Published in 2nd International Workshop on Software Mining, Palo Alto, California, Nov 11, 2013.

Comparing Incremental Latent Semantic Analysis Algorithms 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

The problem of bug localization is to identify the source files related to a bug in a software repository. Information Retrieval (IR) based approaches create an index of the source files and learn a model which is then queried with a bug for the relevant files. In spite of the advances in these tools, the current approaches do not take into consideration the dynamic nature of software repositories. With the traditional IR based approaches to bug localization, the model parameters must be recalculated for each change to a repository. In contrast, this paper presents an incremental framework to update the model parameters of the Latent Semantic Analysis (LSA) model as the data evolves. We compare two state-of-the-art incremental SVD update techniques for LSA with respect to the retrieval accuracy and the time performance. The dataset we used in our validation experiments was created from mining 10 years of version history of AspectJ and JodaTime software libraries.

Keywords

Information Retrieval, Incremental Learning, Latent Semantic Analysis, Bug Localization, Singular Value Decomposition

1. INTRODUCTION

Much effort in the software engineering community has gone into developing search based tools to aid the programmer/developer in narrowing down the set of source files relevant to a bug. Such algorithms are collectively referred to as *Information Retrieval (IR) based bug localization* techniques [1–9]. These search based tools first construct an *index* from the source files present in the most recent revision of the software. Subsequently, a text model is learned from this index. A bug is localized by treating the textual content of the bug as a query that is then used to fetch the relevant documents from the model [10].

Despite the effectiveness of these approaches, the current state-of-the-art bug localization tools are not efficient for deployment for a real software system that is constantly evolving. As the software evolves, source files may be added, deleted, or modified, which can leave the index and the model out-of-sync with the repository. In order to ensure accurate retrieval, the current approaches to IR based bug localization re-compute the index and the model from scratch for each bug that needs to be localized in a newer version of the software. This traditional approach to bug localization is commonly referred to as the *batch-mode* approach. The batch-mode approach suffers from the following two shortcomings:

- Re-computing the index and the model from scratch is time-consuming and can result in high turn around time or *query latency* [11] (See Table 1).
- Since it is often the case that each commit to a repository affects only a very tiny portion of the overall code base [10], re-learning the model from scratch for each new bug is inefficient and may be unnecessarily computationally expensive.

In a previous contribution [10], we presented an incremental update framework that keeps the index and the model updated at each commit as the software evolves using the Smoothed Unigram Model (SUM) and the Vector Space Model (VSM). That work demonstrated that the SUM and VSM based incremental frameworks gave us enhanced retrieval efficiency with no penalty in retrieval accuracy. In this paper, we extended our experiments to the Latent Semantic Analysis (LSA) model that has been studied extensively in the software engineering community for a variety of software maintenance activities [12–20].

In order to incrementally update the LSA model, we compare the following two state-of-the-art incremental LSA algorithms:

- *iLSI* – This algorithm was proposed by Jiang et al. [21] to incrementally update the LSA model of a dynamic collection of source files and related documentation for the purpose of search-based automated traceability link recovery.
- *iSVD* – This algorithm was proposed by Brand [22] to incrementally update the SVD components of a user preference matrix for movie recommendation systems.

To the best of our knowledge, there has been no prior work in employing incremental approaches for efficient IR based bug localization using the LSA model. Although our work with the *iLSI* algorithm is closely related to the work reported in [21], there are two very important differences between how this algorithm was studied in [21] and how we use it here. Our work makes explicit the limitation that the *iLSI* algorithm is incapable of incorporating new information (source files and terms) as a software library evolves. In other words, we show that the *iLSI* algorithm is not the best choice for incrementally updating the LSA model of an evolving software repository. Secondly, Jiang et al.'s [21] experimental validation consists of just two consecutive releases of the software libraries they worked with. In contrast, our experiments are based on commit-level information tracked over 10 years of commit history of the software libraries on which we have reported our results.

Thus, our main goal in this paper is to compare the retrieval accuracy, modeling error and speed of computation of the *iSVD* and the *iLSI* algorithms mentioned above with the *batch-mode* LSA in the context of IR based bug localization. We also present strategies for retraining the model after a sequence of commits or for large commits (commits that affect a significant portion of the source code) in order to keep the incrementally updated model close to the true model. In order to evaluate our incremental model update framework, we have created a benchmark dataset called `moreBugs` [23] that tracks commit-level changes over 10 years of developmental history of two software repositories: JodaTime and AspectJ.

Table 1: Time spent in different stages of the retrieval process using the LSA model for software repositories of different sizes. As shown in the last column, the query latency can range from 5 minutes to 50 minutes.

bug ID	Dataset	# of source files	# of terms	Preprocessing (seconds)	Indexing (seconds)	Model Learning (seconds)	Retrieval (seconds)	Query Latency (in minutes)
178828	JodaTime	486	10824	264.71	37.06	6.96 (k=60)	0.696	5.16
3192457	JodaTime	864	12174	603.18	92.70	10.47 (k=90)	0.894	11.79
371684	AspectJ	7594	40,256	2942.11	228.47	35.13 (k=100)	2.869	53.47

2. THE BATCH-MODE LSA ALGORITHM

In the LSA model, the source files are first represented by a $|\mathcal{V}| \times M$ *term-document matrix* whose rows correspond to the terms in the vocabulary $\mathcal{V}$ and whose columns correspond to the M source files. We will denote this matrix by A. Subsequently, the dimensionality of the vector space in which the documents are represented is reduced by subjecting A to a Singular Value Decomposition (SVD) and retaining only the top k singular values: $A \approx U_k S_k V_k^T$, where U_k is a $|\mathcal{V}| \times k$ column-wise orthogonal matrix, S_k a $k \times k$ diagonal matrix of the singular values, and V_k an $M \times k$ orthonormal matrix. For retrieval, a query is constructed from the bug report and mapped to the LSA's eigenspace: $q_k = q^T U_k S_k^{-1}$. A cosine similarity between q_k and the columns of V_k is used to compute the ranked list of source files vis-à-vis the query.

