USING TOPIC MODELS TO SUPPORT SOFTWARE MAINTENANCE

by

SCOTT GRANT

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen’s University
Kingston, Ontario, Canada
April 2012

Copyright © Scott Grant, 2012
Abstract

Latent topic models are statistical structures in which a “latent topic” describes some relationship between parts of the data. Co-maintenance is defined as an observable property of software systems under source control in which source code fragments are modified together in some time frame. When topic models are applied to software systems, latent topics emerge from code fragments. However, it is not yet known what these latent topics mean.

In this research, we analyse software maintenance history, and show that latent topics often correspond to code fragments that are maintained together. Moreover, we show that latent topic models can identify such co-maintenance relationships even with no supervision. We can use this correlation both to categorize and understand maintenance history, and to predict future co-maintenance in practice. The relationship between co-maintenance and topics is directly analysed within changelists, with respect to both local pairwise code fragment similarity and global system-wide fragment similarity. This analysis is used to evaluate topic models used with a domain-specific programming language for web service similarity detection, and to estimate appropriate topic counts for modelling source code.
Co-Authorship

All papers resulting from this thesis were co-authored with my supervisor Dr. James R. Cordy, and many included my supervisor Dr. David B. Skillicorn. Our work in Chapter 7 was co-authored by Douglas Martin. In all cases I am the primary author.

Part of Chapter 4 was published in the proceedings of the 18th Working Conference on Reverse Engineering (WCRE’11) co-authored with James R. Cordy and David B. Skillicorn [37]. Part of Chapter 5 was published in the proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC’09) co-authored with James R. Cordy [34]. Part of Chapter 7 was published in the proceedings of the 13th IEEE International Symposium on Web Services Evolution (WSE’11) co-authored with Douglas Martin, James R. Cordy, and David B. Skillicorn [39], and received the Best Paper Award. Part of Chapter 8 was published in the proceedings of the 10th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM’10) co-authored with James R. Cordy and David B. Skillicorn [35]. Part of Chapter 9 was published in the proceedings of the 15th Working Conference on Reverse Engineering (WCRE’08) co-authored with James R. Cordy and David B. Skillicorn [36] and in the proceedings of the 2008 Workshop on Link Analysis, Counterterrorism and Security (LACTS ’08) with David B. Skillicorn and James R. Cordy [40].

Chapter 5 is also based on presentations and the ensuing discussion from invited talks at the Consortium for Software Engineering (CSER), specifically “The Value

A summary of Chapters 4, 7, and 8 was presented at the Early Research Achievements satellite track at the 16th European Conference on Software Maintenance and Reengineering (CSMR’12) [38].
Acknowledgments

To my family and friends, who stood by and permitted me to toil around in theory for a few years. Thank you!

Gemma, for the support and patience; it isn’t a stretch to say that you’ve done far more work for this than I have. Thank you, Frau Doktor! Let’s go grab some poutine to celebrate. And to my two beautiful boys Milo and Cole, who always had a smile ready. Daddy will write some more papers for you soon. Care for another Chuck the Dump Truck paper? Or will it be the Ninja Turtles today? Love you.

To my mom, Marlene, who told me from an early age that I’d be a doctor one day. I’m fairly certain you meant a medical doctor, but hopefully this works as well! And to my sister Laura, Nathan, and their beautiful boy Oliver, thank you for all of the support, snacks, and for finding us a wonderful house in the township close to you!

Thanks to the long list of labmates who have suffered through my distractions. Chris, Pooja, Chanchal, Adrian, Doug, Karolina, Eric, and Paul, you’ve specifically endured the most Youtube forwards and dumb jokes. I’ll send you some more shortly. I hope the Prank Wars live on long after I leave the lab! I would also like to thank the members of the School of Computing who have helped me. In particular, thank you to Tad Stach, Irene Lefleche, Debby Robertson, Ben Hall, and anyone who has ever had the misfortune of receiving an absent-minded or outright silly question from me. Thank you for your patience!
To Dr. David B. Skillicorn, who initiated my interest in this area thanks to his graduate course on Data Mining. In order to make me a better writer, you’ve put up with a lot of nonsensical phrases (does this one count?). Thank you for always being honest and fair, and for making me a stronger academic.

And most importantly, it is difficult, if not impossible, to adequately express how influential my advisor Dr. James R. Cordy has been on my industrial and academic careers. Any success I have had is a direct result of his decision to mentor me as an undergraduate. It has been my privilege to learn from him during my time as a graduate student. His guidance in maintaining a balance between work and family allowed me to continue researching and publishing while celebrating the birth of my two sons. You will always be a mentor and a friend. Jim, thank you for making all of this possible. Thank you, thank you.
Statement of Originality

I hereby certify that all of the work described within this thesis is the original work of the author. The research was conducted under the supervision of Dr. James R. Cordy and Dr. David B. Skillicorn. Any published (or unpublished) ideas and/or techniques of others are fully acknowledged in accordance with the standard referencing practices.

Scott Grant
April 2012
# Table of Contents

Abstract i

Co-Authorship ii

Acknowledgments iv

Statement of Originality vi

Table of Contents vii

List of Tables xi

List of Figures xii

## Chapter 1:

- **Introduction** .................................................. 1  
  - 1.1 Goals of the Thesis ........................................ 3  
  - 1.2 Thesis Outline ........................................... 5

## Chapter 2:

- **Background** .................................................. 7  
  - 2.1 Latent Variable Models ................................... 7  
  - 2.2 Vector Space Model ....................................... 12  
  - 2.3 Latent Semantic Indexing ................................. 13
9.5 Summary ......................................................... 125

Chapter 10:

Conclusion ......................................................... 126
10.1 Contributions .................................................. 127
10.2 Threats to Validity ............................................. 129
10.3 Future Work .................................................... 130

Bibliography ......................................................... 132

Appendix A:

Mathematical Techniques ........................................ 147
A.1 Principal Component Analysis ............................... 149
A.2 Singular Value Decomposition ............................... 149
List of Tables

2.1 Summary of Latent Variable Models ........................................... 9
2.2 Hellinger distances between documents ................................. 20

7.1 Similar operations before (upper) and after (lower) contextualizing .. 88
7.2 Relevant WSCell and WSDL operations using LDA ....................... 90

8.1 Source code packages used in this study ................................... 99
8.2 Source code results where a clear proximity score peak emerges ... 102
8.3 Topic proximity scores without document proximity score peaks ... 102
8.4 Latent topic estimates using heuristics and Equation 8.1 ............ 108

9.1 ICA Topic Tokens for cook ...................................................... 113
9.2 Topic 1: Highest Scoring Functions ........................................ 115
9.3 10 topics extracted from httpd using LDA ......................... 120
9.4 10 topics extracted from httpd using ICA ............................ 121
9.5 Topic functions identified from httpd ................................. 122
9.6 ICA Topic 8 functions (negative) from httpd ....................... 125
List of Figures

4.1 A changelist that modifies a function definition ....................... 42
4.2 Visualizing topics over changelists ................................. 43
4.3 Visualization pattern: Feature Development .......................... 45
4.4 Visualization pattern: Commit, Patch ................................. 46
4.5 Visualising co-maintained topics .................................... 48
4.6 The same set of changes from Figure 4.2 with 50 topics instead of 100 49
4.7 Visualization pattern: Aspect Development .......................... 50
4.8 Visualization pattern: Lexical Change ................................. 50
4.9 Visualization using LSI and LDA ..................................... 51
5.1 POCO: Local Similarity ................................................. 58
5.2 Bluevis: Example Diagram .............................................. 60
5.3 Bluevis: Global Pairwise Visualization ............................... 61
6.1 ICA Signals from $A_{flea}$ .............................................. 65
6.2 Similarity scores from $x_{flea}$ ......................................... 66
6.3 First Percentile Nearest Neighbour .................................. 68
6.4 Tenth Percentile Nearest Neighbour .................................. 69
6.5 Comparing Clones and Topics ........................................... 71
7.1 An example WSCell ..................................................... 78
7.2 “Similar” web services that are actually unrelated ................ 79
Chapter 1

Introduction

Making sense of data is hard. In the social sciences, such as psychology or economics, large quantities of observations are collected. One approach to making sense of this wealth of information is the latent factor, an unobserved variable that explains patterns of relationship among the observed data. For example, happiness is treated as a latent variable in economics. Happiness cannot be measured directly, but it can be inferred from other observable variables like lifespan and education. Studying these latent factors is called factor analysis, and factor analysis is a mainstream tool for researchers in the behavioural and applied sciences.

In natural language, word frequencies are the most common form of observable variables. Instead of measuring a country’s attributes, such as the gross domestic product or a citizen’s average lifespan, attributes of a document are measured, such as word counts. These attributes are provided as input to a statistical structure called a topic model, in which a “topic” describes some relationship between parts of the data. In a domain like economics, it may be the case that a topic relates to a concept such as happiness.

This leads to a question for software developers. Can the same kind of observations taken on a country or a natural language document be made on a software system?
And if so, what are the patterns identified by the latent factors? Factor analysis has already been applied to software systems, but it is not clear if the results can be explained in human-oriented expressions.

Although it is understood that there are many applications for topic models, they are easily applied superficially, leading to “a situation where many important concepts are ignored or poorly understood” [93, p.286]. We may not yet understand the true power of these techniques, and have not gained all of the information we can from them.

The main advantage of topic models is that they provide an improved model of the system being studied: they strip away variation that is not of interest, that may be characterized as noise, and they reveal deeper relationships between the parts of the system, relationships that may be obscured by the particular set of attributes being collected. A second important advantage is that they enable the size of the data to be reduced. A large set of collected attributes is reduced to a much smaller set, making subsequent computations more practical. Therefore prediction and clustering can be done using the latent factors instead of the collected attributes with more robust results at less cost.

In this thesis, our research shows that the latent factors found in commonly used topic models relate to the development history of a software system. While it is not always possible to describe software factors with human-oriented concepts like happiness, it is demonstrable that these factors can identify historical maintenance relationships in source code. Specifically, when a developer makes a change to a software project, it is common for a significant part of that change to relate to a single factor. Two significant results can be drawn from this conclusion: latent topic models identify co-maintenance relationships with no supervision, and topic models can be used to support the maintenance phase of software development. We support these conclusions by examining the project history and topic models generated by
a wide set of open source systems ranging from a few dozen source code methods (functions) to the entire Linux kernel, with over 150,000 methods.

To show the relationship between topic models and software maintenance, we introduce visualizations with three goals in mind. These visualizations show how topics are distributed over changelists taken from a project’s revision history, explore local pairwise relationships between code fragments, and plot global structure as extracted from a model. These visualizations allow topic models to be evaluated on their own and against other models. Our approach is also applied in a case-study for a set of web service operations to show how domain-specific programming languages can be best modelled. We also use knowledge about code structure to estimate an appropriate number of topics for modelling software systems. Finally, we use the techniques developed in this thesis to evaluate a blind source separation technique previously unused as a topic model.

