history of software components can be used to guide engineers in future development tasks. Motivated by these and similar other studies, our goal here is to mine the defect and file modification related knowledge that is always buried in the software repositories and to incorporate this knowledge in well-principled retrieval models for fast and accurate bug localization in large software projects.

With regard to using version histories for bug localization, the work by Hassan and his collaborators has demonstrated that defects are mainly associated with high software modification complexity [9]. Many modifications committed by several programmers during a short period of time is a strong predictor for future defects [10]. Besides complex prior modification history, the defect histories of the files in a software project is also a good predictor of future defects. A buggy file in the early stages of the project development is likely to produce defects throughout the life cycle of a project unless the project undergoes a fundamental design change [12], [13].

We must mention that there is a rich history of prior work on bug localization through an analysis of either the dynamic or the static properties of software [14], [15]. While dynamic approaches rely on passing and failing executions of a set of test cases to locate the parts of the program causing the bug, static approaches do not require execution of the program and aim at leveraging the static properties or interdependencies to locate bugs. The main deficiency of the dynamic approaches is that designing an exhaustive set of test cases that could effectively be used to reveal defective behaviors is very difficult and expensive. Static properties of a software project as derived from, say, call graphs, on the other hand, tend to be language specific. The static and dynamic approaches also are not able to take into account non-executable files, such as configuration and documentation files, that can also be a source of bugs in modern software.

In comparison with the dynamic and static approaches of the sort mentioned above, the IR based approaches to bug localization may be found preferable in certain software development contexts because they can be used in interactive modes for the refinement of the retrieved results. That is, if a developer is not satisfied with what is retrieved with a current query (because, say, the retrieved set is much too large to be of much use), the developer has the option of reformulating his/her query in order to make the next retrieval more focused. The set of artifacts retrieved in response to the current query can often give clues as to how to reformulate one's query for better results the next time. Being able to query the source code flexibly in this manner can only be expected to increase programmers' understanding and knowledge on the software, which, we believe, is the key to efficient bug localization.

This paper is organized as follows. Section II presents our framework for estimating the *prior defect and modification probabilities* of the files in a given version of a project. Section III explains how these prior probabilities can be incorporated into the state of the art retrieval models to improve the bug localization accuracies. Section IV presents the experimental evaluation of our approach and Section VI concludes the paper.

II. Estimating Defect & Modification Based Prior Probabilities

After a software product has entered the market place, any further evolution of the software typically takes place in small steps in response to change requests such as those for adding a new feature, modifying an existing feature, bug fixing, and so on. At each step, new files may be added to the code base of the project, or existing files may be removed or altered to implement the requested change. Software Configuration Management (SCM) tools such as SVN create a new revision of the project by incrementally storing these *change-sets* with every commit operation.

The modifications made to a specific set of files in response to a change request suggest strong *empirical* dependencies among the changed files that may not be captured otherwise via dynamic or static properties of the software such as call graphs, APIs or execution traces created by running a set of test cases.

Naturally, not all change-sets imply strong interdependencies among the involved files. For example, the change-sets for what are usually referred to as General Maintenance (GM) tasks tend to be very large. As a case in point, a change-set for removing unnecessary import statements from all Java files in a code base [16] does not carry much useful information with regard to any co-modification dependencies between the files. We do not use such change-sets in our models. We regard all non-GM change-sets accumulated during the life cycle of a software project as Feature Implementation (FI) change-sets. We consider an FI change-set to be of type BF (Bug Fixing) if it is specifically committed to implement a bug fix. The common approach to determining whether a change-set is BF is to lexically analyze the commit message associated with it [17]. Those commit messages include key phrases such as "fix for bug" or "fixed bug" etc.

Over the years, researchers have proposed several process and product related metrics to predict the defect potential of software components in order to help the software managers make smarter decisions as to where to focus their resources. Several studies have shown that bug prediction approaches based on process metrics outperform the approaches that use product metrics [7], [8], [9]. Along the same lines, in our work we use FI and BF change-sets to compute the modification and the defect probabilities respectively.

We denote the kth change-set of a software project by r_k; this represents the set of the modified files in response to the kth change request for $k = 1 \cdots K$ during software development or maintenance. After the kth commit, with some files having being altered, a new collection of files C_k is created. If C_k exhibits defective behavior, the defect and the modification probabilities of the files in C_k can be modeled by a *multinomial distribution*[1] where $P(f|C_k)$ represents the probability of a file f to be responsible for the defective behavior reported. Obviously, $\sum_{f \in C_k} P(f|C_k) = 1$. We assume that this probability associated with a file is independent of the rest of the files in the collection. We refer to this as the "bag of files" assumption.

In the rest of this section, we first propose two base models that determine from the version histories the prior defect and modification probabilities associated with the files in a software project. We have named them *Modification History based Prior* (**MHbP**) and *Defect History based Prior* (**DHbP**). Then we extend these models by incorpo-

[1]The multinomial distribution and the categorical distribution are used exchangeably in the IR community. Here we adhere to the tradition.

Table I
The Notation Used

K	Total number of change-sets
C_k	The collection of the software files after the kth commit
R_k	The set of change-sets up to and including the kth change-set
β_1	The decay parameter for modification probabilities
β_2	The decay parameter for defect probabilities

rating in them a time decay factor. We add the suffix 'd' to the acronyms of the base models to indicate the decay-based versions of the two models by **MHbPd** and **DHbPd** respectively. Table I summarizes the notation used in these models.