As SVD is one of the fundamental operations in manipulating matrices in general, much research has been devoted in the past to the development of numerically efficient algorithms for such a decomposition [24–27]. We have used the popular Lancsoz SVD algorithm [28] designed for large sparse matrices as our batch-mode LSA algorithm. The computational complexity of this algorithm is O($|\mathcal{V}|Mk^2$) and it is implemented as a part of the ARPACK software library[1].

3. INCREMENTAL FRAMEWORK FOR BUG LOCALIZATION

Figure 1 shows the framework for LSA-based incremental bug localization that can be used as an alternative to the batch-mode framework. In the following discussion, the superscript t indicates the current state of the repository. The framework shown in Figure 1 entails the following steps:

Change Processing: For each new commit, the set of source files that are affected, called the *change-set*, is checked out and subject to the text preprocessing steps described in [10].

Index Update: Recall that the columns of A^t correspond to the source files $A^t = [A_1^t A_2^t ... A_M^t]$. For the sake of developing the notation, assuming for a moment that only a single file is involved in either the addition to the library, or its modification, or deletion, we can then represent the matrix A^{t+1} as follows:

- Addition: $A^{t+1} = [A^t A_{M+1}^{t+1}]$.
- Modification of j^{th} file $A^{t+1} = [A_1^t A_2^t ... A_j^{t+1} ... A_M^t]$
- Deletion of j^{th} source file $A^{t+1} = [A_1^t A_2^t ... \mathbf{0} ... A_M^t]$

In general, a single commit may involve a combination of the above mentioned changes, the notation shown generalizes in an obvious manner. Although not shown explicitly in the notation above, addition of new terms appends new rows to the term-document matrix A. If there are M_a new source files and $|\mathcal{V}_a|$ new terms added to the vocabulary, then the new A^{t+1} is of size $\{|\mathcal{V}| + |\mathcal{V}_a|\} \times \{M + M_a\}$.

[1] http://www.caam.rice.edu/software/ARPACK/

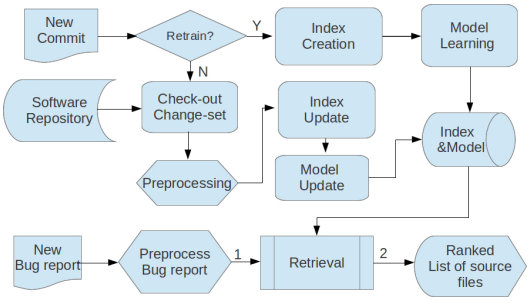


Figure 1: Incremental update framework for bug localization.

Updating the Model: Given the SVD decomposition at commit t as $A^t \approx U_k^t S_k^t V_k^{t^T}$, and the updated term-document matrix A^{t+1}, the goal of the model update is to estimate U_k^{t+1}, S_k^{t+1} and V_k^{t+1}. In Section 3.2, we will briefly review the two incremental SVD algorithms that we are comparing in this paper.

Retrieval for a bug report: The search process for a new bug report does not involve the overhead of preprocessing, index and model creation. It merely consists of two steps: (a) preprocessing of the bug report, and (b) retrieval of the source files using the equations presented in Section 2.

3.1 When to Retrain?

Since the dimensionality reduction achieved by ignoring the smallest of the singular values calculated by SVD can always be expected to introduce some error in an LSA model, any incremental approach involving LSA must make provision for occasionally re-acquiring the true model through batch-mode learning. That raises the question of how often must the batch-mode algorithm be invoked?

Additionally, as the reader will see later, the time gains achieved with the incremental LSA algorithm are based on the assumption that $\{M_a + M_d\} << M$. That is, time gains are achieved only when the number of source files affected is a small fraction of the total number of files — an assumption easily satisfied for most commits in a real software repository. Obviously, should it happen that a commit affects a large portion of the source code, this assumption would not be satisfied, and the amount of time taken to update the model would begin to approach the time taken to re-compute the LSA model through the batch-mode SVD. Thus, on encountering a revision in which a significant proportion of the source files has changed, it might be computationally more efficient to re-compute the SVD afresh.

Gauging when an incrementally updated model is no longer a good approximation to the true model can only be handled by a heuristic. This is owing primarily to the fact that we do not have access to the true state of the model as it is modified incrementally. One possible policy is to retrain the model at every major release. We will refer to this as the *major_releases* policy. Along the same lines, if the invocation of batch-mode learning for re-acquiring the true model is to be triggered by how much of the library was changed at a commit, we need to set a heuristic threshold on the size of the *change-set*. This true model re-acquisition policy will be denoted by *major_changes* and we will use n_{thresh} to denote the

Table 2: Notation used in Algorithms 1 and 2

$\mathbf{0}_{mxn}$	A matrix containing m rows and n columns of all zeros
$\mathbf{I}_n$	A n x n identity matrix
subscript $_k$	The number of eigenvalues retained
superscript ^{T}	Transpose of a matrix
superscript ^{t}	The commit (revision) number identifier of a repository
subscript $_a$	Denotes new or added information
subscript $_d$	Denotes deleted source files

threshold on the size of the *change-set* to trigger this policy. The incremental update framework shown in Figure 1 has a decision box labeled "*retrain?*" to allow for this policy. Note that even when the model is computed afresh, only the index needs to be re-created; there is no need to subject *all* the source files to the preprocessing steps mentioned earlier.

3.2 Review of Incremental Calculations of the SVD

Given A^{t+1} and A^t, U_k^t S_k^t and V_k^t, the goal of the incremental SVD update algorithm is to estimate U_k^{t+1}, V_k^{t+1} and S_k^{t+1}. Table 2 briefly reviews the notation used in the presentation in this section. Due to lack of space we skip the detailed steps of these algorithms and interested readers are directed to the relevant papers that have been cited.

3.2.1 The *iLSI* algorithm [21]

This algorithm is based on the intuitive plausibility of the steps involved (Algorithm 1). The overall complexity of *iLSI* is $O(\{|\mathcal{V}|+|\mathcal{V}_a|\}\times k\times\{M+M_a\}+k^4+\{|\mathcal{V}|+|\mathcal{V}_a|\}\times k^2+\{M+M_a\}\times k^2)$. Since $M_a << k << M$, and $|\mathcal{V}_a| << |\mathcal{V}|$ the overall time taken to update the SVD components is significantly reduced to $O(|\mathcal{V}|\times k\times M+k^4)$.

