Information Retrieval Methods in Concept Location

by

Scott Grant

Queen's University
Kingston, Ontario, Canada
Republished September 2010

Technical Report 2010-575

Copyright © Scott Grant, 2009-2010

Contents

Contents i

List of Figures iii

1 Introduction 1

1.1 Concept Location and Program Comprehension 1

1.2 Terminology 3

1.3 Organization of Paper 4

2 Latent Variable Models 5

2.1 Latent Variables 5

2.2 Modelling Latent Data 6

2.3 Factor Analysis 7

2.4 Latent Class Analysis 8

3 Mathematical Techniques 9

3.1 Introduction 9

3.2 Mathematical Overview 10

3.3 Singular Value Decomposition 12

3.4 Principal Component Analysis 13

3.5 Independent Component Analysis 17

3.6 Non-negative Matrix Factorization 18

4 The Vector Space Model 18

4.1 Definition 18

4.2 Term Weighting 19

4.3 Document Scoring 21

5 Latent Semantic Indexing **22**

5.1 Definition 22

5.2 LSI in Concept Location 24

5.3 Traceability Recovery 29

5.4 Conceptual Coupling and Cohesion 31

5.5 Describing Data 33

6 Probabilistic IR Models **33**

6.1 Probabalistic Latent Semantic Indexing 33

6.2 Latent Dirichlet Allocation 36

7 Conclusion **38**

7.1 Summary 38

7.2 Open Questions 38

Bibliography **43**

List of Figures

1 Latent Variable Models 7
2 Principal Component Visualization 14
3 PCA Transformation 15
4 SVD Transformation 15
5 PCA vs. SVD Transformations 16
6 Using LSI in Concept Location 27
7 Data Representations using LSI 34
8 Comparing Latent Models 39
9 Comparing Concept Location Models 40
10 Overview of Concept Location Model Comparison 41

1 Introduction

1.1 Concept Location and Program Comprehension

> *A person understands a program when able to explain the program, its structure, its behaviour, its effects on its operational context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program. (Biggerstaff, et al. [6])*

Concept and feature location techniques are designed to extract related subsets of program code in order to aid program comprehension. These location techniques seek to identify related blocks of code, whether supervised or not, and aim to ease the difficult process of making sense of large code bases. This can remove a great deal of overhead when trying to understand a set of code, and can even work to prevent related methods from going unnoticed when developing an understanding of unfamiliar source code.

The initial roots of concept and feature location can be traced back to program comprehension theories [10, 11, 62], and these early works attempted to determine how a programmer developed the comprehension necessary to debug, modify, or document code. From these, Biggerstaff identified the *concept assignment problem* [5, 6], and described it as the problem of discovering individual human oriented concepts and assigning them to their implementation oriented counterparts for a given program.

As the idea of a computer program began to shift toward something that required human comprehension and away from simply being input to a machine, researchers began to present theories on how to aid programmers in this process.

In the 1970's, a great deal of research was performed in order to focus on the human-oriented nature of programming. Determining what it meant for a human to *understand* a program, and separating the human-oriented issues from the machine-oriented ones,

can be expected to lead to improved design techniques and language features.

One of the first full cognitive models came from Shneiderman and Mayer [62]. In their research, they developed an information processing model that included a long-term store of semantic and syntactic knowledge, along with a working memory in which problem solutions were constructed. The semantic knowledge stored in the long-term memory as described, can range from low level details to high level concepts.

Ruven Brooks produced an early paper on program comprehension in 1978 designed to demonstrate how he believed a programmer went about developing an understanding about the program they were working with. "Rather than seeing it solely as an object for machine consumption - one which is only compiled or executed - the program is increasingly viewed as an object for programmer consumption as well; programmers read it, understand it, and modify it." [10]

In his paper, Brooks identifies a set of cues for understanding a program. These include internal cues like comments, pretty-printing, structure, and external cues, such as user's manuals, flowcharts, and published algorithm descriptions. He argues that by using this set of information, the programmer is able to aggregate this knowledge to gain an understanding of the program, and "this information is described in terms of the psychological concept of knowledge domains." It was through bridging together the problem domain and the executing program using a succession of knowledge domains that a programmer gained program comprehension.

As these program comprehension theories developed, the specific role of actual conceptual units and their value as descriptive elements became apparent. Rajlich and Wilde explained the role of concepts in program comprehension in 2002 [57], and suggested that instead of focusing only on top-down or bottom-up methodologies, researchers should consider the role of concepts. They note that concepts are fundamental building blocks of human learning, and that the disciplines of program comprehension and human learning share many similarities.

Information retrieval techniques are designed to search through document sets in order to identify the data that is relevant to the searcher. In program comprehension, the approaches range from textual querying to automated clustering, and are often unsupervised. They commonly model source code methods as documents, and form relationships between the methods using a combination of explicit links such as function calls or class sharing and derived latent context.

1.2 Terminology

While the definition of concepts and features have been established for some time, newer terms like concerns are still ambiguous, and often defined on a per-paper basis. In order to clearly define the terms used in this paper, the following core definitions are used.

A **concept** is a human-oriented expression of computational intent [6].

A **feature** is an observable behaviour of the system that can be triggered by the user. A feature is a concept.

Concept location is the act of identifying and isolating concepts in a computer program.

Some related and widely used terms also exist, but some background must be provided in order to define them appropriately.

The phrase **separation of concerns** was introduced by Dijkstra in 1982 [20]. In his words,

> *Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects. We know that a program must be correct and we can study it from that viewpoint only;*

> *we also know that it should be efficient and we can study its efficiency on another day, so to speak. In another mood we may ask ourselves whether, and if so: why the program is desirable. But nothing is gained –on the contrary!– by tackling these various aspects simultaneously. It is what I sometimes have called "the separation of concerns", which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of. This is what I mean by "focussing one's attention upon some aspect": it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously.*

Separation of concerns has come to be associated with concept location. What is important to note is the subtle difference between its original use and the currently assumed definition. Dijkstra referred to the separation of concerns as a way of not only separating concepts in source code (as defined above), but also of qualities of the overall program. Specific concerns referred to in the original essay include the separation of correctness and desirability, implementability, and general acceptance. These qualities are not well described as concerns, as defined in current literature.

A **concern** is a conceptual set of behaviours needed by a computer program. As such, a concern is a concept.

1.3 Organization of Paper

This paper presents an overview of the current state of information retrieval research in concept location, and supports the directions by reflecting on the original goals of the *concept assignment problem*. It will use the terminology provided above, in an attempt to summarize the current research methods related to information retrieval methods as used in concept location, and their relation to each other. It will contrast the approaches

to extracting semantic information from source code, and compare the ways that information retrieval techniques are used to derive this data. Finally, it will attempt to outline the roadmap for future research in these areas.

Section 2 explains the theory behind latent variable models, and how latent data such as conceptual informaton is embedded within the original data. Section 3 describes the mathematical techniques used to convert the original source code into a model, along with ways to reduce large data sets into manageable approximations of the original. Section 4 describes the Vector Space Model, a commonly used algebraic model that represents documents as vectors in a dimensional space that corresponds to the number of tokens spanning the original document set. Section 5 explains Latent Semantic Indexing, the most popular information retrieval technique currently used in concept location, and explains some of the ways it has been used. Section 6 looks at some of the probabilistic techniques that have been developed in order to provide a more rigorous statistical underpinning to the language models. Finally, Section 7 summarizes the research in this area, and provides an overview of the ongoing and future research.

2 Latent Variable Models

2.1 Latent Variables

A latent variable model specifies the distribution of a set of random variables in which some additional variables are assumed to exist and be unobservable. The observable variables are referred to as manifest variables, and have been directly measured in some way. The unobservable variables are called latent variables, and are inferred somehow from the manifest variables. Latent variable models differ from traditional statistical models only in the sense that in addition to the observed data, we assume some hidden substructure to be present [4].

Manifest and latent variables can represent data in two ways. Metrical variables may be formed from values in the set of real numbers, and may be discrete or continuous. Categorical variables assign values from a set of categories, and may or may not offer an intrinsic ordering on the values. The representation used for the manifest and the latent variables helps define the type of model necessary, and should be chosen carefully.

In the social sciences, latent variables are used to represent highly abstract concepts like intelligence, social class, power, and expectations [8]. Economics uses the theory of latent variables aggressively, and considers concepts like quality of life, morale, and happiness as theoretical values that are hidden inside real data. The analysis of source code and program documentation has been looking at latent variables for over a decade, and although there have been promising results, no real specific latent variables have been consistently identified.

2.2 Modelling Latent Data

Latent variable models are often used for two main reasons. First, models derived from a large set of data may be too big to process in any meaningful way. Using a latent variable model to extract latent variables as new components can act as a dimensionality reduction technique, which can transform a large matrix into a smaller close representation of the data. Many of these latent models can even provide a value for the accuracy maintained in the new representation, like a rank-reduced matrix approximation. Second, extracting the latent variables can help to detect structure in the relationships between the manifest variables. Identifying correlations in this way can demonstrate information about the original data that may not have been immediately clear. This data itself may help refine the model to represent the data more appropriately.

The fundamental premise behind a latent variable model is that there is some covariation observed among the manifest variables that can be explained by a mathematical

relationship between them, and that these relationships can be extracted as latent variables.