Software development tends to be dominated by maintenance [92]. With this in mind, this thesis shows that topic models can be beneficial during software maintenance by identifying the relationship between co-maintenance relationships uncovered by mining software repositories and clusters identified by topic models. This thesis is therefore useful because it points to methods for predicting the necessity for co-maintenance without the heavyweight overhead of mining or maintaining historical relationships.

1.1 Goals of the Thesis

Evaluating topic models is generally a subjective process. Topic count selection has even been described as art instead of science in some cases [66]. As a result, it is nearly impossible to accurately gauge one technique’s performance compared to another. The strengths and weaknesses of the techniques may be hypothesized, but
it is difficult to objectively judge the relative performance of each technique. The goal of this research work is to improve understanding of latent topic models by outlining an evaluation approach based on observable historical data. As much as possible, we would like to make models based on science instead of art.

We begin by examining the topics identified by several topic models in Chapter 4, and explore the relationship between topics and co-maintenance history through a visualization of the changelist history. This information is used in Chapter 5 to motivate two new topic model visualizations and to provide a local pairwise similarity view for documents and a global system-wide structural view. Our global structural view is used in Chapter 6 to identify a similarity between topic similarity and software clone detection. These visualizations are then applied in Chapter 7 to a domain-specific language to find web service similarity. Chapter 8 uses the information to tune topic models and find parameters that provide appropriate models for use with software maintenance. Finally, in Chapter 9, we show how a previously unused blind-source-separation technique called Independent Component Analysis can be used to identify topics in source code.

**Thesis Statement:** Through a combination of visualization and software analysis, latent topic models can be shown to identify co-maintenance relationships even with no supervision. This correlation can be used to categorize and understand maintenance history, and to compare latent topic models to one another.

In this work, we look at latent topic models as software developers or maintainers who want to use such a model to obtain usable information about their software project. With this goal in mind, we want to identify how this user can understand what information is being captured by the topics.
This research was motivated in part by the fact that there is no clear objective technique for comparing latent topic models. Some preliminary work has been done in this area [22, 73, 74], but an objective evaluation is still not available. For example, with results that are subjective or empirical, it is not clear how to compare two models made by different topic modelling techniques. Additionally, the results that are returned are rarely compared with expert evaluations on topic accuracy as seen in the source code. While it is true that some related methods are often grouped together, the topic quality is difficult to evaluate.

1.2 Thesis Outline

Chapter 2 provides a summary of latent variables and latent variable models. The chapter includes a brief survey of some common topic models, including the pairwise similar metrics used.

Chapter 3 surveys previous work in the program comprehension community using topic models for tasks including concept location, impact analysis, and traceability recovery. The chapter also provides a history of the area, touching on early human-oriented software development, and on the early desire to identify concepts in source code using unsupervised approaches.

Chapter 4 introduces a visualization to plot the topic distribution across changelists. These changelists are the atomic changes tracked by revision control systems. By reviewing changelists, the maintenance history of a project can be observed directly. We explain the visualization and how it is generated, and examine emergent patterns of maintenance history that recur over the project’s lifetime. We discuss what information can be obtained by these patterns, and how the information justifies topic models as an aid for software maintenance.
Chapter 5 introduces two visualizations that provide insight about different characteristics of the topic models. In the first, a local per-document view of topic similarity is rendered. This allows us to take individual documents, which in this case are software methods, and to examine the most similar code fragments as determined by the topic model. In the second, a global system-wide view is plotted. This global view indicates where the most similar sections of the software project are located. Such a view can be used to compare different topic models, or to evaluate how well a topic model is able to extract meaningful similarities from a collection of source code.

Chapter 6 applies Independent Component Analysis, a latent variable model previously unused in program comprehension, to perform software clone discovery. The discovery of software clones is related to similarity detection. Documents in the vector space produced by Independent Component Analysis provide an estimate of the likelihood that two code fragments are clones, as opposed to a boolean conclusion.

Chapter 7 applies the visualizations developed earlier to examine how contextualization can improve topic model quality for domain-specific languages. These visualizations provide direct evidence validating contextualization.

Chapter 8 uses software system architecture to find an appropriate value for the topic count. By improving parameter selection using topic models by observing the software system in question, we show how a model can capture architectural information more accurately.

Chapter 9 returns to Independent Component Analysis, discussed as a software clone detection technique in Chapter 6, and shows how it can be used as a topic model. The chapter provides an implementation of ICA as a topic model, and examines two software systems in detail. A direct comparison is made to LDA using the visualizations introduced earlier in this thesis.

Appendix A provides a brief overview of some algebraic and statistical techniques used in the thesis.
Chapter 2

Background

Statistical models are often used to identify relationships between variables. In natural language processing, these variables generally correspond to token presence in the sampled documents. For example, in a document set built from a collection of ten-thousand terms, a statistical model with ten-thousand variables can be constructed to evaluate the relationships between terms and documents. This idea has been adopted in source code analysis, where source code fragments such as methods, functions, or classes are used in place of the natural language documents. This chapter describes latent variable model theory, and provides a detailed description of several common models that are currently used.

2.1 Latent Variable Models

2.1.1 Latent Variables

A latent variable model uses a collection of observed random variables to identify a smaller set of unobserved variables. The observable variables which have been directly measured in some way are observed attributes. The unobservable variables
are called latent variables, and are inferred somehow from the observed 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 [9].

Both observed and latent variables can represent data in two ways. Metrical variables are formed from 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. This ordering may occur with sizes that range from small to large and can be compared to one another, or may not occur for categories like words that have no intrinsic ordering. The representation used for the observed and the latent variables helps decide the model, and should therefore be chosen carefully. For example, it is possible to classify an individual’s height as either a categorical variable (tall or short) or as a metrical variable (height in centimetres).

In the social sciences, latent variables are used to represent highly abstract concepts like intelligence, social class, power, and expectations [17]. Economics considers concepts like quality of life, morale, and happiness as theoretical values that are hidden inside observed data. In program comprehension, no analogous latent variables have been consistently identified. Simple abstractions like quality of life do not seem to be applicable with the input data obtained through source code analysis methods.

2.1.2 Modelling Latent Data

Latent variable models are used for two main reasons. First, models derived from a large set of data may be too big to process in a meaningful way. Using a latent variable model to extract latent variables as new representations of the data can act as a dimensionality reduction technique to transform a large matrix into a smaller close representation or approximation of the data. Many of these latent models can
even provide a value for the accuracy maintained in the new representation, such as a rank-reduced matrix approximation\(^1\). Second, extracting the latent variables can help to detect structure in the relationships between the observed variables. Identifying correlations in this way can show information about the original data that may not have been immediately clear. In the economics example, it may be possible to show a correlation between the latent morale and happiness variables that is not as obvious in the original observed variable set. This data may be used to 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 observed variables that can be explained by a mathematical relationship between them, and that these relationships can be extracted as latent variables.

Figure 2.1 shows the commonly used techniques when dealing with latent variable models and certain variable types [79]. The two latent class models used most commonly in program comprehension are factor analysis and latent class analysis. Both of these models assume that the observed and latent variable types used in each model are the same. Factor analysis is based on the assumption that the observed metrical variables are composed of linear combinations of factors, which are latent

\(^1\)See Appendix A for a detailed description. A rank-reduced matrix approximation can be obtained by choosing a threshold percentile of information to retain. The sum of the retained singular values divided by the total sum of singular values can be used to estimate how much information is captured in the new rank-reduced matrix. The number of retained dimensions can then be increased or decreased to keep more or less of the original data.
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 [42].

2.1.3 Factor Analysis

Factor analysis, as used to describe a wide range of related techniques that measure relationships between metrical variables, is the most common latent variable model currently used in program comprehension. This analysis originated in psychology to condense a set of children’s academic scores down to a smaller set of values representing general mental ability. Generally, factor analysis will operate over a correlation matrix. A correlation matrix for $n$ random variables is a square $n \times n$ matrix where an entry at $i,j$ is the correlation (roughly, the relationship) between the $i$th and $j$th random variables. This matrix is used 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 [24].

These methods extract factors from a correlation matrix, resulting in a vector of numbers that represents each observed variable’s relevance 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 indicates the related observed variable’s relevance 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 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
that 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 observed and latent variables, is closely related to factor analysis and discussed in Appendix A.1. The Vector Space Model [99] and Latent Semantic Indexing [30] are also related to factor analysis by the assumption that latent structure exists and can be identified and extracted.

2.1.4 Latent Class Analysis

Latent Class Analysis is a technique for analysing relationships in categorical data [79], and is often used for organizing sets of data in clusters. The main justification for its use is the fact that many variables, observed 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 known categorizations, and to show that the assumptions adequately represent the data. Latent class analysis 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. This necessitates a different approach for extracting these results.

A latent class model specifies how many classes should be extracted from the data; this number is analogous to the number of factors to identify. From this, the latent class probabilities can be determined; probability distributions for documents over latent classes that suggest membership in the latent classes for each document.

Probabilistic Latent Semantic Analysis (PLSA) and Latent Dirichlet Allocation (LDA) are two examples of latent class analysis.
2.2 Vector Space Model

The vector space model (VSM), introduced in 1975 by Gerard Salton [99], uses an algebraic approach to document modelling. For a set of documents with a total vocabulary of \( n \) terms, each document is identified by a vector in an \( n \)-dimensional vector space. For a document \( D_i \), the \( n \)-dimensional vector representation can be represented by Equation 2.1.

\[
D_i = (w_{1,i}, w_{2,i}, \ldots, w_{n,i})
\]  

(2.1)

Each dimension in the vector space corresponds to a term, and therefore if a term occurs in the document, it will result in a non-zero value in that position in the vector. Vectors are often sparse, and are always non-negative without normalization, since there can never be a negative number of instances of a term in a document.

The non-zero values at each position in the vector can be computed in several different ways. One simple approach is to use each document’s term frequency. Term frequency is not ideal, as it can result in extreme imbalances between short and long documents that share terms. Since longer documents are likely to use terms more frequently, there can be extreme magnitude differences along each axis. One remedy for this is tf-idf weighting, where the term frequency (tf) and the inverse document frequency (idf) are used to scale term weights proportionally by term frequency in the document and in the entire corpus. If a term occurs frequently across the entire set of documents, as is common with words like “and”, it will receive a small weight in documents where it appears. If a term occurs many times in only a few documents, such as a domain specific term like “refactor”, the weight of that term will be relatively larger than common terms.

With a vector representation, the similarity between pairs of documents can be obtained using techniques that evaluate the similarity of vectors. One similarity
metric that is frequently used is based on the cosine similarity between the vectors. For a pair of documents $d_1$ and $d_2$, and the angle $\theta$ between those documents, the cosine similarity can be obtained with Equation 2.2.

$$\cos \theta = \frac{d_1 \cdot d_2}{|d_1||d_2|}$$

(2.2)

The cosine similarity ranges from $-1$ to 1, where $-1$ indicates two documents that are complete dissimilar and 1 indicates two documents that are the same. In a non-negative term-frequency vector that can be derived from a document, the cosine similarity will always be in the range from 0 to 1. However, after normalization, this may no longer be true.

