An Incremental Update Framework for Efficient Retrieval from Software Libraries for Bug Localization

Shivani Rao, Henry Medeiros, and Avinash Kak
School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN, USA
{sgrao, hmedeiro, kak}@purdue.edu

Abstract*—Information Retrieval (IR) based bug localization techniques use a bug reports to query a software repository to retrieve relevant source files. These techniques index the source files in the software repository and train a model which is then queried for retrieval purposes. Much of the current research is focused on improving the retrieval effectiveness of these methods. However, little consideration has been given to the efficiency of such approaches for software repositories that are constantly evolving. As the software repository evolves, the index creation and model learning have to be repeated to ensure accuracy of retrieval for each new bug. In doing so, the query latency may be unreasonably high, and also, re-computing the index and the model for files that did not change is computationally redundant. We propose an incremental update framework to continuously update the index and the model using the changes made at each commit. We demonstrate that the same retrieval accuracy can be achieved but with a fraction of the time needed by current approaches. Our results are based on two basic IR modeling techniques - Vector Space Model (VSM) and Smoothed Unigram Model (SUM). The dataset we used in our validation experiments was created by tracking commit history of AspectJ and JodaTime software libraries over a span of 10 years.

I. Introduction

IR based bug localization techniques follow a multi-step process shown in Figure 1 in order to identify source files relevant to a bug [1]. The raw source files are first preprocessed and subsequently indexed to create an internal representation of the source files. The index is then used to learn the parameters of an IR model chosen to represent the software repository. The bug report is preprocessed in the same manner as the files and used to query the software repository through its IR model.

Much research effort has gone into improving the retrieval accuracy of these algorithms, by using sophisticated text models [2][3], incorporation of additional information such as version histories [4], bug-fixing history [5], class relationships [6] and so on. Effort has also gone into studying and improving the quality of the query to improve retrieval accuracy [1][7][8][9][10]. In other words, research in the area of IR based bug localization has focused primarily on retrieval effectiveness.

However, the current approaches to IR based bug localization are not efficient for continuously changing software repositories. Software systems are constantly evolving for a variety of reasons such as bug fixes, removal of security vulnerabilities, addition of features, adaptations to operating system changes or new software/hardware architectures, etc.

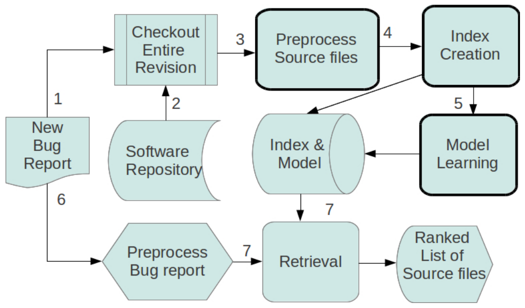


Fig. 1. A typical bug localization process shown for a single bug.

These changes cause addition, deletion or modification of source files causing the index and the model to be out of sync with the software repository. The straight forward approach to ensuring retrieval accuracy is to re-create the index and the model for each bug that needs to be localized on a newer version of the software. Henceforth, we refer to this approach as the *batch mode* approach. However, this can be time consuming and can lead to high query latency[1]. Alternatively, one could re-compute the index and the model at major releases and ignore commit-level changes (henceforth, we refer to this approach as the *limited update*). With this approach it is not possible to guarantee retrieval accuracy as a retrieval carried out using an out dated index or model may be meaningless or erroneous. In other words, with the current state of the art IR based bug localization techniques, it is not possible to achieve effective and efficient retrieval at the same time.

In this paper, we propose an *incremental approach* to IR based bug localization that achieves the same retrieval accuracy as that of *batch mode approach* at significantly lower query latency. We draw inspiration from the incremental indexing techniques developed for web collections in the domain of text-retrieval [12][13]. The proposed framework continuously updates the index and the model using only the source files changed in each commit — these files are typically referred to as the *change set*. Since the size of the *change-set* is generally lower than the size of the software repository, the computational cost of keeping the index and the model updated is expected to reduce considerably. Additionally, since

[1]Query latency is the amount of time a user needs to wait before the retrieved list is available [11].

the index and the model do not need to be re-computed before querying for each new bug, the query latency is greatly reduced as well.

We demonstrate the effectiveness of our method by comparing the retrieval accuracy obtained with the *incremental approach* and the *batch mode approach* using two different but commonly used text modeling methods, Vector Space Model (VSM) and Smoothed Unigram Model (SUM). We have also carried out rigorous empirical evaluation of the time-savings in various stages of the retrieval process to demonstrate the efficiency of our approach. Evaluation has been carried on a benchmark dataset called `moreBugs`[2], that tracks commit-level changes over 10 years of history of two software systems: AspectJ and JodaTime. In the next section, we present some motivating examples to emphasize the need for an *incremental approach* to IR based bug localization.

II. The need for incremental approach to Bug Localization

Recall that, in order to localize a bug using an IR based bug localization technique, the following four steps need to be executed (as shown in Figure 1): (a) Text Preprocessing (b) Index Creation (c) Model Learning and (d) Retrieval. Table I shows the time taken by each of the above steps for three typical bugs in the JodaTime and AspectJ software repository. It is worthwhile to note that the text preprocessing and the index creation are the most time-consuming parts of the process. The query latency increases more than 10 times as the size of the repository grows from 486 files to 7594.

For an evolving software repository, the *batch mode* approach ensures accurate retrieval by repeating the above steps for each bug filed on a newer version of the software. However, this approach is sub-optimal in terms of computational effort, since each new commit is likely to change only a small portion of the code base. We illustrate this very important fact through the results we obtained by mining 7477 revisions of AspectJ spanning 10 years of its developmental history. Figure 2 displays a histogram of the number of source files affected (added, deleted, modified, renamed, or copied) at each commit during these 10 years. As shown in the figure, it is unlikely that more than 5 source files are changed in a single commit. Thus, recomputing the index and the model from scratch for each new bug is not only computationally expensive but also sub-optimal.

The reader may argue that, given that a commit affects only a small portion of the code-base, one can ignore commit-level changes and update the index and re-compute the model only at major software releases. While this *limited update* approach would obviously reduce the query latency for a bug, it would be at the cost of retrieval accuracy. In Table II we compare retrieval precision of *batch mode* approach with the *limited update* for sample bugs in JodaTime and AspectJ software repositories. As shown in Table II, for some queries a reduction in retrieval precision by up to 98% may occur when the model is re-computed only at major releases[3]. Furthermore, if a bug report relates to source files introduced into the repository after the model and the index were created last, any file retrievals in response to that bug would be meaningless. In our evaluation dataset, we found that for about 67/321 (20%) bugs in AspectJ and 4/43 (9%) bugs in JodaTime, at least one of the relevant source files was found missing in the index built on a previous release. Thus, *limited update* approach to re-computing the index and model at major releases may not guarantee retrieval accuracy.

In summary, it would be ideal to keep the software repository in sync with the index and the model as this would ensure accurate retrieval as well as low query latency.

