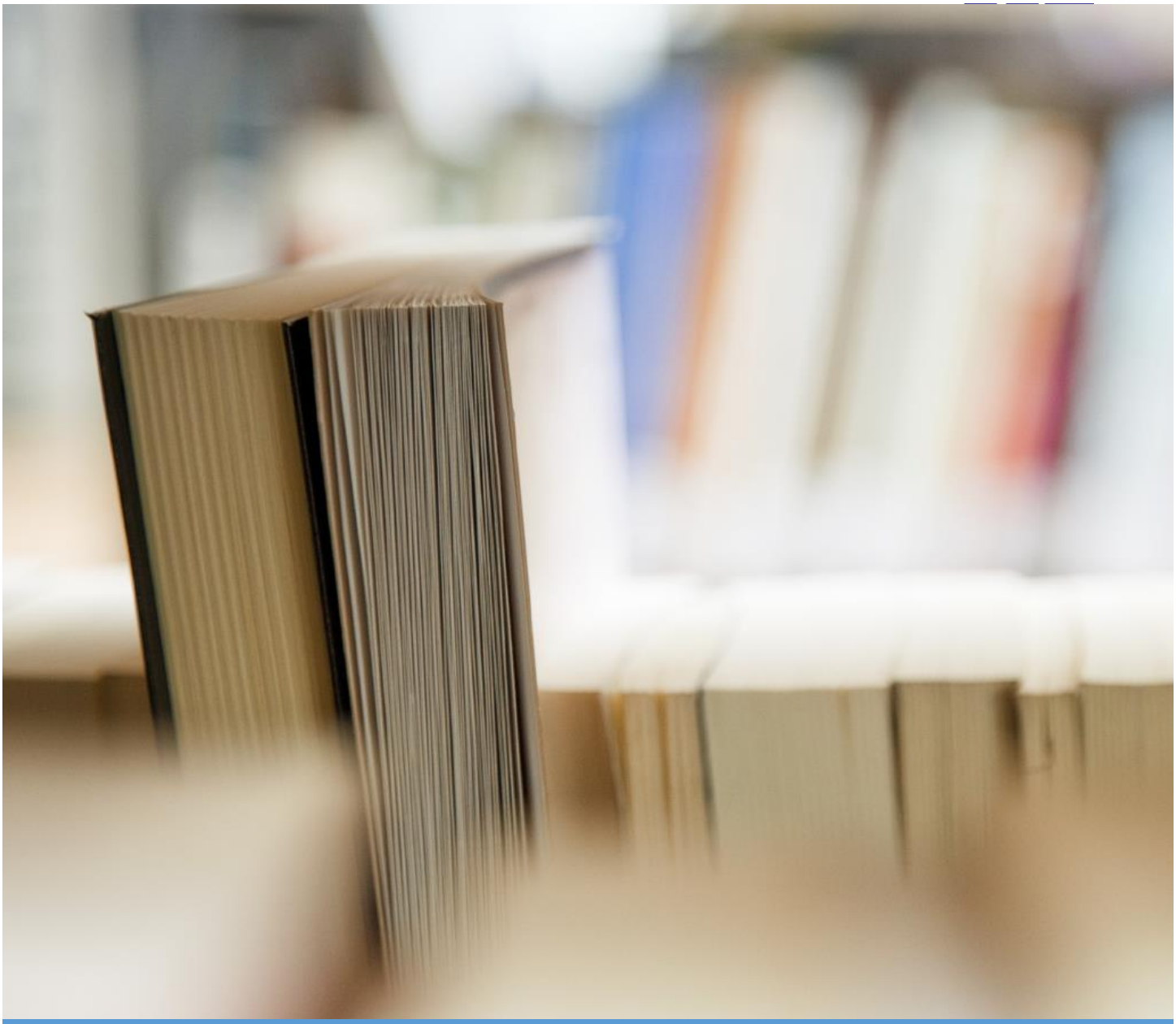

SPL-III Report

A Research and Development Project…

M. A. Nur Quraishi

12/14/17

SE-801: SPL-3

Project Name:

"Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing"

Written by

M. A. Nur Quraishi
Roll: BSSE-0615
Institute of Information Technology,
University of Dhaka

Supervised by

Rayhanur Rahman
Lecturer,
Institute of Information Technology,
University of Dhaka

_____ _____

Signature of the Supervisor Signature of the Student

Letter of Transmittal

Amit Seal Ami

Lecturer,

and

Dr. B M Mainul Hossain

Assistant Professor,

Institute of Information Technology,

University of Dhaka

December 14, 2017

Honorable Sir,

With due respect and humble submission, I want to inform you that, I have submitted the enclosed research report on “Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing” for your approval. This report covers all the knowledge fields to understand the work and also records the key findings of the experiments based on the evaluation.

The principal purpose of this document is to depict the literature survey and methodology along with experimental results and analysis required for the research and development project of Software Project Lab-III. The report contains each and every information in detail that I have followed to conduct the research and development project.

Sincerely yours,

M. A. Nur Quraishi (BSSE-0615)

Institute of Information Technology

University of Dhaka

Session: 2013-14

Enclosure: Research Report

IIT
University of Dhaka

Abstract

An information retrieval method named latent semantic indexing is applied to diagnose and analyze the traceability links from system documentation to program source code automatically. It is a very cost effective and highly flexible method to apply with regards to preprocessing and/or parsing of the source code and documentation. The experimental results of two Java projects to identify links in those applications (i.e., the Braille to Text Translator and the BD Travelers) are demonstrated. The method presented in this paper proves to give good and effective results. Moreover, it is of low cost, a highly flexible approach to apply considering the preprocessing and parsing of the source code and documentation.

Acknowledgements

At first, I want to express my heartiest gratitude and show unmeasurable fidelity to the Almighty, ALLAH for completing the project in due time and without any hassles.

Next, I would like to thank especially my project supervisor Rayhanur Rahman and course instructors Amit Seal Ami and Dr. B M Mainul Hossain for their kind and valuable supervision, advice, instruction and time throughout the project. I will be always grateful to my project supervisor for his kind support and knowledge sharing. Without his assistance, it may be impossible to accomplish the project in due time with proper effectiveness and efficiency.

I would also like to thank my honorable elder brother, Dipok Chandra Das for his assistance and support. I am also grateful to my elder brother, Minhas Kamal and my fellow Sharafat Ahmed Sabir for sharing their project.

IIT
University of Dhaka

Table of Contents

1 Introduction

Most of the documentations including Software Requirement Specification (SRS), design documents, user manuals and also annotations of individual programmers and teams associated with an application or large software system are free text documents in form of either PDF or DOC interpreted in a natural language. Moreover, these documents often cover the available knowledge of the application domain. Even when (semi-)formal models are applied, free text is largely adopted either to add semantics and context information in the form of comments or ease the understanding of the formal models to nontechnical readers. [1]

The whole software engineering community (both research and industrial) in this era stride to improve the explicit connection of documentation and source code. A number of integrated development environments and Computer Aided Software Engineering (CASE) tools are particularly focused on this issue. These tools and techniques have played an important role in documentation to source code traceability for the development of new software systems. Miserably, several of these methods intrinsically incompatible for existing and/or legacy systems.

The need for tools and applications to recover documentation to source code traceability links in legacy systems is particularly important for various software engineering tasks. Such as- general maintenance tasks, impact analysis, program comprehension, and more encompassing tasks such as reverse engineering for redevelopment and systematic reuse. [2]

The preeminent obstacle in link recovery process is that the links are rarely explicit and based on the semantic meaning of the prose in the documentation. Contrasting some sort of natural language analysis of the documentation with that of the source code is an obviously difficult problem. Here, we are offering a cost effective and more pragmatic solution to this problem by utilizing an advanced information retrieval technique (latent semantic analysis) to extract the meaning (semantics) of the documentation and source code. Then use this information to identify traceability links based on similarity measures.