The vector space model is a simple approach to applying algebra to document similarity. This model is language independent, is easily understood, and once documents have been converted to vectors, has a straightforward approach to document similarity and document ranking by query. However, it suffers from issues with polysemy and synonymy, the ability for words to have multiple meanings and for multiple words to represent the same meaning. For example, the term “love” has an extremely different semantic meaning when dealing with romance than it does with tennis.

### 2.3 Latent Semantic Indexing

Latent semantic indexing (LSI), or latent semantic analysis, was introduced in 1988 [30]. LSI was described as a way to take advantage of a higher-order latent structure found by the association of terms across documents, and used to identify relevant documents by textual queries [29]. These term-associations are 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 an algebraic problem. This approach builds on the vector space model by using a matrix decomposition to approximate the original document-term representation of the data set. With the decomposition, a set of latent data can be extracted and interpreted as the latent semantic information.

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\(^2\). The improvement over the basic VSM model comes from this matrix decomposition. The singular value decomposition identifies the most significant features of the data set and throws away the relatively unimportant noise. The matrix approximation is used instead of the original document-term matrix with the cosine similarity (or some other similarity calculation) to perform queries like the VSM.

Equation 2.3 gives the notation for a rank-reduced approximation of a matrix using the singular value decomposition (SVD). We consider a document-term matrix \(A\), where rows correspond to documents in some original set, and columns 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 Chapter 2.1, 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.

\(^2\)Technically, LSI can be applied to either a term-document matrix or a document-term matrix. Swapping the position of the rows and columns changes which matrix in the decomposition holds the terms and which holds the documents. In this thesis, a document-term matrix as described in this chapter is used unless otherwise noted.
CHAPTER 2. BACKGROUND

\[ A_k = U_k \Sigma_k V_k^T \] (2.3)

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 document vectors, and the rows of \( V_k \) represent the term vectors. These vectors maintain most of the semantic information about the document and term space, and as linearly independent components, are often considered to be independent patterns that are composed into the inputs. That said, the goal of LSI is not necessarily to accurately describe the patterns 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 documents and terms 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 \( A_k \).

To compare two documents, the rows are compared against one another. Given the approximation matrix \( A_k \), the dot product between two row vectors provides the extent that two documents are related. The matrix \( A_k A_k^T \) is the square symmetric matrix that contains the entire set of relations, and it can be shown that:

\[ A_k A_k^T = U_k \Sigma_k^2 U_k^T \] (2.4)

Comparing two terms using LSI can be used to show 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 column vectors provides the extent that two terms are related. The matrix \( A_k^T A_k \) is the
square symmetric matrix that contains the entire set of relations, and from the SVD decomposition, it can be shown that:

\[ A_k^T A_k = V_k \Sigma_k^2 V_k^T \] (2.5)

LSI is also language independent, and does not require 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.

While LSI uses a singular value decomposition to calculate the document-document and term-term similarity matrices, LSI and SVD are not synonymous. LSI is not a topic model. LSI is used to identify similarities between documents and terms to allow similarity queries to be posed. The analogous topics are the singular vectors identified during the singular value decomposition.

The essence of LSI is the normalization. In this thesis, document-term matrices used with LSI are normalized using the standard z-score. A standard score converts elements of the document-term matrix from tf-idf weights into units of standard deviation from the mean score of the column. Standard scores are negative when the value is below the column mean, and positive when the value is above the column mean. This normalization produces columns centered around zero. Without this normalization, all of the points in the document-term matrix will be positive, and the most significant singular vector will be directed toward the center of the entire set of points instead of identifying descriptive attributes of the document set.

### 2.4 Probabilistic 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 [45]. PLSI is strongly based on latent class analysis. 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 document-term 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. One of the primary problems with PCA, and by extension SVD and LSI, is the requirement that each axis is orthogonal to the others. This optimal reduction of the data works fairly well for describing the original set as a more distinct number of components, but may not describe underlying features that lie on non-orthogonal axes.

PLSI is based on the maximum likelihood statistical method to produce a generative statistical model, and assumes the existence of an unobserved latent class variable $Z$ that independently generates each of the observed words 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. When solving a system using PLSI, the input is given as a document-term matrix representing the observations over the document set. As a generative model, it attempts to explain the observed pair $(d, w)$ for a document $d$ and a word $w$ by a probability that it was generated from the latent classes given by $Z$.

As a generative model, it gives the probability $P(d, w)$ that the word $w$ will be observed in document $d$. This is done by selecting a document $d$ with probability $P(d)$, selecting a latent class $z$ from $Z$ with probability $P(z|d)$, and generating a word $w$ from $z$ with probability $P(w|z)$. The observation is then explained by Equation 2.6.
\begin{align*}
P(d, w) = P(d)P(w|d) = P(d) \sum_{z \in Z} P(w|z)P(z|d) 
\end{align*}

To estimate the unknown probabilities (generally referred to as the posterior probabilities) used in Equation 2.6, PLSI uses expectation maximization (EM). EM uses two steps to estimate posterior probability. The expectation step computes the expectation of the log-likelihood function for the model, which is used as a measure for how well the model explains the possibility that the document set could have been generated. The maximization step updates and improves the estimated probabilities using information taken from the expectation step.

### 2.5 Latent Dirichlet Allocation

Latent Dirichlet Allocation (LDA) is an improvement over the PLSI model that also defines a generative prior topic distribution over the documents [16]. As in PLSI, LDA assumes each document can be considered a mixture of the latent topics. The primary difference is that the probability distributions of each topic generating a given document are assumed to have a Dirichlet distribution.

LDA 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. In particular it assumes that each document has a particular distribution of topics, and that each word is chosen by first choosing its topic and then choosing a particular word from that topic. A topic is therefore a related set of words which, in turn, induces relationships on the documents. Algorithmically, if the document-term matrix is $n \times m$, and the LDA model is asked to find $k$ topics, the result is an $n \times k$ matrix giving the membership probability of each document in each topic, and a $k \times m$
matrix giving the membership probability of each term in each topic.

Like PLSI, LDA attempts to infer an unobserved set of topic assignments from the original data, taken as word observations from a set of documents. The introduction of probability distributions to the model requires that the posteriors must be approximated using techniques like variational inference or Gibbs sampling [65].

As the documents are represented as probability distributions in the model, the similarity between two documents can be computed using probability similarity metrics like the Hellinger distance [15]. The Hellinger distance is a similarity metric that, in this context, helps to find pieces of data that have similar topic relationships. For example, in a simple latent topic model with two topics, each document is represented as a distribution over them, or essentially as a pair of probabilities. If two pieces of data have similar probabilities of being related to the topics, those pieces of data would have a low Hellinger distance, and therefore a high likelihood of being similar. When the Hellinger distance data is normalized to a value in the range $0 < x < 1$, and then subtracted from 1 (we want low distance scores to correspond to high probability of relationship, so distance 0 should be similarity 1), each individual distance score provides the means to estimate the probability that two documents are conceptually related to one another. Other normalizations are possible, including the reciprocal, but do not affect the order of the normalized documents.

The following table gives the probability distribution for each of the documents for two sample topics $t_1$ and $t_2$, and the Hellinger distance from each other document (larger numbers indicate less similarity).

By using LDA as a model and calculating the Hellinger distance between the topic distributions, it can be seen that $d_1$ and $d_2$ are the most similar documents. After normalizing the data, smaller distances like the one between $d_1$ and $d_2$ will be close to 1, and larger distances like the one between $d_1$ and $d_3$ will end up near 0.
2.6 Independent Component Analysis

Independent Component Analysis (ICA) [23, 47] is a blind-source-separation technique that separates a set of input signals into statistically independent components. ICA 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 topics from source code [74]. The primary difference is that instead of focusing on signals that are simply uncorrelated, ICA extracts signals that are statistically independent of one another. This is a stronger property, and when used in a domain like program comprehension, can ensure that the difference between extracted signals is more significant than SVD, along with 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 combine 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 data obtained from the microphones, the focus on statistical independence instead of uncorrelation allows ICA to isolate the original source signals, and individual voice data for each of the attendees can be recovered.

<table>
<thead>
<tr>
<th></th>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$h(d_1)$</th>
<th>$h(d_2)$</th>
<th>$h(d_3)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_1$</td>
<td>0.33</td>
<td>0.67</td>
<td>-</td>
<td>0.003</td>
<td>0.258</td>
</tr>
<tr>
<td>$d_2$</td>
<td>0.40</td>
<td>0.60</td>
<td>0.003</td>
<td>-</td>
<td>0.210</td>
</tr>
<tr>
<td>$d_3$</td>
<td>0.95</td>
<td>0.05</td>
<td>0.258</td>
<td>0.210</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 2.2: Hellinger distances between documents.
ICA can be described by the equation $X = AS$, factoring an original data matrix $X$ into a transformation, or mixing matrix $A$, and a source signal matrix $S$, where the extracted independent signals are stored. If $X$ is an $m \times n$ matrix, and $k$ independent signals are specified, $A$ 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 were combined to produce the original set of mixed signals [102].

The signals are assumed to be mixed together into the observed data matrix, so ICA is also discussed as a generative model. When compared to LDA, ICA has seen relatively little use as a textual topic model, and is instead widely applied to numerical or signal data analysis.

### 2.7 Summary

Latent variables provide a convenient abstraction for large data sets. They permit dimensionality reduction, and can identify unobservable factors in data that explain relationships.

Factors can be extracted using a wide range of techniques. Latent variable models, including Latent Semantic Analysis and Latent Dirichlet Allocation, provide a language-independent approach to extracting latent relationships from observed data.

In the next chapter, a survey of topic model use with source code is presented. We begin with the history of human-oriented software development, and the idea that source code is developed with a set of concepts or topics in mind. Topic models are then applied to source code for program comprehension tasks. We focus on how Latent Semantic Indexing and Latent Dirichlet Allocation are used to extract relationships from source code and program documentation to benefit software developers.
Chapter 3

Related Work

Modern feature or concept location in software is often defined by Biggerstaff’s seminal paper discussing the concept assignment problem [12, 13], discovering individual human oriented concepts and assigning them to their implementation oriented counterparts for a given program. The concept assignment problem built upon earlier research examining how humans were able to relate elements of a program into human-oriented conceptual knowledge, and suggested that a human-oriented approach to identifying concepts would help human users understand source code.

This chapter summarizes the feature and concept location literature, starting from early research leading to a proper problem definition, up to the current ongoing work on making sense of source code. Topic models are widely used in concept location and program understanding in general, and thus we focus primarily on the application of topic models, branching out for context as necessary.

3.1 Human-Oriented Software Development

As computer programs were understood to require human comprehension instead of being simple machine input, researchers presented theories on how to aid programmers
with efficient software development.

In the 1970’s, a great deal of research was performed 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, was expected to lead to improved design techniques and language features. One of the first full cognitive models came from Shneiderman and Mayer [101]. 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 [19] on program comprehension in 1978 designed to show how he believed programmers 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” [19, p.196]. In his paper, Brooks identifies a set of cues for understanding a program. These include internal cues like comments, pretty-printing, and structure, and external cues, such as user manuals, flowcharts, and published algorithm descriptions. He argues that 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 is by bridging the problem domain and the executing program using a succession of knowledge domains that a programmer gains program comprehension.