- The problem with current approaches is that they cannot simultaneously achieve these goals.
- We look for avenues of optimization in the current method by eliminating repeated computation on the source files that did not change after previous model learning. Efficiency can be achieved if incremental methods are used only on the source files changing with each commit. Furthermore, query latency is reduced as index and the model need not be computed before responding to each query.

The incremental update framework proposed in this paper starts with an index and a model built by a batch-mode algorithm (as shown in the next section).

III. Current approach to IR based Bug Localization

In this section we expand on the *batch mode* retrieval process shown in Figure 1 with focus on the steps that can be optimized using *incremental update*.

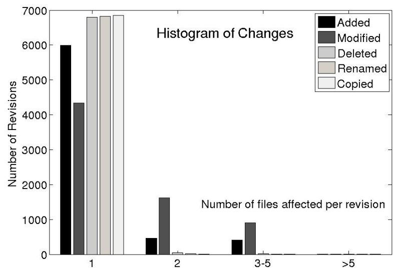


Fig. 2. Modification statistics over 7477 revisions of the AspectJ software in the `moreBugs` repository.

TABLE II
Comparing retrieval accuracy using Average Precision for sample bugs in JodaTime and AspectJ softwares using a model that is updated only at software releases (column labeled as *limited update*) with the *batch mode* learned model. Last column shows the % reduction in retrieval accuracy.

Software	Model	BugID	Batch Mode	Limited Update	% Reduction
JodaTime	SUM	3520651	0.1060	0.0788	25.66
AspectJ	VSM	33011	0.1438	0.0800	44.37
AspectJ	VSM	75129	0.2250	0.0688	69.42
JodaTime	SUM	3161586	0.1820	0.0516	71.66
JodaTime	VSM	1887104	0.5147	0.0406	92.11
JodaTime	SUM	2461322	1.0000	0.0625	93.75
AspectJ	VSM	70794	0.2536	0.0031	98.78

[2]Our benchmark dataset is being made publicly available at https://engineering.purdue.edu/RVL/Database/moreBugs/.

[3]A more detailed analysis is presented in Section VII-A2

TABLE I
TYPICAL TIME SPENT (IN SECONDS) IN DIFFERENT STAGES OF THE RETRIEVAL PROCESS FOR SOFTWARE REPOSITORIES OF DIFFERENT SIZES.

bug ID	Dataset	Number of source files	Number of terms	Model	Pre-processing (a) (in seconds)	Indexing (b) (in seconds)	Model Learning (c) (in seconds)	Retrieval (d) (in seconds)	Query latency (in minutes)
178828	JodaTime	486	10,824	VSM	264.70	37.06	0.69	0.23	5.04
3192457	JodaTime	864	12,174	VSM	603.17	92.70	0.89	0.42	11.62
371684	AspectJ	7,594	40,256	VSM	2942.11	228.47	2.869	2.43	52.93

A. Text Preprocessing and Index Creation

For each new bug, *all* the source files in the bug's prefix revision of the repository are checked out. The text preprocessing of the all the source files in the repository is carried out using with the following steps. First, the source files are tokenized and Unicode character strings, numerical literals, etc. are deleted. Next, the identifier names formed by concatenation of terms (e.g. "PrintHandler", "Print_Request", "submitcommand") are split into more generic terms [2]. This is followed by stop-word removal in which commonly occurring programming language constructs like "for" and "while" are dropped. The same is done to other commonly occurring words of the English language, such "the," "for," "up", "on", etc. Finally, all surviving tokens are stemmed to their roots. The preprocessed source files are then used to create an *index*. The *index* contains the vocabulary (denoted by $\mathcal{V}$) extracted from the entire repository and an internal representation for each of the source files. One such representation is the term-document matrix A, where the source files correspond to the columns and the terms to the rows. If there are M source files in the repository, A is a $|\mathcal{V}| \times M$ term-document matrix and $A_m(w)$ denotes the frequency count of the w^{th} word in the vocabulary $\mathcal{V}$ of the m^{th} source file.

B. Learning parameters of the text model

Any text model of a software repository typically contains the document-level parameters and collection-level parameters. The document-level parameters refer to the distribution of the terms in each source file. On the other hand, the parameters that exist at the collection-level characterize the distribution of the terms over *all* the source files. For both models studied in this paper: VSM and SUM, we show how these collection-level and document-level parameters are learned from the index. For retrieval, a bug report's textual fields like the title (summary) and description are extracted and preprocessed to construct a query which can then be represented as a $|\mathcal{V}|$-dimensional vector A_q.

1) Vector Space Model (VSM): The VSM model [14] associates three different frequencies with a term, the *term frequency*, the *document frequency*, and the *inverse document frequency*. The term frequency vector representation of m^{th} source file (d_m) is nothing but the m^{th} column of A ($\overline{A}_m$). The document frequency, denoted $df(w)$, is the *number of documents* that contain the term indexed at w. In other words, $df(w) = |m : A_m(w) > 0|$. The inverse document frequency is denoted $idf(w)$, and is given by:

$$idf(w) = log(\frac{M}{df(w) + 1}) \tag{1}$$

The terms that are common to all documents will have a $df \approx M$ value and an $idf \approx 0$. Thus, the idf value exercises control over the relative importance of the terms in a document with regard to how well they help in distinguishing this document from other documents. A source file d_m is represented in the VSM model as a weighted vector (called as the *tf-idf* representation) of length $|\mathcal{V}|$, where each term $A_m(w)$ is weighted by $idf(w)$: $A_m(w)idf(w)$. Similarly, the query is represented by a $|\mathcal{V}|$- dimensional term-frequency vector A_q and weighted by the *idf* values as follows: $A_q(w)idf(w)$. A cosine similarity between the two vectors is used to compute the score and this score is used for ranking the source file vis-à-vis the query. Owing to its simplicity, the VSM model has been used extensively for retrieval from software libraries [1][5][6][15][16].

2) Smoothed Unigram Model (SUM): The SUM [17] fits a single multinomial distribution to the term frequencies in each file. The representation of the m^{th} source file (d_m) under SUM is a $|\mathcal{V}|$-dimensional probability vector $p_{uni}(w|d_m)$ whose elements must add up to 1, that is,

$$\begin{aligned} p_{uni}(w|d_m) &= \mu \frac{A_m(w)}{dl(m)} + (1-\mu)p_c(w) \\ p_c(w) &= \frac{cf(w)}{\sum_w cf(w)} \\ cf(w) &= \sum_m A_m(w) \quad \& \quad dl(m) = \sum_w A_m(w) \end{aligned} \tag{2}$$

where $dl(m)$ is the length of d_m. $p_c(w)$ is called the *collection model* that represents the multinomial distribution of the terms over the entire collection of source files. $p_c(w)$ is the normalized version of the collection-wide term frequencies cf. μ is the smoothing parameter that controls the degree of importance given to the term frequencies in the document and to the collection-wide term frequencies. The query can be represented in a similar fashion as a $|\mathcal{V}|$-dimensional representation, by smoothing the normalized term frequencies $A_q(w)$ with the collection model $p_c(w)$. A Kullback−Leibler (KL) divergence between the two probability vectors is used to compute the retrieval score of the source files vis-à-vis the query [18]. Due to its simplicity and robustness, the SUM has gained popularity for numerous software engineering (SE) problems like bug localization [2], program comprehension [19], concern location [20] and so on.