The main drawback of *iLSI* is that it ignores new information. In Steps 1 and 2 of the algorithm shown in Algorithm 1, the rows of U_k^t and V_k^t are appended with zeros to create U' and V'. The product $U'^T A^{t+1} V'$ thus formed causes the new information in A^{t+1} to be literally ignored. As we will show, this results in an approximate update of the model parameters, leading to a higher degree of modeling error and relatively lower retrieval accuracy.

Algorithm 1 *iLSI* algorithm proposed by Jiang et. al [21].

Require: U_k^t of size $|\mathcal{V}|\times k$, V_k^t of size $M\times k$, S_k^t of size $k\times k$, and A^{t+1} of size $\{|\mathcal{V}|+|\mathcal{V}_a|\}\times\{M+M_a\}$

1: Append rows of zeros to U_k^t and V_k^t to account for M_a new source files and $|\mathcal{V}_a|$ new terms respectively. $U' = \begin{bmatrix} U_k^t \\ \mathbf{0}_{|\mathcal{V}_a|\times k} \end{bmatrix}$ and $V' = \begin{bmatrix} V_k^t \\ \mathbf{0}_{|Ma|\times k} \end{bmatrix}$

2: Compute the new *central matrix* via: $\hat{S} = U'^T A^{t+1} V'$. Note that since A^{t+1} contains additional information, $\hat{S}$ is of size $k\times k$ and may not be perfectly diagonal.

3: Diagonalize $\hat{S}$ by SVD decomposition: $\hat{S} = \widetilde{U}_k \widetilde{S}_k \widetilde{V}_k$. All the three components are of size $k\times k$.

4: Compute the updated U_k^{t+1}, S_k^{t+1} and V_k^{t+1} matrix as $U_k^{t+1} = U'\widetilde{U}_k$, $V_k^{t+1} = V'\widetilde{V}_k$ and $S_k^{t+1} = \widetilde{S}$

3.2.2 The *iSVD* algorithm [22]

This is a mathematically rigorous algorithm that incrementally updates the components of SVD decomposition by using just the modifications made to the term-document matrix (A^t), and not the entire matrix. The changes to A^t are encoded using two matrices X and Y, where X contains the column vectors corresponding to the changes for each affected source file and the columns of Y are the indicator vectors where exactly one element (the one corresponding to the affected source file) is set to 1 and the rest to 0.

The *iSVD* algorithm (summarized in Algorithm 2) computes the residual energy in X and Y with respect to U and the V eigenvectors that needs to be accounted for in the update. Thus, the *iSVD* algorithm incorporates new information (source files/terms) that appear in A^{t+1} more directly in the update equations. Despite the additional QR step, the time-complexity of the *iSVD* stays at $O(|\mathcal{V}|\times k\times M+k^4)$.

Algorithm 2 Incremental SVD (*iSVD*) proposed by Matthew Brand [22].

Require: U_k^t of size $|\mathcal{V}|\times k$, V_k^t of size $M\times k$, S_k^t of size $k\times k$, X of size $\{|\mathcal{V}|+|\mathcal{V}_a|\}\times\{M_a+M_d\}$, Y of size $\{M+M_a\}\times\{M_a+M_d\}$

1: Compute Projection of X on U and Y on V. Both m and n are of size $k\times\{M_a+M_d\}$:
$m = \begin{bmatrix} U_k^t \\ \mathbf{0}_{|\mathcal{V}_a|\times k} \end{bmatrix}^T X$ and $n = \begin{bmatrix} V_k^t \\ \mathbf{0}_{|Ma|\times K} \end{bmatrix}^T Y$

2: Compute the residual energy in U and V space. m_r is of the same size as X and n_r is of the same size as Y:
$m_r = X - \begin{bmatrix} U_k^t \\ \mathbf{0}_{|\mathcal{V}_a|\times K} \end{bmatrix} m$ and $n_r = Y - \begin{bmatrix} V_k^t \\ \mathbf{0}_{Ma\times K} \end{bmatrix} n$

3: Compute the QR decomposition of the residuals m_r and n_r as follows. P is of size $\{|\mathcal{V}|+|\mathcal{V}_a|\}\times k_m$ and R_x is of size $k_m\times k_m$. Q is of size $\{M+M_a\}\times k_n$ and R_y is of size $k_n\times k_n$. k_m and k_n are the ranks of the m_r and n_r matrices.
$PR_x \xleftarrow{QR} m_r$ and $QR_y \xleftarrow{QR} n_r$

4: Compute the central matrix $\hat{S}$ and its SVD decomposition. $\widetilde{U}$ is of size $\{k+k_m\}\times k$ and $\widetilde{V}$ is of size $\{k+k_n\}\times k$ and $\widetilde{S}$ is of size $k\times k$

$$\hat{S} = \begin{bmatrix} S_k^t & \mathbf{0}_{k\times k_n} \\ \mathbf{0}_{k_m\times k} & \mathbf{0}_{k_m\times k_n} \end{bmatrix} + \begin{bmatrix} m \\ R_x \end{bmatrix} \begin{bmatrix} n \\ R_y \end{bmatrix}^T$$
$$\hat{S} \approx \widetilde{U_k} \quad \widetilde{S_k} \quad \widetilde{V_k}^T \tag{1}$$

5: Compute the updated U_k^{t+1} and V_k^{t+1} matrix
$U_k^{t+1} = [\ U_k^t \quad P\ ]\,\widetilde{U_k}$ & $V_k^{t+1} = [\ V_k^t \quad Q\ ]\,\widetilde{V_k}$ & $S_k^{t+1} = \widetilde{S}$

4. EXPERIMENTAL VALIDATION

4.1 The Dataset

Similar to our previous work [10], we have used the `moreBugs` [23] dataset to perform our experimental validation. The dataset contains all the necessary information to evaluate both the *batch-mode* and the *incremental-mode* approaches to IR based bug localization, namely: (a) the commit-level changes taking place in the repository; (b) the release history of the software; and (c) a set of closed/resolved issues/bugs. For each of bug in item (c), the following information is available: (i) the bug report's textual content like title, description, comments and so on, (ii) the source files that were fixed in order to resolve the bug (we call this list of source files the *patch-list* or *relevance list* for the bug), and (iii) the prefix snapshot of the software repository. While (c) alone suffices for evaluation of the *batch-mode* approach, (a) and (b) are additionally required for evaluation of the *incremental update* framework. This publicly available benchmark dataset was created by mining 10 years of commit history, release history and bug-fixing history for AspectJ (7477 commits) and JodaTime (1573 commits) projects. Table 3 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/.