Figure 1 shows the commonly used techniques when dealing with latent variable models and certain variable types [47]. The two latent class models used most commonly in concept location are factor analysis and latent class analysis. Both of these models assume that the types of the metrical and latent variables used in each model are the same. Factor analysis is based on the assumption that the original metrical manifest variables are actually composed of linear combinations of factors, which are latent metrical variables, plus error terms. On the other hand, latent class analysis deals explicitly with discrete observed variables, and is most commonly used to allocate cases into a discrete latent classification [24].

Variables		
Observed	**Latent**	**Model**
Metrical	*Metrical*	Factor analysis
Metrical	*Categorical*	Latent profile analysis
Categorical	*Metrical*	Latent trait analysis
Categorical	*Categorical*	Latent class analysis

Figure 1: Latent Variable Models

2.3 Factor Analysis

Factor analysis, as used to describe a wide range of related techniques that measure relationships between metrical variables, is the most common form of latent variable model currently used in concept location. It originated in psychology as an attempt to condense a set of academic scores in children down into a smaller set of values that represent general mental ability. Generally, it is assumed that a factor analysis will operate over a correlation matrix in order to identify a number of components that best represent the relationships in the original data. These new factors explain the correlations

among the observed variables, and by using a smaller set of factors, the data set can be described using a smaller number of variables [15].

A number of methods exist to extract factors from a correlation matrix, and all result in a vector of numbers that represents the relevance of each observed variable on that factor. If n variables have been observed in a data set, each factor will consist of n numbers, and each value in the factor will indicate the relevance of the related observed variable for that particular factor. For example, if a factor provides the value 0.9 for a particular observed variable, it indicates that the correlation between the observed variable and the factor has a value of 0.9. After each factor is extracted from the original correlation matrix, it is eliminated from the observations in order to determine whether it is necessary to obtain successive factors.

Factor analytic methods can help to provide a more accurate understanding of the complicated relationships found in large sets of variables, including those data sets which contain errors due to faulty collection or imprecise observations. These methods can also help researchers identify the most important variables in a set, and to provide insight about how further research or refinement should be directed.

Principal Component Analysis, a mathematical technique that assumes metrical types for both manifest and latent variables, is closely related to factor analysis and discussed in Section 3.4. Latent Semantic Indexing, which is discussed in more detail in Section 5, is also related to factor analysis in the assumption that latent structure exists and can be identified and extracted.

2.4 Latent Class Analysis

Latent Class Analysis is a technique for analysing relationships in categorical data [47], and is often used for organizing sets of data in clusters. The main justification for its use is the fact that many variables, both manifest and latent, are simply not continuous. As an example, many metrics are boolean, and a feature is either present or absent. Based

on the collective answers obtained from a set of metrics, an ontology of discrete classes can be defined to explain the results. These methods have been used as a way of using latent classes to empirically validate existing categorizations, and to demonstrate that the assumptions adequately represent the data. It is similar to factor analysis in the sense that each attempts to extract a set of underlying latent information hidden in the data. The primary difference is that the variables must be categorical, necessitating a different mathematical approach for extracting these results.

A necessary parameter for the latent class model is the number of classes to extract, which is analogous to the number of factors to identify. From this, the latent class probabilities can be determined, which give the derived prevalence of classes over the entire data set. Additionally, within each class, the conditional probabilities are calculated, and give the probability of class membership for each member in the original set. For the conditional probabilities, variables are considered to be statistically independent of one another.

Probabilistic latent semantic analysis and latent Dirichlet allocation are two examples of latent class analysis based on a non-negative matrix factorization that have been used in concept location, and are discussed in Section 6. Their development grew out of a desire to apply statistically sound assumptions to standard latent semantic indexing approaches, along with the recognition that term frequencies and probabilities can never take on negative values.

3 Mathematical Techniques

3.1 Introduction

For years, automated techniques have converted document corpora into matrices, which are then processed using an unsupervised mathematical technique to derive knowledge

from data that is often purely syntactic. By removing the requirement for a person to categorize these massive datasets, great savings in time and energy can be made. The mathematical techniques described here were not invented for use in concept location, but have been adapted in order to leverage their innate ability to manipulate data. By converting document sets into matrices or deriving probabilistic models that describe the data, it is possible to interpret the results in a way that demonstrates latent relationships hidden in the original data.

3.2 Mathematical Overview

A brief overview of some basic mathematical background is necessary in order to accurately describe the methods used in this section. Although it is possible to perform techniques like dimensionality reduction without a thorough understanding of the mathematics, an appreciation for the specific results can help clarify exactly what is happening.

A *linear transformation* is a mapping from one vector space to another while maintaining scalar multiplication and addition. Every linear transformation can be represented as a nonsingular matrix, and for a vector $\boldsymbol{x}$ and a matrix A that defines a transformation, we can write $A\boldsymbol{x}$ to say that A acts on $\boldsymbol{x}$ by left multiplication to produce a new vector called $A\boldsymbol{x}$ [30].

If we consider a square matrix A, and a non-zero vector $\boldsymbol{x}$, we can define an *eigenvalue* λ of A as a scalar value that satisfies

$$A\boldsymbol{x} = \lambda\boldsymbol{x} \tag{1}$$

As seen in Equation 1, $\boldsymbol{x}$ does not have its direction changed after having the linear transformation defined in A applied, but only a potential change in its magnitude. If $\boldsymbol{x}$ is a nontrivial, or non-zero solution of $A\boldsymbol{x} = \lambda\boldsymbol{x}$, $\boldsymbol{x}$ is referred to as an *eigenvector* of A, and λ is an eigenvalue of A that corresponds to $\boldsymbol{x}$. By extracting the eigenvectors from

A, it becomes possible to rewrite any vector in the space of A as a linear combination of the eigenvectors.

Given a vector $\boldsymbol{x}$, the *mean* of the elements $\overline{\boldsymbol{x}}$ is the sum of the values in the matrix divided by the number of elements. If $\overline{\boldsymbol{x}}$ is subtracted from each of the elements in the vector, the new vector will have a mean of zero, and is described as being *zero mean*. If the rows or columns of a matrix A are individually considered as vectors, and each has its mean subtracted from each of the elements in that vector, the mean of each row or column will be zero.

The *covariance* of two variables is a measure of how much they change together, and provides a metric about how correlated the two variables are. If the two variables tend to vary with one another in some way, the covariance of the two variables will be non-zero. For two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ of length n, the covariance can be calculated using Equation 2.

$$\operatorname{cov}(\boldsymbol{x},\ \boldsymbol{y}) = \sum_{i=1}^{n} \frac{(\boldsymbol{x}_i - \overline{\boldsymbol{x}})(\boldsymbol{y}_i - \overline{\boldsymbol{y}})}{n} \tag{2}$$

If the covariance is zero, the variables are *uncorrelated* with one another. Given an $m \times n$ input matrix A, the *covariance matrix* is the $n \times n$ square matrix of covariances between the rows of A. The diagonal elements of the covariance matrix correspond to the variance of the associated rows, and off-diagonal elements correspond to the variance between two rows in A.

Matrix decomposition, or matrix factorization, involves expressing a matrix as the product of two or more matrices. This may be done in order to simplify operations on the original matrix, to obtain information about the matrix itself, or as is common in concept location techniques, as a way to approximate the original data as a matrix with smaller rank.

3.3 Singular Value Decomposition

The absolute value of the eigenvalues of a matrix measure the amount that the linear transformation represented by the matrix stretches or shrinks the eigenvectors. Therefore, if the eigenvalue with the greatest magnitude is λ_1, then the corresponding eigenvector $\boldsymbol{x}_1$ indicates the direction in which the stretching effect is greatest [30]. If we assume that $\boldsymbol{x}$ is a unit vector, then $||\boldsymbol{x}|| = 1$, and from Equation 1,

$$||A\boldsymbol{x}|| = |\lambda| \tag{3}$$

From Equation 3, finding the vector $\boldsymbol{x}$ that maximizes $||A\boldsymbol{x}||$ with the constraint that $||\boldsymbol{x}|| = 1$ results in finding the eigenvector aligned along the direction that A stretches most in its transformation. The absolute value λ_1 is the largest singular value of A, and is the length of the vector $A\boldsymbol{x}$. Additionally, the square roots of the eigenvalues corresponding to the eigenvectors of A are called the *singular values* of A.

A Singular Value Decomposition (SVD) is a factorization of a real $m \times n$ matrix A into the product:

$$A = U\Sigma V^T \tag{4}$$

In the decomposition in Equation 4, U is an $m \times n$ matrix with columns $u_{1..n}$ comprised of the left singular vectors, Σ is an $n \times n$ diagonal matrix, and V is an $n \times n$ matrix with rows $v_{1..n}$ comprised of the right singular vectors. The singular vectors form an orthonormal basis, giving $u_i \cdot u_j = 1$ when $i = j$, and $u_i \cdot u_j = 0$ otherwise. With an orthonormal basis, it is possible to rewrite any vector in the space in terms of the singular vectors themselves. The diagonal elements s_k of Σ are the singular values of A, and are constructed to be non-negative and in descending order.

For practical use, the orthonormal columns of U and V can be considered to be transformations on the original vectors in A into and out of a new space, and the diagonal

elements of Σ correspond to the importance of each axis in describing the original data set.