IV. THE PROPOSED APPROACH

In this section, we present our *incremental framework* to update the index and the incremental algorithms to update the model parameters. Figure 3 shows the main steps of this proposed framework. In this framework, the index and the model once created are not re-computed but incrementally updated as the software evolves. In the rest of the paper we use A^t to indicate the state of the term-document matrix after the

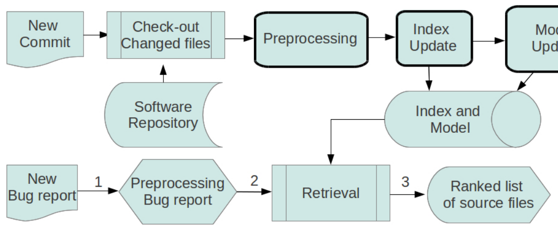


Fig. 3. Incremental update framework for bug localization.

t^{th} commit operation. Since both SUM and VSM are linear models, the update equations shown below yield a model that is a replica of the model learned from batch-mode techniques.

A. Change Preprocessing and Index Update

For each new commit, the source files in the *change-set* are checked-out and subject to the preprocessing steps described in Section III-A. Recall that the columns of A^t correspond to the source files $A^t = [A^t_1 A^t_2 ... A^t_M]$, and thus changes to the software alters the term-document matrix as follows:

- Addition: $A^{t+1} = [A^t Add]$.
- When the j^{th} source file is modified: $A^{t+1} = [A^t_1 A^t_2 ... A^{t+1}_j ... A^t_M]$
- When the j^{th} source file is deleted: $A^{t+1} = [A^t_1 A^t_2 ... \mathbf{0} ... A^t_M]$

In general, a given commit may involve a combination of the above changes. Although, not shown explicitly in the notation, new terms may be added to the index which increases the number of rows of the term-document matrix. If there are M_a new source files added and $|\mathcal{V}_a|$ new terms then the resulting A^{t+1} is of size $\{|\mathcal{V}| + |\mathcal{V}_a|\} \times \{M + M_a\}$.

B. Text Models and the Incremental Update formulas

In this section we show how updates to the index and the term-document matrix affect the document-level and collection level parameters for the two models: VSM and SUM as the software evolves.

1) Vector Space Model (VSM): Due to the simplicity of the VSM model, only the collection-level parameters $df(w)$ and $idf(w)$ need to be updated as the software evolves. The document-level parameters of the source files are simply the columns of the term-document matrix (A), which are updated with the index. As is obvious from Eq. 1, in order to update idf incrementally, we just need to keep the df updated. For each source file d_m that is affected, and each term w in that source file, the df is updated by the following equation:

$$df^{t+1}(w) = df^t(w) + sign(A^{t+1}_m(w) - A^t_m(w)) \qquad (3)$$

where, $sign(x) = 1$ if $x > 0$, -1 if $x < 0$ and 0 if $x = 0$. If a source file is added to the repository, then for each unique term in each such file, we simply increment the value of df for that term by 1 as $A^t_m(w) = 0$ for such cases. For each file that is deleted from the repository, and for each unique term in the file, we decrement the value of df for that term by 1 as $A^{t+1}_m(w) = 0$ for such cases. Last but not the least, for each file that is modified, the above formula can be used as-is for updating the df values.

2) Smoothed Unigram Model (SUM): With regard to updating the SUM incrementally, we just need to keep cf and the dl updated since the rest of the probabilities can be calculated from these by normalization. The logic for updating cf and dl incrementally is exactly the same as presented previously for updating df incrementally. The only difference is that unlike the df, the cf and dl are increased by the actual frequency count of the term in the source files. For the m^{th} source file that is affected, we can use the following formulas to update these count variables:

$$\begin{aligned} cf^{t+1}(w) &= cf^t(w) + A^{t+1}_m(w) - A^t_m(w) \\ dl^{t+1}(m) &= dl^t(m) + A^{t+1}_m(w) - A^t_m(w) \end{aligned} \qquad (4)$$

C. Retrieval for a query

With the proposed framework the index and the model are always in sync with the underlying software. Retrieval for a new bug report now consists of merely two steps (a) preprocess the bug report using the steps mentioned in Section III-A, (b) retrieve the source files that are relevant to the query by matching the two entities in the model-space. The query latency is reduced because there is no overhead of computing the index and the model before retrieval. Since the size of the *change-set* is typically small, we can also expect to save time spent on most of the stages of the retrieval process namely preprocessing, index creation and model learning.

V. Time Complexity Analysis

For both SUM and VSM models, the computational complexity of the update for each document is proportional to the size of the vocabulary i.e. $O(|\mathcal{V}|)$ or $O(u)$, where u is the average number of terms per source file. If there are n files modified/added/deleted in a commit, then the overall complexity of *incremental update* is $O(n \times |\mathcal{V}|)$ or $O(nu)$. The corresponding time for *batch mode* algorithm for both models is $O(M|\mathcal{V}|)$ or $O(Mu)$.

VI. Experimental Validation

A. The Evaluation Dataset

In order to evaluate an automatic bug localization framework, one needs a set of closed/resolved issues/bugs for a particular software system and for each of these bugs the following information: (a) the bug report's textual content like title, description, comments and so on; (b) the source files that were fixed in order to resolve the bug (we call this list of sources files the *relevance list* for the bug); and (c) the prefix-snapshot of the software repository. Thanks to the availability of open-source software code-bases (like Mozilla, Rhino, JodaTime, Eclipse, Chrome etc.), researchers have successfully mined the bug-tracking systems and the version control systems associated with these projects and linked them together in order to collect the necessary data for the evaluation datasets [5][21]. Although these benchmark datasets are useful and have been used extensively in IR based bug localization research, they lack the commit-level changes that are needed to evaluate our *incremental update* framework.

We have therefore created a new and publicly available benchmark dataset called `moreBugs` [22] by mining ten years of commit history for AspectJ and JodaTime projects.

TABLE III
moreBugs SPECIFICATIONS

	AspectJ	JodaTime
Version Control System	Git	Git
Number of tags/releases	77	32
Number of revisions	7477	1537
Total duration of the project analyzed	Dec'02-Feb'12	Dec'03-June'12
Average number of source files/bug	5214	556
Bug tracking system	Bugzilla	SourceForge
Number of bugs used for evaluation	321	43

Based on 7477 revisions for AspectJ and 1573 revisions for JodaTime, the dataset contains the commit-level changes and the release history for the two software libraries. Table III displays quantitatively the contents of `moreBugs`. A technical report detailing the creation of the dataset as well as how to obtain free public access to the same is available through https://engineering.purdue.edu/RVL/Database/moreBugs/.