The mechanism uses all the comments (internal documents) and identifier names within the source code to produce semantic meaning with respect to the entire input document space. This experimental study is well-supported by the work of Anquetil [3] and others in determining the importance of this information in existing software. This entails the postulates that the comments and identifiers are reasonably named however, the alternative bares little hope of deriving a meaning automatically (or even manually).

The major advantage of using this method is that it does not depend on predefined vocabulary or grammar (including sentence structure) for the documentation and source code. As a result, this procedure does not require large amounts of preprocessing or manipulation of the input, which drastically reduces the costs of link recovery.

2 Background Study

There is a wide range of information retrieval (IR) mechanisms. Conventional approaches [4] such as signature files, inversion, classifiers, and clustering. Other methods applying parsing, syntactic information, natural language processing techniques, neural networks, and advanced statistical methods that attempt to capture more information about the documents in order to achieve better performance. Much of this work deals with natural language text and a large number of techniques exist for indexing, classifying, summarizing, and retrieving text documents. These methods produce a profile for each document where the profile is an abbreviated description of the original document that is easier to manipulate. This profile is typically represented as vector, often real valued. The method, we are going to implement also has an underlying vector space model. We are now going to present a discussion of these type of representations. [2]

2.1 Vector Space

A vector space which is also known as linear space consists of a set of objects named vectors, which can be added with each other and multiplied by numbers named scalars. Scalars are commonly anticipated to be real numbers. But A number of vector spaces exist with scalar multiplication by complex numbers, rational numbers or any field. The operations of vector addition and scalar multiplication must fulfill certain set of requirements which is called axioms.

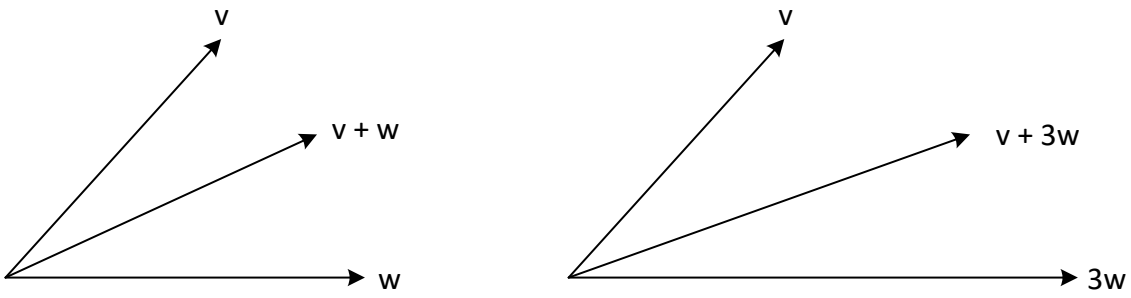

Figure 2.1.1: Geometric view of vector space.

The axioms are listed below:

- It must demonstrate physical quantities or measures like forces: any two forces (are of the same category) can be added together to produce a third one and the result of multiplication of a force vector by a real multiplier is another force vector. Geometrically, vectors illustrating displacements in the plane or in three-dimensional space also create vector spaces. The vectors in vector spaces do not essentially need to be arrow like objects as it is shown in the figure 2.1.1. Here, vectors are considered as abstract mathematical objects with certain attributes. In some cases, these can be visualized through arrows. For an example- Euclidean vectors

- Vector spaces are the topic of linear algebra and are precisely distinguish by their dimension. At large, the dimension defines the number of independent directions in the space. Vector spaces with infinite-dimension may arise in mathematical analysis as function spaces, whose vector elements are functions. These vector spaces are commonly illustrated with additional structure which may be a topology by considering the issues of proximity and continuity. Among these topologies, those that are specified by a norm or inner product are more commonly used as having a notion of distance between two vectors. For example- Banach spaces and Hilbert spaces.

2.2 Vector Space Model

The vector space model (VSM) [5] is a widely used classic method for constructing vector representations for documents. It encodes a document collection by a term-by-document matrix whose [i, j]th element indicates the association between the ith term and jth document. In typical applications of VSM, a term is a word, and a document is an article. However, it is possible to use different types of text units. For instance, phrases or word/character n-grams can be used as terms, and documents can be paragraphs, sequences of n consecutive characters, or sentences.

The importance of VSM is that it represents one type of text unit (documents) by its association with the other type of text unit (terms) where the association is calibrated by explicit evidence based on term occurrences in the documents. A geometric view of a term-by-document matrix is as a set of document vectors occupying a vector space spanned by terms. We call this vector space VSM space.

The similarity between documents is typically measured by the cosine or inner product between the corresponding vectors, which increases as more terms are shared. In general, two documents are considered to be similar if their corresponding vectors in the VSM space point in the same (general) direction. [2]

2.3 Latent Semantic Indexing

Latent Semantic Indexing (LSI) [6] is based on VSM for inducing and representing aspects of the meanings of words and passages reflective in their usage.

Experiment applying LSI to natural language text by [7] has shown that LSI not only covers significant portions of the meaning of individual words but also of whole passages such as sentences, paragraphs, and short essays. The basic idea of LSI is that the information about word contexts in which a particular word appears, or does not appear, provides a set of mutual constraints that determines the similarity of meaning of sets of words to each other. [2]

2.4 Singular Value Decomposition and LSI

For text analysis, LSI uses a user-constructed corpus to create a term-by-document matrix. Then it applies Singular Value Decomposition (SVD) [5] to the term-by-document matrix to construct a subspace, called an LSI subspace. New document vectors (and query vectors) are obtained by orthogonally projecting the corresponding vectors in a VSM space (spanned by terms) onto the LSI subspace. Following the mathematical formulation of LSI, the term combinations which are less frequently occurring in the given document collection like to be excluded from the LSI subspace. According to our examples above, one could argue that LSI performs “noise reduction” if it was true that less frequently co-occurring terms are less mutually-related, and therefore less sensible.

The factor behind using SVD is rather complex and lengthy to elaborate here. The interested reciter is referred to [5] for details. Apparently, in SVD [2] a rectangular matrix X is decomposed into the product of three other matrices. One component matrix (U) describes the original row entities as vectors of derived orthogonal factor values. Another component (V) describes the original column entities in the same way, and the third is a diagonal matrix (Σ) containing scaling values such that when the three components are matrix-multiplied, the original matrix, X is reconstructed (i.e., $X = U\Sigma V^T$). The columns of U and V are the left and right singular vectors, respectively, corresponding to the monotonously decreasing (in value) diagonal elements of Σ which are called the singular values of the matrix X. When fewer than the necessary number of factors are used, the reconstructed matrix is a least-squares best fit. One can reduce the dimensionality of the solution simply by deleting coefficients in the diagonal matrix, ordinarily starting with the smallest. The first k columns of the U and V matrices and the first (largest) k singular values of X are used to construct a rank-k approximation to X through $X_k = U_k\Sigma_k V_k^T$. The columns of U and V are orthogonal, such that $U^TU = V^TV = I_r$, where r is the rank of the matrix X. X_k constructed from the k-largest singular triplets of X (a singular value and its corresponding left and right singular vectors are referred to as a singular triplet), is the closest rank-k approximation (in the least squares sense) to X.

With regard to LSI, X_k is the closest k-dimensional approximation to the original term-document space represented by the incidence matrix X. As stated previously, by reducing the dimensionality of X, much of the “noise” that causes poor retrieval performance is thought to be eliminated. Thus, although a high-dimensional representation appears to be required for good retrieval performance, care must be taken to not reconstruct X. If X is nearly reconstructed, the noise caused by variability of word choice and terms that span or nearly span the document collection won't be eliminated, resulting in poor retrieval performance. [2]

The implementation of LSI has been empirically studied. The experiment in [8] explores the effects of several term weighting schemes to instantiate the input term-by-document matrix. Evaluation was based on the precision/recall curves on the retrieval tasks with the dimensionality of the LSI subspace being fixed. Several term-weighting schemes, which combine global weights

(i.e., statistics in the collection) and local weights (i.e., statistics within each document), were investigated. Log Entropy, which is a combination of a local log weight and a global entropy weight, showed superiority over the combinations of the local term frequency and global weighting schemes or no global weighting. Two well-known global weightings (i.e., Gfldf and Normal) produced performance worse than no global weighting.