A. The MHbP Model

Several authors have established that the modification history of a file is a good predictor of its fault potential. Hassan used the code change complexity to estimate the defect potential of the files on the basis of the rationale that as the number of modifications to the files increases, the defect potential of the files must also increase [9]. Nagappan et al. [10] showed that the *change bursts* during certain periods of software development are good indicators of future defects. The main intuition behind their approach is that implementing many changes in a short period of time complicates the development process, leading to defects.

With the MHbP model, we translate the frequencies with which the files are mentioned in the change records into file modification probabilities. Using Maximum Likelihood (ML) estimation, the modification probability of a file f in a given collection C_k can be expressed as

$$P_{MHbP}(f|C_k, R_k) = \frac{\sum_{i=1}^{k} I_m(f, r_i)}{\sum_{f' \in C_k} \sum_{i=1}^{k} I_m(f', r_i)} \quad (1)$$

where

$$I_m(f, r_i) = \begin{cases} 1, & f \in r_i \ \& \ r_i \in FI \\ 0, & otherwise. \end{cases} \quad (2)$$

In the formulation, R_k represents the set of change-sets from the beginning of the development to the kth change-set. Obviously $R_K = FI \cup GM$. The model assigns bigger probability mass to the files modified more frequently with $\sum_{f \in C_k} P_{MHbP}(f|C_k, R_k) = 1$.

B. The DHbP Model

Bug fixing change-sets mark the defect producing files during the life cycle of a project. The files mentioned in these change-sets are highly likely to produce bugs in the future as the buggy files tend to remain buggy [12], [13].

Similar to the MHbP model, we estimate the defect probability of a file by

$$P_{DHbP}(f|C_k, R_k) = \frac{\sum_{i=1}^{k} I_b(f, r_i)}{\sum_{f' \in C_k} \sum_{i=1}^{k} I_b(f', r_i)} \quad (3)$$

where

$$I_b(f, r_i) = \begin{cases} 1, & f \in r_i \ \& \ r_i \in BF \\ 0, & otherwise. \end{cases} \quad (4)$$

$I_b(f, r_i)$ is an indicator variable that becomes one if r_i implements a bug fix that results in a modification in f. This probability gives the ML estimate for the defect probabilities of the files.

C. Modeling the Priors with Temporal Decay

MHbPd: After a change request is implemented, it takes time for the files to stabilize and become bug-free. Indeed, implementing certain change requests may even take more than one commit operation perhaps from several developers [18]. However after the files have been stabilized, we expect a decrease in the modification probabilities. Therefore, even if a file had been modified frequently during a certain period of time in the past, if it has not been modified recently, the modification probability should decrease [8]. We incorporate a time decay factor into the formulation of the modification probabilities to take that facet of software development into account as follows:

$$P_{MHbPd}(f|C_k, R_k) = \frac{\sum_{i=1}^{k} e^{\frac{1}{\beta_1}(t_i - t_k)} I_m(f, r_i)}{\sum_{f' \in C_k} \sum_{i=1}^{k} e^{\frac{1}{\beta_1}(t_i - t_k)} I_m(f', r_i)} \quad (5)$$

where t_i represents the time at which the ith change-set was committed. This type of decay models has been used commonly in the past [12], [9], [8]. The parameter β_1 governs the amount of decay and it is related to the *expected time* for the files to stabilize with the implementation of a change request. As β_1 decreases, the amount of decay increases and therefore the expected stabilization time decreases.

DHbPd: Similar to the MHbPd model, recency of the bugs is an important factor in estimating the defect probabilities. We incorporate a time decay factor into the defect probabilities to emphasize the recent bug fixes in the estimation of prior defect probabilities as follows:

$$P_{DHbPd}(f|C_k, R_k) = \frac{\sum_{i=1}^{k} e^{\frac{1}{\beta_2}(t_i - t_k)} I_b(f, r_i)}{\sum_{f' \in C_k} \sum_{i=1}^{k} e^{\frac{1}{\beta_1}(t_i - t_k)} I_b(f', r_i)}. \quad (6)$$

III. Document Retrieval Models

Our main goal is to score the files in the code base of a software project in order to rank them according to their relevance to a given bug. The models we have presented in the previous section estimates the prior probability of a file to be defective. Now we want to incorporate that prior probability into the bug localization process for an increased accuracy. Although in different contexts, incorporating query independent prior knowledge into IR systems has been extensively studied in the literature [19].

A. Document Priors

In the context of using document priors in a probabilistic retrieval framework, the Language Modeling (LM) framework of Ponte and Croft [20] and the Divergence From Randomness (DFR) framework of Amati and Risjbergen [21] are the two main approaches to text retrieval that have been shown to produce strong empirical results. While LM presents a probabilistic approach in the Bayesian framework, DFR is an information theoretic approach that evaluates the appropriateness of a document to a query on the basis of the divergence of document feature probabilities from pure non-discriminative random distributions.