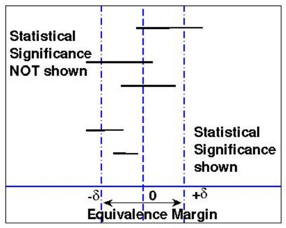
B. Evaluation metrics for the Incremental Update Framework

We have evaluated the *incremental update* algorithm using two types of metrics. The first kind compares the retrieval accuracy obtained using the incrementally updated model with that obtained using the true model (which can be acquired at any time through *batch mode* learning). The second kind compares the computational effort in terms of the time spent on preprocessing, index creation and update, model creation and update, and the retrieval time using the two approaches.

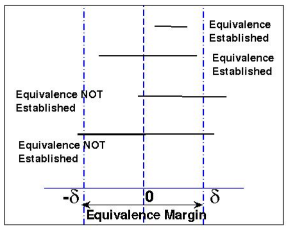
1) Measuring Retrieval Accuracy: The metrics used to evaluate retrieval accuracy of a search engine are computed by examining the ranked list of the source files returned by it in response to a query. The set of the top N_r source files in the ranked list is called the *retrieved set* which is compared with the *relevance list* to compute the following metrics. For a more focused assessment of the retrieval performance, one computes *Precision*-at-rank-r, denoted $P@N_r$, to measure the fraction of the set of retrieved files up to rank r that was deemed relevant. By the same token, *Recall*-at-rank-r, denoted $R@N_r$, measures the fraction of the set of all relevant source files that were retrieved up to rank r. In this paper, we report P@1, P@5, P@10 and R@1, R@5 R@10. An overall metric of retrieval accuracy, known as *Average Precision* (AP), is defined as the area under the Precision-Recall curve. AP has essentially the same significance as that of Precision. The higher the value of AP the better the retrieval engine. We report Mean Average Precision (MAP), which is the average of the AP values over all the bugs in a database.

Another way to gauge retrieval accuracy is by using rank-based metrics. This measure computes the number of bugs for which at least one relevant source file was retrieved at rank r [3]. For example, the rank measure at $r = 1$ is the number of bugs that were localized correctly by retrieval of at least one relevant source file at rank 1. In our validation experiments, we have presented rank measures for the following values of r: $r = 1$, $2 \leq r \leq 5$, $6 \leq r \leq 10$ and $r > 10$.

2) Note on Statistical Significance Testing: To guard against the noise introduced in the retrieval performance by the quality of the query and the variability in the completeness


Statistical Significance NOT shown
Statistical Significance shown
-δ
0
+δ
Equivalence Margin

(a) Standard significance testing



(b) Equivalence testing using TOST.

Fig. 4. Standardized t-test to prove that there is a statistical difference (a) and equivalence tests to show that the two algorithms are equivalent within a margin of δ (b).

of the relevance list, it is recommended that an AP-based result be subject to statistical significance testing [23]. These tests are designed to show that the retrieval performance (as measured by MAP) computed by the proposed algorithm (Algorithm B) is superior (or not similar) to the state-of-art approach (Algorithm A). Using standard statistical significance testing, the null hypothesis H_0 attempts to show that the two algorithms are the same ($dMAP = MAP_A - MAP_B = 0$) and then reject the null hypothesis in favor of the alternate hypothesis $H_1 : dMAP = |MAP_A - MAP_B| \neq 0$. In order to reject the null hypothesis using the student's pair-wise t-test, one needs to show that the confidence interval of the distribution of $dMAP$ does not contain 0 (See Figure 4(a)).

However, our goal is to show that the retrieval accuracy computed on the two modes: *batch mode* vs. *incremental mode* are equivalent, thereby requiring us to prove the null hypothesis. However, standard significance tests are designed to reject the null-hypothesis in favor of the alternate hypothesis. Concluding that two retrieval algorithms are equivalent just because we were unable to reject the null hypothesis would be invalid. Therefore, while a demonstration that the confidence interval of $dMAP$ contains the origin is sufficient to say that we were unable to establish a significant difference, it does not suffice to show that H_0 can be accepted.

Thanks to the on-going work in clinical trials of drugs in the health industry, *equivalence tests* have been designed and used widely to show that two drugs are equivalent to each other [24]. The null hypothesis of the equivalence test is to show that the two algorithms A and B differ, that is $H_0 : dMAP > \delta$, where δ is called the equivalence margin. In order to disprove the hypothesis, a significance test called Two One Sided Test (TOST) [25] can be used, that aims to shows that the *confidence interval* of the distribution of $dMAP$ is completely contained within the interval $[-\delta, +\delta]$ (see Figure 4 (b)). If this is indeed the case, the null hypothesis H_0 is rejected in favor of the alternate hypothesis $H_1 : dMAP <= \delta$. The choice of δ is critical and is often based on the nature of the experiment (prior knowledge). We have selected δ to be 0.005 as this indicates a difference in the average rank of relevant documents at ranks $r > 200$. In this paper, we have subjected our AP values to two statistical significance tests namely, the student's pair-wise t-test and the randomization test [23] and the TOST based equivalence test.

3) Measuring improvements in time: In this section, we present metrics to measure the time spent on each stage of the retrieval process for the *batch mode* approach and for the *incremental approach*.

- Preprocessing: For *batch mode* case, this is the time taken to preprocess all the source files in the repository and we refer to it as the Batch Preprocessing Time (BPT) and for the *incremental update* it is the time taken to preprocess the source files in the *change-set* and we call it the Change Preprocessing Time (CPT).
- Indexing: For the *batch mode* approach this is the time taken to create an index from the preprocessed files and we refer to this as the Index Creation Time (ICT). For *incremental approach* this is the time taken to update the index and the vocabulary and we refer to it as the Index Update Time (IUT).
- Model learning: For *batch mode* approach this metric measures the time taken to learn the model parameters using *all* the source files and can be called as Model Creation Time (MCT). For *incremental* approach this time can be referred to as Model Update Time (MUT).
- Retrieval: Retrieval Time (RT) measures the time taken to construct the query from its bug-report, create its model-space representation and subsequently carry out the matching with documents to compute a ranked list of the source files in decreasing order or relevance. RT remains the same for both modes of operation.

Note that BPT, ICT and MCT vary with the size of the repository, and CPT, IUT and MUT vary with the size of *change-set*.

C. Research Questions

Our experiments have been designed to answer the following research questions (RQ):

*1) **RQ1**: Does the proposed approach impact the retrieval accuracy compared to the batch mode approach?:* The evaluation metrics used for this comparison are listed in Section VI-B1. We subject the AP values to both standard pair-wise t-test, randomization test and the equivalence testing to show that (a) we were not able to establish significant differences between the two algorithms and (b) to show that the two algorithms are indeed equivalent within a margin of δ. Section VII-A shows our findings.

*2) **RQ2:** Does the retrieval accuracy suffer if the index and the model are re-computed only at major releases?:* We have already indicated using sample bugs from both software libraries that retrieval accuracy degrades for some bugs when one uses an outdated index and model for retrieval (see Table II). In Section VII-A, we present overall results using all the bugs.