The SVD of a matrix can be used for dimensionality reduction, and is a common approximation used in concept location techniques to avoid processing extremely large matrices. As the diagonal elements of Σ are ordered from the most significant to the least, the strategy for obtaining the best approximation at lower dimensions involves retaining the k highest values of Σ and setting the rest to zero. In this way, the least significant elements of U and V are ignored. Determining the closest approximation of A with rank r can be determined by the following equation.

$$A^r = \sum_{k=1}^{r} u_k s_k v_k^T \tag{5}$$

This rank-reduced decomposition of a matrix A into an $m \times k$ approximation is often written as:

$$A_k = U_k \Sigma_k V_k^T \tag{6}$$

3.4 Principal Component Analysis

Principal Component Analysis (PCA) is a technique related to factor analysis, and linearly transforms an original set of manifest variables into a smaller set of uncorrelated latent variables representing a good approximation of the original. It is often used as a way to reduce dimensionality while maintaining the best set of information about the original data and suppressing the redundant information.

It is fairly common to consider PCA as a method to identify sets of related information in data in order to discover some internal structure. The result of PCA is a linear transformation into a new basis aligned with the eigenvectors of the source matrix. The eigenvalues corresponding to each eigenvalue and each coordinate in the matrix act as a

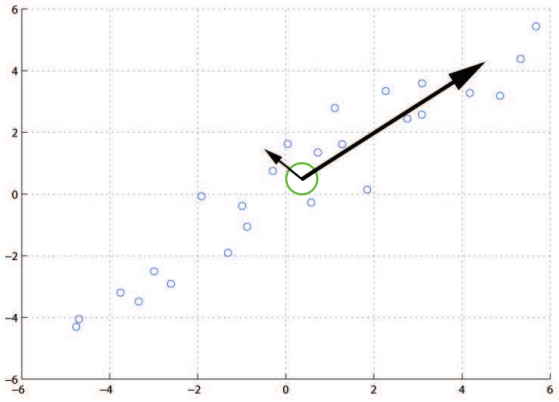

Figure 2: Principal Component Visualization
A visualization of the two principal components constructed from 25 points scattered around the line $x = y$. The first principal component is indicated by the thicker arrow. The average of all 25 points is shown by the largest circle. To demonstrate how the points are transformed into a new basis, consider the transformation from the standard basis into one in which the principal components represent the lines of axis, and the average of the points is the new origin. This transformation is shown in Figure 3.

measure on the overall importance of each axis. The axis with the greatest eigenvalue can be considered to contain the most information about the original data, and correspondingly, the axis with the smallest eigenvalue contains a relatively small amount of information. In this way, PCA can be used for dimensionality reduction, by eliminating the points along each axis that is below a certain relevancy threshold; the goal is to uncover the most meaningful basis to express a set of data. A visualization of the principal components for a small data set can be seen in Figure 2.

For a covariance matrix X with zero mean columns, we are interested in identifying the most relevant basis that corresponds to the eigenvectors of the original data. The unit eigenvectors of the covariance matrix X are called the principal components, and are ranked in order by the magnitude of the eigenvalues corresponding to those eigenvectors.

Figure 3 demonstrates how PCA transforms a set of 25 points scattered around the line $x = y$. The first eigenvector sits almost on top of the scatter line, and the second is

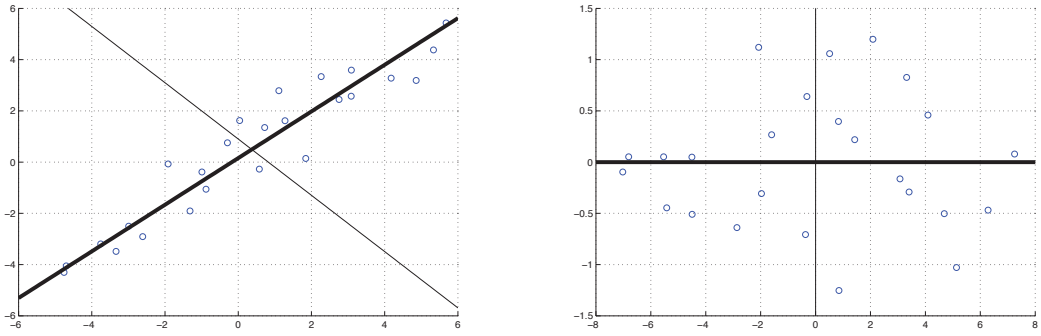

Figure 3: PCA Transformation

In the left graph, 25 points are scattered around the line $x = y$, and the stronger principal component is drawn with a dark solid line. The right graph shows the points after undergoing the transformation to the new space. PCA aligns the matrix around the origin by reducing the mean deviation to zero for all vectors.

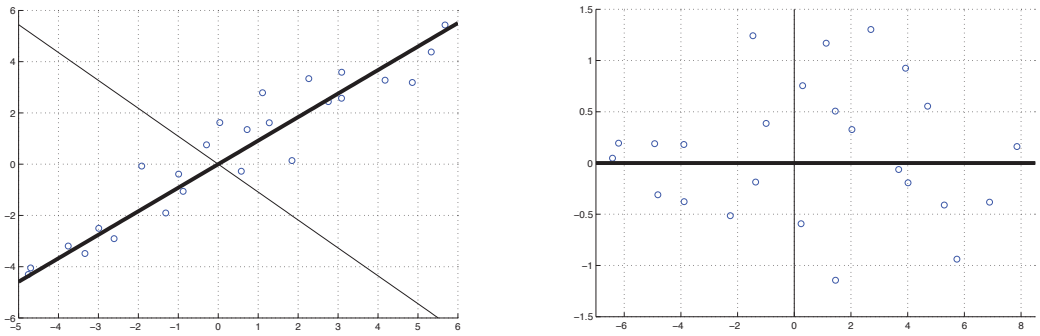

Figure 4: SVD Transformation

The left graph plots the same points used in Figure 3, and the right shows the result of $U * S$ after a singular value decomposition. SVD does not center the points around the origin like PCA, and although the two plots look very similar, SVD linearly transforms the points back to their original space, instead of leaving them at zero mean.

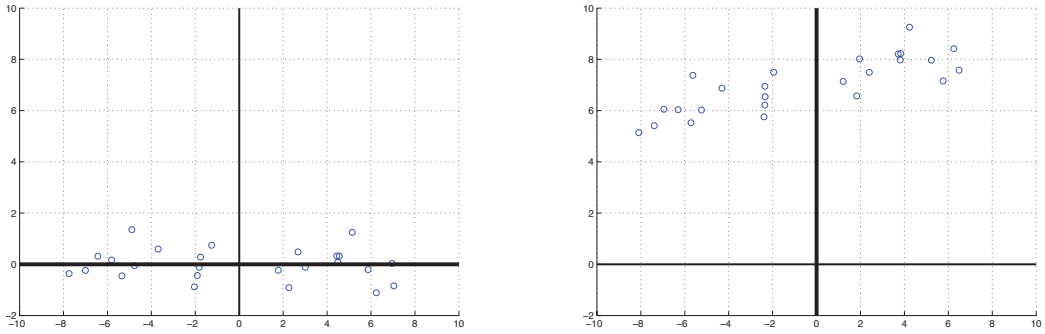

Figure 5: PCA vs. SVD Transformations
25 new points are instead scattered around the line $x = y - 10$. The left graph shows the new orientation after PCA is applied, and the right graph after SVD. Due to the zero mean requirement, PCA centers the points around the origin, and looks much different than the SVD translation, which does not.

almost perpendicular to it. If the points are mapped down to a single axis represented by the first eigenvector, the resulting one-dimensional data will be the best approximation of the original transformed set. In Figure 4, a transformation using SVD on the same set of points is demonstrated. The two appear very similar, but due to the fact that PCA requires a covariance matrix where either the rows or columns have zero mean, a slight change in alignment occurs. This can be seen visually in Figure 5, where the original points have been scattered around the line $x = y - 10$. SVD is a matrix decomposition that has no requirement on the averages of the data. This is an important difference due to the new structure of the data, and affects the choice of comparison functions. When using SVD, it is common to use the cosine distance between two points, as the location along the significant axes is important. For PCA, dealing with a zero mean covariance matrix also means dealing with a different comparison function for measuring the linear dependence between variables.

3.5 Independent Component Analysis

Independent Component Analysis (ICA) [14, 26] is a blind signal separation technique that separates a set of input signals into statistically independent components. It operates in a similar way to Singular Value Decomposition, which is often used in Latent Semantic Indexing and has previously been explored as a way to extract information about concepts from source code [45]. The primary difference is that instead of focusing on signals that are simply decorrelated, ICA extracts signals that are *mutually independent* of one another. This is a stronger bound, and when used in a domain like program comprehension, can ensure a stronger difference between the extracted signals, and a correspondingly stronger similarity between fragments with similar signal profiles. ICA involves the factorization of a source matrix comprised of a set of mixed data signals into two new matrices. One of the matrices describes a number of independent components, and the other is a mixing matrix that holds information about how the independent components themselves were combined to produce the original set of mixed signals.

The original example of ICA as a technique is the idea of a set of microphones hung over a crowded room, wherein a number of people are engaged in conversations. If we examine the set of data obtained from the microphones, the focus on statistical independence as a bound instead of decorrelation allows ICA to isolate the original source signals, and individual voice data for each of the attendees can be recovered.