Particularly, various experimental studies have shown that the performance of LSI significantly varies over the dimensionalities of the LSI subspace [6]. In practice, the dimensionality is chosen experimentally, or blindly picked by following the existing work.

After the documents are imitated in the LSI subspace [2], the user can compute similarities measures between documents by the cosine between their corresponding vectors or by their length. These measures can be used for clustering similar documents together, to identify concepts and topics in the corpus. This method is typically used for text analysis tasks. The LSI representation can also be applied to map new documents (or queries) into the LSI subspace and find which of the existing documents are similar (relevant) to the query. This usage is typical for information retrieval tasks.

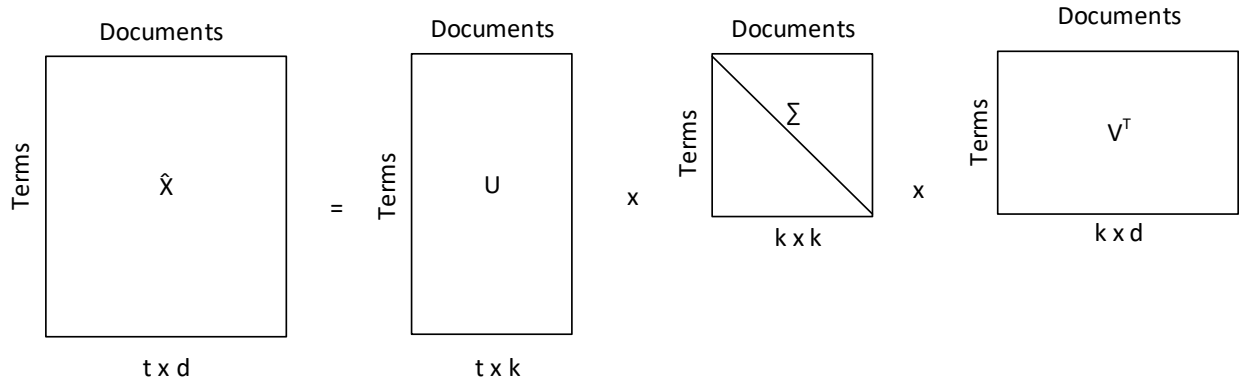

Figure 2.2.1: Singular value decomposition of the term-by-document matrix, X (Source: [6]).

Here,

$\hat{X}$ is the approximate matrix of X

U has orthogonal, unit-length columns ($UU^T = I = U^TU$)

V has orthogonal, unit-length columns ($VV^T = I = V^TU$)

$\sum$ is the diagonal matrix of singular values

t is the number of rows of X

d is the number of columns of X

m is the rank of X ($\leq \min(t, d)$)

k is the chosen number of dimensions in the reduced model ($k \leq m$)

2.5 Why LSI?

A major drawback of VSM is that it does not consider relations between terms. For instance, having "automobile" in one document and "car" in another document does not contribute to the similarity measure between these two documents.

The fact that VSM produces zero similarity between text units that share no terms is an issue, especially in the information retrieval task of measuring the relevance between documents and a query submitted by a user. Usually, a user query is tiny and does not consider all the vocabulary for the target concept. In VSM, “car” in a query and “automobile” in a document do not contribute to retrieving this document (synonym problem). LSI attempts to overcome this problem by selecting linear combinations of terms as dimensions of the representation space. The examples in [6] & [9] show that LSI may solve this synonym problem by generating positive similarity between related documents sharing no terms.

As the LSI subspace grabs the most important factors (i.e., those contains the largest singular values) of a term-by-document matrix, it is anticipated to capture the relations of the most repeatedly simultaneously occurring terms. This fact is understood when we realize that the SVD factors a term-by-document matrix into the largest one-dimensional projections of the document vectors, and that each of the document vectors can be regarded as a linear combination of terms. From this perspective, LSI can be viewed as a corpus-based statistical method. However, the relations among terms are not modeled explicitly in the computation of LSI subspace, which makes it hard to understand LSI in general. Despite an LSI subspace provides the best low rank and cost effective approximation of the term-by-document matrix, it does not denote that the LSI subspace approximates the “true” semantics of documents.

Another criticism of this method is that when it applied to natural language texts, it does not capitalize on word order, syntactic relations, or morphology. Nevertheless, very good representations and results are acquired without this information. This feature is very much suitable to the domain of source code and internal documentation (Programmer annotations or comments). Because much of the informal abstraction of the problem concept may be embodied in names of key operators and operands of the implementation. Here, the word ordering is meaningless. Source code is hardly English prose but with the convention of selective naming, much of the high level meaning of the problem-at-hand is transmit to the reader (the programmer). Internal source code documentation (comment) is also commonly written in a subset of English that also utilized in IR methods. This makes automation exceptionally easier and directly supports programmer defined variable names that have implied meanings (e.g., avg) which are not in the English language vocabulary. The meanings are determined from their usage rather than a predefined dictionary. This is a certain advantage over using a traditional natural language processing approach.

Like other IR methods, LSI does not use a grammar or a predefined or assumed vocabulary. However, it uses a list of “stop words” (like- to, and, the etc.) that can be extended by the user.

These words are excluded from the analysis. Regardless of the IR method used in text analysis, in order to identify two documents as similar they should have in common concepts represented by the association of terms and their context of usage in the document. In other words, two documents written in different languages will not appear similar. In the case of source code, our main assumption is that developers use the same natural language (e.g., English, French, German etc.) in writing internal documentation (comments) and external documentation (SRS). In addition, the developer should have some sense of humor and consistency in defining and using identifiers. [2]

3 Related Works

The research topic on which we are working in this paper specifies two distinct problems: applying IR methods to support software engineering tasks and recovering source code to documentation links. The previous works [10] & [11] related to this research uses a technique named indexing reusable components as information retrieval method in order to identify the traceability link from documentation to source code. Remarkably, the work of Maarek [11] on the use of information retrieval method for automatically constructing software libraries contributes most among them. The success of this work as well as the incompetence and less cost effectiveness of constructing the knowledge base associated with natural language parsing methods to this problem [12] are the main motivations behind our work. In brief, it is very costly (and often unrealistic) to build the knowledge base(s) necessary for parsing methods to derive more significant semantic information from source code and corresponding documentation. Employing IR methods (based on statistical and heuristic methods) may not generate a result close to the actual one, but they are pretty cost effective to implement. From these stuidies, we can conclude that it may generate close and low cost results.

More recently, [13] used LSI to extract similarity measures among source code elements. Later, these measures were used to cluster the source code to support the identification of abstract data types in procedural code. Moreover, these measures were used to define a cohesion metric for components. The work on which we are working extends these results in a new direction.

Concurrently, Antoniol and others inspected the use of IR methods to support the traceability link recovery process. In particular, they used both a probabilistic method [14] and a vector space model [15] to identify links between source code and documentation and between source code and requirements. Their results were decent in each case. Moreover, this work also supports the selection of vector space models over probabilistic IR. Applying LSI makes the work presented here quite different in many aspects and yet provides complementary results.

SRS and design documents are generally written in natural languages like- English, Roman etc. It represents almost all key features and scenarios. Nowadays, the process of traceability link identification is relied on information retrieval techniques like- probabilistic approach, vector space model, and latent semantic indexing. Former works consider both documentation and

source code as a content of plain text. But the quality of retrieved links can be enhanced by prescribing additional structure as they are software engineering documents. In the research work of [16] presents four improved strategies to boost up the traditional LSI method based on the special characteristics of internal and external documentation and source code. These strategies are source code clustering, identifier classifying, similarity thesaurus, and hierarchical structure enhancement. The empirical study notifies that the first three strategies can increase the precision of retrieved links by 5% to 16%. Whereas, the last strategy improves the precision about 13%.