*3) **RQ3**: Does the proposed framework reduce the query latency for bugs?:* Using the time measures presented in Section VI-B3, the query latency can be quantified as $BPT + ICT + MCT + RT$ for the *batch mode* case. For retrieval by the *incremental update* framework, the query latency is the time taken for constructing the query from the bug report and the time taken to perform matching, which is nothing but the RT.

*4) **RQ4:** Does the proposed approach save on the computational effort/time spent at each commit to keep the model updated?:* The net effort spent in keeping the index and the model updated is a sum of the time taken in preprocessing the source files in the *change sets*, and the time taken to update the index and the model parameters. Using the time measures presented in Section VI-B3, we quantify the amount of effort that goes into keeping the model updated as the sum $CPT + IUT + MUT$ (see Section VII-B).

*5) **RQ5:** At what point is it more beneficial to re-compute the index or the model from scratch as opposed to incrementally updating it?:* The main inspiration of the *incremental approach* to bug localization is that each commit only changes a small portion of the entire repository. The question that remains to be answered is "what happens at large commits?". Our model update formulas and equations do not introduce any approximations into the incrementally updated model. Hence, there is no reason to re-compute the model for the purposes of retrieval accuracy. However, on the account of efficiency, it is worthwhile to explore the relationship between the size of the *change-set* and the time-taken to preprocess the source files (CPT) and update the index (IUT). We attempt to explore the upper bounds on the size of the *change-set* at which the time-benefits of *incremental update* is lost.

VII. Results

A. Comparing Retrieval Accuracy

In this section, we demonstrate that the retrieval accuracy of the *incremental framework* is as good as that of the *batch mode* approach for each of the two models: VSM and SUM. Using the retrieval performance metrics described in Section VI-B1 and statistical significance tests detailed in Section VI-B2, we show the overall results in Table IV and Table V for JodaTime and AspectJ, respectively. The MAP values shown in column 14 of the two tables were subject to student's pair-wise significance tests (Column 15 shows the p-value) and equivalence testing (Column 16 shows confidence interval (ci) computed using TOST) to confirm our findings. Note that the ci is completely contained within the equivalence margin of $[-0.005, 0.005]$. Although not shown in the tables due to space restrictions, we have carried out randomization test and additionally confirmed that the differences are not statistically significant.

1) Parameter Sensitivity Analysis: We now report on the impact of the parameter μ of the SUM model on the retrieval accuracy. Figure 5 shows the variation in the retrieval accuracy using the two approaches for different types of queries w.r.t μ using the SUM model for JodaTime and AspectJ. Note that, with the VSM model, since we used a very basic tf-idf weighting scheme, there are no parameters to be examined. Figure 5 confirms that the retrieval effectiveness of the two approaches are equivalent and robust to variation in parameters.

Answer to RQ1: Retrieval accuracy of the *batch mode* and the *incremental approach* are equivalent (see Tables IV and V and Figure 5).

TABLE IV
COMPARING RETRIEVAL ACCURACY USING PRECISION AND RECALL AND THE RANK-BASED METRICS FOR 43 BUGS IN JODATIME. $R1$ MEANS $r = 1$, $R5$ MEANS $2 \leq r \leq 5$, $R10$ MEANS $6 \leq r \leq 10$, $R11$ MEANS $r > 10$. THE EQUIVALENCE MARGIN IS $\delta = 0.005$.

Model	QueryType	mode	P@1	R@1	P@5	R@5	P@10	R@10	$R1$	$R5$	$R10$	$R11$	MAP	p-value	ci ($\times10^{-3}$)
SUM $\mu = 0.9$	title	batch	0.302	0.178	0.149	0.374	0.119	0.588	13	9	10	11	0.341	0.56	[-0.06,0.03]
		inc	0.302	0.178	0.149	0.374	0.119	0.588	13	9	10	11	0.341		
	description	batch	0.535	0.291	0.251	0.646	0.147	0.705	23	13	1	6	0.539	0.32	[-0.02,0.08]
		inc	0.535	0.291	0.251	0.646	0.147	0.705	23	13	1	6	0.539		
	title+ description	batch	0.535	0.291	0.256	0.669	0.156	0.760	23	14	2	4	0.551	1.00	[0,0]
		inc	0.535	0.291	0.256	0.669	0.156	0.760	23	14	2	4	0.551		
VSM	title	batch	0.209	0.105	0.112	0.295	0.077	0.389	9	10	5	20	0.236	1.00	[0,0]
		inc	0.209	0.105	0.112	0.295	0.077	0.389	9	10	5	20	0.236		
	description	batch	0.279	0.124	0.116	0.261	0.077	0.350	12	6	3	23	0.234	0.323	[0,0.02]
		inc	0.279	0.124	0.116	0.261	0.077	0.350	12	6	3	23	0.234		
	title+ description	batch	0.256	0.116	0.126	0.296	0.079	0.374	11	8	3	22	0.243	0.3230	[0,0.01]
		inc	0.256	0.116	0.126	0.296	0.079	0.374	11	8	3	22	0.243		

TABLE V
COMPARING RETRIEVAL ACCURACY USING PRECISION AND RECALL AND THE RANK-BASED METRICS FOR 321 BUGS IN ASPECTJ. $R1$ MEANS $r = 1$, $R5$ MEANS $2 \leq r \leq 5$, $R10$ MEANS $6 \leq r \leq 10$ AND $R11$ MEANS $r > 10$. THE EQUIVALENCE MARGIN IS $\delta = 0.005$.

Model	QueryType	mode	P@1	R@1	P@5	R@5	P@10	R@10	$R1$	$R5$	$R10$	$R11$	MAP	p-value	ci ($\times10^{-3}$)
SUM $\mu = 0.9$	title	batch	0.139	0.055	0.072	0.143	0.05	0.194	35	50	27	205	0.127	0.71	[-0.5,0.8]
		inc	0.136	0.055	0.071	0.143	0.05	0.193	34	51	27	205	0.126		
	description	batch	0.215	0.081	0.102	0.196	0.069	0.257	53	54	23	186	0.185	0.97	[-1.8,1.7]
		inc	0.209	0.079	0.101	0.195	0.067	0.255	51	55	23	187	0.185		
	title+ description	batch	0.235	0.089	0.111	0.218	0.074	0.268	59	59	22	179	0.201	0.23	[-0.5,2.1]
		inc	0.235	0.089	0.110	0.217	0.074	0.268	59	59	22	179	0.201		
VSM	title	batch	0.077	0.038	0.059	0.138	0.049	0.23	24	59	43	174	0.112	0.492	[-0.7,0.3]
		inc	0.077	0.038	0.060	0.14	0.049	0.232	24	60	44	172	0.112		
	description	batch	0.094	0.04	0.055	0.13	0.052	0.243	29	46	54	179	0.116	0.706	[-0.2,1.4]
		inc	0.101	0.041	0.055	0.131	0.052	0.242	31	45	53	179	0.116		
	title+ description	batch	0.097	0.040	0.064	0.154	0.057	0.261	30	57	51	172	0.121	0.79	[-0.2,0.3]
		inc	0.100	0.04	0.064	0.154	0.057	0.260	31	56	51	172	0.120		