Table 3: `moreBugs` specifications.

	AspectJ	JodaTime
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
Number of bugs used in evaluation	321	43
Average number of source files/bug	5214	556
Average number of relevant source files/bug	3.36	2.13

4.2 Evaluation Metrics

We have evaluated the incremental update algorithms using the following three types of metrics:

4.2.1 Measuring the Modeling Error

As mentioned in section 3.1, the incrementally updated model is a close approximation to the batch-mode learned model, and the degree of approximation can be measured through a metric called the *Relative Modeling Error* [26]. RME is nothing but the ratio of the reconstruction error

Table 4: Quantifying the time spent in various stages of the retrieval process.

Stage	Batch-mode	Incremental-mode
Preprocessing	Batch Preprocessing Time (BPT)	Change Preprocessing Time (CPT)
Indexing	Index Creation Time (ICT)	Index Update Time (IUT)
Model Learning	Model Creation Time (MCT)	Model Update Time (MUT)
Retrieval	Retrieval Time (RT)	Retrieval Time (RT)

and is computed as follows: If U^t_{kG} V^t_{kG} and S^t_{kG} denote the batch-mode trained model parameters at any revision and U^t_k, S^t_k and V^t_k are the SVD components obtained through incremental updating, the RME is given by:

$$RME = log\left(\frac{||A^t - U^t_k S^t_k {V^t_k}^T||}{||A^t - U^t_{kG} S^t_{kG} {V^t_{kG}}^T||}\right)$$

$RME \geq 0$ with equality taking place only when the incremental update approach introduces no additional error.

4.2.2 Measuring Retrieval Performance

The metrics used to evaluate the retrieval accuracy of a search engine are computed by examining the ranked list of source files returned by the search engine in response to a query. The top N_r source files in the ranked list is called the *retrieved set* and is compared with the *relevance list* to compute the Precision and Recall metrics (denoted by $P@N_r$ and $R@N_r$ respectively). In this paper, we report P@1, P@5, P@10 and R@1, R@5 R@10. Precision and Recall share an inverse relationship, in that, the Precision is higher than Recall for lower values of N_r and vice versa for higher values of N_r. An overall metric of retrieval accuracy that is independent of the cut-off N_r is known as *Average Precision* (AP), and is defined as the area under the Precision-Recall curve. Higher values of AP indicate a more effective retrieval engine. In this work, we report the Mean Average Precision (MAP), which is the average of the AP values over all the bugs in the software.

Another way to gauge retrieval accuracy is by using rank-based metrics [3] which measure the number of bugs for which at least one relevant source file was retrieved at rank r. In our validation experiments, we have presented rank measures for the following values of r: $r = 1$, $2 \leq r \leq 5$, $6 \leq r \leq 10$, $11 \leq r \leq 20$ and $r > 20$.

4.2.3 Measuring Improvements in Retrieval Efficiency

The time spent on each stage of the retrieval process for the *batch mode* approach and for the *incremental approach* can be quantified using metrics shown in Table 4. BPT, ICT and MCT vary with the size of the repository, and CPT, IUT and MUT vary with the size of the change-sets. Additionally, MCT and MUT vary with the complexity of the model (k). RT remains the same regardless of the mode of operation.

From the above quantities, the following two quantifiable metrics can be computed: (a) the *Query Latency (QL)* of a retrieval system is measured as the time taken for computing the ranked list given a query as input, and (b) the *Net Computational Effort (NCE)* is measured as the time spent in keeping the model updated at each commit. For the batch-mode framework QL can be quantified as $BPT + ICT + MCT + RT$, whereas for the incremental update framework it is merely RT. NCE is the 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 in the incremental update mode ($CPT + IUT + MUT$).

4.3 Research Questions

We designed our validation experiments to answer the following research questions (RQ):

1. **RQ1**: How similar is the incrementally updated model (computed by the *iSVD* and *iLSI* algorithms) to the model learned using batch-mode SVD?

2. **RQ2**: How does the retrieval accuracy obtained with the *iSVD* and *iLSI* algorithms compare with the batch-mode approach?

3. **RQ3:** How do the two incremental update algorithms compare with the batch-mode SVD approach in terms of retrieval efficiency?

4. **RQ4:** Under what conditions is it wise to retrain the model instead of incrementally updating it?

5. RESULTS

The experimental framework was set up on a 2.4 GHz desktop computer with 4 cores and 6 GB RAM. Two parameters affect the retrieval accuracy of the batch-mode and the incremental mode approaches to bug localization using the LSA model: (a) The number of eigenvalues retained k, and (b) the type of query. We analyzed the sensitivity of the incremental update approaches to the parameter k, and we experimented with three types of queries created from the title and description fields of the bug report: (a) title only (b) description only, and (c) title + description.

In the incremental update framework, there is yet another design choice to consider – the decision as to when to retrain the LSA model. One can either train at major releases (*major_releases*) or at commits where a significant portion of the source files is affected (*major_changes*). In our previous work [10] we found that in general when more than 100 source files are affected in a commit, the time taken to incrementally update the index approaches the time taken to re-compute it. Thus, for the results presented in sections 5.1, 5.2 and 5.3, we used the retraining threshold $n_{thresh} = 100$ to identify commits at which the model is re-computed (*major_changes*). In section 5.4 we vary n_{thresh} and explore other design choices for retraining the LSA model and attempt to answer RQ4.