In the Bayesian framework, given the description of a bug B as a query, we compute the posterior probability of a file f to pertain to the defective behavior by

$$P(f|B, R_k, C_k) = \frac{P(B|f, C_k)P(f|R_k, C_k)}{P(B)}. \quad (7)$$

Since we are only interested in ranking the files and the denominator in Eq. 7 does not depend on the files, it can be ignored. Taking the logarithm for computational convenience, we compute the final score of a file being relevant to a given bug with the prior belief by

$$\begin{aligned} s_{LM}(f|B, R_k, C_k) = log_2[P(B|f, C_k)] + \\ log_2[P(f|R_k, C_k)]. \end{aligned} \quad (8)$$

In the DFR framework, the final score of a file in response to a query is altered to take the prior belief into account as follows [22]:

$$\begin{aligned} s_{DFR}(f|B, R_k, C_k) = s_{DFR}(f|B, C_k) + \\ log_2[P(f|R_k, C_k)]. \end{aligned} \quad (9)$$

In the previous section, we presented the estimation of $P(f|R_k, C_k)$. In the following two subsections, we explain the estimation of $P(B|f, C_k)$ in LM and the estimation of $s_{DFR}(f|B, C_k)$ in the DFR framework.

B. Language Modeling

The language modeling approach uses the notion of query likelihood which ranks the documents in a collection according to the likelihood of relevance to the query. Given a bug B as a query, in order to compute the posterior defect probability of a file f, we need to compute $P(B|f)$.

The terms in the query, as well as in the documents, are regarded as occurring independently of one another (this being the *bag of words* assumption), therefore we can write

$$P(B|f) = \prod_{w \in B} P(w|f) \quad (10)$$

where $P(w|f)$ is the likelihood of a bug term w in f and it is computed with ML estimation. Given a term w from the vocabulary V of the collection C, the ML estimate we seek is given by $P_{ML}(w|f) = tf(w, f)/\sum_{w' \in f} tf(w', f)$ where $tf(w, f)$ is the term frequency of w in $f \in C$. Scoring the set of files in this way is problematic as all the query terms may not be present in a given document, leading to zero probabilities. To overcome this problem, several *smoothing* techniques have been proposed over the years [20]. Using the collection model for smoothing, Hiemstra Language Model (HLM) [20] computes the file likelihood of a term as follows:

$$P_{HLM}(w|f) = \lambda \cdot P_{ML}(w|f) + (1 - \lambda) \cdot P_{ML}(w|C) \quad (11)$$

where $P_{ML}(w|C)$ is the collection likelihood of the term and it is given by $P(w|C) = tf(w, C)/\sum_{w' \in V} tf(w', C)$ where $tf(w, C)$ represents the term frequency of w in the collection. The parameter λ is called the mixture variable and governs the amount of smoothing.

Another powerful smoothing approach is the Bayesian Smoothing with Dirichlet Priors [20]. If the Dirichlet parameters are chosen as $\mu P(w|C)$ for each term $w \in V$ then the file likelihood of a term is given by

$$P_{DLM}(w|f) = \frac{tf(w, f) + \mu P(w|C)}{\sum_{w' \in f} tf(w', f) + \mu} \quad (12)$$

where μ is the smoothing parameter. We denote this model by DLM (Dirichlet Language Model).

C. Divergence from Randomness

That brings us to the second major approach to document retrieval in probabilistic frameworks: the Divergence from Randomness (DFR) based approach. As mentioned earlier, DFR is an information theoretic approach that evaluates the appropriateness of a document to a query on the divergence of document feature probabilities from pure non-discriminative random distributions. The core idea in DFR is that the terms that do not help discriminate the documents in a collection are distributed randomly while the discriminative terms tend to appear densely only in a small set of *elite*[2] documents. These content bearing terms or the *specialty terms* should not follow a random distribution and the amount of divergence from randomness determines the discriminatory power of the term for retrieval. The higher the divergence, the higher the importance of the term in the retrieval. The DFR framework allows us to avoid parameter tuning to a great extent since the models are *non-parametric*.

In this framework, the score of a document with respect to a single query term is given by the product of two information content:

$$s_{DFR}(w, f) = [1 - Prob_2(w, f)] \cdot [-log_2 Prob_1(w, f)]. \quad (13)$$

[2]A document is regarded as an elite document for a term if the term appears at least once in the document.

$Prob_1$ is the probability of having tf occurrences of the term in the document by pure chance and as this probability decreases, the information content $-log_2 Prob_1$ of the document vis-a-vis the term increases. $(1 - Prob_2)$, on the other hand, is related to the risk of choosing the query term as a discriminative term and works as a normalization factor. Using different probability distributions in these two information contents results in different retrieval models.

Similar to the LM, the models we present from DFR also use the bag of words assumption. Therefore the score of a document with respect to a query is given by

$$s_{DFR}(f|B) = \sum_{w \in B} s_{DFR}(w, f). \quad (14)$$

Tf-Idf[3] Models for $Prob_1$: Assuming that the terms are being distributed in the documents randomly, having tf occurrences of a term in a document by pure chance is given by $Prob_1 = p^{tf}$ where p is the probability of a term to appear in any document. In order to compute the posterior distribution for p, usually a beta prior with parameters $\alpha_1 = -0.5$ and $\alpha_2 = -0.5$ is assumed. The evidence in computing p is given by the probability of the term to land in E elite documents out of N documents in a collection, which can be modeled by a binomial distribution $P(E|p, N) = \binom{N}{E} p^E \cdot (1-p)^{N-E}$. In this case, the posterior will also be in the form of the beta distribution and the expected value of p is given by $(E+0.5)/(N+1)$. Therefore,

$$-log_2 Prob_1 = tf \cdot log_2 \frac{N+1}{E+0.5}. \quad (15)$$

This information content is denoted by "In" (Inverse document frequency).