As these program comprehension theories developed, the role of conceptual units and their value as descriptive elements became apparent. In psychology, researchers began to speculate on how programming was able to encourage experienced developers to “think in terms of higher level units” instead of simply acting as a code generator.
With the Vector Space Model, described in Section 2.2 and introduced in 1975 by Gerard Salton, the source code is preprocessed and organized into a corpus of documents. These documents can be represented by vectors and 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. All documents within a given empirically determined distance from the query vector are representative of the concept.

### 3.2 Latent Semantic Indexing

The VSM was in use for a decade before Latent Semantic Indexing, described in Section 2.3, was introduced in 1988 [30, 31]. By building on the VSM and incorporating information about the latent variables identified by a singular value decomposition, LSI is able to find relationships that are undetectable by naive approaches like a simple vector space model.

In 1999, Maletic and Valluri [69] and Maletic and Marcus [67] began exploring LSI’s potential in software by performing a handful of clustering and classification experiments against source code and documentation. Their 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 clusters that represent documents within some cosine distance from each other. The early tests were promising, suggesting that even without a grammar or solutions to polysemy and synonymy, LSI can 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 that it “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 [68], and defined a number of metrics for comprehension to assess the semantic cohesion of the documents with each other. These metrics use the profile generated by the application of LSI to the source matrix, and are easy to compute, albeit 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 file cluster, give information on how the clusters conceptually fit the structure of the 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 [68]. They used the data to show that in several large systems, 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 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 by using LSI to map concepts expressed in natural language to the relevant parts of the source code [74]. They noted that a common action by software engineers, when tracking down a particular piece of relevant code was to use the grep utility, which provides a string matching search technique over code. Based on this observation, they attempted to show that 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 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.

Several related approaches began to appear, leveraging the ability of LSI to provide a straightforward, language-independent way to identify relationships between documents. SNIAFL [106], a Static Non-Interactive Approach to Feature Location, revealed connections between features and functions using LSI, and then used the results to generate a Branch-Reserving Call Graph to recover relationships between the retrieved functions. IRiSS [89], an Information Retrieval based Software Search, is a Visual Studio plugin based on the “find” feature that uses LSI to search projects using natural language queries. The authors also developed JIRiSS [88], an Eclipse plugin that searches Java source code for the implementation of concepts. These tools use class or method granularity, and return confidence metrics about how close each returned value is to the original query.

3.2.1 Traceability Recovery

The language-independence of VSM and LSI allows researchers to combine source code and documentation with almost no additional overhead. New files can be simply added to the document set and treated like other pieces of data. This was identified fairly soon after LSI’s first introduction to source code, and led naturally into 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 [103]. Requirements traceability involves describing and following a requirement in forward and backward directions [32]. While there are many benefits in retaining information about a requirement’s life, the lack of automated techniques for generating traceability links can often make recovery a costly process.

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 [5]. 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 semi-automatic method for recovering traceability links. The results of their study indicated that these information retrieval techniques afford some ability to recover traceability links in a semi-automatic way. Marcus and Maletic [70] expanded the analysis to include LSI as a querying model, and determined that, when 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 requirements [60], and experimented with different link selection strategies and case studies.

The idea of using LSI as a tool to support software developers in identifying the proper traceability links was examined in a study involving the ADAMS system [26]. The primary difference between ADAMS and the previous work was that LSI was used in an environment designed to 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 150 artefacts produced. The document-term 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 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 [27, 80] focused on choosing a good cut-off value and LSI’s inability 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 150 students and 17 software projects, to determine how effective a tool based on LSI could be [61]. Each team was asked to evaluate LSI’s ability to determine 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 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 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 [61] confirmed that IR-based traceability recovery tools were effective in reducing the time spent by engineers manually tracing through the code [28].

3.2.2 Conceptual Coupling and Cohesion

The importance of tracking metrics in the software development cycle is a well-demonstrated need [20]. 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 [87], and conceptual cohesion is a measure of the degree that elements of a class belong together [71].

Poshyvanyk and Marcus used LSI to identify the amount of conceptual coupling 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 were 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 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 that appeared to leverage the latent semantic information contained in the set of identifiers and comments.

Impact analysis involves estimating how changing one class affects other classes in the code. This analysis 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 [72, 90], much like in their earlier work from 2006 [87]. They strip comments and structural information like other approaches, convert the code into a document-term 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.

### 3.2.3 Evaluating and Tuning LSI

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 [67, 69], and a recent study showed “islands of stability” around 300 to 500 dimensions for documents sets in the millions, with degrading performance outside of that range [18]. 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 document count in their data set is smaller than most natural language corpora [53]. Unfortunately, the authors do not give a 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. To access
the information obtained from dynamic traces of execution scenarios, Poshyvanyk et al. [91] took two techniques they had previously developed, one static and one dynamic, and combined them to identify concepts and features in the source. An LSI implementation [73] was combined with a Scenario Based Probabilistic ranking of events [6] 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 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. [22] introduced the cognitive assignment technique, which uses information flow and co-occurrence information derived from non-source code artefacts to carry out 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, 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 showed some 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.

3.3 Latent Dirichlet Allocation

The first application of LDA to source code was in 2007, when Linstead et al. [58, 59] began to use LDA to visualize the emergence of topics over several versions of a project. They looked at seven version of Eclipse, and the entire history of ArgoUML, to provide an unsupervised technique to identify feature integration and design refactoring milestones. These topic distributions were plotted, and showed 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: the emergence of new functionality, the refactoring of functionality, and the concerns whose prevalence are related only to the size of the codebase under inspection. The authors were able to match the emergence of new functionality in situations like major version changes that added a new feature. The refactoring of functionality, a fairly common activity in code maintenance, led to topic distribution plots that were often flat as new code was added and removed. Code programming practices like string manipulation and logging, which can be expected to grow along with the overall size of the code, resulted in topics that were seen more or less frequently based on the size of the codebase under inspection.

Shortly afterwards, in 2008, Maskeri et al. [77] showed LDA’s application to
CHAPTER 3. RELATED WORK

the extraction of business topics from source code. In a similar approach taken by researchers working with LSI, PLSI, and other IR techniques, they provided 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 topic count 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.

Lukins et al. [62] used LDA and LSI in a study comparing performance for bug localization in software. Bug localization involves using known information about a bug to identify the source code needed to correct the problem. To show how LDA compared to LSI, several case studies were used to show that the LDA-based bug localization technique performed at least as well as LSI-based techniques, and in many cases, performed much better.

Savage et al. [100] introduce TopicXP, an Eclipse plug-in designed to present LDA topics to developers while programming. TopicXP uses two visualizations to show the relationships between topics. A Topic Dependency View lists topics as boxes labelled with the most relevant tokens and Java packages. Dependency links between the topics show how frequently methods in one topic call methods in another topic. A Topic Contents View consists of boxes representing classes in the system, with the size of the box relating to the degree that the class belongs to the topic. The authors performed a user study to show that the tool can effectively assist developers
in maintenance tasks such as concept location.

Oliveto et al. [81] undertook an empirical study to evaluate the equivalence of traceability techniques including the VSM, LSI, LDA, and another information retrieval technique called Jensen-Shannon. Each technique’s recall and precision for identifying traceability links was determined. The surprising result in their research is that LDA, while finding information that is unique to the techniques considered in the study, is significantly outperformed with respect to accuracy by all three other approaches.

The Diff model introduced by Thomas et al. [105] is a topic evolution model that takes advantage of the fact that software is incrementally updated between versions. Topic evolution models explicitly include time in the model, analysing how topics change over that time-frame. The state-of-the-art approach is the Hall model, which applies LDA to all versions of all documents by using the full instance of each document at each point in time. The Diff model improves on this for software by noting that most files are not changed between software versions, and that the changes that do occur are very small. These observations lead to a model that eliminates excessive data duplication and, as demonstrated in an empirical evaluation, lead to better topic evolution models for software systems.

3.4 Independent Component Analysis

ICA has been used successfully in natural language topic detection, by considering each extracted signal as a topic [14, 50]. We have also used the technique successfully to segment large document sets in the same way, identifying the major topics that are used [40]. LSI has seen more use in extracting information from source code [74, 73], and shares many similarities. However, due to the difference in assumptions about the nature of the signals, ICA may be better suited to detecting smaller unique
topics than LDA, as it expects statistically independent signals. This stronger bound should force results that are significantly more distinct than techniques that assume uncorrelated signals.

3.5 Web Service Similarity

With the growth of the web, along with the corresponding growth of web services, a concerted effort has been made to provide automated web service discovery. Paolucci et al. [83] discussed the claim that a semantic representation of web services would enable matching for related web service capabilities. In particular, they identified WSDL’s lack of semantic information for this problem, and proposed DAML-S, a language for service description based on the DARPA Agent Markup Language (DAML). The authors suggested that an XML-based standard for web service description lacked sufficient semantic information, and that “two identical XML descriptions may mean different things depending on the context of their use.” This observation remains true today, and was a motivating factor in our decision to use contextualization. The goals of the semantic web continued to evolve towards rich semantic specifications for web services and operations [57, 75].

The first approach to using information retrieval techniques for web service discovery was in 2005, when Platzer and Dustdar used WSDL files in conjunction with the vector space model [86]. The authors used the vector space model to tokenize each web service description and to compare them against each other using the cosine similarity, a common similarity metric. Although WSDL files can be sparse, they discovered that even a basic analysis of the keywords found in WSDL sources can be enough to build a usable search engine for web service discovery based on their approach. Paliwal et al. [82] extended this idea with a novel approach to web service discovery by applying Latent Semantic Indexing (LSI) to information derived from
WSDL service descriptions. In their work, they also faced the issue of limited information in the service descriptions, and addressed the problem by linking results with a domain ontology. Our shared solution goal was the addition of implicit semantic information, and by using an ontology, the implicit data becomes explicit, allowing models like LSI and LDA to find good relationships.

Ma et al. [64] also developed an approach for web services discovery and an initial divide and conquer strategy followed by a singular value decomposition. To address the ever-present issue of a lack of context in web service description files, the authors clustered sets of operations together to discover a relevant group of services. Once they found a clustered group that was most similar to their desired query, they then applied a singular value decomposition to identify the most similar web service. With their approach, the authors aim to eliminate as many of the irrelevant services from the problem set to give their algorithm a better chance of finding an appropriate solution. The authors also experimented with Probabilistic Latent Semantic Indexing [63], but they do not provide a great deal of data to evaluate.