In the work of [17] , represents several case studies related to information retrival technique, basically LSI. They proposed an enhanced term weighting scheme, i.e., Developers Preferred Term Frequency/Inverse Document Frequency (DPTF/IDF). It uses the perception of the developers' preferred types of Source Code Entities to draw more attention to these Source Code Entities into the term weighting scheme. They integrated and tested the DPTF/IDF with LSI. They experiment three systems, such as- iTrust, Lucene, and Pooka to represent the system they proposed statistically increases the accuracy of the recovered links over a method based on LSI and the usual TF/IDF weighting scheme.

In [18], Palomba and others proposed a textual-based method that identifies the textual information like comments and identifiers included in source code and relies on textual similarity between code elements representing a code component. They mainly focused on detecting four code smell detection. Such as- Feature Envy, Blob, Promiscuous Package, and Misplaced Class using LSI as information retrieval technique with TF/IDF as term weighting schemes.

4 Proposed Methodology

Here, I try to cover the conceptual and technical view of the proposed methodology. In the conceptual view, the basic steps required for constructing corpus, mathematical knowledge required for the similarity measure computation and link recovery process are described. In the technical view section, I explain the tools and technology (why it is needed and how to use it) required for implementing the acquired conceptual knowledge.

4.1 Conceptual View

The traceability link recovery process is pivoted on Latent Semantic Indexing (LSI). It is almost an automated process. Yet, the user input is required.

Figure 4.1.1 (in the next page) represents the prime components in the traceability link recovery process. The whole process is designed following a pipeline architecture. The product of one stage delivers as the input of other stage. The user's engagement in the procedure ensues at the starting of the method for providing Java source code and corresponding documentation file (like- SRS, Manual, Design document) in PDF format.

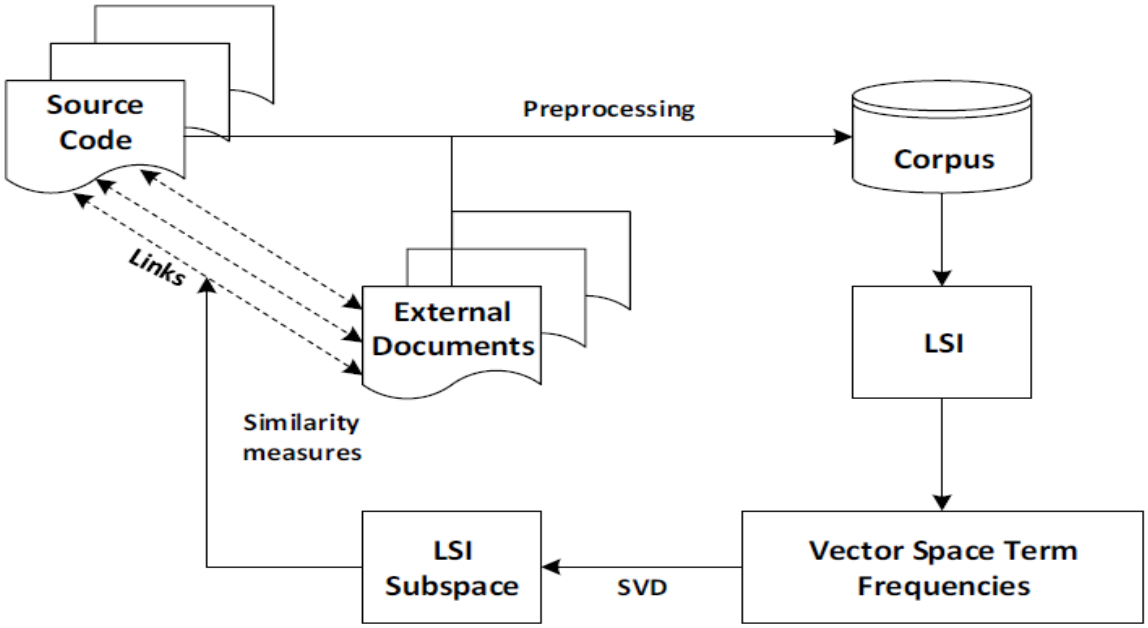

Figure 4.1.1: The traceability link recovery process (Source: [2]).

Then the user mentions the LSI subspace dimension, which will be considered as the dimension reduction unit in LSI. The user also decides the threshold value, which will be used in detecting the traceability links.

As stated earlier, identifying the links between source code and documentation may support numerous software engineering activities. Different activities usually ask for different types of information. For instance, times completeness is imperative in some of the case. That means the user requires to identify all the links that are correct even if identifying many incorrect ones at the same time is not a matter of concern. On the other hand, precision is chosen and the user confines the search space. So that, all the links which are recovered will be the correct ones, even if finding them all is not a matter of concern.

4.1.1 Constructing the Corpus

The input data contains source code and corresponding external documentation. In pursuance of constructing a corpus that goes with LSI, a simple preprocessing of the input texts is required. Both the source code and external documentation must be fragmented into the appropriate granularity to represent the documents, that will be turned out to vectors later.

When LSI is applied to natural text or document, a paragraph or section is adopted as the granularity of the input document. Sentences likely to be pretty tiny and chapters are awfully large. In case of source code, the similar perceptions are function, structure, module, file, class, etc. Apparently, the granularity statement is too cramped. Moreover, the choice of the granularity level is controlled by the distinct software engineering task. In earlier experiments

associated with LSI and source code, I apply functions as documents in procedural source code [19, 20] and class declarations in object oriented source code [21]. There, the target was to cluster elements of the source code pivoted on the semantic similarity, instead of mapping them to documentation.

In this operation, a portion of the external documentation may point to different structures or elements in the source code (like- a class, a hierarchy of classes, a set of functions or methods, a data structure, etc.). Consequently, here I specify each file as a document in order to allow flexibility and simplicity for automation. Definitely, some files will too large to be broken up. In those cases, the files can be fragmented into pieces approximately the size of the average document in the corpus. As a result, most of the documents have a close number of words and thus may map to vectors of equal lengths. Certainly, this fragmentation of the files could be rather unfortunate in some cases by mapping some parts from the source code to the wrong sections of the external document. It is a trade-off and I am wishing to take in favor of simplicity and low-cost of the preprocessing.

From the perspective of external documentation, the preferred granularity is decided by the division in sections of the documents, specified by the original authors (generally embodied in the table of contents).

Some text conversions are needed to prepare the source code and documentation to create the corpus for LSI. At first, most of the non-textual tokens from the input text are eradicated (like- operators, special symbols, some numbers, keywords of the programming language etc.). After that, the names of identifiers in the source code are split into pieces based on familiar coding standards. For examples all the following identifiers are fragmented into the words “subspace” and “dimension”: “subspace_dimension”, “Subspace_dimension”, “subspace_Dimension”, “Subspace_Dimension”, “SubspaceDimension”, “subspaceDimension”. The original name of the identifier is also kept in the documents. As I do not consider n-grams, the order of the words is not important. Last but not the least, the white spaces in the document are normalized or eliminated, blank lines separate documents, and at last the source code and documentation are combined.

One of the point to be noted in this process is that, grammar based parsing or natural language processing of the source code is not required. LSI does not utilize a predefined vocabulary, or grammar, so no morphological analysis or transformations are needed.

One can disagree that the mnemonics and words used in defining the identifier may not exist in the external documentation. It is certainly true. In fact, it is the main reason for using the internal documentation (comments) in creating the corpus. It has been observed [12] that internal source code documentation is generally composed of a subset of the language of the developer, analogous to that of external documentation. In these circumstances, the performance of LSI is beneficial to correlate the terms in the text that are in appropriate natural language (and also appear in the external documentation) with the mnemonics from the identifiers. Eventually,

these mnemonics will contribute to the similarity between two components of source code that use the same identifiers. Here I have an assumption that, developers must define and use the identifiers by considering some rationale and not entirely randomly.