Using the expected number of elite documents E_e instead of E in the formula above results in a separate model "InExp" where $E_e = N \cdot P(tf \neq 0)$. If the probability of a term appearing in a document out of N documents is given by $1/N$, then $P(tf \neq 0) = 1 - (\frac{N-1}{N})^{TF}$ where TF is the total number of occurrences of the term in the collection.

Normalizing Information Content ($Prob_2$): $Prob_1$ by itself is not sufficient to accurately discriminate specialty terms since the terms with high frequencies will always produce small $Prob_1$ and thus become dominating during retrieval. To normalize the information content, two main methods have been proposed [21]. Normalization L[4] estimates the probability of having one more token of the same term in the document by $Prob_2 = tf/(tf + 1)$. Normalization B, on the other hand, assumes a new token of the same term added to the collection which already has TF tokens of the term. With this new token, the probability of having $tf + 1$ occurrences of the term in a document can be estimated by the binomial probability $Binom(E, TF + 1, tf + 1) = \binom{TF+1}{tf+1} p^{tf} \cdot q^{TF-tf}$ where $p = 1/E$ and $q = 1 - p$. Then the incremental rate between $Binom(E, TF, tf)$ and $Binom(E, TF + 1, tf + 1)$ gives the normalization factor:

$$Prob_2 = 1 - \frac{Binom(E, TF + 1, tf + 1)}{Binom(E, TF, tf)}. \quad (16)$$

Document Length Normalization: Document length is another important factor in retrieval. It has been shown that the relevancy of a document to a query is dependent on the document length [20].

Normalization 2 as proposed in [21] uses the assumption that the term frequency density in a document is a decreasing function of the document length. If the effective document length is chosen as the average document length in the collection, then the normalized term frequency is given by

$$tfn = tf \cdot log_2(1 + \frac{avg_l}{l}) \quad (17)$$

where l is the length of the document and avg_l is the average document length in the collection. Instead of the regular tf, tfn is used in the computation of $Prob_1$ and $Prob_2$.

TF-IDF Retrieval Models: Another set of widely used retrieval algorithms are grouped under the TF-IDF scheme [20]. We present this approach in the DFR framework as the structures of the scoring functions are the same. In this class of retrieval models, the score of a document with respect to a single query term is given by the multiplication of the term frequency with the inverse document frequency. One algorithm that produced strong empirical results in our experiments uses Robertson's tf which is given by

$$a_1 \cdot \frac{tf}{(tf + a_1 \cdot (1 - a_2 + a_2 \cdot l/avg_l))} \quad (18)$$

and Spark Jones' Idf which is given by $log_2(\frac{N}{E+1})$. The parameters a_1 and a_2 in Robertson's tf provide non-linearity to the term frequencies for scoring the documents.

Table II summarizes the models explained in this section.

Table II
Retrieval Models

Language Models (Section III-B)	
HLM	Hiemstra Language Model
DLM	Dirichlet Language Model
Divergence from Randomness (Section III-C)	
InB2	Inverse Document Frequency + Normalization B + Normalization 2
InExpB2	Inverse Document Frequency with expected number of elite documents + Normalization B + Normalization 2
InL2	Inverse Document Frequency + Normalization L + Normalization 2
Tf-Idf	Robertson's tf + Sparck Jones' Idf

[3]Tf-Idf stands for Term Frequency - Inverse Document Frequency

[4]L stands for Laplace as this probability is given by so-called Laplace's law of succession

IV. Experimental Evaluation

In order to evaluate the proposed algorithms, we need to have the complete repository of a software project with a set of documented bug descriptions Q. Unfortunately, software repositories and bug tracking databases are usually maintained separately [18]. Therefore, in general, we may not know the actual change-sets in the repository that are committed for implementing the fixes for bugs. One approach that has been used to get around this limitation is to look for pointers in the commit messages to the bug tracking database. The iBugs dataset created by Dallmeier and Zimmerman [23] uses this approach and it is therefore an appropriate testbed for our experiments.

A. Data Preparation for Bug Localization

We have evaluated the bug localization performance of our approach on AspectJ, an aspect-oriented extension to the Java programming language. The iBugs dataset for AspectJ contains a set of reported bugs, the collection of the files before and after a fix was implemented for each bug, and the files modified to implement the fix. We have used the pre-fix versions of the project as the corpora for our retrieval experiments.

Besides the iBugs dataset, we obtained the complete CVS repository of the AspectJ project which is publicly available[5]. Unfortunately, CVS repositories do not record the change-sets. In line with the approach presented in [24], we reconstruct the change-sets by grouping the files that are committed by the same author with the same commit message using a time fuzziness of 300 seconds. Table III gives various properties of the AspectJ project.

1) Indexing the Source Code: Since the code base of a software project keeps changing during development, in general, there can be significant differences in the underlying code base for any two bugs. In order to keep track of the changes in the code base, we create a distinct index for each collection that a bug $B \in Q$ was reported on. For indexing the source code and performing retrieval, we extended Terrier[6], a flexible open source platform developed in Java [22].