5.1 RQ1: Measuring Model Update Error

In order to answer RQ1, we have plotted RME results computed using the *iSVD* and *iLSI* algorithms in Figure 2 for JodaTime and AspectJ respectively. The model is re-calculated at *major_changes* with $n_{thresh} = 100$. Note that at the times when the model is retrained, the error drops and then starts gradually increasing again as the model is incrementally updated. For both software systems – JodaTime and AspectJ, the RME of the *iSVD* algorithm is significantly lower than that of the *iLSI* algorithm.

Answer to RQ1: The incrementally updated model is closer to the true model when using the *iSVD* algorithm as compared to the *iLSI* algorithm.

5.2 RQ2: Comparing Retrieval Accuracy

Tables 5 and 6 compare the retrieval accuracy of the incremental framework (with retraining at major changes at $n_{thresh} = 100$) and the batch-mode approach for 43 JodaTime bugs and 321 AspectJ bugs using different combinations of the title and description of the bug report as the query. The last column shows the p-value computed from pairwise student's t-test to additionally confirm the statistical significance of our findings. It can be seen in the tables that for most cases the retrieval accuracy obtained by *iLSI* is significantly lower than that obtained using batch-mode. On the other hand, when using the *iSVD* algorithm, we were unable to establish statistically significant differences between the incremental and *batch-mode* approaches.

Table 5: Comparing the retrieval accuracy using Precision and Recall at different points and the rank-based metrics for 43 bugs in JodaTime using the LSA model (k=40). Columns labeled as $R1$ means $r = 1$, $R5$ means $2 \leq r \leq 5$, $R10$ means $6 \leq r \leq 10$, $R20$ means $11 \leq r \leq 20$ and $R21$ means $r \geq 21$.

Query Type	mode	P@1	R@1	P@5	R@5	P@10	R@10	$R1$	$R5$	$R10$	$R20$	$R21$	MAP	p
title	batch	0.1860	0.0860	0.1023	0.2876	0.0721	0.3822	8	9	7	3	16	0.2047	
	iSVD	0.1860	0.0860	0.1023	0.2876	0.0744	0.3899	8	10	6	2	17	0.2040	0.6752
	iLSI	0.0930	0.0395	0.0605	0.1636	0.0605	0.3163	4	8	8	1	22	0.1326	0.006
description	batch	0.1163	0.0628	0.1256	0.3008	0.0791	0.3899	5	16	2	4	16	0.2145	
	iSVD	0.1628	0.0822	0.1209	0.2853	0.0744	0.3667	7	13	2	3	18	0.2206	0.466
	iLSI	0.1163	0.0628	0.0837	0.1953	0.0581	0.2814	5	9	4	6	19	0.1593	0.008
title + description	batch	0.1395	0.0744	0.1163	0.2845	0.0837	0.4093	6	14	3	4	16	0.2197	
	iSVD	0.1395	0.0744	0.1209	0.2891	0.0814	0.3977	6	14	2	4	17	0.2177	0.7055
	iLSI	0.1163	0.0628	0.0791	0.1915	0.0628	0.3047	5	8	5	5	20	0.1625	0.007

Table 6: Comparing the retrieval accuracy using Precision and Recall at different points and the rank-based metrics for 321 bugs in AspectJ using the LSA Model (k=60). Columns labeled as $R1$ means $r = 1$, $R5$ means $2 \leq r \leq 5$, $R10$ means $6 \leq r \leq 10$ and $R20$ means $11 \leq r \leq 20$ and $R21$ means $r \geq 21$.

Query type	mode	P@1	R@1	P@5	R@5	P@10	R@10	$R1$	$R5$	$R10$	$R20$	$R21$	MAP	p
title	batch	0.0648	0.0321	0.0346	0.0811	0.0287	0.1248	21	27	29	31	216	0.0692	
	iSVD	0.0586	0.0290	0.0340	0.0774	0.0265	0.1183	19	28	23	38	216	0.0666	0.131
	iLSI	0.0463	0.0223	0.0309	0.0699	0.0250	0.1159	15	28	24	40	217	0.0603	0.009
description	batch	0.0342	0.0162	0.0329	0.0832	0.0289	0.1277	11	37	28	34	212	0.0642	
	iSVD	0.0342	0.0162	0.0348	0.0844	0.0283	0.1208	11	39	25	31	216	0.0634	0.682
	iLSI	0.0373	0.0172	0.0298	0.0796	0.0252	0.1165	12	32	27	30	221	0.0592	0.088
title + description	batch	0.0463	0.0238	0.0333	0.0788	0.0312	0.1312	15	35	31	31	212	0.0700	
	iSVD	0.0463	0.0238	0.0340	0.0784	0.0318	0.1349	15	35	35	27	212	0.0697	0.851
	iLSI	0.0340	0.0180	0.0315	0.0777	0.0281	0.1281	11	36	32	29	216	0.0627	0.016

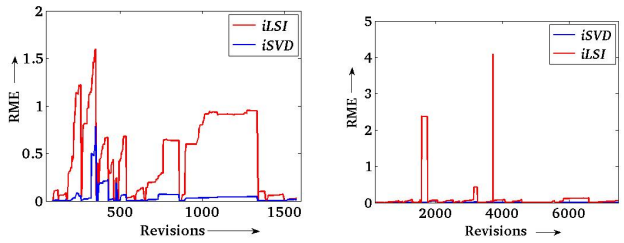


Figure 2: Relative Modeling Error of the incrementally updated LSA model for JodaTime at k=20 (left) and AspectJ at k=100 (right) (see in color).

5.2.1 Sensitivity to the Parameter k

Figures 3 (a)-(c) and 4 (a)-(c) show the variation in the retrieval accuracy with respect to k for the batch-mode and the two incremental algorithms for JodaTime and AspectJ respectively using different combinations of the title and the description fields of the bug report as the query. For both datasets, the ranking of the algorithms in terms of retrieval accuracy is as follows: *iLSI* $\prec$ *iSVD* $\preceq$ batch.

Answer to RQ2 The retrieval accuracy of *iSVD* is comparable to that of batch-mode and the retrieval accuracy suffers when using the *iLSI* algorithm.