4.1.2 Specifying the Semantic Similarity Measure

Before providing a detailed elaboration of this method, I need to illustrate some of the mathematical background and definitions which are necessary in this operation.

Notation: A bold lowercase alphabet (e.g., **y**) designates a vector. A vector is equivalent to a matrix having a single column. The i^{th} entry of vector y is designated by $\mathbf{y}_{[i]}$.

Notation: A bold uppercase alphabet (e.g., **X**) designates a matrix. The corresponding bold lowercase alphabet with subscript i (e.g., $\mathbf{x}_i$) designates the matrix's i^{th} column vector. The [i, j]th element of matrix **X** is designated by $\mathbf{X}_{[i, j]}$. I write $\mathbf{X} \in \mathbf{R}^{mxn}$ when matrix **X** contains m rows and n columns and its elements are real numbers.

Definition: A diagonal matrix $\mathbf{X} \in \mathbf{R}^{nxn}$ contains zeroes (0) in its non-diagonal elements, and is designated by $\mathbf{X} = \text{diag}(\mathbf{X}_{[1, 1]}, \mathbf{X}_{[2, 2]}, \ldots, \mathbf{X}_{[n, n]})$.

Definition: An identity matrix is a kind of diagonal matrix whose diagonal elements are all one (1). I designate the identity matrix in $\mathbf{R}^{mxm}$ by $\mathbf{I}_m$. For any $\mathbf{X} \in \mathbf{R}^{mxn}$, $\mathbf{XI}^n = \mathbf{I}^m\mathbf{X} = \mathbf{X}$. I exclude the subscript when the dimensionality is prominent from the context.

Definition: The transpose of matrix **X** is a matrix whose rows are the columns of X, and is designated by $\mathbf{X}^T$, i.e., $\mathbf{X}_{[i, j]}\ \mathbf{X}^T_{[j, i]}$. The columns of **X** are orthonormal if $\mathbf{X}^T\mathbf{X} = \mathbf{I}$. A matrix **X** is orthogonal if $\mathbf{X}^T\mathbf{X} = \mathbf{X}\mathbf{X}^T = \mathbf{I}$.

Definition: The vector 2-norm of $\mathbf{x} \in \mathbf{R}^m$ is designated by $|\mathbf{x}|_2 = \sqrt{\mathbf{X}^T\mathbf{X}} = \sqrt{\sum_{i=1}^{m}(\mathbf{X_i})^2}$ I also invoke it as the length of **x**.

Definition: The inner product of **x** and **y** is $\mathbf{x}^T\mathbf{y}$. So the cosine of **x** and **y** is the length-normalized inner product, designated by-

$$\cos(\mathbf{x}, \mathbf{y}) = \sqrt{\frac{\mathbf{x}^T\mathbf{y}}{|\mathbf{x}|_2 * |\mathbf{y}|_2}} \quad \ldots (1)$$

For each $\mathbf{x}, \mathbf{y} \neq 0$; $\cos(\mathbf{x}, \mathbf{y}) \in [-1, 1]$. A higher cosine value illustrates that geometrically **x** and **y** points to the similar directions. Particularly, if **x** = **y** then $\cos(\mathbf{x}, \mathbf{y}) = 1$ and **x** and **y** are orthogonal if and only if $\cos(\mathbf{x}, \mathbf{y}) = 0$.

Definition: In this procedure, a source code document (or simply document) d is any contiguous set of lines of source code. Usually, a document is a file of source code or a program entity such as a class, function, interface, etc.

Definition: An external document (e) is any contiguous set of lines of text from external documentation (like- manual, design documentation, SRS, test suites, etc.). Usually, an external document is a section, a chapter, or maybe an entire file of text.

Definition: The external documentation is also defined as a set of documents $M = \{e_1, e_2, \ldots, e_m\}$. The total number of documents in the documentation is $m = |M|$.

Definition: An application is consists of a set of documents (source code and external documentation) $S = D \cup M = \{d_1, d_2, \ldots, d_n\} \cup \{e_1, e_2, \ldots, e_m\}$. The total number of documents in the system is $|S| = n + m$.

Definition: A file f_i, is consists of a number of documents and the union of all files is S. The size of a file (f_i) is the number of documents in the file, denoted by $|f_i|$.

LSI treats the set $S = \{d_1, d_2, \ldots, d_n, e_1, e_2, \ldots, e_m\}$ as input and figure out the vocabulary V of the corpus. The number of words (or terms) in the vocabulary can be denoted by $v = |V|$. Each term is weighted with a combination of a local log weight and a global entropy weight (TF/IDF) on the basis of the frequency of the occurrence of the terms in the documents and entire document space. A term-by-document matrix $\mathbf{X} \in \mathbf{R}^{vxn}$ is formed. Then, SVD produces the LSI subspace based on the user-selected dimensionality (k). The term-by-document matrix is then reduced to the k-dimensional LSI subspace. Each document $d_i \in D \cup M$, will be represented as a vector $\mathbf{x_i} \in \mathbf{X}$ projected onto the LSI subspace.

Definition: For any two documents d_i and d_j, the semantic similarity between them is computed by the cosine between their corresponding vectors such that, $sim(d_i, d_j) = \cos(x_i, y_i)$. The value of the computation must be situated between [-1, 1] and value (almost) 1 represents that the two are (almost) identical.

One of the most important aspect needed to be consider is the granularity of the documents. The external documentation is generally composed of paragraphs, sections, or chapters. These are the common alternatives in demonstrating the definition of an external document in particular cases. The organization of the source code varies from one programming language to another. The easiest and also simplest way to choose the documents granularity is applying the file decomposition. Certainly, this will not suffice for all software engineering tasks. So, choosing classes, functions, or interfaces as source code documents is often time more enticing. As one of the targets for the framework is to be as flexible as possible, so applying a full parser for each possible language is absolutely impractical. Here I have applied a simple and open source java parser named “JavaParser” that can be employed to slice the Java source code into documents of different granularity levels (like- functions, methods, interfaces, and classes).

4.1.3 Recovering Traceability Links

Finally, the similarity between each pair of documents from M X D are measured. The user needs to a value of ε as threshold for the similarity measure, which distinguishes the valid links lied

between the corresponding documents from others. Particularly, among all the pairs from M X D, only those will be recovered which have a calculated similarity value larger than ε. As stated, a conventional, and effective, heuristic is ε = 0.7. This measure associated with a 45° angle between the corresponding vectors. This value of threshold has achieved better results in terms of accuracy [19] when measuring the similarity between entities of source code. The bigger the threshold chosen the closer the weights on the terms occurring in the documents are. I need to consider one of the most important things here is that the documents from the M or external document are practically different either in structure or in nature than the ones from D or source code. So, a smaller valued threshold definitely results in a higher similarity. The contention for the best threshold value for this type of corpus (combination of source code and external documentation) is still a matter of research and further analysis is needed to resolve it.

Here I have used two evaluation matrices or measures to figure out the effectiveness of the application result. Again, these two measures are very familiar for determining the quality of the result of an experiment with IR methods. These are: recall and precision. Broadly, for a provided document (d_i), similarity measure and the defined threshold will be applied to recover a number of documents (N_i), pivoted on the LSI subspace that seems similar to d_i. C_i which is less than or equal N_i are really similar to d_i among these N_i documents. Assume that, R_i which is greater than or equal C_i documents are indeed similar to d_i. I can specify the recall and precision for d_i using these notions as follows:

$$\text{Recall} = \frac{C_i}{R_i} = \frac{\#\,(correct \cap recovered)}{\#\,correct} \text{ x } 100\%$$

$$\text{Precision} = \frac{C_i}{N_i} = \frac{\#\,(correct \cap recovered)}{\#\,recovered} \text{ x } 100\%$$

Both of the matrices must have values that is in the range of [0, 1]. If recall = 100%, that means all the correct links are recovered, while there could be some incorrect ones among all the retrieved links. If the precision = 100%, that means all the retrieved links are correct, while there could be some other correct ones that were not retrieved or considered in the search space.