Modern software projects tend to include a substantial number of non-executable files such as configuration files, documentation files, help files, and so on. Therefore finding only the executable files may not constitute a complete answer to the bug localization problem. XML files, for example, are used heavily in project configuration and it may be necessary to make modifications to these files to fix certain bugs. For obvious reasons, we regard the files with the extensions ".java" and ".aj" as executable. The rest of the files in the code base that may also cause defective behavior are regarded as non-executable. Figure 1 depicts the evolving

[5]http://archive.eclipse.org/arch/tools-cvs.tgz

[6]http://terrier.org/

Table III
AspectJ Project Properties

K	$\|FI\|$	$\|BF\|$	$\|Q\|$	Analysis Period
6,271	5,165	1,214	291	2001-01-16 - 2008-10-06

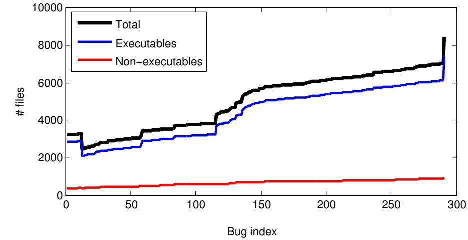

Figure 1. *The size of the AspectJ project as a function of the time of the bug report for each bug $B \in Q$.*

size of the AspectJ project during the bug reporting period. On average, the non-executable files constitute 13.58% of the source code. It is important to note these different types of files have different characteristics. So, as described below, they are subject to slightly different tokenization procedures.

Treating each source-code file as a *bag of words*, we first use an initial stop list to remove the programming language specific tokens such as "public," "private," "transient" etc. from the executable files. Then we split the compound terms that use camel-case and punctuation characters. After these steps, we apply an English stop list to remove the noise terms such as "a," "an," "the" and so on, and apply Porter's Stemming algorithm to reduce the terms to their root forms [25]. All the terms obtained through this process are then lowercased and incorporated in the index. The non-executable files, also considered to be a bag of words, are not subject to the programming-language-specific stop list. However, we split the compound tokens in such files also.

2) Preprocessing Bug Reports: Typically, a bug report is written in ordinary English and it may include tokens from the code base. For example, a bug report may include the trace of an exception caused to be thrown by the bug, or it may include method names that are possibly related to the defective behavior.

We apply the same preprocessing steps to the bug reports as we did for the non-executables in the code base. That is, we carry out compound-term splitting and use stopping and stemming rules to prune the reports. We also drop those bug reports that have no associated files in the code base. After this preprocessing, we ended up with 291 bug reports and 1124 files associated with them, implying 3.86 files per bug on average. Figure 2 illustrates the bug localization process.

B. Retrieval Results

The accuracy of search engines is commonly measured by precision and recall metrics [20]. Let's say we have a query for which there exist four relevant files in the collection that we want the search engine to retrieve. If three of the top

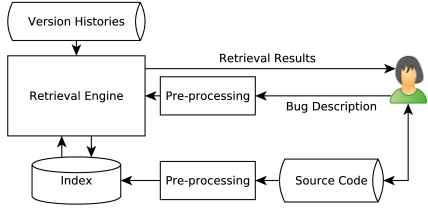

Figure 2. *An illustration of bug localization process.*

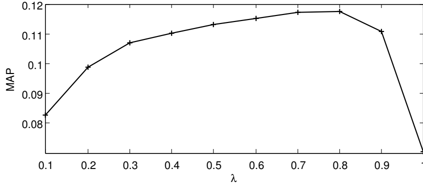

Figure 3. *The effect of varying λ in HLM.*

$t = 10$ retrieved files in the ranked list are relevant then the precision would be calculated as 3/10 and the recall as 3/4. A high precision indicates that a large fraction of the retrieved set is relevant. Recall, on the other hand, measures the completeness of the results.

We have tabulated the retrieval performance using mean precision at rank t ($P@t$), mean recall at rank t ($R@t$) and Mean Average Precision (MAP) metrics. The average precision (AveP) for a query $B \in Q$ is given by

$$AveP(B) = \frac{\sum_{t=1}^{T} P@t \times I(t)}{rel_B}, \tag{19}$$

where $I(t)$ is a binary function whose value is 1 when the file at rank t is a relevant file, and 0 otherwise. rel_B is the total number of relevant files in the collection for B. MAP is computed by taking the mean of the average precisions for all the queries.

The parameters λ for HLM and μ for DLM need to be tuned according to the characteristic of the underlying collection. Figure 3 and Figure 4 plot the retrieval accuracies in terms of MAP for various values of these parameters. For retrievals with HLM, we obtained the highest baseline accuracy with $\lambda = 0.8$ which assigns a higher weight to the file likelihoods as compared to the collection likelihoods of the query terms. On the other hand, the optimum value of μ for DLM is 2400, although the accuracies are not very sensitive to the variations in the $[2000 - 4000]$ interval. The constants a_1 and a_2 in the Tf-Idf model are set to 1.2 and 1.0 respectively. In all of the experiments with DLM, HLM and Tf-Idf, we used these fixed values for the parameters to solely compare the improvements with the proposed defect and modification based models for the priors.

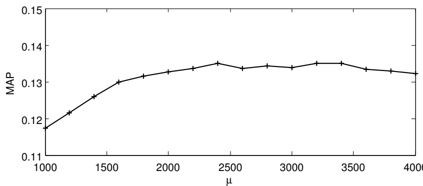

Figure 4. *The effect of varying μ in DLM.*

Table IV
Baseline Retrieval Results

Model	MAP	P@1	P@5	R@5
HLM ($\lambda = 0.8$)	0.1174	0.0859	0.0715	0.1607
DLM ($\mu = 2400$)	**0.1349**	**0.1271**	**0.0851**	**0.1642**
InB2	0.1240	0.1233	0.0774	0.1491
InExpB2	0.1327	0.1237	0.0735	0.1485
InL2	0.1268	0.0993	0.0719	0.1555
Tf-Idf ($a_1 = 1.2$, $a_2 = 1.0$)	0.1264	0.1062	0.0712	0.1488

Table V
Hypotheses

H1	Using the prior modification probabilities of the files (MHbP) in a software project enhances the bug localization accuracy.
H2	Using the prior defect probabilities of the files (DHbP) in a software project enhances the bug localization accuracy.
H1a	MHbPd outperforms MHbP when they are employed in IR based bug localization.
H2a	DHbPd outperforms DHbP when they are employed in IR based bug localization.
H3	Prior defect history is superior to prior modification history when they are employed in IR based bug localization.