ICA is a blind source separation method designed to extract the statistically independent components of a non-Gaussian source signal. It is described by the equation $x = As$, and factors an original data matrix x into a transformation, or mixing matrix, referred to as A, and a source signal matrix s, where the extracted independent signals are stored. If x is an $m \times n$ matrix, and we are interested in k independent signals, A will be an $m \times k$ matrix, and s will be $k \times n$. One of the matrices describes a number of independent components, representing the individual extracted voices from the party described above. The other matrix is a mixing matrix, and holds information about how

the independent components themselves were combined to produce the original set of mixed signals [63].

3.6 Non-negative Matrix Factorization

Non-negative Matrix Factorization (NMF) involves factoring a non-negative matrix A into two matrices W and H, which are also non-negative themselves. If A is an $m \times n$ matrix, W is approximately factored into an $m \times k$ matrix, and H into a $k \times n$ matrix. Often, the value of k is chosen to be smaller than m or n, which leads to the fact that A is often approximated by a NMF. Given this definition, we can write the approximation as Equation 7.

$$A \approx WH \tag{7}$$

To avoid the potential of one matrix growing too large and causing the other to decay in size, it is common to normalize the columns of W and/or the rows of H. These vector sets each form a new basis of strictly additive non-negative components, in contrast to the components of other factorings which may contain negative elements. This is seen as a benefit of using NMF in areas like data mining, where term frequencies in the input matrix can never take on negative values.

4 The Vector Space Model

4.1 Definition

The Vector Space Model (VSM) is often attributed to Salton's 1975 paper entitled "A Vector Space Model for Automatic Indexing" [61]. It is a statistical model where documents are conceptually represented by a vector of keywords extracted from the document, with associated weights representing the importance of the keywords in the document

[31]. Queries can be submitted to the model by representing the request as a vector in the term space, and determining which of the document vectors are most similar.

Assume that we have a set of n documents that we would like to represent using the vector space model, and that the document set is described by a total of k unique terms. For each document in the set, a vector v with rank k can be used to describe the document. The elements of v correspond to the set of unique terms, and v_i is non-zero if the ith term occurs in the document. The aggregation of all vectors generated from the documents in the set into a matrix can be considered the vector space representation of the original document set. This matrix will almost certainly be quite sparse, and will be $n \times k$ in size. Natural language text is usually stemmed, reducing words to their base form in order to simplify queries and relate similar documents.

The VSM offers an interesting abstraction of the original data set, but it does suffer from some inherent limitations. Due to the fact that the vector representation of each document considers only the presence or absence of a term, the ordering of terms is lost. The problems of polysemy, the use of many different words to represent the same meaning, and synonymy, when a single word represents many different concepts, can be especially problematic. In particular, polysemy can frequently result in false negatives, when documents are required to contain any relevant keyword in order to have a non-zero value for a term being queried. Long documents have small dot products, and therefore make it difficult to compare similarity since specific matching terms that are being queried end up dwarfed under the large scope of terms found in the document.

4.2 Term Weighting

When a term is present in a document, its position in the weight matrix is non-zero. Many options are available for the value used to represent the statistical importance of a word to a document. This value is referred to as the term *weight*, and we refer to each term in the weight matrix as w_{td}, for the dth document and the tth term. Basic values

that can be used as term values include a boolean choice, where $w_{td} = 1$ when the term is present, and $w_{td} = 0$ when the term is absent.

The next logical step is to assume that documents that mention a term more often has more to do with that term, and should receive a higher score then other documents that do not make frequent references. One way of achieving this is the Term Count Model, in which $w_{td} = tf_{td}$, and tf_{td} is defined as the number of times a given term occurs in the dth document. This particular method suffers from abuse due to term repetition, and in the case of longer documents, are bound to have a greater score for more queries simply because they are longer, not more relevant.

One of the most popular term weighting schemes attempts to alleviate the term repetition problem, and is called *term frequency-inverse document frequency*, or tf-idf [59, 60]. The tf-idf weighting combines term frequency, or the number of time a term occurs in a document, with the inverse document frequency, a value that scales based on the number of documents in the set that mention the term. The document frequency for a term t (called df_t) is defined as the number of documents in a set of n documents that reference t. This leads to the definition of the inverse document frequency of a term t (called idf_t) as:

$$idf_t = log \frac{n}{df_t} \tag{8}$$

The logarithm is applied in an attempt to help alleviate the problem of term repetition. From Equation 8, tf-idf can be defined as:

$$tf\text{-}idf_{td} = tf_{td} \times idf_t \tag{9}$$

By looking at the term count along with the inverse document frequency, terms that appear in too many documents receive lower weights, and are less important to the overall score. Uncommon terms that appear in a smaller number of documents receive a higher

weight.

4.3 Document Scoring

In the VSM, each document is treated as a vector in some high dimensional space, with each axis of the dimensional space representing some aspect of the frequency of a term. Some information about the original set is lost, including the ordering of the documents, and information about the ordering of terms within each document. However, this mapping from documents to vectors allows for a new way to quantify the similarity between each other.

The most common way of determining how related two documents are when using the vector space model is by taking the cosine similarity between the vectors that represent the documents. Cosine similarity takes the cosine of the angle between the vectors, and results in a value between -1 and 1. If we are interested in finding the distance between two documents d_1 and d_2, where the angle between them is represented by θ, we can use the following equation:

$$\cos\theta = \frac{d_1 \dot{d}_2}{|d_1||d_2|} \tag{10}$$

When the VSM is used in concept location, it is common to generate a matrix in place of the original document set with rows corresponding to blocks of source code, and columns corresponding to some feature about the global set of code blocks. Comments may or may not be included, and although language specific operators are often omitted, some structural information or metrics may be maintained.

It is also possible to perform "queries" against the VSM in order to determine the documents that best match a particular case. The initial method of converting documents into vectors and combining them into a document-term matrix can be used to convert a query string into a vector itself. By parsing the original query string and converting the

tokens into their representation under the model, where elements in the vector correspond to the presence or absence of terms, a new vector in the space is defined. The cosine distance between this vector and the existing document vectors can be used to determine which documents are most similar.

5 Latent Semantic Indexing

5.1 Definition

Latent semantic indexing (or latent semantic analysis, as it is also commonly referred to) was introduced in 1988 [21], and was described as a way to take advantage of higher-order structure in the association of terms with documents in order to improve the detection of relevant documents on the basis of terms found in queries [19]. These term-associations can be interpreted as the latent semantic structure of the document set. By assuming that some latent semantic structure exists in a set of documents, the problem of term association can be treated as a statistical problem.

Although latent semantic analysis was originally described as a description for any technique that extracted unobservable information from textual data sets, the latent semantic indexing model that was described in the early papers has become associated with both names. Latent Semantic Indexing (LSI) applies a singular value decomposition on the vector space representation of the input documents, where rows correspond to documents, and columns correspond to some weighted term value. It offers an improvement over the raw VSM model by performing the extra step of extracting "latent" relationships by smoothing out the matrix through a dimensionality reduction that identified statistical correlations in the observed variables and redefines the larger set in terms of the smaller extracted one.

Equation 6 gives the notation for a rank-reduced approximation of a matrix using

the singular value decomposition. It is commonly assumed that we are considering a term-document matrix A, where columns correspond to documents in some original set, and rows refer to the frequency of the terms used in those documents. LSI reduces the matrix using SVD to k dimensions, generating a new matrix A_k. As discussed in Section 2, we presume some latent structure to exist in the data in the form of hidden correlations between the observable variables. Each of the k dimensions represents one of those latent variables.

Dimensionality reduction also provides approximations for the U and V matrices, with U_k as a matrix of size $m \times k$ and V_k as a matrix of size $n \times k$. The rows of U_k represent the term vectors, and the rows of V_k represent the document vectors. These vectors maintain most of the semantic information about the term and document space, and as linearly independent components, are often considered to be independent concepts that are composed into the inputs. That said, the goal of LSI is not necessarily to accurately describe the concepts that are extracted in a meaningful way, but simply to represent the data in a general way that eliminates as many of the issues of polysemy and synonymy as possible.

The decomposition represents terms and documents as vector sets, and it becomes straightforward to compute the similarity between document-document pairs, term-term pairs, and document-term pairs. Document-term pairs are simply the approximated values in the matrix A_k.

Comparing two terms using LSI can be used to demonstrate the likelihood that the terms are related across the document set. High values for the term-term score may indicate synonyms, or simply that documents that contain one term are likely to contain the other. Given the approximation matrix A_k, the dot product between two row vectors provides the extent that two terms are related. The matrix $A_k A_k^T$ is the square symmetric matrix that contains the entire set of relations, and from the SVD decomposition, it can be shown that:

$$A_k A_k^T = U_k \Sigma_k^2 U_k^T \tag{11}$$

Comparing two documents uses a similar approach to the comparison of terms, except that due to the alignment of the input as a term-document matrix, the columns must be compared against one another. Given the approximation matrix A_k, the dot product between two column vectors provides the extent that two documents are related. The matrix $A_k^T A_k$ is the square symmetric matrix that contains the entire set of relations, and again, it can be shown that:

$$A_k^T A_k = V_k \Sigma_k^2 V_k^T \tag{12}$$

LSI is language independent, and does not require any knowledge of the grammar used in the document, making it a powerful cross-domain tool for extracting latent information from document sets. Querying the LSI model is similar to querying the VSM model, and is covered in Section 4.3.