5.3 RQ3: Evaluating Improvements in Time Performance

Table 7 presents the mean and the median of the time spent in each step of retrieval for the *batch-mode* and the *incremental mode* 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 revision. Columns 5 and 8 show that the degree of speed-up obtained in each of the stages of the retrieval process is significant. Additionally, since MUT, IUT and CPT depend on the size of the *change-set*, they remain more or less constant for both JodaTime and AspectJ. For example, as shown in the last two rows of Table 7, the median of MUT for *iSVD* and *iLSI* for both JodaTime and AspectJ is under 1 second.

5.3.1 Query Latency

As shown in Table 8, the *Query Latency* (measured in seconds) is significantly reduced with the incremental update framework as the model is always kept up-to-date.

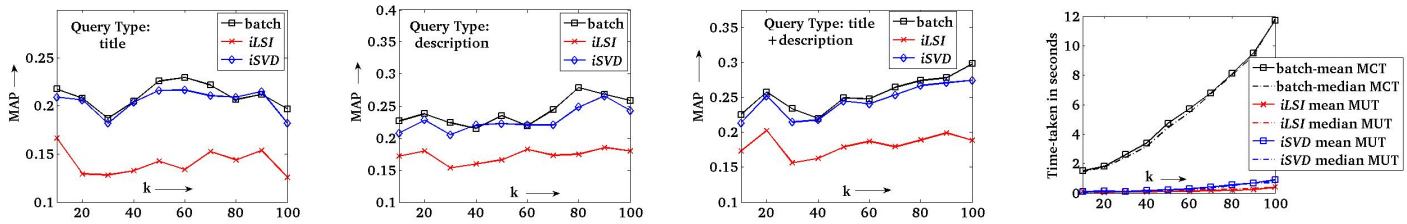
5.3.2 Net Computational Effort in Keeping the Model Updated

Since the *change-set* is relatively small, the overall time spent in keeping the model updated (NCE) is just around 2 seconds for most revisions and 8 seconds on average, as shown in Table 9.

5.3.3 Sensitivity of MCT and MUT to k

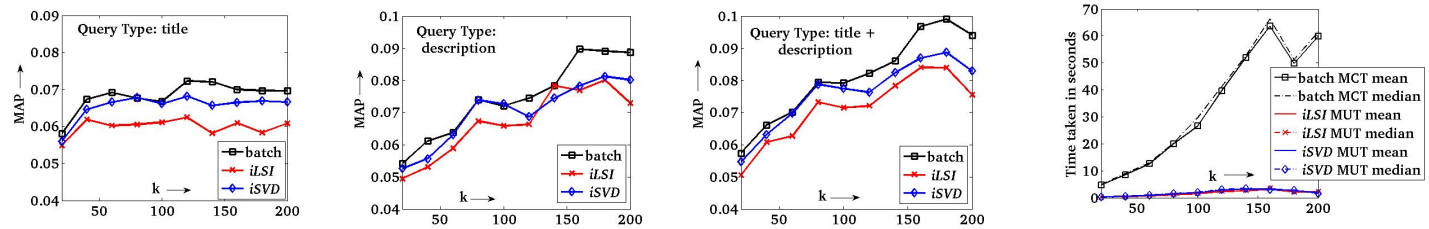
The variation of MCT and MUT to the parameter k is plotted in Figures 3(d) and 4(d) for the two software repositories. As k increases, the figure shows that MCT increases more rapidly than MUT. In other words, incremental update techniques save time in the model update stage. The figures also compare the *iSVD* and *iLSI* algorithms in terms of MUT indicating that *iLSI* is marginally faster than *iSVD*.

Answer to RQ3: With the incremental update framework, significant speed-ups can be achieved in various stages of the retrieval process. The Net Computational Effort (NCE) required at each commit to keep the model updated was found reasonable within a few seconds. Both *iSVD* and *iLSI* are significantly faster than the batch-mode LSA algorithm.



(a) (b) (c) (d)

Figure 3: (a)-(c) Sensitivity of retrieval accuracy to the parameter k of the LSA model using different types of query for 43 bugs in JodaTime (d) Variation of MCT and MUT with k for JodaTime (see in color).



(a) (b) (c) (d)

Figure 4: (a)-(c) Sensitivity of retrieval accuracy to the parameter k of the LSA model using different types of query for 321 bugs in AspectJ (d) Variation of MCT and MUT with k for AspectJ (see in color).

Table 7: Summary of the time taken (in seconds) by each of the stages of the batch-mode and the incremental mode framework.

		JodaTime (k=40)			AspectJ (k=60)		
		mean	median	gain	mean	median	gain
# of files	batch	556	494		5214	5309	
	inc	5.41	2		4.429	1	
Pre-proc	BPT	412.7	303.7	89	1628	1052.7	246
	CPT	7.83	2.07	270	6.61	1.96	536
Index--ing	ICT	44.97	36.23	133	170.15	153.68	240
	IUT	0.34	0.19	188	0.71	0.29	607
iSVD	MCT	3.39	3.18	12-	12.58	12.93	15
	MUT	0.27	0.20	15	0.85	0.87	
iLSI	MCT	3.39	3.18	26 -	12.58	12.93	18
	MUT	0.13	0.11	28	0.71	0.71	

Table 8: Comparing the Query Latency (in seconds) of the batch-mode and incremental mode approaches to bug localization.

	JodaTime (k=40)		AspectJ (k=60)	
Model	mean	median	mean	median
batch	459.352	341.49	1806	1214.18
iSVD	0.3185	0.2483	0.3185	0.2483
iLSI	0.8131	0.725	1.3869	1.202

Table 9: Net Computational Effort (in seconds) to keep the model updated using the incremental update framework.