1) Comparison of the Retrieval Models: Table IV presents the baseline retrieval performance across the models without incorporating the defect or the modification probabilities. That is, we assume a *uniform* prior for the results in Table IV. While DLM performs the best amongst the models, HLM performs the worst. The models InB2, InExpB2 and InL2 from the DFR framework do not require parameter tuning while performing as well as and sometimes better than the models in LM.

Table V presents the hypotheses we have formulated to investigate as to what extent using the defect and modification priors influences the retrieval of the files likely to be defective. The retrieval performance results themselves are presented in Tables VI and VII. We use pairwise student's t-test for significance testing. The columns p-H1, p-H1a, p-H2 and p-H2a in the tables give the computed p-value of the pairwise significance tests for the corresponding hypotheses. The highest score in each column is given in bold. We also report the improvement percentages for MAP and P@1 compared to the baseline results. Highest improvement in each column is designated by the '*' character.

All of the improvements in the retrieval results are statistically significant at a significance level of 1% i.e. $\alpha = 0.01$, therefore the null hypotheses for H1, H1a, H2 and H2a are rejected. As can be seen in Tables VI and VII, using the defect or the modification priors estimated from the version histories improves the bug localization accuracies signifi-

Table VI
Retrieval performances across the models with MHbP and MHbPd.

Model	MHbP				MHbPd ($\beta_1 = 1.0$)			
	MAP (Imp.)	P@1 (Imp.)	P@5	p-H1	MAP (Imp.)	P@1 (Imp.)	P@5	p-H1a
HLM	0.1318 (+12.27%)	0.0825 (-3.96%)	0.0859	8.64e-07	0.1924 (+63.88%)	0.1856 (+116.07%)	0.1313	4.14e-08
DLM	**0.1556** (+15.34%)	**0.1375** (+8.18%)	0.0907	7.30e-11	0.1896 (+40.55%)	0.2131 (+67.66%)	0.1340	5.11e-05
InB2	0.1329 (+6.83%)	0.1306 (+5.58%)	0.0818	2.90e-03	0.1704 (+36.98%)	0.1924 (+55.54%)	0.1141	4.91e-05
InExpB2	0.1447 (+9.04%)	0.1271 (+2.75%)	0.0838	2.49e-04	0.1975 (+48.83%)	0.2268 (+83.35%)	0.1265	4.24e-06
InL2	0.1481 (+16.43%)	0.1203 (+20.66%)*	**0.0914**	1.43e-04	0.2007 (+57.78%)	0.2337 (+134.40%)*	0.1388	3.07e-05
Tf-Idf	0.1508 (+18.93%)*	0.1271 (+19.34%)	0.0893	4.61e-06	**0.2121** (+67.27%)*	**0.2474** (+132.30%)	**0.1416**	3.50e-06

Table VII
Retrieval performances across the models with DHbP and DHbPd.

Model	DHbP				DHbPd ($\beta_2 = 5.0$)			
	MAP (Imp.)	P@1 (Imp.)	P@5	p-H2	MAP (Imp.)	P@1 (Imp.)	P@5	p-H2a
HLM	0.1474 (+25.55%)	0.1100 (+28.06%)	0.0962	9.28e-09	0.2114 (+80.07%)*	0.2199 (+156.00%)*	0.1326	3.84e-10
DLM	**0.1660** (+23.05%)	**0.1546** (+21.64%)	0.0928	5.51e-07	0.2041 (+51.30%)	0.2268 (+78.44%)	0.1423	1.50e-03
InB2	0.1500 (+20.58%)	0.1409 (+13.90%)	0.0907	2.73e-06	0.1847 (+48.47%)	0.2165 (+75.02%)	0.1175	1.93e-07
InExpB2	0.1592 (+19.97%)	0.1512 (+22.23%)	0.0955	1.18e-05	0.2047 (+54.26%)	0.2268 (+83.35%)	0.1320	4.29e-07
InL2	0.1640 (+28.93%)	0.1409 (+41.32%)*	0.1031	3.06e-07	0.2194 (+72.48%)	0.2509 (+151.65%)	0.1471	5.78e-07
Tf-Idf	0.1651 (+30.21%)*	0.1409 (+32.30%)	**0.1065**	2.96e-08	**0.2258** (+78.08%)	**0.2646** (+148.45%)	**0.1512**	1.77e-07

cantly. Especially the amount of improvement in precision is extremely high with highest improvement being 156% in P@1 for the HLM model incorporating DHbPd.

The InL2 model from the DFR framework stands out in the experiments. This model does not need any parameter tuning and it performs comparably well, gaining the highest improvements in P@1 with MHbP, MHbPd and DHbP, and second to the highest improvement in P@1 with DHbPd. With a MAP value of 0.2258, we obtained the highest retrieval accuracy with the Tf-Idf model incorporating DHbPd. The InL2 model incorporating DHbPd came out as the second best with a MAP of 0.2194.