In a recent survey of survey discovery approaches, Rambold et al. [94] performed a comparison of 42 different approaches. Of the techniques evaluated, only a handful dealt with related approaches like the vector space model or LSI. In Kokash et al. [49], a combination of lexical and structural matching techniques are used to evaluate the similarity between topics. Lee et al. [56] also build a vector space model representation of the web services using a clever system for encoding service information into elements of an ordered tree, and execute similarity requests through SQL queries to a standard database.
3.6 Summary

Topic models are widely used in program comprehension research to identify relationships in source code. They have been applied to mining software repositories for bug detection, extracting topics in source code, and evaluating the evolution of source code. Some studies have been made to evaluate the performance of models including the VSM, LSI, and LDA, but no firm consensus has emerged for all uses. Nonetheless, topic models have been shown to address many maintenance problems with source code by helping to categorize and relate similar code fragments.

The notion of a topic is still debated. Many authors propose topics to be concepts that can often be labelled appropriately [7, 44], although Blei et al. emphasize in their original publication about LDA that they “refer to the latent multinomial variables in the LDA model as topics, so as to exploit text-oriented intuitions, but we make no epistemological claims regarding these latent variables beyond their utility in representing probability distributions on sets of words” [16, p.996]. It seems that topics are not always neatly paired with human-oriented concepts, such as features or concerns.

In the next chapter, topic models will be used with the revision history of a software project to show that models like LDA can cluster code fragments that are likely to be co-maintained. A visualization is introduced to show how topics are maintained over the life of a project. We argue that understanding the type of clustering that occurs when modelling source code is an important part of understanding how topic models can be applied in software maintenance.
Chapter 4

Visualizing Topic Similarity

A topic model is a statistical model used to identify a set of latent topics in a data set. The fundamental premise behind a topic model is that there is some correlation in the patterns of use of tokens in documents of the data set that can be explained by a mathematical relationship between them, and that these relationships can be extracted as clusters called topics. If two code fragments are strongly related to the same topic, they are likely to be similar to one another. We would like to understand the nature of the similarity to make better use of topics during software development and maintenance.

This chapter uses a visualization to explore the relationship between topic models and co-maintenance history. Co-maintenance is defined as an observable property of software systems under source control in which source code fragments are modified together in some time frame, and the co-maintenance history of a project is the revision history. Each set of files changed at the same time are recorded in a changelist. We work at this changelist granularity to show that code fragments that are modified together often share a conceptual relationship as discovered by topic modelling techniques such as Latent Dirichlet Allocation. This result suggests that latent topic models can be shown to identify co-maintenance relationships even with
4.1 Background

Topic models are described in terms of the tokens that are most relevant for each topic, along with the documents (in this case, the code fragments) that are most strongly related to that topic. However, the topics are only anecdotally related to software maintenance; clusters of related code fragments are induced from topics but their validation is weak.

This prompts a simple question: are code fragments that are maintained together clustered by topic? Although we suspect that co-maintained code would be found to be related in a topic model like LDA, there is little evidence showing the existence of such a relationship. To address this issue, we developed a visualization to show the relationship between co-maintenance and topic clustering. By explicitly showing how topic clustering occurs within changelists, this visualization motivates the role of topic models in software maintenance.

This view of a project history can provide insight about the semantic architecture of the code. For example, we have used the set of visualizations to identify patterns that characterize particular kinds of maintenance tasks. These include visual cues about where features are developed and implemented, where potential errors in the code were introduced, and where large aspect-like or lexical changes are made. We then suggest how these topic models can be used to predict co-maintenance of source code.

4.2 Method

Consider the following example of a document-term matrix.
\[ s_1 = \text{My dog has fleas} \]
\[ s_2 = \text{That dog has fleas} \]
\[ s_3 = \text{My ukulele has fleas} \]
\[ s_4 = \text{My team won the football game} \]
\[ s_5 = \text{That dog ate all the turkey} \]

Five input documents (or code fragments, when parsing source code) are given, named \( s_1 \) through \( s_5 \). In total, there are six tokens used in the input, of which five are non-unique. Since the token 'a' is only used in \( s_2 \), it is only relevant for \( s_2 \) and can be omitted, as it is outside the scope of tokens used in the comparison.

\( \text{tokens} = \{ \text{all, ate, dog, fleas, football, game, has, my, team, that, the, turkey, ukulele, won} \} \)

\( \text{non-unique} = \{ \text{dog, fleas, has, my, that, the} \} \)

The input matrix, here referred to as \( x_{\text{flea}} \), will necessarily be a \( 5 \times 6 \) matrix, with the five rows representing the input documents \( s_1 \) through \( s_5 \), and the six columns representing the non-unique tokens above in the order they are encountered when parsing the input documents. Each of the entries in \( x_{\text{flea}} \) represents the frequency of that term in that document.

\[
x_{\text{flea}} = \begin{bmatrix}
1 & 1 & 1 & 1 & 0 & 0 \\
1 & 1 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 \\
1 & 0 & 0 & 0 & 1 & 1 
\end{bmatrix}
\]

The set of documents used in the topic models consists of the complete set of source code methods in the package. Each of these source code fragments has some probability of membership in each of the topics found by the model. In each of these
models, we identify the largest probability, and use that as the most significant topic for a document.

The majority of this chapter uses the Apache \textit{httpd} web server, written in C, as the example system from which to generate the visualizations. However, we have evaluated this visualization with dozens systems across several languages. We also use either Subversion \cite{Subversion} or Git \cite{Git} repository logs as the source of maintenance history for the systems. A model of the historical maintenance data is generated using Latent Dirichlet Allocation with either Mallet \cite{Mallet} or GibbsLDA++ \cite{GibbsLDA++}. Mallet and GibbsLDA++ use two different algorithms for generating the latent model, although the results are largely similar. Latent Semantic Indexing models are created with Matlab.

First, a snapshot of the project under revision history is taken. A parser from the NiCad clone detection tool \cite{NiCad} is used to extract all of the functions from the package. Comments are stripped out, and the camel-case and underscore-separated compound tokens are split to give a larger set of meaningful words. Both full and split tokens are retained, and no stemming is used. These terms are used to build a document-term matrix similar to $x_{flea}$, the example matrix constructed above. Document-term matrices used with LSI are built with tf-idf weighting and z-score normalized, as described in Section 2.3. The LDA input files use term frequency in each document, and normalize probability distributions during the LDA algorithm such that distributions sum to 1.

Next, the full revision history is extracted from the source code repository logs. From these logs, all of the changelists that modify a source code function are identified. For each of the changelists with meaningful code changes, a row is generated in the visualization. We take the list of modified code fragments, identify the related topics for each of the fragments, and plot the row based on those results. The size of the circle representing topic changes inside of a changelists is proportional to the number
Figure 4.1: A changelist that modifies a function definition. It will not be possible to identify this function naively when looking at the history of a software project, so we look for patterns in the function prototype and attempt to identify functions that have changed over time.

of code fragments modified in each topic.

Some functions are moved, renamed, or modified over the life of a software project, so we attempt to associate as many of them as possible. Figure 4.1 shows that the `mpm_get_completion_context` function has been modified to accept a single parameter. Since the LDA model was created using a snapshot of the software system, only one of these methods is found in the model.

Figure 4.2 shows how the visualization is structured as an interactive HTML page. The rows of the display are the list of relevant changelists over the history of the project, starting with the oldest, and progressing down towards the newest revisions. Each column in the display corresponds to one of the topics described by the model. The circles of varying size and colour on each row indicate how many code fragments are modified in that changelist. Large red circles indicate five or more code fragments associated with a topic, ranging down to small grey circles that indicate single code fragments.

The actual markers are a combination of `div` and `span` HTML elements that may include a `title` parameter. Detailed information can be obtained by hovering over elements on the screen. The blue squares on the left side are changelist indicators. Hovering over each displays a pop-up with the revision id, author, date of revision,
Figure 4.2: Visualizing topics over changelists. We plot approximately 35 consecutive changelists for the Apache httpd web server using an LDA model with 100 topics. Each row in the table corresponds to a changelist, with the oldest changes at the top of the table, and the most recent at the bottom. The circles are colour-coded and sized to indicate the number of code fragments they represent. In the legend, a topic change indicates a modification to a method related to the topic.

In general, the results are sparse, indicating that each changelist involves relatively few topics. Although there are many changelists with a small number of methods from different topics, it appears that many of the larger changes are, for the most part, related to relatively few topics. This indicates a correlation between co-maintenance history and topic clustering. If we had instead discovered a relatively small amount
of topic clustering within changelists, it would be an indication that topics did not
capture a human-oriented perspective of revision history.

4.3 Patterns

In our visualization of systems, we identified a number of patterns that appeared reg-
ularly. In this section we classify and show by example how these patterns are found,
and discuss why they appear and how they correspond to particular maintenance
activities.

4.3.1 Feature Development

Iterative and incremental development is a common software development practice
[54]. When combined with source code versioning systems, the addition of a new
feature is often characterized by a long series of related changelists. When viewed
in this visualization, a sequence of related changelists shows up as a vertical line of
coloured circles. This indicates a series of changes to source code methods that are
related in the topic model, and is a good demonstration that topic models like LSI
or LDA are capturing co-maintenance relationships.

Figure 4.3 describes as the start of a major overhaul of mod\_include’s filter parser
beginning with the uppermost changelist (revision 101036). A set of consecutive
changes introduces a code wrapper, removes old code, refines the API, and improves
the efficiency and cleanliness of the code. Interestingly, shortly after the vertical
feature implementation line, a horizontal pattern can be seen (revision 101154) that
switches to the new API. This large change necessarily touches many topics, and is
described in the actual changelist as “switch to APR 1.0 API”. After a long run of
feature changes, a larger aspect-like change was submitted to implement the feature
CHAPTER 4. VISUALIZING TOPIC SIMILARITY

Figure 4.3: Visualization pattern: Feature Development. The addition of a new feature is often characterized by a long series of related changelists, and a sequence of related changelists often shows up as a vertical line of coloured circles.

in the code, and this feature addition and move to the new system can be seen in the visualization.

If we return to Figure 4.2, the view of the system also gives at least one example of a feature in development. The long run of vertical red, green, and orange dots in the upper left is described in the changelists as a series of changes to the `apr_send` functionality (revision 88625), and the related code changes that go with it. The long vertical run on the bottom right corner, and the associated changes found in other topics, are described as a significant change to the FTP proxy code (revisions 88721 and forward).

4.3.2 Commit, Patch

It is common to find instances of a single large conceptual change followed by several smaller ones. Figure 4.4 provides an example with two larger check-ins that are followed by a series of smaller revisions. Examples like this often show up visually as
Figure 4.4: Visualization pattern: Commit, Patch. A single larger check-in, followed by small optimizations or bug fixes. This appears as a larger green, orange, or red circle, followed by a trail of grey circles.

a larger green, orange, or red circle, followed by a trail of grey circles. This pattern is seen throughout the code, and indicates a single larger check-in, followed by small optimizations or bug fixes.

Figure 4.4 shows a change (revision 984188) made to mod_proxy that improves communication handling. The next five check-ins fix comments and code, add some additional checks, and refactor code slightly. Other examples include data structure changes that require additional fixes after submission, or propagations that cannot be made in a single large submission.