5.2 LSI in Concept Location

The original paper describing the use of LSI in program comprehension was tremendously influential [40], and led to a great deal of further research in the area. In software, the source code is preprocessed and organized into a corpus of documents, which can be represented by vectors, and can be compared using a similarity measure corresponding to the difference between the vector representations. Concept location is done by formulating a query that represents the desired concept, converting that query into a vector in the new document space, and determining the nearest neighbours from the source code documents. It is presumed that all documents within a given empirically determined distance from the query vector are representative of the concept.

Starting in 1999, Maletic and Valluri [40] and Maletic and Marcus [38] began exploring LSI's potential in software by performing a handful of clustering and classification experiments against source code and documentation. The initial tests sought to determine LSI's ability to cluster groups of related code together by extracting subsets of code into documents, performing a decomposition of the resulting vector space matrix, and counting the number of clusters that represent documents within a certain cosine distance from each other. The early tests were promising, and suggested that even without a grammar or solutions to the problems of polysemy and synonymy, LSI could be used to support some aspects of the program understanding process. The results from Marcus' original LSI study suggest that as a concept location technique, it is "almost as easy and flexible to use as grep" and "provides better results". LSI is also language independent, and as they state, "source code preprocessing is simpler than building a dependence graph."

Maletic and Marcus continued their work [39], and began to define a number of metrics for comprehension, and to assess the semantic cohesion of the documents with respect to each other. These metrics use the profile generated by the application of LSI to the source matrix, and are easy to compute, albeit somewhat less accurate than other methods of extracting semantic information. Clustering the code fragments and generating a graph with edges that represent relationships like semantic similarity and structural relations provides the data structure used in aiding program comprehension. The set of simple metrics, including derivable data like the semantic cohesion of a cluster with respect to files, give information on how the clusters conceptually fit the structure of the actual source data. An important note is that the authors conclude that the clusters produced represent an abstraction of the source code based on a semantic similarity, which should relate to higher-level concepts [39]. They used the data to demonstrate that in several large systems, it can be shown that concepts from the problem domain are often spread over multiple files, and that files contain multiple concepts.

The earliest analysis of LSI's relevance to source code tended to focus on determining whether or not it was effective for classifying related sets of documents or queries together correctly. Once promising results were being identified, research began to focus on what it meant to define clusters as conceptual groups. Marcus et al. directly related LSI to concept location in 2004, and used LSI to map concepts expressed in natural language to the relevant parts of the source code [45]. They noted that a common activity undergone by software engineers when tracking down a particular piece of relevant code was using the *grep* utility, which provides a string matching search technique over the code. From this, they sought to determine if using LSI to issue string queries against the source code was an effective technique compared to other methods like grep and program dependency graphs. Although the ultimate conclusion was that none of the techniques was perfect, some interesting results were obtained from the case study. The use of *grep* is a common and effective technique for basic search, but it lacks ranking of results, and often returns either far too many results, or far too few if synonyms are considered. The accuracy of a program dependency graph appears to be better than LSI, but is more complicated and time consuming to construct. A visualization of the concept location process using LSI was provided, and has been reproduced in Figure 6.

Several related approaches began to appear, leveraging the ability of LSI to provide a straightforward, language-independent way to identify relationships between documents. SNIAFL [69], a Static Non-Interactive Approach to Feature Location, attempted to reveal the connections between features and functions using LSI, and to then use the results to generate a Branch-Reserving Call Graph to recover relationships between the retrieved functions. IRiSS [54], Information Retrieval based Software Search, is a Visual Studio plugin based on the existing "find" feature that uses LSI to search projects using natural language queries. From this, the authors also developed JIRiSS [53], an Eclipse plugin that searches Java source code for the implementation of concepts. These tools feature class or method granularity, and return confidence metrics about how close they believe

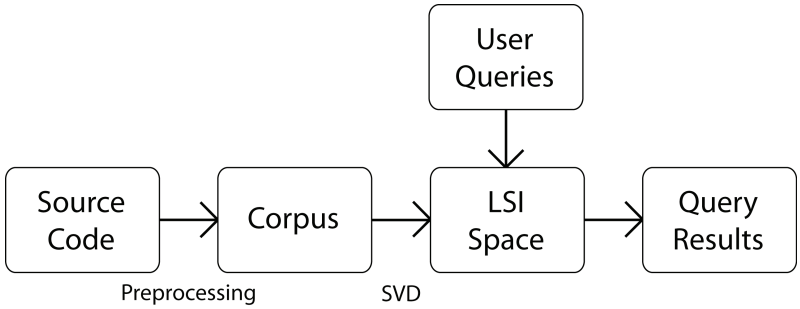

Figure 6: Using LSI in Concept Location

An overview of the process for using LSI in concept location. The original source code or document set is preprocessed to remove unnecessary tokens and characters, and often to perform some kind of stemming to normalize the identifiers. The documents obtained from preprocessing are referred to as the corpus, and from this, a term-document matrix is generated. The term-document matrix is decomposed using SVD, and the decomposition is used to generate the LSI vector space. User queries are formulated using terms found in the corpus to create vector representations, and the nearest corresponding points by cosine similarity in the LSI space are considered to be the query results.

each returned value is to the original query.

The decision about how many dimensions to retain when performing a singular value decomposition has been fairly subjective. Many authors propose somewhere in the range of 200 to 300 dimensions [38, 40], and a recent study demonstrated "islands of stability" around 300 to 500 dimensions for documents sets in the millions, with degrading performance outside of that range [9]. Kuhn et al. suggest using a value of $(m \times n)^{0.2}$, and suggest that a smaller number of dimensions is warranted as the number of documents in their data set is smaller than most natural language corpora [29]. Unfortunately, the authors do not give any comparison or explanation beyond this, and it is difficult to be sure that the choice is sound.

LSI is a static analysis technique that is generally applied to blocks of source code or documentation, and does not take dynamic data into account. In order to access the information obtained from dynamic traces of execution scenarios, Poshyvanyk et al. [56] took two existing techniques they had previously developed, one static and one dynamic,

and combined them in order to identify concepts and features in the source. An existing LSI implementation [44] was combined with a Scenario Based Probabilistic ranking of events [3] that analysed dynamic traces to obtain a list of methods and classes that were likely to be associated with a feature given some scenario. The combination is based on an assumption that each method is considered an expert, and that if it was possible to assume a confidence measure on how well the experts were expected to perform, the sum of the expert scores would provide a valid relevance score. An interesting result of the study was that the accuracy of their retrieval was not greatly affected by weighing one technique's contribution to the relevance over the other, but that combining the two scores resulted in better results than using either technique on its own. Additionally, they looked at varying the number of dimensions when using LSI, and for their large example set with 68,190 documents and 85,439 words, a larger number of dimensions worked better than a smaller number. For their data, they found 1500 dimensions worked better than a typical count like 300.

A recent analysis and comparison of several information retrieval based concept location techniques by Cleary et al. [13] introduced the cognitive assignment technique, which uses information flow and co-occurrence information derived from non-source code artefacts to implement a query-expansion-based concept location technique. What this means is that instead of building a semantic space from the source code and comments, they leverage the external documentation and other non-source code pieces to construct a semantic space from which to derive meaning. They suggest that it may be true that source code is the primary mechanism used by software engineers to express their intent, but situations exist in which those same engineers benefited by using other techniques like comments or bug reports to record concerns not easily expressed in source code. While this approach is not new in itself, this team effectively shifted the focus of the language model away from the source code and into the domain of the non-source code artefacts. An interesting side-effect of their research was the demonstration that their

cognitive assignment technique did not entirely outperform other approaches in all areas, but demonstrated certain sizes of concepts that were discovered with a higher success rate. To them, this seemed to indicate evidence for the conclusion that for different types of concepts, not all concept location techniques are equal, and that several techniques should be used in parallel to find the optimal results.

5.3 Traceability Recovery

Requirements are a specification of what should be implemented. They are descriptions of how the system should behave, or descriptions of a system property or attribute, and they may act as a constraint on the development process of the system [64]. Requirements traceability refers to the process of describing and following the life of a requirement, in both a forwards and backwards direction [23]. While there are many benefits in retaining information about the life of a requirement, the lack of automated techniques for generating traceability links can often make recovery a costly process. Information retrieval techniques like LSI offer an interesting way to bridge the gap using an unsupervised method.