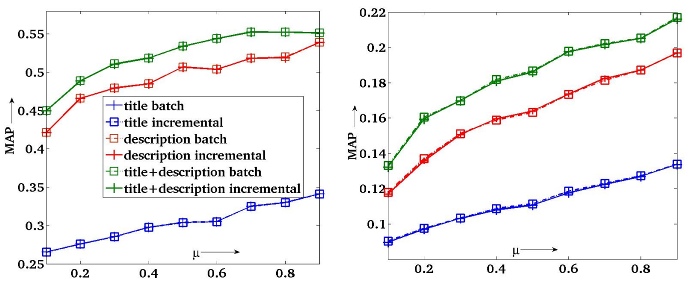


Fig. 5. Sensitivity of retrieval accuracy to the parameter μ using the SUM for JodaTime (left) and AspectJ (right). The legend for the right graph is the same as that for the left one. It has been omitted for the sake of clarity and space restrictions. (see in color)

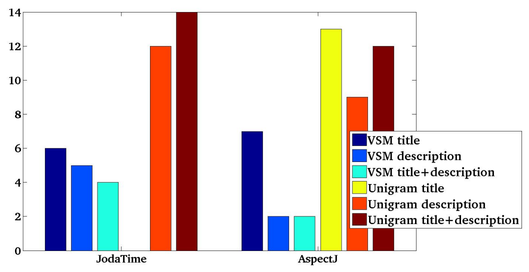


Fig. 6. The number of queries for which retrieval performance deteriorates severely ($|Q_{det}|$) when using *limited update* compared to *batch mode*. (see in color)

2) Comparing retrieval accuracy of batch mode and limited update: Table VI shows the retrieval accuracy obtained by *limited update* technique, and compares it to the retrieval accuracy obtained using the *batch mode* technique. The last column shows the % reduction in retrieval accuracy as measured by MAP. Note that depending on the software system used and the model used to represent the source files, the degree of deterioration of retrieval performance can vary from 0.69% to 24.67%. We confirmed with equivalence testing that the retrieval accuracy computed from the two approaches are not equivalent. We also computed the set of queries Q_{det}, for which the rank of relevant documents deteriorates severely. That is, the rank of a relevant source files slips from within the top 10 ranks in the *batch mode* case to greater than rank 10 in the case of *limited update*. The cardinality of $|Q_{det}|$ varied from 0 to 14, depending on the software, text model and the type of query (see Figure 6). On the other hand, we observed that $|Q_{det}|$ for the case of *incremental update* was 0 for almost all cases.

Answer to RQ2: Recomputing the index and the model only at releases cannot guarantee the same retrieval accuracy that would have been achieved with the *batch mode* approach (See Figure 6 and Table VI).

TABLE VI
COMPARING RETRIEVAL ACCURACY OF ASPECTJ AND JODATIME SOFTWARE USING MEAN AVERAGE PRECISION (MAP) WITH A MODEL THAT IS UPDATED ONLY AT SOFTWARE RELEASES (COLUMN LABELED AS *limited update*) WITH *batch mode* LEARNED MODEL. LAST COLUMN SHOWS THE % REDUCTION IN RETRIEVAL ACCURACY.

	Query Type	Batch Mode	Limited Up-date	Reduction %
JodaTime SUM ($\mu = 0.9$)	title	0.3413	0.2725	20.19
	description	0.5389	0.4718	12.45
	title + description	0.5514	0.5135	6.87
JodaTime VSM	title	0.2316	0.1860	21.22
	description	0.2343	0.1765	24.67
	title + description	0.2430	0.1871	23.00
AspectJ SUM ($\mu = 0.9$)	title	0.1339	0.1208	9.78
	description	0.1969	0.1872	4.93
	title + description	0.2169	0.2030	6.41
AspectJ VSM	title	0.1102	0.1061	3.72
	description	0.1163	0.1155	0.69
	title + description	0.1204	0.1175	2.41

TABLE VII
SUMMARY OF THE TIME TAKEN BY EACH OF THE STAGES OF THE BATCH MODE AND THE INCREMENTAL APPROACHES TO BUG LOCALIZATION.

		JodaTime			AspectJ		
		mean	median	gain	mean	median	gain
# of Files	batch	556	494		5214	5309	
	inc	5.41	2		4.42	1	
Pre processing	BPT	412.7	303.7	50-150	1628	1052	246-536
	CPT	7.83	2.07		6.62	1.96	
Index Creation	ICT	44.97	36.23	133-188	170.15	153.68	241-529
	IUT	0.33	0.19		0.71	0.29	
SUM	MCT	0.76	0.64	4-9	1.72	1.81	8-25
	MUT	0.17	0.06		0.21	0.07	
VSM	MCT	1.26	0.52	4	1.44	1.27	5-10
	MUT	0.28	0.12		0.28	0.13	

B. Improvements in Retrieval Efficiency

In this section we present a detailed analysis of time gains when using the *incremental update* framework compared to *batch mode* approach as measured by a 2.4 GHz desktop computer with 4 cores and 6 GB RAM. Table VII presents the time spent in each of the stages of retrieval and model update for the *batch mode* and the *incremental* approaches, respectively, for the two software libraries. Note that while MCT, BPT and ICT are measured for each bug, MUT, IUT and CPT are measured for each commit. The first row of Table VII shows the size of the input for each of the modes. While this is the size of the project for the *batch mode case*, it is the size of the *change-set* for the *incremental mode* of operation. The fifth column and the eighth column are labeled as “gain” and measure the degree of speed-up obtained by using the *incremental update* framework compared to the *batch mode*.

Evidently, the *incremental update* framework speeds up the preprocessing, index building and the model learning stages significantly. SUM and VSM are linear models, the speedup in MUT compared to MCT is not as significant. However, since MUT depends on the size of the *change set*, the MUT remains more or less constant regardless of the size of the repository. For example, the value of MUT for the SUM for both repositories is in the range 64-71ms for most revisions.

The query latency time (measured in seconds) for both

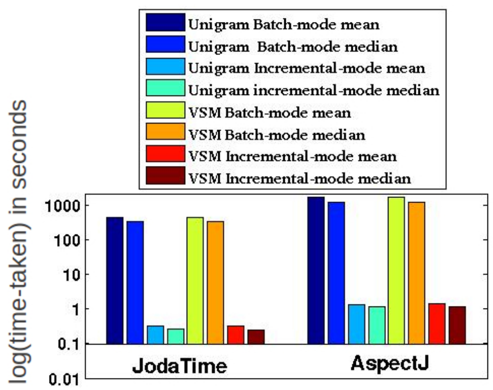


Fig. 7. Comparing the query latency using the *batch mode* and *incremental approach*. (see in color)

TABLE VIII
NET COMPUTATIONAL EFFORT MEASURED IN SECONDS TO KEEP THE MODEL UPDATED USING THE INCREMENTAL APPROACH.