This type of change is often seen as a subset of the feature-development pattern from Chapter 4.3.1, where incremental changes are made, tested, and patched. In fact, it may be true that this pattern is a specialized version of feature development at a much smaller scale.

### 4.3.3 Co-maintained Topics

One surprising artifact of our visualization is the relatively low frequency of co-maintained topics. More specifically, it is much more common to find changes that are either localized to a single topic, or spread out across many topics. For example, it is rare to find a changelist with a large number of method changes that are evenly distributed across two or three topics. Instead, that large number of method changes
is usually concentrated in a single topic (possibly with some sparse distribution in other topics), or widely spread out like an aspect.

We had considered the possibility that large pairs of parallel vertical lines would emerge, indicating a strong relationship between two topics. Figure 4.5 shows that this does happen from time to time, but it is much more frequent to see changes isolated to a single topic, or distributed across a wide range of topics. Figure 4.5 suggests that developing a new feature that is closely tied to old functionality in some way, such as the addition of a proxy worker pool in httpd instead of a single worker, may result in two distinct topics. In this example, the developers were working with some communication code dealing with a proxy worker pool (revision 104557 and onward). A number of code clean-up submissions were made, along with some data structure changes. The largest vertical line along the right side of the diagram is made up of changes to the actual worker pool code (ap_proxy_get_balancer, ap_proxy_get_worker, and so on), and other vertical lines include proxy connection code and socket functions.

Occasionally pairs or groups of vertical runs will occur, but it is rare. This would seem to indicate that it is much more common for developers to work on either a single conceptual area or on a cross-cutting one when making single check-ins.

It is likely that the match of co-maintenance and topics is linked to the number of topics used in the generation of the LDA model. Figure 4.6 shows the same set of changes from Figure 4.2, but with 50 topics instead of 100. The two images are similar, and many of the same patterns are still visible. However, some of the source code fragments have been distributed to different topics. Some of the sparser dots have merged into single topics, and there are some indications of higher co-maintenance between topics (the three or four vertical bands near the top are one example of this).
Visualising co-maintained topics. There is a relatively low frequency of co-maintained topics. More specifically, it is much more common to find changes that are either localized to a single topic, or spread out across many topics.

### 4.3.4 Aspects

Aspect-Oriented Programming is a paradigm that explicitly allows for the separation of cross-cutting topics (or business concerns) into separate *aspects*. In this context, a changelist that modifies an aspect might be seen as one that modifies code fragments across a wide range of topics.

Figure 4.7 shows one example of aspect development with a wide distribution across topics. In the figure, three changelists (revisions 95149, 95150, and 95151) are made to modify the behaviour of the *ap_log_error* and *ap_log_prerror* logging functions. These functions are distributed throughout the code, and as a result, we
see a wide range of topics affected by the change. Logging is a classic example of a cross-cutting concern, and from our observations, aspects cross topics as well.

Baldi et al. [8] has also suggested that aspects are latent topics with high scattering entropy, which may suggest that true source code aspects would fall under a single topic. However, as discussed above, we have observed no clear instances of this in practice.

4.3.5 Lexical Changes

Larger system-wide changes will show as wide horizontal bars that modify many topics. These are usually not indicative of feature or topic changes, but instead relate
CHAPTER 4. VISUALIZING TOPIC SIMILARITY

Figure 4.7: Visualization pattern: Aspect Development. Aspects appear to be cross-cutting topics in addition to cross-cutting source code.

Figure 4.8: Visualization pattern: Lexical Change. Larger system-wide changes will show as wide horizontal bars that modify many topics, but are not usually indicative of feature or topic changes.

to system wide syntax changes or large renamings. In httpd, we only find a small number of instances of these large changes. Each one deals with a superficial system-wide change like the removal of tabbed spacing, for example, and holds relatively little value from a maintenance standpoint.

Figure 4.8 shows an example of a large lexical change. It stands out clearly in the visualization, and ends up involving nearly all of the topics discovered by the model. From our observations, these lexical changes have no meaningful functional impact, even though they are the most distinct visual features in the display.

4.4 Model Comparison

Figure 4.9 gives two images of approximately thirty changelists visualized using LSI and LDA. Each model is generated using 100 topics, and the same snapshot of the code is used to generate each model.

When we looked at the average cluster size for models generated on our data sets, on average, LDA achieved a higher number of code fragments per cluster than LSI. More generally, the average size of each circle in the visualization was larger using LDA, indicating more clustering by co-maintenance was occurring. For example, in httpd, the average cluster size across the entire code history using LDA is 1.66 with a
standard deviation of 1.65, compared to an average cluster size using LSI of 1.25 with a standard deviation of 0.77. LDA’s larger variation and larger cluster sizes matches real changes better.

There is no explicit ordering relationship in LDA between topic 1 and topic 2, and “adjacent” topics in the visualization are not necessarily related to one another. We can impose an artificial ordering by sorting the columns by the overall strength of the topic in the data set, leading to visualizations that will appear more consistent for similar models. For models made with LSI, the ordering of the topics relates to the size of the singular values, and “stronger” topics that are more significant throughout the data will be found on the left side of the visualization.
CHAPTER 4. VISUALIZING TOPIC SIMILARITY

4.5 Summary

The majority of our research work using this visualization has centered on the relationship between co-maintenance history and the topics identified by Latent Dirichlet Allocation. We wanted to demonstrate that code fragments that are maintained together are likely to be conceptually related to one another. However, topics are not the only metric that can be used in the columns of a visualization like this. If the code fragments can be clustered in some way, the distribution can be used to compare the relationship between the clusters over time.

We have introduced a visualization designed to explore the clustering of features over the changelist history of a project. With the visualization, we observed a relationship between topic models and co-maintenance history by identifying and cataloguing patterns that characterize particular kinds of maintenance. Vertical lines indicate related source code methods in the traditional topic modelling sense, where two code fragments found in the same topic are likely to be highly related to one another. Horizontal lines indicate larger system-wide changes, or changes that necessarily cut across topics.

Understanding that topic models capture information about co-maintenance motivates a closer look at the pairwise similarity results. In the next chapter, two visualizations are presented that focus on the individual relationships between code fragments instead of the topic clustering. We also look at the relationship between topic models and clone detection, a related research area, in which similarities between code fragments are also uncovered.
Chapter 5

Pairwise Similarity

If a reference retrieval system’s response to each request is a ranking of the documents in the collection in decreasing probability of relevance to the user who submitted the request, where the probabilities are estimated as accurately as possible on the basis of whatever data have been made available to the system for this purpose, the overall effectiveness of the system to its user will be the best that is obtainable on the basis of those data.

van Rijsbergen, Probability Ranking Principle, 1999

Topic models are used to cluster groups of related documents. These models identify patterns, represented as unobserved variables called topics, shared between members of the document set. In the context of software maintenance, while these topics are clearly able to identify related code fragments, they can be difficult to characterize in a human-oriented way. If topics are so abstract that they cannot be understood, it is difficult to explain how each topic can be used to benefit software maintenance.
As shown in Chapter 4, topic models can do a good job of capturing the co-maintenance relationships found in software systems. Topic clusters group related code fragments together. However, instead of only using the clusters of individual topics, a similarity metric that uses information about the entire set of topics for a code fragment can evaluate the relevance of one code fragment to another. This is much more intuitive for end users; this is the paradigm used in search engines, where only the results are presented, not the means by which the results were obtained. To use a topic model in a practical setting, the type of similarity between pairs of code fragments must be understood. For example, when modifying a method, a developer could ask what the other most similar methods are based on overall topic similarity, and whether they also need to be modified.

Software clone detection is related to pairwise similarity detection using topic models. Code clones are intentionally given vague definitions [11, 51, 97], illustrating how the notion of similarity is subjective. Baxter [10] even goes as far as defining clones as “segments of code that are similar according to some definition of similarity”. General definitions are also found in concept location, a related research area that intentionally defines concepts in vague terms [13].

In topic models and software clones, similar terms and structure lead to similar documents. The replication of terms in a pair of code fragments will lead to those code fragments being identified as similar. In addition, similarity is not boolean; one widely adopted hierarchy of code clone similarity organizes clones into several types ranging from exact (type 1) clones to semantic (type 4) clones, which perform the same activity [97]. In a topic model, similarity is often derived from a distance metric, and is therefore also not boolean. For example, LDA combined with the Hellinger distance provides a probability that each two documents are similar.

This chapter begins with two visualizations that use topic models as tools for identifying local similarity for individual code fragments, and global similarity that
provides an overview of system-wide structure. We intend these visualizations to be used in an exploratory setting, motivating the use of topic models for software maintenance, instead of directly by software developers. The visualizations are used with Independent Component Analysis, a blind signal separation technique discussed in Chapter 2.6, to find similar code fragments and detect software clones. In the context of the thesis, these exploratory visualizations are used to show how latent topic models identify co-maintenance relationships. These visualizations will be used throughout the thesis to explain qualities of the generated models and to evaluate approaches against each other. For example, in Chapter 7, these visualizations are used to show how preprocessing input documents can dramatically improve the quality of a model.

5.1 Pairwise Similarity with Topic Models

Superficially understanding a topic model is easy; words with high probabilities in a topic represent that topic, and documents with high probabilities of topic membership relate to that topic. However, it is harder to understand a topic model deeply. What information is obtained from the lower probability results, or the tokens that represent a topic poorly? What parameters should be chosen when building the model? Have the right tokens been chosen when generating the model in the first place?

With these questions in mind, we designed a pair of visualizations to make it easier to understand a topic model. In each visualization, we attempted to answer one specific question about a model. These visualizations allow us to empirically evaluate a topic model by observation. We used the results to evaluate different models and model parameters, as well as different variations on the input data set.
We focus on two visualizations of the topic model. Local pairwise similarity involves examining a single code fragment and identifying its most related code fragments. Local similarity determines whether the individual results for a single code fragment appear to be related as we would expect. The global pairwise similarity gives a view of the most related source code fragments across an entire project. Global similarity determines whether the overall model structure appears to identify the structural or architectural relationships we expect to be captured.

5.2 Local Pairwise Similarity

POCO (Pairwise Observation of Concepts) is a visualization tool designed to show how topic models actually segment individual source code methods by similarity. POCO takes the pairwise relationships, and provides a heat-map overview along with a list of the most related methods. In an attempt to show whether or not this treatment of topic models provides utility for software developers and managers, the tool uses real data, giving users the ability to judge whether the topic model is able to capture the latent human-oriented relationships found in the source code.

The primary goal of POCO is the practical visualization of topic models. The common approach to displaying topics in a software system is to provide a list of descriptive terms for each of the extracted topics, along with the best code fragment matches. This works well at a high level, and a list of the topics may provide some insight about the complete structure of the source code package. This data is less useful for a programmer, who may be working on a code fragment that scores poorly in all of the identified topics found by the model. This code fragment will still have similarity relationships, but these similarities may spread across several topics, and may not allow the programmer to identify similar code fragments by simply investigating the list of topics. POCO uses a pairwise relationship score to visualize
CHAPTER 5. PAIRWISE SIMILARITY

the full list of related code fragments for a given function as identified by a topic model, and disregards information about the individual topics. POCO offers two distinct visualizations based on the pairwise relationship score which can be immediately utilized in a development environment or a dashboard.