In order to gauge the effectiveness of information retrieval methods, two metrics for measuring recall and precision performance are commonly used. Recall is the ratio of relevant documents retrieved for a given query over the number of relevant documents for that query in the database. Precision is the ratio of the number of relevant documents retrieved over the total number of documents retrieved [22].

$$\text{Precision} = \frac{|\{\text{relevant documents}\} \cap \{\text{ documents retrieved}\}|}{|\{\text{documents retrieved}\}|} \tag{13}$$

$$\text{Recall} = \frac{|\{\text{relevant documents}\} \cap \{\text{ documents retrieved}\}|}{|\{\text{relevant documents}\}|} \tag{14}$$

Antoniol et al. first explored the usefulness of using information retrieval techniques

to analyse project documentation, including specifications, design documents, logs, and other available related text sources, in an attempt to recover traceability links [2]. They compared a probabilistic model and a standard vector space model, performing queries against the two models, and using the resulting vector to perform an indexed search of the documentation. By indexing the documentation against the reduced vocabulary of the documents stored in the model, the authors aimed to provide a semiautomatic method for recovering traceability links. The resuts of their study indicated that these information retrieval techniques afford some ability to recover traceability links in a semiautomatic way. Marcus and Maletic [41] expanded the analysis to include LSI as a querying model, and determined that in the task of recovering links between source code and documentation, LSI performed at least as well as the probabilistic or VSM methods. Lormans and van Deursen also asked the question about whether or not LSI could be used to help reconstruct the traceability requrements [36], and experimented with different link selection strategies and case studies.

The idea of using LSI as a tool to support a software developer in identifying the proper traceability links was examined in a study involving the ADAMS system [16]. The primary difference between ADAMS and the previous work is the use of LSI in an environment designed to specifically manage artefacts, as opposed to the source code of the system. As traceability links are identified manually by the developer, ADAMS is able to suggest similar candidate links whose similarity is greater than or equal to a given threshold. Although some false positives may be displayed, the benefit of reviewing candidates for inclusion would be worthwhile. A case study was performed against a software package under development by students, with a total of 150 artefacts produced. The term-document matrix was constructed using the words in the artefact set, and produced a relatively small matrix. Queries were formed from the identifiers in each code class, and could be used to determine the relevant artefacts by cosine similarity. The results seemed to indicate that in order to achieve a complete recall of the relevant

artefacts, a great deal of precision must be sacrificed, and a large number of false positives must be observed. Further discussion continued later [17, 51], and provided discussion on both the difficulty of choosing a good cutoff value and the inability of techniques like LSI to solve the traceability problem independent of manual quality control by a software developer.

In 2007, ADAMS was used in the first controlled study involving approximately 150 students and 17 software projects, in order to determine the effectiveness of using a tool based on LSI [37]. Each team was asked to evaluate the results of LSI in determining the traceability links discovered in the software by determining whether or not the links were accurate and what the apparent threshold value for relevance appeared to be. The actual traceability matrix was determined by members of the team who were intimately aware of the structure of the project, and therefore had a good understanding of what constituted a correct answer. In their discussion of the results, the researchers determined that the tool worked well, and was certainly faster than a manual trace recovery, but suffered from a problem with the large number of false positives. This is a problem often found with LSI, where large numbers of false positives are mixed in with true positives. In general, the addition of IR techniques gave students a great advantage in recovering the traceability links, but due to the costly nature of discarding false positives, it was found to be prohibitively expensive to recover all of the traceability links. A recent replication of the experiment in [37] was found to confirm the fact that IR-based traceability recovery tools demonstrated their effectiveness in reducing the time spent by engineers manually tracing through the code [18].

5.4 Conceptual Coupling and Cohesion

In object-oriented programming, coupling and cohesion are considered to be good metrics on how well the code has been decomposed into individual modules. Coupling is regarded as the degree to which each class relies on the other models, while cohesion is a measure

of how strongly related the individual parts of a class are to one another. If the goal is to use the coupling and cohesion metrics as an aid to performing concept location, then conceptual coupling is an approach to measure the degree that classes are conceptually related to one another [52], and conceptual cohesion is a measure of the degree that elements of a class belong together [42].

Poshyvanyk and Marcus discussed the use of LSI in order to identify the amount of conceptual coupling occurring in object oriented systems by looking at the information encoded in identifiers and comments. A new set of coupling measures based on the cosine similarity are defined to give metrics between methods and classes in object-oriented systems. In effect, the conceptual similarity between methods is their cosine distance in the vector space created by LSI when positive, and zero otherwise. The conceptual similarity between a method and a class is the sum of similarities between all methods in the class and the target method to compare, divided by the number of methods in the class. The conceptual similarity between two classes is then defined as the average of the similarity measures between all pairs of methods from the two classes. By analysing the results from their study, the researchers were able to discover that a new form of cohesion was being identified, and specifically one that appeared to leverage the latent semantic information contained in the set of identifiers and comments.

Impact analysis involves making estimations on how the modification of one class affects other classes in the code. It often uses information about the coupling and cohesion qualities of the classes, as they provide metrics about how one piece of code is tied to the others. Poshyvanyk et al. performed a study using LSI to determine the conceptual cohesion of the Mozilla code base, and defined equations that give measurements between methods and classes [43, 55], much like in their earlier work from 2006 [52]. They strip comments and structural information like other approaches, convert the code into a term-document matrix, and define the conceptual coupling between methods as the cosine similarity when positive, and zero otherwise. From this comparison between methods,

additional comparisons between methods and classes and pairs of classes are defined.

The importance of tracking metrics through the software development cycle is a well-demonstrated need [12], and monitoring the cohesion and coupling between classes will ideally allow for more disciplined maintenance practices.

5.5 Describing Data

LSI has been applied to a fairly broad range of concept location techniques, and it is interesting to investigate the differences in how the data is represented in each model. In particular, variations occur in the data used in the rows and columns of the term-document matrix used in the singular value decomposition, and occasionally in the vector set in which queries are issued.

Figure 7 offers a glimpse at the evolution of LSI as it relates to program comprehension. It is most common to separate source code at the function level, although some work has investigated using entire classes as documents. The number of dimensions is fairly consistent, although some deviations have occurred. Generally, researchers have chosen to remain around the 300 dimension mark, which is a fairly standard guess in the data mining community that is being revisited due to the increased document set size [9].

6 Probabilistic IR Models

6.1 Probabalistic Latent Semantic Indexing

Probabilistic Latent Semantic Indexing (PLSI) is an approach to automated document indexing that evolved from the desire to apply a sound statistical foundation to typical LSI approaches [25]. The main idea behind LSI is to map documents and terms into a reduced dimensional vector space that reduces noise and amplifies the latent semantic

Research	Input	Columns	Dimensions
Maletic and Valluri 1999 [40]	Source code	Classes / Functions	250
LEDA, C++: ∼2000×144, **MINIX, C**: ∼2000×498			
Marcus and Maletic 2000 [38]	Source code	Source code document	350
Mosaic 2.7, C: 5114×2347			
Marcus and Maletic 2003 [41]	Source code and documentation	Classes / Documents	200/250/300/350/400
LEDA, C++: 3814×803, **Albergate, Java**: 1198×89			
De Lucia 2004 [16]	Software artefacts	Identifiers	?
EasyClinic, C: ?×150			
Marcus 2005 [44]	Source code with comments	Classes / Functions	?
AOI, Java: ?, **Doxygen, C++**: ?			
Lormans 2005 [35]	Software artefacts	Identifiers	20% (∼240)
PACMAN, C: 1200×46			
Kuhn 2006 [29]	Source code with comments	Classes	$(m * n)^{0.2}$ (∼15)
jEdit, Java: 1603×394, **JBoss, Java**: 1379×660			
Poshyvanyk 2006 [56]	Source code	Functions	300/500/750/1500
Mozilla, C/C++: 85436×68190			
Liu 2007 [34]	Source code	Functions	?
jEdit, Java: 7353× ∼5000, **Eclipse, Java**: 56861× ∼89000			

Figure 7: Data Representations using LSI

information contained in the document set. This is achieved through the use of a singular value decomposition in which only a select number of the singular values are maintained, with the rest being set to zero. One of the primary problems with PCA, and by extension SVD, is the requirement that each axis is orthogonal to one another. This optimal reduction of the data works fairly well for describing the original set in terms of a more distinct number of components, but may not actually describe the underlying features that may lie on non-orthogonal axes.

The heart of PLSI lies in latent class analysis, and instead of using a matrix approximation to model term and document relationships, it applies probability theory to make judgements on the likelihood that documents are members of certain classes. This probability theory is applied by performing a non-negative matrix factorization of the original term-document matrix, with the reasoning that probabilities can never take on negative values, and therefore that it does not make sense to allow such values in a solution.

PLSI is based on the maximum likelihood statistical method, and forms a generative statistical model. It assumes the existence of an unobserved latent class variable within each observation of a word in a document. Observations, or the term frequencies for each document, are considered to be independent of one another, which is similar to the bag of words approach taken by other IR techniques like LSI. As a generative model, it attempts to explain the observed pair (d, w) for a document d and a word w by the probability in Equation 15.

$$P(d, w) = P(d) \sum_{z \in \mathcal{Z}} P(w|z)P(z|d) \tag{15}$$

Deriving the probability of observing $P(d, w)$ involves the definition of a set of latent classes, given by $\mathcal{Z}$. Equation 15 also defines the selection of a document d with probability $P(d)$, the generation of a word within a latent topic with probability $P(w|z)$, and the likelihood of observing a latent topic within the document as $P(z|d)$. When solving

a system using PLSI, the input is given as a term-document matrix representing the observations over the document set.

A collection of IR techniques that included PLSI were compared for their ability to perform traceability recovery [1], and found some very unique results. Surprisingly, the researchers found that in their data, PLSI was outperformed by LSI, and even by the VSM, which itself outperformed both other methods. The authors suggest that the poor performance of techniques that perform dimensionality reduction like PLSI suffer with smaller documents, and result in severe overfitting; this claim is not widely acknowledged, and it appears that the number of dimensions chosen may simply not have been sufficient to model the data properly.