	JodaTime		AspectJ	
Model	mean	median	mean	median
SUM	8.2348	2.3265	7.5342	2.3215
VSM	8.4566	2.3823	7.6092	2.3755

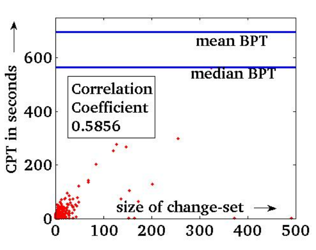
modes of operation is presented in Figure 7. Note the time taken to perform retrieval is significantly reduced with the *incremental update* framework as the model is always kept up-to-date.

Answer to RQ3: Significant reduction in query latency can be achieved using the proposed approach compared to batch-mode approach (see Figure 7).

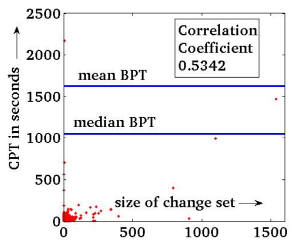
The net amount of computational effort (measured as the time spent) in keeping the model updated at each revision is shown in Table VIII. Since the change set is relatively small (see Figure 2) the overall time spent is just around 2 seconds for most revisions and 8 seconds on the average.

Answer to RQ4: The net computational effort/time spent at each commit to keep the model updated is reasonable within a few seconds (see Table VIII).

1) Sensitivity to the size of change-set: As mentioned earlier, the time to preprocess the source files in the *change-set* (CPT) and update the index (IUT) varies with the size of the change set. In Figures 8 and 9 we plot the size of the *change set* along the x-axis and the time-taken along the y-axis for both software libraries and the horizontal blue lines correspond to the mean and the median of the corresponding *batch mode time* (BPT and ICT). These figures illustrate that as long as the size of a change set is much smaller than the size of the repository (which is likely to be case all the time), the time taken by the *incremental update* framework is significantly less than the time taken by the *batch mode* framework. Note that the IUT is highly correlated with the size of the *change set* except for some outliers. For example, the large commits that take very little time for index updates are commits that delete a significant number of source files. Correlation exists between the CPT and the size of the *change-set* as well

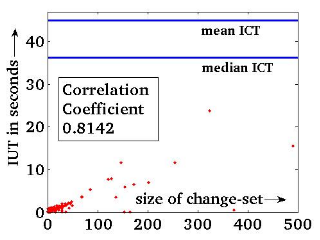


(a) JodaTime

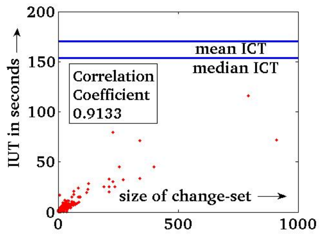


(b) AspectJ

Fig. 8. Variation of CPT with the size of the *change-set*. The horizontal blue lines indicate the mean and median of BPT. (see in color)



(c) JodaTime



(d) AspectJ

Fig. 9. Variation of IUT with the size of the *change-set*. The horizontal blue lines indicate the mean and median of ICT. (see in color)

except for some outliers. The cases where large change-sets are processed quickly correspond to commits where several small source files are added (a typical example is adding test-cases). Similarly, in some cases, a single large file takes a long time for preprocessing. In general, when the size of *change-set* is > 100, the CPT and IUT start to look comparable to that of BPT and ICT respectively.

Answer to RQ5: For commits that affect large number of source files (typically > 100 or $> 10\%$ of the size of the repository) the amount of time taken to preprocess the *change-set* (CPT) and update the index (IUT) is of the same order as that of the time taken by *batch mode* approach (see Figures 8 and 9).

VIII. Threats to Validity

Any empirical research must be subject to an analysis of threats to its validity. Since the retrieval accuracy of the *batch mode* approach and the *incremental framework* are equivalent, an apparent threat to validity is the poor retrieval accuracy of *batch mode* model itself. For queries where the *batch mode* approach gives poor retrieval performance, incremental approach cannot yield a better retrieval accuracy. A query may perform poorly for a number of reasons and there is a lot of research that is focused on identifying such *difficult queries* [9][10] and improving their retrieval accuracy [8][5][4][6] and is out of the scope of this paper. Our approach to *incrementally updating* the index does not eliminate deleted files, and so the size of the index grows monotonically. While this might necessitate rebuilding of the index for the sake of saving memory, retrieval accuracy is not affected as these deleted files are discounted from our retrieval algorithm. Another threat to validity of this work is related to the experimental set-up of our *incremental framework*. The measurements of time-savings are carried out using a 2.4 GHz desktop computer with 4 cores and 6 GB RAM and may vary with configuration of system used, the choice of search tool, level of granularity of the indexer (e.g. lines of code or functions as code entities as opposed to source files) and the profiler used. Both AspectJ and JodaTime are written in Java, hence our conclusions are not generalizable to software libraries written in other languages. A possible threat to the validity of the *moreBugs* dataset we have created is that we have not performed any sort of post-processing on the change history of the software. In the future we plan to use change-distillation [26] to eliminate null changes to the code entities and merge changes that are spread over multiple commits [27]. Additionally, *moreBugs* only tracks changes taking place in the official/main branch of the software. However, our proposed framework can be easily extended to maintain index and model for the various branches of development of a software.

IX. Related Work

The problem of building search engines for dynamic collections is not new and there has been much research on incremental updating of the index for web-scale text collections in the domain of text-IR [12][13][28]. However, the main focus of these algorithms are low level issues related to incremental indexing like storage memory and I/O [29]. In terms of incremental model update, incremental clustering algorithms have been proposed as well [30][31][32]. Mention must also be made of the open-source tools like Terrier[4], Lemur[5] and Lucene[6] for IR-based research. These tools vary in the support they provide for incrementally updating the index and the model [33].

In the context of retrieval from software repositories, there are a few contributions that explore the efficiency aspect of these tools. One such contribution is the Incremental Latent Semantic Indexing (LSI) algorithm for search based automatic traceability link recovery proposed by Jiang et al. [34]. In this paper, the authors propose an incremental approach based on LSA model to update the links between the source code files and the documentation as they both evolve. However, their algorithm ignores information that is added in terms of new terms and new source files. Additionally, their analysis is limited to a dataset built from 2 consecutive releases of candidate software systems. Canfora et al. [35] have also studied the the use of incremental indexing for impact analysis in development process. The authors demonstrate improvements in retrieval of code entities related to a Change Request (CR) by using code entities impacted by similar past change requests.

X. Conclusion

In this paper we have proposed an incremental approach to bug localization that achieves reduced query latency without compromising on the retrieval accuracy achieved by current IR techniques. With our incremental framework, the index and model can be efficiently updated to reflect the changes in the software repositories within a few seconds at each commit.