Figure 5.1 shows a screenshot of a POCO visualization. The screen is segmented into five major sections. The uppermost section gives information about the currently selected source code method, including the file and folder where it is found, and the first line of the method. Immediately below that is a heat-map based on a flattened view of the source code methods, ordered alphanumerically by folder and file. The heat-map, in large systems, provides a way to see whether similar sections of the source code are found near one another. Next is a description of the code fragment over which the mouse is hovering. Below that is a list of the top ten related code fragments, with their id, file name, folder, and the similarity score as determined by the topic model.

The heat-map component gives a visual representation of how each source code method is related to the source code by its architectural layout. For example, we might expect that source code methods dealing with the same topic would often be found near one another in the file-structure.

POCO's heat-map can use either an absolute value threshold or the top percentage of similar code fragments to limit the amount of data rendered to the screen. In the examples given in Figure 5.1, the Apache httpd web server is used as input, using LSI with 250 dimensions as the model, and a heat-map threshold of the top 10% of related code fragments using the similarity score given in Section 2.3. For a particular selected code fragment (dav_method_update on the top, and create_includes_server_config on the bottom), the blue lines indicate a related piece of code for the selected code fragment. Dense blue blocks indicate a large portion of related code in a single location, and sparse coverage of thin blue lines indicates a lack of locality for that
Figure 5.1: POCO: Local Similarity. On the top is an example of a code fragment that is tightly localized to a particular section of the source code. On the bottom is an example of a code fragment with related code fragments that cross-cut the package structure.
these topics may or may not correspond to business concerns, feature implementations, or general conceptual considerations like logging or memory management. They may cluster methods with structural and syntactic similarities, or they may even identify abstract features about the code fragments that are difficult or nearly impossible to adequately describe in human-oriented ways. With the top methods available, it immediately becomes possible to consider the impact of the bug on the other related pieces of source code.

5.3 Global Pairwise Similarity

In a system that is well designed architecturally, the majority of similar pairings (pairs of source code fragments with a similarity score higher than some threshold value) would reside near one another in the file tree. Related methods would be found in the same files or folders, rather than being spread across the system. The idea behind the Bluevis visualization is to explore how the complete file structure of the system relates to its topic distribution.

Each visualization uses an ordered list of the source code functions, first sorted alphanumerically by location in the architecture of the system, and then by the order they are observed in each file. A line extending from one side of the window to the other represents a similarity relationship (above a threshold) between two source code functions. Larger blocks of related code form brighter lines, and random pairings appear as dark and almost invisible. In this way, the global structure of the system can be seen. Although it appears that there are horizontal lines in the plot, these result from similar neighbours (for example, finding code fragments 1234 and 1235 to be similar), and never result from a code fragment being similar to itself. Diagonal lines indicate similar code methods in different architectural locations; the steepness of the diagonal has no relevance.
CHAPTER 5. PAIRWISE SIMILARITY

Figure 5.2: Bluevis: Example Diagram. A visual representation of how Bluevis maps functions to positions on either side of the screen. Strong similarity relationships are marked by a strong line. Any self-references are removed, so the display is not dominated by horizontal lines.

Figure 5.2 shows how these lines are rendered by example. The sample system shows three methods: `create_config`, `merge_config`, and `malloc`. If the topic model determined that `create_config` and `merge_config` were highly related to one another, a line is drawn between the pair of methods. For the two configuration methods in comparison to `malloc`, the scores indicate less relationship to one another, and a faint line (or no line, if a threshold is applied, as in Bluevis) is drawn. The relationship structure emerges in examples like Figure 5.3, and if similar sections of code are consistently near one another, the plot will be dominated by near-horizontal lines.

Of course, we would expect aspects and other cross-cutting concerns to be distributed across the code. Some topics span the entire codebase, and we would not be particularly surprised to see them. The benefit of a plot like Bluevis is that the aspects can be directly visualized using this method. In Figure 5.3, in the left visualization, there are several distinct fans near the top of the graph that stand out.

Smaller systems will not generate meaningful visualizations due to an inability to order the methods effectively. Bluevis uses an alphanumeric ordering by file and folder, and this works fairly well on a larger system. For flat file structures with no meaningful folders, or large folders with many files, there may be significant scattering. Overall, on systems of a reasonable size, this visualization offers a good way to investigating the global layout of the source code package.
Figure 5.3: Bluevis: Global Pairwise Visualization. Screenshots of Bluevis’s interface for two different software systems. The left and right sides of each screenshot correspond to an alphanumerically ordered list of the source code functions, and a connection made between the sides represents a similarity between two different source code fragments above a user-defined threshold. On the left, the large number of horizontal lines indicates a strong local structure, and may indicate that the code is structured in a good way. On the right, the related functions are sparsely distributed on a large scale with only a few small sections of strong locality, and probably indicates poor package design.

We have also begun to use this visualization as a way to compare how effective various models are at capturing similarity structure in a particular data set. For example, given a source code package (in our case, we use the Apache httpd web server as an example), for LDA and LSI with various topic and dimension counts, we can ask how similar the visualizations look for each model. If all of the visualizations
for all of the models look similar, we conclude that the models are not doing anything
different from one another. If a particular model generates images that look chaotic
or lack organization, we conclude that either the source code is poorly structured, or
that the model does a poor job of capturing relationships in this code for some other
reason, such as poor parameter selection.

This visualization takes a higher-level observation of the code than the top nearest
neighbours list found in POCO. A view of the structure of the system can help
determine if the code is well-organized, and can aid a software developer who wants
to know if the section of code they are working on is relevant to the local code or if
it is a cross-cutting concern.

5.4 Summary

We have presented two visualizations that are designed to leverage the pairwise results
obtained by topic models. These tools present some highly esoteric topic structures
in a form that is more easily used.

In the next chapter, we show how Independent Component Analysis, an unob-
served blind-signal separation technique, can be used to identify software clones by
using pairwise similarity metrics that also identify code fragments that are not clones,
but that are still similar, and may be interesting during software maintenance.
Chapter 6

Topics and Clones

Reuse of software code fragments by copy/paste/edit is a common software development practice that leads to a large number of similar code segments, or code clones, in software systems [11, 95, 98]. Code clones can cause problems for software maintenance and evolution [48], making them a popular topic in software comprehension [52].

In the following section, we introduce a method for using Independent Component Analysis to analyse vector representations of code fragments. Our previous work in this area [36, 40] has shown that individual topics found in software can be isolated using this technique. As ICA identifies the signals that are statistically independent from one another, we can be confident that each axis in the vector derived from our original representation is a measure of some significantly different property. We predict that this results in a clearer semantic division between methods than LDA or LSI, and better results when determining syntactic and semantic similarity. With this information, we show how ICA can be applied to the problem of locating clones in software, using a document-term matrix to represent the set of code fragments. The resulting signals can be used to transform the original representation of the code in a vector space into a new set of data based on the magnitude of the statistically
independent topics in each code fragment.

This chapter uses visualizations to understand how latent topic models can be compared to software clones. This comparison is important in understanding what it means for two documents to be similar to one another when represented by a latent topic model.

6.1 Method

To identify the most similar code blocks in our source data, we take the following steps:

- Construct a document-term matrix using the non-unique tokens found in our source code.
- Reduce the dimensionality of our matrix using PCA.
- Apply ICA to our reduced matrix, and save the results.
- Calculate the nearest neighbour scores of each method using the previous matrix.

The source package to be analysed is segmented by method (or another code fragment unit), and a list of the non-unique tokens used across the application is generated. We define non-unique tokens as those tokens that appear in more than one code fragment, and therefore can contribute to some correlation between methods. A document-term matrix is generated from the application methods and the list of non-unique tokens. Entries in the document-term matrix indicate how many times a token was observed in a given document.

A standardized z-score normalization as discussed in Section 2.3 is applied, and PCA is used for dimensionality reduction to reduce the memory footprint. After the
As an example, we process the $x_{rea}$ matrix from Section 4.2 using PCA and ICA. The contribution of each independent signal can be taken as the weight in an axis corresponding to some independent feature about the documents. Taking two principal components and performing ICA results in the following $A_{rea}$ signal distribution.

$$A_{rea} = \begin{bmatrix} -0.92 & -0.27 \\ -0.77 & 1.25 \\ -0.76 & -0.96 \\ 1.23 & -1.11 \\ 1.21 & 1.10 \end{bmatrix}$$
The cosine distance between each of the row vectors provides the similarity of each document to the others. In this case, we can show each source document and each of the nearest neighbours with their similarity scores.

The strings and their nearest neighbours can be seen in Figure 6.2. Each score is the cosine distance between the vectors. As ICA enforces a strong statistical bound on the axes, we expect to see vectors that are quite distinct from one another, and can be observed by the significantly different orientations of the vectors in $A_{\text{flea}}$.

The application of PCA and ICA is a way of remapping the original matrix into a new vector space, where each axis corresponds to some mathematically significant topic or feature found in the original matrix. Using ICA, instead of a technique like singular value decomposition, results in statistically independent components, and should result in axes that are more distinctive than those generated from components that are merely decorrelated. Another interesting way to consider the results is to map to three dimensions and to plot the results; we do this to visually identify clusters that may be clone groups, along with outliers that are unlikely to have similar blocks of code.

The meaning of these results is as follows. By applying ICA to the original document-term matrix generated from our input source code, we can examine the

<table>
<thead>
<tr>
<th></th>
<th>$s_1$</th>
<th>$s_2$</th>
<th>$s_3$</th>
<th>$s_4$</th>
<th>$s_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>1.0</td>
<td>0.26</td>
<td>0.82</td>
<td>-0.52</td>
<td>-0.90</td>
</tr>
<tr>
<td>$s_2$</td>
<td>0.26</td>
<td>1.0</td>
<td>-0.35</td>
<td>-0.96</td>
<td>0.18</td>
</tr>
<tr>
<td>$s_3$</td>
<td>0.82</td>
<td>-0.35</td>
<td>1.0</td>
<td>0.07</td>
<td>-0.99</td>
</tr>
<tr>
<td>$s_4$</td>
<td>-0.52</td>
<td>-0.96</td>
<td>0.07</td>
<td>1.0</td>
<td>0.10</td>
</tr>
<tr>
<td>$s_5$</td>
<td>-0.90</td>
<td>0.18</td>
<td>-0.99</td>
<td>0.10</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Figure 6.2: Similarity scores from $x_{\text{flea}}$. The most similar documents (0.82) are $s_1$ and $s_3$, “My dog has fleas” and “My ukulele has fleas”. The least similar documents (-0.99) are $s_3$ and $s_5$, “My ukulele has fleas” and “That dog ate all the turkey”.

The cosine distance between each of the row vectors provides the similarity of each document to the others. In this case, we can show each source document and each of the nearest neighbours with their similarity scores.