6.2 Latent Dirichlet Allocation

Latent Dirichlet Allocation (LDA) is an improvement over the PLSI model that defines a generative prior topic distribution over the documents [7]. As in PLSI, LDA assumes each document can be considered as a mixture of the latent topics. The primary difference is that the set of probabilities of each topic generating a given document is assumed to have a Dirichlet distribution, which does not result in the same overfitting issues found in PLSI. This generative model assumes a Dirichlet distribution on the generation of documents from latent topics, and also on the generation of words within a document from latent topics, so that individual components of a document can be viewed given their topic distribution.

The first application of LDA to source code was in 2007, when Linstead et al. began to use LDA to visualize the emergence of topics over several versions of a project [33, 32]. They looked at seven version of Eclipse, and the entire history of ArgoUML, in order to provide an unsupervised technique to identify feature integration and design refactoring milestones. These topic distributions are plotted, and can show the evolution of a feature's development as a software project matures and grows. The LDA topic

distributions led to the identification of three general patterns. First, the emergence of new functionality, which the authors were able to match up in situations like major version changes that added a new feature. Second, the refactoring of existing functionality, which is a fairly common activity in code maintenance. As new code is being added and removed, this led to topic distribution plots that were often flat. Third, the concerns whose prevalence are related only to the size of the codebase under inspection. As an example, the authors provide code programming practices like string manipulation and logging, which can be expected to grow along with the overall size of the code itself.

Shortly afterward, in 2008, Maskeri et al. demonstrated its application to the extraction of business topics from source code [46]. In a similar approach taken by researchers working with LSI, PLSI, and other IR techniques, they provide a human-assisted method for identifying topics in source code that uses LDA to automatically locate the related subsets of code. The construction of their input matrix involves determining the vocabulary set, which they define as the program elements like identifiers and comments, and the presence of that vocabulary in the source code files. This source code file-word matrix becomes the input, and the topics are considered to be the classes determined by LDA. Preliminary results indicated that some valid clustering was occurring, that topics were being identified, and interestingly that the number of topics for a large scale software system like Linux appeared to be just under 300. However, it can be argued that these initial results, though promising, were not that different from the typical results determined by other IR techniques when applied to concept location.

Although this probabilistic technique is relatively new, it is among the biggest topics researched in the data mining community with respect to document classification and conceptual analysis. A corresponding lack of interest in older techniques like LSI seems to indicate that newer approaches are producing better results, and are garnering more attention due to their successes. The concept location community would certainly benefit from further study of these techniques.

7 Conclusion

7.1 Summary

The ability to use information retrieval techniques in concept location is still an active area of research, and an overview of several specific methods along with their advantages and disadvantages can be seen in Table 8.

Pattern Matching is often exemplified by the *grep* command line tool, frequently used by programmers to find related blocks of code very quickly. It is simple to use, and provides comparatively simple results, with many unrelated lines of code that must be sorted out by the programmer. Despite the inaccuracies of the tool, it remains widely adopted due to its sheer speed and prevalence.

Program Slicing [67] involves the generation of a number of subsets of the source code, obtained by static analysis or dynamic tracing. The collection of large numbers of these slices can be time consuming, but with a proper graph constructed, the act of locating related subsets of code can occur quite rapidly. The recall and precision metrics can also be problematic, but can depend heavily on well-structured code.

Concern Graphs [58] are abstractions of program source code that attempt to store only relevant structure about the concerns found in the original code. It can be generated directly from the source code, or from some intermediate representation. Due to the fact that these graphs are abstractions, they store an imperfect representation, and like the other methods, suffer from some problems with recall and precision.

7.2 Open Questions

The introduction of information retrieval techniques to source code analysis has begun to open some very interesting doors. Unsupervised methods that operate on existing data in order to pull out latent relationships give programmers a new way to visualize

Model		Advantages		Disadvantages
VSM	✓	Algebraic representation of documents	XX	Words with multiple meanings cause false matches
	✓	Implicit cosine similarity measure	XX	Synonyms are not matched
			X	Large documents have sparse vectors
LSI	✓✓	Some reduction of synonymy and polysemy	XX	Computation on extremely large sets can be slow
	✓	Algebraic representation of documents	X	Large documents have sparse vectors
	✓	Implicit cosine similarity measure		
PLSI	✓✓	EM algorithm can be parallelized	XX	Can result in severe overfitting
	✓	Stronger probabilistic model	X	Cannot handle new documents
LDA	✓✓	EM algorithm can be parallelized		
	✓✓	Strong generative probabilistic model		

Figure 8: Comparing Latent Models

✓✓ *Very Good* ✓ *Good* **X** *Fair* **XX** *Poor*

Pattern Matching		
Speed	✓✓	Robust tools like grep that use regular expressions are very fast. (Knuth 1977 [28])
Storage	✓✓	Aside from the original source code, no additional storage requirements exist. (Knuth 1977 [28])
Recall	X	Searches often retrieve more than necessary, with a different recall format. (Marcus 2004 [45])
Precision	XX	Irrelevant matches and false positives commonly contain the same pattern. (Marcus 2004 [45])
Program Slicing		
Speed	✓	Generating slices may take time, but queries against the graph are fast. (Tip 1994 [65])
Storage	X	Larger programs can potentially generate extremely large graphs. (Tip 1994 [65])
Recall	✓	Good recall due to direct transitions in blocks, best with modular structure. (Meyers 2007 [48])
Precision	X	Approaches like generating concept lattices from slices do not scale well. (Tonella 2003 [66])
Concern Graph		
Speed	X	Program structure can be automatically extracted from source, but benefits from manual tagging. (Janzen 2003 [27], Robillard 2002 [58])
Storage	✓	The graphs are compact, simple, and descriptive, albeit imprecise. (Robillard 2002 [58])
Recall	✓	The imprecise mapping results in a few false positives. (Robillard 2002 [58])
Precision	X	It also results in false negatives when unimportant code has been filtered. (Robillard 2002 [58])
Information Retrieval		
Speed	✓	Many IR techniques can be executed in parallel, and good tool support exists. (Nallapati 2007 [49], Newman 2008 [50], Wolfe 2008 [68])
Storage	X	The memory cost of extremely large matrices can be prohibitive. (Deerwester 1990 [19])
Recall	✓	IR techniques tend to retrieve a high number of relevant documents. (Kuhn 2007 [29])
Precision	X	Despite good recall rate, it is rarely clear where the relevance cut-off point is. (Cleary 2009 [13])

Figure 9: Comparing Concept Location Models
✓✓ *Very Good* ✓ *Good* **X** *Fair* **XX** *Poor*

	Pattern Matching	Program Slicing	Concern Graph	Information Retrieval
Speed	✓✓	✓	X	✓
Storage	✓✓	X	✓	X
Recall	X	✓	✓	✓
Precision	XX	X	X	X

Figure 10: Overview of Concept Location Model Comparison
✓✓ *Very Good* ✓ *Good* **X** *Fair* **XX** *Poor*

structure in code that may not have been apparent.

Although some significant progress has been made, there are some clear avenues for improving research in this area.

First and foremost, the actual meaning of a latent variable in the source code domain is not yet understood. If a document-term matrix is derived from a large set of source code, and a latent variable that explains either a significant or an insignificant correlation between the original code, what can be said about the meaning of the latent variable itself? In fields like economics and psychology, latent variables are explained as immeasurable concepts like happiness. If it was possible to define ways of explaining this latent structure in source code, we would gain a valuable insight into the correlations derived from the data.

It may be argued that the size of the input matrices for industry-sized source code packages are simply too large and sparse to gain any valuable insight. If a matrix spans hundreds of thousands of rows and millions of columns, and requires a reduced dimensionality of hundreds or thousands of latent variables in order to have any real statistical significance, it may not be the case that the data is being modelled appropriately. New ways of forming the input matrices for these information retrieval methods could conceiveably model the problem in a more interesting way, and allow for the extraction of fewer latent variables that were better understood.

The shift in the data mining community from techniques like LSI to those like LDA

that have a stronger foundation in statistics seem to indicate the future direction for researchers in concept location.

References

[1] A. Abadi, M. Nisenson, and Y. Simionovici. A traceability technique for specifications. In *The 16th IEEE International Conference on Program Comprehension (ICPC '08)*, pages 103–112, June 2008.

[2] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.

[3] G. Antoniol and Y.-G. Gueheneuc. Feature identification: A novel approach and a case study. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM '05)*, pages 357–366, Washington, DC, USA, 2005. IEEE Computer Society.

[4] D. Bartholomew and M. Knott. *Latent variable models and factor analysis*. Oxford University Press Inc., Arnold, 2nd edition, 1999.

[5] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th International Conference on Software Engineering (ICSE '93)*, pages 482–498, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[6] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–82, 1994.

[7] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *The Journal of Machine Learning Research*, 3:993–1022, 2003.

[8] K. A. Bollen. *Structural Equations with Latent Variables*. Wiley-Interscience, New York, NY, USA, 1989.