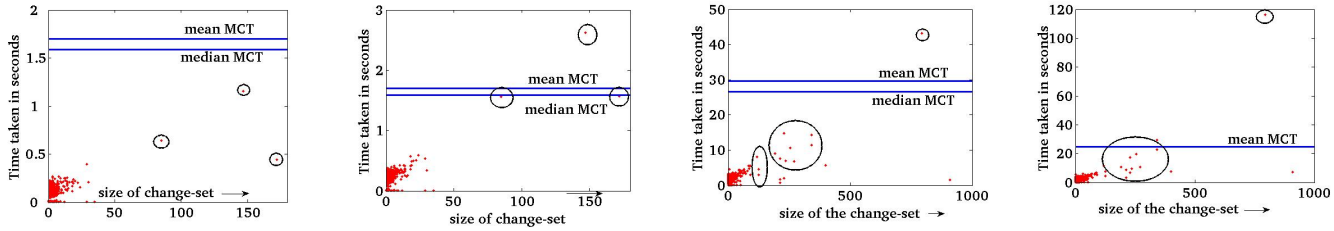
	JodaTime (k=40)		AspectJ (k=60)	
Model	mean	median	mean	median
iSVD	8.378	2.359	8.296	3.137
iLSI	8.329	2.352	7.6092	2.469

5.4 RQ4: When to Retrain?

Recall from section 3.1 that it may be necessary to retrain the model from scratch and the decision of when to retrain can only be handled by a heuristic. One possible policy is to re-compute the model only at major releases of the software (*major_releases*). We found that while this approach guarantees that the model error measure (RME) and the retrieval accuracy are not impacted[2], the efficiency may suffer. In order to demonstrate this, we plot the variation in MUT with respect to the size of the *change-set* for the two software repositories and the two incremental update models in Figure 5. Note that for some revisions in JodaTime's history the time taken to update the model is comparable to that of batch-mode time. These commits typically affect $16 - 34\%$ of the (or > 100) source files and hence the assumption of $M_a << M$ is no longer valid. In other words, the design choice of retraining at *major_releases* may guarantee retrieval accuracy but not retrieval effectiveness.

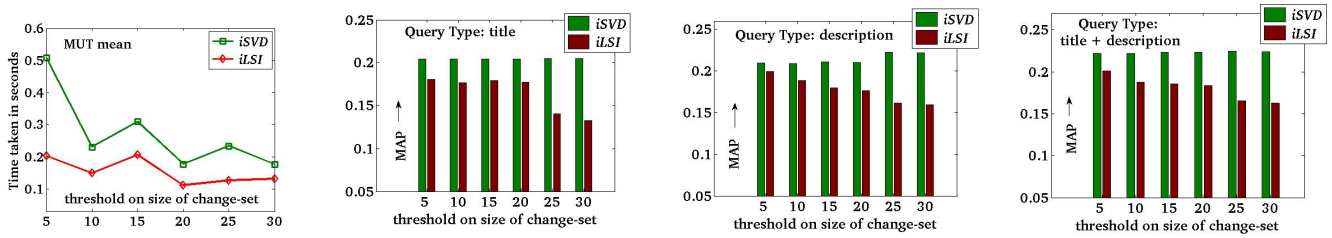
Alternatively, one could re-compute the model when a significant portion of the source files are affected (*major_changes*). In order to identify such commits, a threshold is set on the size of the *change-set* (n_{thresh}). When a commit affects more source files than the set threshold, we retrain the model; otherwise the model is incrementally updated. We study the impact of this threshold on the retrieval efficiency of the two incremental update algorithms *iLSI* and *iSVD*. When n_{thresh} is high, the model is retrained less frequently and the mean MUT is lower (see Figures 6 & 7 (a)). We also studied the variation in the retrieval accuracy (using MAP) with respect to n_{thresh}. Figures 6 & 7 (b)-(d) show the variation in the retrieval accuracy computed using the *iSVD* and the *iLSI* algorithms. Note that since the *iLSI* ignores new information (source files/terms) present in A^{t+1}, retrieval accuracy suffers when this threshold (n_{thresh}) is increased. On the other hand, since the *iSVD* algorithm more directly incorporates new information into the model, the retrieval accuracy stays intact, even when the threshold (n_{thresh}) is increased. Thus with the *iLSI* algorithm one may need to retrain the model more frequently than when the *iSVD* algorithm is used.

[2]Retrieval accuracy and RME follow similar trends as the ones shown in Sections 5.2 and 5.1. We have omitted the results to save space and avoid duplication.



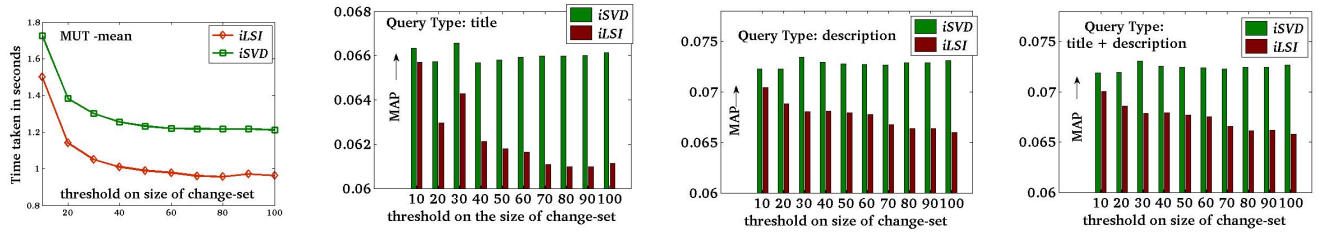
(a) JodaTime-*iLSI* (b) JodaTime-*iSVD* (c) AspectJ-*iLSI* (d) AspectJ-*iSVD*

Figure 5: Sensitivity of MUT to the size of *change-set* with retraining at *major_releases* policy. The circled data points correspond to revisions where a major fraction of source files ($> 16\%$ or > 100 source files) were affected. Observe that for these commits, MUT becomes comparable to the MCT (see in color).



(a) MUT (b) title (c) description (d) title+description

Figure 6: (a) Sensitivity of retrieval efficiency to n_{thresh} (threshold on size of change-set) with retrain at *major_changes* policy (b)-(d) Sensitivity of retrieval effectiveness to n_{thresh} for 43 bugs in JodaTime software (see in color).



(a) MUT (b) title (c) description (d) title+description

Figure 7: (a) Sensitivity of retrieval efficiency to n_{thresh} (threshold on size of change-set) with retrain at *major_changes* policy (b)-(d) Sensitivity of retrieval effectiveness to n_{thresh} for 321 bugs in AspectJ software (see in color).