[4]http://terrier.org/
[5]http://www.lemurproject.org/
[6]http://lucene.apache.org/

We have demonstrated based on extensive empirical evaluation using two software repositories, AspectJ and JodaTime modeled with VSM and SUM, that the query latency reduces from several minutes to a fraction of a second. For future work, we plan to add to the proposed framework incremental model update algorithms for sophisticated models like Latent Semantic Analysis (LSA) and Latent Dirichlet Allocation (LDA). The proposed incremental update framework can also be used for other software engineering problems that have to deal with dynamically evolving software repositories.

Acknowledgments

This work is supported by Infosys. Thanks are owed to Josh Zapf and anonymous reviewers for their valuable feedback.

References

[1] G. Gay, S. Haiduc, A. Marcus, and T. Menzies, "On the Use of Relevance Feedback in IR-based Concept Location," *Software Maintenance, IEEE International Conference on*, pp. 351–360, 2009.

[2] 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*, ser. MSR '11. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 43–52.

[3] S. Lukins, N. Kraft, and L. Etzkorn, "Source Code Retrieval for Bug Localization using Latent Dirichlet Allocation," in *15th Working Conference on Reverse Engineering*, 2008.

[4] B. Sisman and A. Kak, "Incorporating Version Histories in Information Retrieval Based Bug Localization," in *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, June 2012.

[5] J. Zhou, H. Zhang, and D. Lo, "Where Should the Bugs be Fixed? - More Accurate Information Retrieval-Based Bug Localization Based on Bug Reports," in *Proceedings of the 2012 International Conference on Software Engineering*, Zurich, Switzerland, 2012.

[6] N. Ali, A. Sabane, Y.-G. Gueheneuc, and G. Antoniol, "Improving Bug Location Using Binary Class Relationships," *Source Code Analysis and Manipulation, IEEE International Workshop on*, pp. 174–183, 2012.

[7] B. Sisman and A. C. Kak, "Assisting Code Search with Automatic Query Reformulation for Bug Localization," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13, 2013, pp. 309–318.

[8] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies, "Automatic Query Reformulations for Text Retrieval in Software Engineering," in *Software Engineering (ICSE), 35th International Conference on*, 2013.

[9] S. Haiduc, "Automatically Detecting the Quality of the Query and its Implications in IR-based Concept Location," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, 2011, pp. 637–640.

[10] S. Haiduc, G. Bavota, R. Oliveto, A. Marcus, and A. De Lucia, "Evaluating the Specificity of Text Retrieval Queries to Support Software Engineering Tasks," in *Software Engineering (ICSE), 2012 34th International Conference on*, 2012, pp. 1273–1276.

[11] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. New York, NY, USA: Cambridge University Press, 2008.

[12] E. W. Brown, J. P. Callan, and W. B. Croft, "Fast incremental indexing for full-text information retrieval," in *Proceedings of the 20th International Conference on Very Large Data Bases*, ser. VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 192–202.

[13] T. Chiueh and L. Huang, "Efficient Real-Time Index Updates in Text Retrieval Systems," Experimental Computer Systems Lab, Department of Computer Science, State University of New York, Tech. Rep., 1999.

[14] G. Salton, A. Wong, and C. S. Yang, "A Vector Space Model for Automatic Indexing," *Commun. ACM*, vol. 18, no. 11, pp. 613–620, Nov. 1975.

[15] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of Duplicate Defect Reports using Natural Language Processing," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, 2007, pp. 499–510.

[16] 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*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 125–135.

[17] T. Tao, X. Wang, Q. Mei, and C. Zhai, "Language Model Information Retrieval with Document Expansion," in *Proceedings of the Human Language Technology Conference of the North American Chapeter of the ACL*. ACL, 2006.

[18] J. Lafferty and C. Zhai, "Document Language models, Query Models, and Risk Minimization for Information Retrieval," in *24th ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 2001.

[19] B. Cleary, C. Exton, J. Buckley, and M. English, "An Empirical Analysis of Information Retrieval based Concept Location Techniques in Software Comprehension," *Empirical Software Engineering*, vol. 14, no. 1, pp. 93–130, 2009.

[20] S. Wang, D. Lo, Z. Xing, and L. Jiang, "Concern Localization using Information Retrieval: An Empirical Study on Linux Kernel," in *Reverse Engineering (WCRE), 2011 18th Working Conference on*, 2011, pp. 92–96.

[21] 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*. ACM, 2007, pp. 433–436.

[22] S. Rao and A. Kak, "moreBugs: A New Dataset for Benchmarking Algorithms for Information Retrieval from Software Repositories (tr-ece-13-07)," Purdue University, School of Electrical and Computer Engineering, Tech. Rep., 04 2013.

[23] M. D. Smucker, J. Allan, and B. Carterette, "A Comparison of Statistical Significance Tests for Information Retrieval Evaluation," in *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management*, ser. CIKM '07. ACM, 2007, pp. 623–632.

[24] E. Walker and A. S. Nowacki, "Understanding Equivalence and Noninferiority Testing," *General Internal Medicine*, February 2011.

[25] K. Phillips, "Power of the Two One-Sided Tests Procedure in Bioequivalence," *Journal of Pharmacokinetics and Biopharmaceutics*, vol. 18, no. 2, pp. 137–144, 1990.

[26] D. Kawrykow and M. P. Robillard, "Non-essential Changes in Version Histories," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, 2011, pp. 351–360.

[27] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining Version Histories to Guide Software Changes," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 563–572.

[28] A. Tomasic, H. Garcia-Molina, and K. Shoens, "Incremental Updates of Inverted Lists for Text Document Retrieval," Stanford Infolab, Technical Report 1994-2, 1994.

[29] S. Büttcher and C. L. A. Clarke, "Indexing Time vs. Query Time: Trade-offs in Dynamic Information Retrieval Systems," in *Proceedings of the 14th ACM international conference on Information and knowledge management*, ser. CIKM '05. New York, NY, USA: ACM, 2005, pp. 317–318.

[30] M. Charikar, C. Chekuri, T. Feder, and R. Motwani, "Incremental Clustering and Dynamic Information Retrieval," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, ser. STOC '97. ACM, 1997, pp. 626–635.

[31] D. Widyantoro, T. Ioerger, and J. Yen, "An Incremental Approach to Building a Cluster Hierarchy," in *Data Mining, 2002. ICDM 2002. Proceedings. 2002 IEEE International Conference on*, 2002, pp. 705 – 708.

[32] F. Can, "Incremental Clustering for Dynamic Information Processing," *ACM Transactions on Information Systems*, vol. 11, pp. 143–164, April 1993.

[33] C. Middleton and R. Baeza-yates, "A Comparison of Open Source Search Engines," Universitat Pompeu Fabra Department of Technologies, Tech. Rep., 2007.

[34] H. Jiang, T. Nguyen, I.-X. Chen, H. Jaygarl, and C. Chang, "Incremental Latent Semantic Indexing for Automatic Traceability Link Evolution Management," in *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, 2008, pp. 59–68.

[35] G. Canfora and L. Cerulo, "Fine Grained Indexing of Software Repositories to Support Impact Analysis," in *Proceedings of the 2006 international workshop on Mining software repositories*, ser. MSR '06, 2006, pp. 105–111.