2) MHbP vs. DHbP: Intuitively, we expect the BF change-sets to be more descriptive in terms of the defect potential for the files. Indeed, comparing the models using the prior defect history with the models using the prior modification history, we observe that DHbP consistently outperforms MHbP at a significance level of 1% i.e. $\alpha = 0.01$ for all the models except DLM for which the p-value is 0.026, indicating a significance level of 5%. From these results, we conclude that using the defect histories of the files results in superior bug localization in comparison to the modification histories. Therefore, the null hypothesis for H3 is rejected.

3) Discussion: It is reasonable to assume that the effects of modification and bug-occurrence events associated with a file should decay with time. Obviously, as developers fix the faulty parts of a software project in response to the reported bugs, some files may have caused the bugs to occur on just a one-off basis, while others may require repeated fixes for the same set of bugs. So, one could argue that the weight given to a file in relation to a given bug with just a one-off occurrence of the bug that was fixed a long time ago

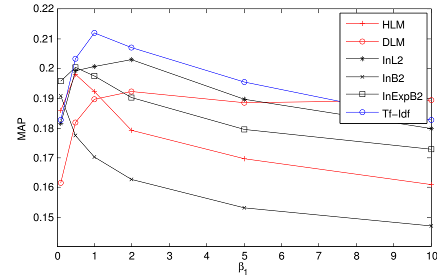

Figure 5. *The effect of varying β_1 in MHbPd.*

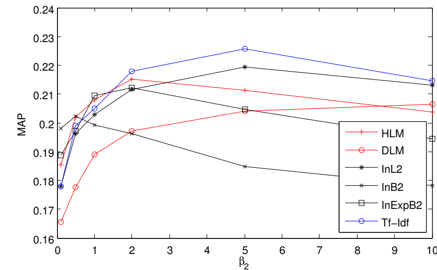

Figure 6. *The effect of varying β_2 in DHbPd.*

should be low and this weight should become even lower with the passage of time. On the other hand, a file requiring repeated fixes for a bug should get a higher weight as being a likely source of the same bug again and this weight should diminish only slowly with time.

As explained in Section II-C, we incorporate time decay through the parameters β_1 and β_2. Figure 5 and Figure 6 plot the retrieval accuracies for several different values of these parameters. Retrieval with the Tf-Idf model performed the best with $\beta_1 = 1.0$ and $\beta_2 = 5.0$.

Table VIII
Recall with InL2 incorporating version histories

Model	R@5 (0.1%)	R@10 (0.2%)	R@25 (0.5%)	R@50 (1%)	R@100 (2%)
MHbP	0.1926	0.2644	0.3966	0.5040	0.6588
DHbP	0.2048	0.2837	0.4189	0.5231	0.6676
MHbPd($\beta_1 = 1.0$)	0.2392	0.3170	0.4478	0.5388	0.6421
DHbPd($\beta_2 = 5.0$)	0.2519	0.3532	0.4750	0.6025	0.6863
Baseline	0.1555	0.2351	0.3468	0.4485	0.5745

What is interesting is that the optimum value for the decay parameter β_1 is always less than the optimum value for β_2 that resulted in the highest MAP for the analyzed models as can be seen in Figure 5 and Figure 6. These results suggest that the expected stabilization time of the BF change-sets tends to be longer than that of the non-BF change-sets, i.e. the bug fixes take longer to be finally resolved.

4) Completeness of Retrievals: Table VIII presents the recall values at several cut-off points in the ranked list with the proposed models for the priors. The row "Baseline" presents the recall results with a uniform prior. Since the size of the source code is not the same for each bug, we use the average size of the source code to designate the percentage of the code at the reported ranks in parentheses. Here, we only report the results for the InL2 model because of space limitations. By analyzing 1% of the code on the average, 60.25% of the buggy files were localized with the InL2 model incorporating DHbPd.

C. Comparison with Relevant Work

Using the iBugs dataset, Dallmeier and Zimmerman [23] experimented with FindBugs [15], a static bug pattern detection tool, and Ample [14], a dynamic bug localization tool. These experiments allow us to indirectly compare our results with those obtained through static and dynamic analysis. Their evaluation shows that FindBugs was not able to locate any of the 369 bugs in the iBugs dataset. On the other hand, the experiments with Ample in [14] are restricted to the 44 bugs that require a single class to be fixed and that have at least one failing test. Ample locates 40% of those bugs by searching at most 10% of the executed classes. For comparison with our work, notice that P@1 for the Tf-Idf model incorporating DHbPd is 0.2646, which indicates that, for 77 of the 291 bugs, the first file in the retrieved list is actually a relevant file. These results clearly show that our approach works extremely well for a larger set of bugs.

Another relevant bug localization tool is BugScout by Nguyen et al. [6]. Their experiments on AspectJ are restricted to a single collection of 978 files and 271 bug reports. The accuracy of BugScout is reported in terms of *hitrate*. If BugScout correctly retrieves at least one relevant file for a bug in a ranked list of a certain size, it is considered to be a *hit*. The hitrate for a project is given by the ratio of the total number of hits to the total number of bugs. BugScout's hitrate for AspectJ with a ranked list of 10 files is reported as 35%. For a comparison with our work, note that the hitrate with a ranked list of 10 files for the InL2 model incorporating DHbPd is 63.5%, indicating more than 80% improvement. Additionally, P@10 for InL2 incorporating DHbPd in our experiments is 0.1065. This result indicates that, on the average, the developer is guaranteed to locate a relevant file with our approach if s/he is willing to examine the top 10 files.