[9] R. B. Bradford. An empirical study of required dimensionality for large-scale latent semantic indexing applications. In *Proceeding of the 17th ACM Conference on Information and Knowledge Management (CIKM '08)*, pages 153–162, New York, NY, USA, 2008. ACM.

[10] R. Brooks. Using a behavioral theory of program comprehension in software engineering. In *Proceedings of the 3rd International Conference on Software Engineering (ICSE '78)*, pages 196–201, Piscataway, NJ, USA, 1978. IEEE Press.

[11] R. E. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, 1983.

[12] G. Canfora and M. D. Penta. New frontiers of reverse engineering. In *2007 Future of Software Engineering (FOSE '07)*, pages 326–341, Washington, DC, USA, 2007. IEEE Computer Society.

[13] B. Cleary, C. Exton, J. Buckley, and M. English. An empirical analysis of information retrieval based concept location techniques in software comprehension. *Empirical Software Engineering*, 14(1):93–130, 2009.

[14] P. Comon. Independent component analysis, a new concept? *Signal Processing*, 36(3):287–314, 1994.

[15] A. L. Comrey. *A first course in factor analysis*. Academic Press New York, 1973.

[16] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora. Enhancing an artefact management system with traceability recovery features. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM '04)*, pages 306–315, Washington, DC, USA, 2004. IEEE Computer Society.

[17] A. De Lucia, R. Oliveto, and P. Sgueglia. Incremental approach and user feedbacks: a silver bullet for traceability recovery. In *Proceedings of the 22nd IEEE International*

Conference on Software Maintenance (ICSM '06), pages 299–309, Washington, DC, USA, 2006. IEEE Computer Society.

[18] A. De Lucia, R. Oliveto, and G. Tortora. Assessing IR-based traceability recovery tools through controlled experiments. *Empirical Software Engineering*, 14(1):57–92, 2009.

[19] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[20] E. W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, 1982.

[21] S. T. Dumais, G. W. Furnas, T. K. Landauer, S. Deerwester, and R. Harshman. Using latent semantic analysis to improve access to textual information. In *Proceedings of the ACM CHI 88 Human Factors in Computing Systems Conference*, pages 281–285. ACM Press, 1988.

[22] W. B. Frakes and R. A. Baeza-Yates, editors. *Information Retrieval: Data Structures & Algorithms*. Prentice-Hall, 1992.

[23] O. C. Z. Gotel and A. C. W. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of the 1st International Conference on Requirements Engineering (ICRE '94)*, pages 94–101, 1994.

[24] T. Heinen. *Latent Class and Discrete Latent Trait Models*. SAGE Publications, Thousand Oaks, California, USA, 1996.

[25] T. Hofmann. Probabilistic latent semantic indexing. In *Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '99)*, pages 50–57, New York, NY, USA, 1999. ACM.

[26] A. Hyvarinen, J. Karhunen, and E. Oja. *Independent Component Analysis*. J. Wiley, New York, 2001.

[27] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD '03)*, pages 178–187, New York, NY, USA, 2003. ACM.

[28] D. E. Knuth, J. James H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[29] A. Kuhn, S. Ducasse, and T. Gírba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, 2007.

[30] D. C. Lay. *Linear Algebra and Its Applications*. Addison Wesley, second edition, 1997.

[31] D. Lee, H. Chuang, and K. Seamons. Document ranking and the vector-space model. *IEEE Software*, 14(2):67–75, Mar/Apr 1997.

[32] E. Linstead, C. Lopes, and P. Baldi. An application of latent dirichlet allocation to analyzing software evolution. In *Proceedings of the 2008 7th International Conference on Machine Learning and Applications (ICMLA '08)*, pages 813–818, Washington, DC, USA, 2008. IEEE Computer Society.

[33] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi. Mining concepts from code with probabilistic topic models. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*, pages 461–464, New York, NY, USA, 2007. ACM.

[34] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *Proceedings of*

the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE '07), pages 234–243, New York, NY, USA, 2007. ACM.

[35] M. Lormans and A. van Deursen. Reconstructing requirements coverage views from design and test using traceability recovery via LSI. In *Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE '05)*, pages 37–42, New York, NY, USA, 2005. ACM.

[36] M. Lormans and A. Van Deursen. Can LSI help Reconstructing Requirements Traceability in Design and Test? In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR '06)*, pages 47–56, Washington, DC, USA, 2006. IEEE Computer Society.

[37] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology*, 16(4):13, 2007.

[38] J. I. Maletic and A. Marcus. Using latent semantic analysis to identify similarities in source code to support program understanding. In *Proceedings of the 12th IEEE International Conference on Tools with Artificial Intelligence (ICTAI '00)*, page 46, Washington, DC, USA, 2000. IEEE Computer Society.

[39] J. I. Maletic and A. Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE '01)*, pages 103–112, Washington, DC, USA, 2001. IEEE Computer Society.

[40] J. I. Maletic and N. Valluri. Automatic software clustering via latent semantic analysis. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE '99)*, page 251, Washington, DC, USA, 1999. IEEE Computer Society.

[41] 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)*, pages 125–135, Washington, DC, USA, 2003. IEEE Computer Society.

[42] A. Marcus and D. Poshyvanyk. The conceptual cohesion of classes. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM '05)*, pages 133–142, Washington, DC, USA, 2005. IEEE Computer Society.

[43] A. Marcus, D. Poshyvanyk, and R. Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, 34(2):287–300, 2008.

[44] A. Marcus, V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev. Static techniques for concept location in object-oriented code. In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC '05)*, pages 33–42, Washington, DC, USA, 2005. IEEE Computer Society.

[45] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE '04)*, pages 214–223, Delft, Netherlands, 2004.

[46] G. Maskeri, S. Sarkar, and K. Heafield. Mining business topics in source code using latent dirichlet allocation. In *Proceedings of the 1st Conference on India Software Engineering Conference (ISEC '08)*, pages 113–120, New York, NY, USA, 2008. ACM.

[47] A. L. McCutcheon. *Latent Class Analysis*. SAGE Publications, Newbury Park, California, USA, 1987.

[48] T. M. Meyers and D. Binkley. An empirical study of slice-based cohesion and coupling metrics. *ACM Transactions on Software Engineering Methodologies*, 17(1):1–27, 2007.

[49] R. Nallapati, W. Cohen, and J. Lafferty. Parallelized Variational EM for Latent Dirichlet Allocation: An Experimental Evaluation of Speed and Scalability. In *Proceedings of the 7th IEEE International Conference on Data Mining Workshops (ICDMW '07)*, pages 349–354, Washington, DC, USA, 2007. IEEE Computer Society.

[50] D. Newman, A. Asuncion, P. Smyth, and M. Welling. Distributed inference for latent dirichlet allocation. In *Neural Information Processing Systems (NIPS)*, pages 1081–1088, 2008.

[51] R. Oliveto. Traceability management meets information retrieval methods "strengths and limitations". In *Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering (CSMR '08)*, pages 302–305, Washington, DC, USA, 2008. IEEE Computer Society.

[52] D. Poshyvanyk and A. Marcus. The conceptual coupling metrics for object-oriented systems. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM '06)*, pages 469–478, Washington, DC, USA, 2006. IEEE Computer Society.

[53] D. Poshyvanyk, A. Marcus, and Y. Dong. JIRiSS - an Eclipse plug-in for Source Code Exploration. *14th IEEE International Conference on Program Comprehension, 2006 (ICPC '06)*, pages 252–255, 0-0 2006.

[54] D. Poshyvanyk, A. Marcus, Y. Dong, and A. Sergeyev. IRiSS - A Source Code Exploration Tool. In *Industrial and Tool Proceedings of 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 69–72, 2005.

[55] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering*, 14(1):5–32, 2009.

[56] D. Poshyvanyk, A. Marcus, V. Rajlich, Y.-G. Gueheneuc, and G. Antoniol. Combining probabilistic ranking and latent semantic indexing for feature identification. In *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC '06)*, pages 137–148, Washington, DC, USA, 2006. IEEE Computer Society.

[57] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPC '02)*, page 271, Washington, DC, USA, 2002. IEEE Computer Society.

[58] M. P. Robillard and G. C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*, pages 406–416, New York, NY, USA, 2002. ACM.

[59] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. In *Information Processing and Management*, pages 513–523, 1988.

[60] G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill Book Company, 1984.

[61] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

[62] B. Shneiderman and R. Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Parallel Programming*, 8(3):219–238, June 1979.

[63] D. Skillicorn. *Understanding Complex Datasets: Data Mining with Matrix Decompositions*. CRC Press, 2007.

[64] I. Sommerville and P. Sawyer. *Requirements Engineering: A Good Practice Guide*. John Wiley & Sons, Inc., New York, NY, USA, 1997.

[65] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[66] P. Tonella. Using a concept lattice of decomposition slices for program understanding and impact analysis. *IEEE Transactions on Software Engineering*, 29(6):495–509, 2003.

[67] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE '81)*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[68] J. Wolfe, A. Haghighi, and D. Klein. Fully distributed EM for very large datasets. In *Proceedings of the 25th International Conference on Machine Learning (ICML '08)*, pages 1184–1191, New York, NY, USA, 2008. ACM.

[69] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. SNIAFL: Towards a Static Non-Interactive Approach to Feature Location. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pages 293–303, Washington, DC, USA, 2004. IEEE Computer Society.