Answer to RQ4: It is more efficient and effective to retrain the LSA model at commits where significant portion of the source files are affected, as opposed to retraining at major releases. The *iSVD* algorithm incorporates new information (source files/terms) more directly into the model, leading to lower model error. Thus the *iSVD* algorithm requires less retraining compared to the *iLSI* algorithm.

6. THREATS TO VALIDITY

Any empirical research must be subject to an analysis of threats to its validity. In our previous contribution [10], we have highlighted the threats to validity of the incremental update framework to IR based bug localization. Due to space restrictions, we refrain from repeating such limitations. Regarding the incremental LSA methods, one threat to the validity is our vocabulary update mechanism. While incorporation of new source files into the index is straightforward (see Section 3), vocabulary updates are not so direct. Most often, when computing the LSA model, the removal of the n most frequent/least frequent terms in the collection is a common preprocessing step [2]. Let us denote this set as $\mathcal{V}_{elim}$. As the vocabulary evolves, heuristics need to be applied to incorporate these terms selectively into the index. In our current implementation, a new term is added to the vocabulary only if it does not belong to $\mathcal{V}_{elim}$. One threat to validity is that $\mathcal{V}_{elim}$ is computed only when an index is computed afresh and we do not update this set as the software evolves. Nevertheless, software vocabulary evolves at a very slow rate [29] [23], and thus we do not expect this to impact the conclusions we derive from our experimental findings.

7. RELATED WORK

SVD is a fundamental matrix operation that requires intensive computation, both in terms of time and storage space. Thus a number of optimizations have been proposed in the past. Although we have used the popular Lancsoz SVD algorithm [28], alternate fast-SVD algorithms exist. One such approach is called the Stochastic SVD algorithm (SSVD) [24] that uses randomized algorithms to achieve extremely fast SVD computation at a small sacrifice to the accuracy. Incremental versions of the SSVD algorithm have also been proposed to process large term-document matrices in smaller chunks [26]. Such incremental SVD algorithms achieve speed up by just keeping track of U and S components and eliminating the need to store A. Unfortunately, the incremental versions of the SSVD algorithm are essentially sequential in nature and cannot consider cases in which existing rows and columns are deleted or modified. Thus, we have not explored this avenue of research any further.

8. CONCLUSION

In this paper, we have presented an incremental approach to IR based bug localization using the LSA model. Our proposed approach keeps the model updated with changes in the software from one commit to the next. We have compared two state-of-the-art incremental SVD algorithms

— Brand's incremental SVD [22] and Jiang et al.'s *iLSI* [21]. For experimental validation, we created a publicly available benchmark dataset called *moreBugs* that tracks commit-level changes over 10 years of history of the following software repositories: AspectJ (7477 commits) and JodaTime (1573 commits). We also presented strategies for retraining the model using batch-mode from time to time. We have demonstrated that significant speed-up in the retrieval process can be achieved using the presented approach. Our analysis and findings reveal that the *iLSI* algorithm is not suitable for incrementally updating the model of a software repository as it ignores information found in new source files and terms. This causes high model error and poor retrieval accuracy compared to the *iSVD* algorithm and the *batch-mode* algorithm. Consequently, one might need to retrain the model more frequently when using the *iLSI* algorithm as opposed to *iSVD* algorithm.

9. 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*. 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, pp. 50 –59.

[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 *Software Engineering (ICSE), 2012 34th International Conference on*, June 2012, pp. 14 –24.

[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] 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.

[8] 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.

[9] 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*, 2013, pp. 309–318.

[10] S. Rao, H. Medeiros, and A. Kak, "An Incremental Update Framework for Efficient Retrieval from Software Libraries for Bug Localization," in *Working Conference on Reverse Engineering*, 2013.

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

[12] J. I. Maletic and A. Marcus, "Supporting Program Comprehension Using Semantic and Structural Information," in *Proceedings of the 23rd International Conference on Software Engineering*, ser. ICSE '01, 2001, pp. 103–112.

[13] A. Kuhn, S. Ducasse, and T. Girba, "Enriching Reverse Engineering with Semantic Clustering," in *Reverse Engineering, 12th Working Conference on*, 2005, p. 10.

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

[15] D. Poshyvanyk and A. Marcus, "Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code," in *Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on*, 2007, pp. 37–48.

[16] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Combining Probabilistic Ranking and Latent Semantic Indexing for Feature Identification," in *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, 2006, pp. 137–148.

[17] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An Information Retrieval Approach to Concept Location in Source code," in *In Proceedings of the 11th Working Conference on Reverse Engineering*, 2004, pp. 214–223.

[18] A. Marcus and J. I. Maletic, "Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing," in *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, 2003, pp. 125–135.

[19] A. De Lucia, R. Oliveto, and G. Tortora, "Adams Re-Trace: Traceability Link Recovery via Latent Semantic Indexing," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08, 2008, pp. 839–842.

[20] A. Marcus and J. Maletic, "Identification of High-Level Concept Clones in Source Code," in *Automated Software Engineering, 2001. (ASE 2001). Proceedings. 16th Annual International Conference on*, 2001, pp. 107–114.

[21] H. yi 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.

[22] M. Brand, "Fast Low-Rank Modifications of the Thin Singular Value Decomposition," *Linear Algebra and its Applications*, vol. 415, no. 1, pp. 20 – 30, 2006.

[23] 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.

[24] N. Halko, P. G. Martinsson, and J. A. Tropp, "Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions," *SIAM Review*, vol. 53, no. 2, pp. 217–288, May 2011.

[25] H. Zha and H. D. Simon, "On Updating Problems in Latent Semantic Indexing," *SIAM J. Sci. Comput.*, vol. 21, no. 2, pp. 782–791, Sep. 1999.

[26] R. Řehůřek, "Subspace Tracking for Latent Semantic Analysis," in *European Conference on Information Retrieval*, 2011, pp. 289–300.

[27] "Stochastic svd," http://code.google.com/p/redsvd/.

[28] G. H. Golub and C. F. V. Loan, *Matrix Computations*, 3rd ed. The Johns Hopkins University Press, Oct 1996.

[29] S. Abebe, S. Haiduc, A. Marcus, P. Tonella, and G. Antoniol, "Analyzing the Evolution of the Source Code Vocabulary," in *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on*, March 2009, pp. 189 –198.