Considering all of these measures, selecting a larger threshold for the link identification process will consequently result in larger percentage of precision, meanwhile decreasing the value of threshold will result in larger percentage of the recall. Usually the relation between precision and recall is asymmetric or non-proportional.

Applying this threshold based measure, with a pretty high threshold value will achieve a very high precision. Determining the correct threshold is literally tricky. The user has is only the heuristic of past experience. As stated earlier, threshold value of 0.7 provided good results in earlier experiments of text based analysis or information retrieval process. The same valued threshold may or may not be best fitted for different software engineering tasks. Finding a more generalized heuristic for choosing the appropriate threshold is a controversy under analysis and future research will be conducted on it.

As, I am going to experiment two student projects which are not precisely and/or effectively documented. So, I have considered a very low value [0.05 - 0.10] of threshold as standard.

4.2 Technical View

There are two groovy files named- Corpus.groovy and LSIAlgorithm.groovy under the lsi package that is used to implement the acquired concept in the previous section.

4.2.1 Program Structure

Here, I describe the core classes along with their methods. The core classes are:

i. **Corpus.groovy:**

It contains methods and data structures to prepare and fetch term frequency (TF*IDF) values in the form of a double dimension array to construct a term by document matrix. The constructor takes an attributes which indicates the path to the file containing stop list words that need to be precluded from the original document. There are two additional methods named- parsePDFFile and parseJavaFile that takes the path to the PDF file and the java files directory respectively. Then they parse the whole PDF file and java files in order to construct a combined corpus.

ii. **LSIAlgorithm.groovy:**

This class consists of methods to compute the similarity measure between each pair of documents (Each document of the source code to all the documents of the manual or SRS) in the corpus applying Latent Semantic Indexing. The constructor takes five attributes (path to PDF file, path to zip file, path to stop list file, dimension of LSI subspace and the similarity threshold value).

4.2.2 Corpus Creation

Here I need to create a combined corpus of source code and external documentation. I have to parse the PDF file using open source parser named “Apache PDF Box”. At first, all the documents are identified by parsing the table of contents where each mentioned section is treated as an individual document. Next, I need to parse the text under each document and remove special characters and words in stop word list before computing the TF*IDF value of each term in order to create a reliable corpus.

Then, I need to traverse the zip file in order to parse all the Java files inside it using an open source parser named “JavaParser”. I need to write two classes named MethodVisitor.groovy and ClassVisitor.groovy under jvp package in view of parsing the methods and classes respectively. At

first, I remove all the special characters from the source code. Then, I extract the identifiers name along with methods and parameters. After that, I split the camel case words using the method named splitCamelCaseWord. Finally, I calculate the TF*IDF value for each refined terms in the source code document in order to populate the already constructed corpus. Here, I treat each Java file as a document.

4.2.3 Solution Explanation

The constructor for LSI takes five arguments: path to PDF file, path to zip file, path to stop list file, dimension of LSI subspace and the similarity threshold value. When this constructor is called, it automatically creates an object of Corpus class with the values it gets. Next, the term-by-document matrix is formed invoking the createTermByDocumentMatrix method. This method fetches TF*IDF values in the form of a double dimension array where rows denote terms and columns denote documents and forms termByDocumentMatrix of the Matrix class defined in the open source JAMA (JAva MAtrix) package developed by NIST. The application performs singular value decomposition by calling the performSingularValueDecomposition method. Here, a new object (called singularValueDecomposition) of the class SingularValueDecomposition class defined in the open source JAMA (JAva MAtrix) package developed by NIST is created by passing the already formed term-by-document matrix. This class contains methods to retrieve the left singular matrix (by using the getU() method), right singular matrix (by using the getV() method) and singular value matrix (by using the getS() method).

Once I have acquired these values, the system checks if the value entered by the user denoting the number of singular values or dimension of LSI subspace. If it is not 0, then the system generates the reduced versions of the left singular matrix, right singular matrix and singular value matrix by invoking the prepareMatrixForLSI method. If the value is 0 or it does not meet the validity condition, then it is overlooked and no reduction of dimension is done.

Finally, the system is now ready to perform similarity checks between the various documents in the corpus. Next, it performs similarity checking for each source code document matrix with all the matrix of external documentation by invoking findSimilarities method which utilises getIndividualDocumentMatrix method to get each document in the LSI supspace. It also records the similarity results between two corresponding vectors in an object of TreeMap class named similarityResultOfEachDocument.

Although, Similarity is actually calculated or measured in the measureSimilarity method which applies calculateVectorModulus method for core similarity computation. It computes and returns similarity (cosine value) between the external document vector and source code document vector by using the relationship:

$$\cos(\mathbf{x}, \mathbf{y}) = \sqrt{\frac{\mathbf{d}'\mathbf{e}}{|\mathbf{d}|_2 * |\mathbf{e}|_2}}$$

Here d' designates the transpose of the source code matrix, e designates the external document matrix in the concept space. $|d|_2$ and $|e|_2$ designates the length of this two matrix respectively.

The getMaxSimilarities method find out only those relations which are absolutely larger than the mentioned threshold value among all. The calculateSimilarityInPercentage method provides the percentage of the recovered traceability links with the value greater than the threshold in terms of all the retrieved links.

The Matrix class in JAMA package provides pre-built methods to perform matrix multiplication (using times (Matrix) method) and for finding inverse (using inverse () method) and transpose (using transpose () method).

5 Experiments and Results

In this section, the main goal is to evaluate the performance of LSI in this type of software engineering task. As stated earlier, I experimented this application on two student projects named- Braille to Text Translator and BD Travelers which are not so precisely and/or effectively documented. Therefore, I have considered a very low value [0.05 - 0.10] of threshold as standard.

5.1 Experiment and Results for Braille to Text Translator

The system which is used for result analysis and evaluation is "Braille to Text Translator" which is developed and engineered by Minhas Kamal on Java platform. The software, Bengali Braille to Text Translator takes in scanned image of Bengali Braille writing. Then, it applies pattern recognition and translate it to text. I used the SRS as external documentation and the whole project source code directory as the input of this application.

Table 5.1.1 contains the size of the source code and SRS of the software under investigation. It also contains the dimensionality used for the LSI subspace and the determined vocabulary.

Table 5.1.1: Elements of the Braille to Text Translator source code, SRS and LSI settings.

Element name	Number of elements
Source code files	64
SRS sections	50
Total documents	114
Vocabulary	1530
LSI dimension used	200 - 400

I decided to use the entire SRS and the whole source code directory to assure the production of an affluent semantic space and vocabulary.

Although I understood that using the whole source code files might rotten the results in lowering the precision, I decided to go with this approach since it imitates the situation when the system to be evaluated is completely new to the engineers, developers and they do not actually know what is the core component of the system.

Moreover, as the size of source code files is much smaller than the size of SRS documents, I chose to identify the links from the source code to the SRS, rather than doing it vice versa. I assumed that the user can read the SRS with ease, and the source code is the undiscovered factor. A general query would be to trace which parts of the SRS are illustrated by a given source code file. Obviously, in a bottom-up type of evaluation a user might ask which source code file describes a given section of the SRS. It can be established with one user query. Recovering all the links, I trust that starting from the smaller size set produces better similarity results.

I manually discovered 63 correct links which will be used to compute the recall and precision of LSI.

I assumed that a threshold value around 0.15 will produce higher precision and lower recall. Table 5.1.2 depicts the results that is obtained through the process of recovering the traceability links between SRS and source code for Braille to Text Translator.

The first column (Cosine threshold) represents the threshold value, second column (Correct links retrieved) represents the number of correct links recovered, third column (Incorrect links retrieved) represents the number of incorrect links recovered, fourth column (Total links retrieved) represents the total number of recovered links (correct + incorrect) and the last three columns represents the Precision, Recall and Similarity result for each threshold.

Table 5.1.2: Recovered links, recall, precision, similarity result using different threshold for the Braille to Text Translator.