V. Threats to Validity

Although our retrieval framework offers a lightweight and highly scalable solution to IR based bug localization, we must mention the following limitations. (1) The accuracy of the proposed approaches relies heavily on the quality of the bug reports. The software development background of the person filing a bug report obviously has a great deal to do with the quality of the report. (2) The language modeling approaches and the time decay models require the tuning of the parameters involved. Depending on the nature of a dataset, the optimum values for these parameters may vary significantly from project to project. (3) We have evaluated the performance of the proposed approaches on only the AspectJ project. Although this project is commonly used in bug localization studies, the accuracy of the retrievals on other projects is open to debate.

VI. Conclusion & Future Work

We presented a theoretically well-principled approach for incorporating version histories of software files in an IR based framework for bug localization. Our approach used the information stored in software versioning tools regarding the frequency with which a file is associated with defects and its modifications in order to construct estimates for the prior probability of any given file to be the source of a bug. Incorporating these priors in an IR based retrieval framework, as we have done, significantly improves the retrieval performance for bug localization. What is even more remarkable is that when we associate a time decay factor with the priors, the improvement in bug localization goes up even more dramatically. For the retrieval itself, our work used two different algorithms, one based on Bayesian reasoning and other on the Divergence from Randomness principle.

Regarding future extensions of our work, the following two come to mind immediately: (1) Validating our bug localization approach with other software repositories; and (2) Using our formalism for feature and concept localization in software repositories.

Acknowledgments

This work was supported by Infosys. Many thanks to Gaurav Srivastava, Chad Aeschliman and Shivani Rao for their valuable comments.

REFERENCES

[1] G. Canfora and L. Cerulo, "Impact analysis by mining software and change request repositories," in *Software Metrics, 2005. 11th IEEE International Symposium*, sept. 2005, pp. 9 pp. –29.

[2] S. Lukins, N. Kraft, and L. Etzkorn, "Source code retrieval for bug localization using latent dirichlet allocation," in *Reverse Engineering, 2008. WCRE '08. 15th Working Conference on*, oct. 2008, pp. 155 –164.

[3] B. Ashok, J. Joy, H. Liang, S. Rajamani, G. Srinivasa, and V. Vangala, "Debugadvisor: a recommender system for debugging," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2009, pp. 373–382.

[4] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models," in *Proceeding of the 8th working conference on Mining software repositories*. ACM, 2011, pp. 43–52.

[5] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. Collard, "Blending conceptual and evolutionary couplings to support change impact analysis in source code," in *Reverse Engineering (WCRE), 2010 17th Working Conference on*, oct. 2010, pp. 119 –128.

[6] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. Nguyen, "A topic-based approach for narrowing the search space of buggy files from a bug report," in *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, nov. 2011, pp. 263 –272.

[7] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*. IEEE, 2008, pp. 181–190.

[8] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. Ieee, 2010, pp. 31–41.

[9] A. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 78–88.

[10] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, "Change bursts as defect predictors," in *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*. Ieee, 2010, pp. 309–318.

[11] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering*, pp. 429–445, 2005.

[12] A. Hassan and R. Holt, "The top ten list: Dynamic fault prediction," in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. IEEE, 2005, pp. 263–272.

[13] A. Bernstein, J. Ekanayake, and M. Pinzger, "Improving defect prediction using temporal features and non linear models," in *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*. ACM, 2007, pp. 11–18.

[14] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight bug localization with ample," in *Proceedings of the sixth international symposium on Automated analysis-driven debugging*. ACM, 2005, pp. 99–104.

[15] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM SIGPLAN Notices*, vol. 39, no. 12, pp. 92–106, 2004.

[16] A. Hindle, D. German, and R. Holt, "What do large commits tell us?: a taxonomical study of large commits," in *Proceedings of the 2008 international working conference on Mining software repositories*. ACM, 2008, pp. 99–108.

[17] A. Mockus and L. Votta, "Identifying reasons for software changes using historic databases," in *Software Maintenance, 2000. Proceedings. International Conference on*, 2000, pp. 120 –130.

[18] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The missing links: Bugs and bug-fix commits," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2010, pp. 97–106.

[19] W. Kraaij, T. Westerveld, and D. Hiemstra, "The importance of prior probabilities for entry page search," in *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2002, pp. 27–34.

[20] C. Manning, P. Raghavan, and H. Schutze, *Introduction to information retrieval*. Cambridge University Press Cambridge, 2008, vol. 1.

[21] G. Amati and C. Van Rijsbergen, "Probabilistic models of information retrieval based on measuring the divergence from randomness," *ACM Transactions on Information Systems (TOIS)*, vol. 20, no. 4, pp. 357–389, 2002.

[22] C. Macdonald, B. He, V. Plachouras, and I. Ounis, "University of glasgow at trec 2005: Experiments in terabyte and enterprise tracks with terrier," in *Proceedings of TREC 2005*, 2005.

[23] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 433–436.

[24] T. Zimmermann and P. Weißgerber, "Preprocessing cvs data for fine-grained analysis," in *Proceedings 1st International Workshop on Mining Software Repositories (MSR 2004)*, 2004, pp. 2–6.

[25] M. Porter, "An algorithm for suffix stripping," *Program: electronic library and information systems*, vol. 14, no. 3, pp. 130–137, 1980.