Cosine threshold	Correct links retrieved	Incorrect links retrieved	Total links recovered	Precision	Recall	Similarity result
0.05	56	145	201	27.86 %	88.89 %	87 %
0.07	43	38	81	53.09 %	68.25 %	62 %
0.10	15	2	17	88.24 %	23.81 %	20 %

Figure 5.1.1 (in the next page) depicts the whole scenario of the calculated precisions and recalls. Here, I applied 0.10 as initial similarity threshold. Although this threshold provided a very good precision value (88.24%), the recall (23.81%) was rather worst. So, I decided to relax the selection criteria by decreasing the value of threshold (0.07). As anticipated, the recall (68.25%) improved, but the precision (53.09%) crumbled. Again, I decreased the threshold (0.05) value. As a result, the value of the recall (88.89%) enhanced. But the precision (27.86%) became worst.

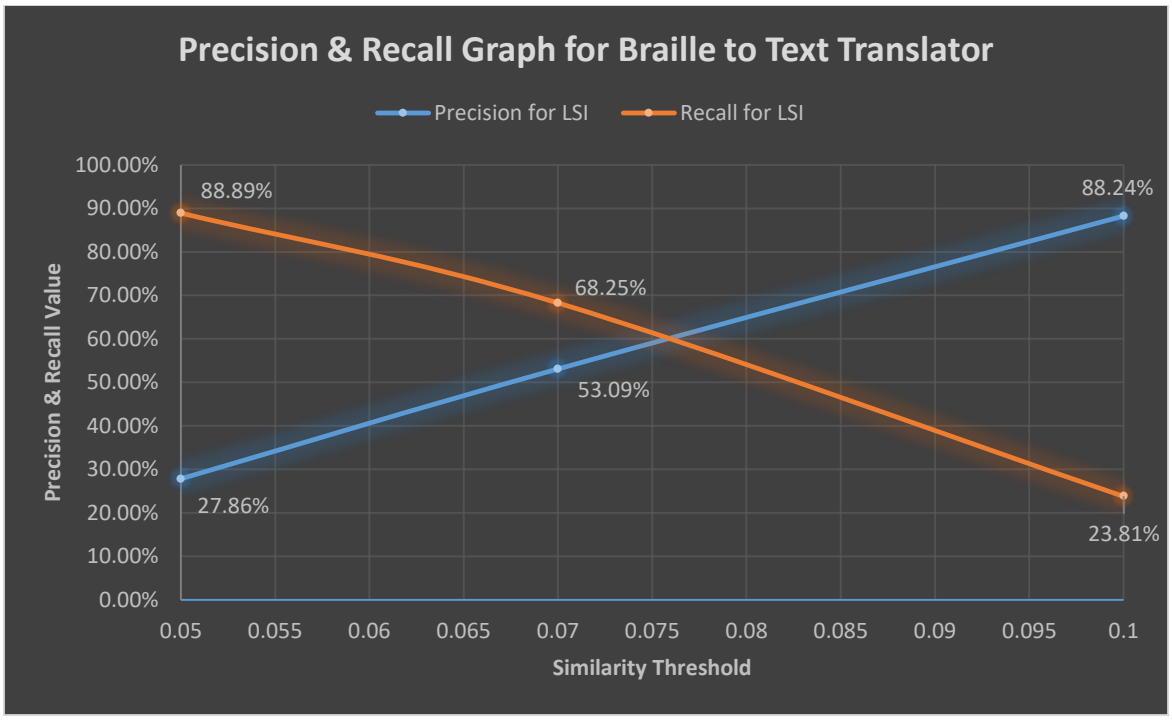

Figure 5.2.1: Precision and recall graph for Braille to Text Translator.

Here, my goal was to analyze the correlation between the increase and decrease of the two measures and select the best value of threshold that maximizes both of these measures. In this case, the most appropriate threshold value is 0.076.

In addition, applying LSI will definitely recover the pairs of similar documents where the similarity hints that the source code file includes the methods described in the documentation. Literally, this is a valid similarity measurement technique and one can claim that, in fact the external documentations page also relates to those source code files.

5.2 Experiment and Results for BD Travelers

Here, I have used a project named “BD Travelers” developed and engineered by Sharafat Ahmed Sabir on android platform using Java as programming language for result analysis and evaluation purpose. It provides information to the users about their desired places, help them to make plan for tours and also will keep track of expenditures they made. I used the SRS as external documentation and the whole project source code directory as the input of this application.

I decided to use the entire SRS and the whole source code directory to assure the production of an affluent semantic space and vocabulary.

Table 5.2.1 contains the size of the source code and SRS of the software under investigation. It also contains the dimensionality used for the LSI subspace and the determined vocabulary.

Table 5.2.1: Elements of the BD Travelers source code, SRS and LSI settings.

Element name	Number of elements
Source code files	21
SRS sections	11
Total documents	32
Vocabulary	1190
LSI dimension used	200 - 400

I decided to use the entire SRS and the whole source code directory to assure the production of an affluent semantic space and vocabulary.

Although I understood that using the whole source code files might rotten the results in lowering the precision, I decided to go with this approach since it imitates the situation when the system to be evaluated is completely new to the engineers, developers and they do not actually know what is the core component of the system.

Moreover, as the size of source code files is much smaller than the size of SRS documents, I chose to identify the links from the source code to the SRS, rather than doing it vice versa. I assumed that the user can read the SRS with ease, and the source code is the undiscovered factor. A general query would be to trace which parts of the SRS are illustrated by a given source code file. Obviously, in a bottom-up type of evaluation a user might ask which source code file describes a given section of the SRS. It can be established with one user query. Recovering all the links, I trust that starting from the smaller size set produces better similarity results.

I manually discovered 20 correct links which will be used to compute the recall and precision of LSI.

I assumed that a threshold value around 0.15 will produce higher precision and lower recall. Table 5.2.2 depicts the results that is acquired through the process of recovering the traceability links between SRS and source code for BD Travelers.

The first column (Cosine threshold) represents the threshold value, second column (Correct links retrieved) represents the number of correct links recovered, third column (Incorrect links retrieved) represents the number of incorrect links recovered, fourth column (Total links retrieved) represents the total number of recovered links (correct + incorrect) and the last three columns represents the precision, recall and similarity result for each threshold.

Table 5.2.2: Recovered links, recall, precision, similarity result using different threshold for the BD Travelers.

Cosine threshold	Correct links retrieved	Incorrect links retrieved	Total links recovered	Precision	Recall	Similarity result
0.05	15	20	35	42.86 %	75 %	57 %
0.07	6	5	11	54.55 %	28.57 %	38 %
0.10	1	0	1	100%	4.76%	4%

Figure 5.3.2 depicts the whole scenario of the calculated precisions and recalls. Here, I applied 0.10 as initial similarity threshold. Although this threshold provided a very good precision value (100%), the recall (4.76%) was rather worst. So, I decided to relax the selection criteria by decreasing the value of threshold (0.07). As anticipated, the recall (28.57%) improved, but the precision (54.55%) crumbled. Again, I decreased the threshold (0.05) value. As a result, the value of the recall (75%) enhanced. But the precision (42.86%) became worst. Here, my goal was to analyze the correlation between the increase and decrease of the two measures and select the best value of threshold that maximizes both of these measures. In this case, the most appropriate threshold value is 0.061.

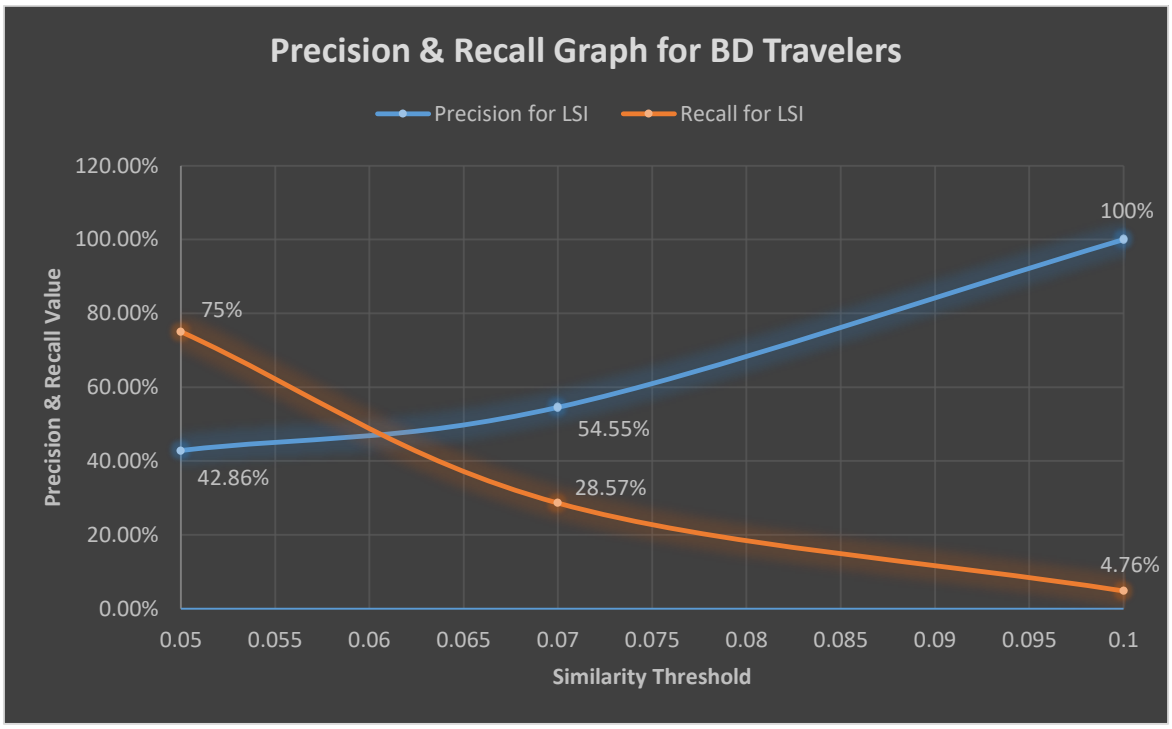

Figure 5.4.1: Precision and recall graph for BD Travelers.

In addition, applying LSI will definitely recover the pairs of similar documents where the similarity hints that the source code file includes the methods described in the documentation. Literally,

this is a valid similarity measurement technique and one can claim that, in fact the external documentations page also relates to those source code files.

6 Conclusion and Future Work

The experimental study presents an approach for recovering the traceability links between documentation and source code by applying an information retrieval technique named- Latent Semantic Indexing (LSI). A set of experiments was performed and the results are presented and analyzed in this report.

The techniques applying LSI performs at least as well methods applying probabilistic and VSM IR methods with the combination of full parsing of the source code and enormous morphological analysis of the documentation.

This approach requires less pre-processing of the source code and documentation and also less computation. Although, I have developed it for only Java source code, but it can be developed for other programming languages too. Therefore, it is language, programming language, and paradigm independent. So it is more flexible and better suited for automation than any other method for retrieving information for text based analysis. These characteristics allow programmers or developers to use the internal documentation in the analysis that allows LSI to generate better and effective results. The BD Travelers project is an instance of this hypothesis. Though it has almost no comments in the source code, LSI does perform at least as well as the other methods.

The results of the experiments are encouraging enough to stipulate further research on it. I will definitely repeat these experiments over and over on similar or other types of software systems (i.e., coded in different programming languages and with different type of documentation). Moreover, I will research more with the view of improving the similarity results by mixing up structural and semantic information extracted from the source code and its corresponding documentation.

Although the results are rather auspicious, I believe the results could be improved further more. The semantic similarity measure defined using LSI (or any other IR method) could be enhanced by applying structural information of the program.

Finally, I am attempting to figure out some good heuristics that the users, developers and the system can apply to select the appropriate threshold value for similarity measures. It is also relied on the maintenance task that is mentioned- the programming language, the quantity and quality of the documentation.

7 References

[1] G. Antoniol, G. Canfora, G. Casazza and A. De Lucia, "Recovering traceability links between code and documentation," *IEEE Transactions on Software Engineering,* vol. 28, no. 10, pp. 970 - 983, October 2002.

[2] A. Marcus and J. I. Maletic, "Recovering Documentation-to-Source-Code Traceability Links using," in *ICSE '03 Proceedings of the 25th International Conference on Software Engineering*, Portland, Oregon, 2003.

[3] N. Anquetil and T. Lethbridge, "Assessing the Relevance of Identifier Names in a Legacy Software System," 1998.

[4] C. Faloutsos and D. W. Oard, "A Survey of Information Retrieval and Filtering Methods," Technical Report CS-TR-3514, 1995.

[5] G. Salton and M. McGill, Introduction to Modern Information Retrival, McGraw-Hill, 1983.

[6] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer and R. Harshman, "Indexing by Latent Semantic Analysis"," American Society for Information Science, 1990.

[7] T. K. Landauer and S. T. Dumais, "A Solution to Plato's Problem: The Latent Semantic Analysis Theory of the Acquisition, Induction, and Representation of Knowledge," 1997.

[8] S. T. Dumais, "Improving the retrieval of information from external sources," vol. 23, no. 2, pp. 229-236, 1991.

[9] T. K. Landauer, P. W. Foltz and D. Laham, "An Introduction to Latent Semantic Analysis," vol. 25, no. 2&3, pp. 259-284, 1998.

[10] B. Fischer, "Specification-Based Browsing of Software Component Libraries," in *ASE*, 1998.

[11] Y. S. Maarek, D. M. Berry and G. E. Kaiser, "An Information Retrieval Approach for Automatically Constructing Software Libraries," *IEEE Transactions on Software Engineering,* vol. 17, no. 8, pp. 800-813, 1991.

[12] L. H. Etzkorn and C. G. Davies, "Automatically Identifying Reusable OO Legacy Code," *IEEE Computer,* vol. 30, no. 10, pp. 66-72, October 1997.

[13] A. Marcus and J. I. Maletic, "Supporting Program Comprehension Using Semantic and Structural Information," in *23rd International Conference on Software Engineering (ICSE 2001)*, Toronto, Ontario, Canada, 2001.

[14] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia and E. Merlo, "Tracing Object-Oriented Code into Functional Requirements," in *8th International Workshop on Program Comprehension (IWPC'00)*, Limerick, Ireland, 2000.

[15] G. Antonio, G. Canfora, G. Casazza and A. De Lucia, "Information Retrieval Models for Recovering Traceability Links between Code and Documentation," in *IEEE International Conference on Software Maintenance (ICSM'00)*, 2000.

[16] X. Wang , G. Lai and C. Liu, "Recovering Relationships between Documentation and Source Code based on the Characteristics of Software Engineering," in *Electronic Notes in Theoretical Computer Science*, Beijing, China, 2009.

[17] N. Ali, . Z. Sharafi, Y.-G. Guéhéneuc and G. Antonio, "An empirical study on the importance of source code entities for requirements traceability," *Empirical Software Engineering,* vol. 20, no. 2, pp. 442-478, 2015.

[18] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto and A. Zaidman, "A Textual-based Technique for Smell Detection," *IEEE xplore,* 2016.

[19] A. Marcus and J. I. Maletic, "Identification of High-Level Concept Clones in Source Code," in *Automated Software Engineering (ASE'01)*, San Diego, CA, 2001.

[20] J. I. Maletic and A. Marcus, "Supporting Program Comprehension Using Semantic and Structural Information," in *23rd International Conference on Software Engineering (ICSE 2001)*, Toronto, Ontario, Canada, 2001.

[21] J. I. Maletic and N. Valluri, "Automatic Software Clustering via Latent Semantic Analysis," in *14th IEEE International Conference on Automated Software Engineering (ASE'99)*, Cocoa Beach Florida, 1999.