The strings and their nearest neighbours can be seen in Figure 6.2. Each score is the cosine distance between the vectors. As ICA enforces a strong statistical bound on the axes, we expect to see vectors that are quite distinct from one another, and can be observed by the significantly different orientations of the vectors in $A_{\text{flea}}$.

The application of PCA and ICA is a way of remapping the original matrix into a new vector space, where each axis corresponds to some mathematically significant topic or feature found in the original matrix. Using ICA, instead of a technique like singular value decomposition, results in statistically independent components, and should result in axes that are more distinctive than those generated from components that are merely decorrelated. Another interesting way to consider the results is to map to three dimensions and to plot the results; we do this to visually identify clusters that may be clone groups, along with outliers that are unlikely to have similar blocks of code.

The meaning of these results is as follows. By applying ICA to the original document-term matrix generated from our input source code, we can examine the
A matrix representing the contribution of each signal for the documents in a new vector space. The rows of $A$ can be plotted as vectors, and the cosine distance between two vectors can be interpreted as a measure of their similarity, since each axis in this new space corresponds to the strength of some statistically independent topic.

6.2 Results

To show the effectiveness of our approach, we look at how well it identifies clones in the Linux kernel directory (hereafter referred to as kernel). This source code is developed in C, and is nearly 40,000 SLOC. The source is segmented into 2731 individual methods, and was preprocessed to only include tokens greater than three characters in length. 9327 tokens were extracted, of which 6828 appeared more than once. The time needed to perform ICA and to generate our results for kernel is approximately one minute on a standard desktop machine.

If two methods are found to be similar, the vectors derived from applying ICA to our input matrix will be extremely close. Since we are treating each signal in the matrix derived from ICA as a topic, for whatever those topics are determined to be, the two methods share that similarity. In our tests, this has frequently (but not necessarily) shown itself by correlating to some subset of the code functionality as in other topic models such as LDA. For example, it may be the case that one signal has high values for methods that handle memory management. Each topic axis is determined automatically when ICA is applied, and is not seeded or predetermined by the user.

A score below a constant threshold is not necessarily a definite identifier of a clone, but there is a clear ordering in structure that can be observed when looking at the matches. A nearest neighbour score for a vector in the first percentile means that the code fragment represented by that vector is within the top one percent of potential
static unsigned long source_load (int cpu, int type)
    struct rq *rq = cpu_rq (cpu);
    unsigned long total = weighted_cpuload (cpu);
    if (type == 0) return total;
    return min (rq->cpu_load[type - 1], total);

static unsigned long target_load (int cpu, int type)
    struct rq *rq = cpu_rq (cpu);
    unsigned long total = weighted_cpuload (cpu);
    if (type == 0) return total;
    return max (rq->cpu_load[type - 1], total);

Figure 6.3: First Percentile Nearest Neighbour. This pair of methods is among the most similar pair identified by ICA, and would also be discovered by a software clone detector.

clone candidates.

A sample clone, as indicated by its nearest neighbour score, is the method source_load seen in Figure 6.3. When we have calculated the distances between the source_load vector and every other vector corresponding to methods in our source code, it can be seen that the vector for the target_load method is extremely close. Visually, these methods share a great deal of similarity, and the only differences are in the method name and in the usage of min or max.

Figure 6.4 is an interesting example, as there is a great deal of structural similarity, but several key semantic differences. In both cases, the entry variable is declared and assigned using create_proc_entry, a comparison is made to proc_fops, and the value 0 is returned. However, a different string value and file system mode is given, different constants are used, and the proc_fops comparison is made twice in ioresources_init. These methods are not clones of each other, and we have no reason to assume that a nearest neighbour score in the tenth percentile would indicate a clone in kernel. As a similar pair, they have been determined to be more alike than 90% of the other methods in the source code.

This is one of the interesting strengths of this method; it may not provide a boolean truth test for whether or not two blocks of code are clones of each other,
static int __init kallsyms_init(void)
struct proc_dir_entry *entry;
    entry = create_proc_entry
        ("kallsyms", 0444, NULL);
    if (entry) entry->proc_fops =
        &kallsyms_operations;
    return 0;

static int __init ioresources_init(void)
struct proc_dir_entry *entry;
    entry = create_proc_entry
        ("ioports", 0, NULL);
    if (entry) entry->proc_fops =
        &proc_ioports_operations;
    entry = create_proc_dir_entry
        ("iomem", 0, NULL);
    if (entry) entry->proc_fops =
        &proc_iomem_operations;
    return 0;

Figure 6.4: Tenth Percentile Nearest Neighbour. This pair of methods are less similar
than the pair in Figure 6.3, and would not be determined to be clones by any clone
detection techniques. However, they are similar pieces of code, and may require
similar maintenance.

but it can provide a reliable estimate on the likelihood that the methods are clones
relative to the rest of the source. With a boolean result, there is a necessary trade-off
between recall and precision. With an ordered list of results, the fuzzy nature of clone
detection may be improved.

Although it is not possible to give a magic number that acts as a cut-off value
when identifying clones by their nearest neighbour scores, for all inputs we can show
that a meaningful gap appears between a source vector and any vectors representing
code fragments that are not clones. In fact, for the kernel system, distances between
nearest neighbours that are clones compared to ones that are not clones will often be
a factor of a thousand smaller. If the distance between a first percentile clone match
is 0.001, the nearest non-clone neighbour in the vector space will often have a score
around 3.0 or 5.0. Again, these are relative scores, but non-clones are considerably
less similar than those that are considered potential clones.

By analysing the nearest neighbour score for each method, we can also identify
outliers that are far from other methods, and therefore bad candidates for clone matching. Although the nearest neighbour distance score scale varies between corpora, it is not unreasonable to say that the values that are farthest apart can be excluded outright. Common culprits for outliers that match poorly with other methods are large singular functions that arguably could benefit from being refactored. Overuse of tokens in a method will not result in high similarity scores to other methods; rather, the absence of tokens shared between two methods will adversely affect the distance score.

6.3 Comparing LDA and Software Clones Using Bluevis

NiCad, a loose acronym for Accurate Detection of Near-miss Intentional Clones [95, 97], is a tool that allows for the identification and extraction of the set of potential clones, the flexible pretty-printing, normalization and filtering of the potential clone set, the clustering of potential clones to minimize comparison cost, and the reporting of results as original source. To determine whether two potential clones really are clones of each other, the unique line count in each pair is used as a measure of similarity. In particular, the size-sensitive Unique Percentage of Items (UPI) is computed for each potential clone. If the UPI for both line sequences is zero or below a predefined UPI threshold, the pairs are considered to be clones.

We can use Bluevis to revisit the similarity between software clones and topic models. This visualization can be used to observe the relationship between the two domains by examining the most similar pairs identified in each technique. In the case of software clones, we use NiCad to identify the full set of clones in a software system for a given UPI threshold, and generate a Bluevis plot for that full set of relationships.
For an LDA model, we calculate the full set of similarity scores between all pairs of code fragments, and plot the top percentage of those relationships in Bluevis. This gives us an overview of where the most similar sections of the software system are found. If two different techniques produce Bluevis plots that look alike, these techniques uncover similar structure in the original data set.

Figure 6.5 shows three Bluevis plots, each generated using the httpd system introduced in Chapter 4 as input. On the left, the full set of software clones obtained using NiCad at UPI threshold 0.1 is plotted. In the center, the full set of software clones obtained using NiCad at UPI threshold 0.3 (recommended by the authors of NiCad as the highest useful UPI for useful clone detection) is plotted. On the right, the top 25,000 most similar pairs of code fragments identified using LDA is plotted.

From this we see that topic models and clone detection techniques are finding similar relationships between code fragments. To a certain degree, this is unsurprising; duplication of terms between code fragments is expected in software clones. What is most interesting is the relationship between NiCad clones extracted using a UPI threshold of 0.3 (center Bluevis plot in Figure 6.5). These are not exact clones, but
are instead software fragments whose source lines differ by 30%. The global similarity turns out to be similar to the top 25,000 most similar pairs identified using LDA with no knowledge or expectation of clone detection. What this indicates is that topic models like LDA or ICA are implicitly detecting clones, and therefore topic models have value as a clone detection technique.

6.4 Summary

ICA identifies similar methods in source code, without built-in knowledge about program language or syntax. By mapping the methods to vectors using a document-term matrix and applying ICA to extract the statistically independent components that correspond to the original dataset, we can use a distance metric to determine how similar the original methods are to each other. Further, this gives us a way to estimate the possibility that these methods might be clones of one another.

The size of the matrices determined has been problematic, and running ICA on a matrix with several hundreds of thousands of methods can be prohibitive. We are looking at ways to reduce the size of the data in a meaningful way to tackle the problem of extremely large sets of data.

Additionally, in this work we have focused on a raw distance metric rather than cosine similarity; looking at the vector space using the cosine distance might help isolate similar features found in methods, as opposed to the similarity of each document. We have used a similar approach previously to identify related methods.

In the next chapter, these visualizations are used in a case study of similarity detection with a domain-specific programming language. LDA models are generated from an original data set and a marked-up version, and the visualizations are used to explicitly demonstrate an improvement after mark-up.
Chapter 7

Case Study: Web Service Similarity Using LDA

This thesis approaches topic models as a set of techniques for predicting the potential co-maintenance of source code fragments. A related research topic addressed in this chapter is how analysing a topic model generated from service specifications written in a domain-specific language can be used to uncover web service similarity. To do this, we use the visualizations introduced in Chapter 5 to evaluate the topic models produced from different forms of data.

Web services are software components used to communicate over a network. These web services are often described using domain-specific languages, outlining the operations that are available, the type of messages that can be sent, and other information about the provider.

The structure of service descriptions written in the Web Service Description Language (WSDL) [21], or other such domain-specific languages, makes reading and understanding the descriptions difficult. Operations and messages are defined abstractly, and are not designed to be easily read by a human. This problem makes
it even more difficult to discover relationships between service operations when considering a large repository of web services. Latent topic models can be used to find these relationships, but without adaptation to the specifics of WSDL, they can produce irrelevant noise. Because of the sparsity of local syntax in WSDL operation descriptions, there are not enough tokens in them to support meaningful semantic conclusions.

In this chapter, we use a strategy for restructuring WSDL documents into a set of contextualized operations in an empirical analysis of a repository of web services showing that, by using these contextualized operations, we can find more meaningful relationships when performing Latent Dirichlet Allocation. We use a similarity metric to identify related web service operations based on the derived model, and show by example how significant improvements are made through contextualizing.

This chapter demonstrates how the visualizations show that latent topic models identify meaningful relationships between source code documents. Additionally, the visualizations are used to compare latent topic models generated from different data sets against one another, and to evaluate which model is better for the required task.

### 7.1 Motivation

Martin and Cordy [76] performed web service analysis with WSDL to discover similar web service operations using clone detection. The authors discovered that the organization of these descriptions makes it impossible to identify meaningful operation descriptions to compare. In general, analysis techniques assume that related components are grouped together; however, this is not the case with WSDL. Using whole WSDL descriptions is a viable option, but does not yield the desired granularity. It may be possible to tell which services are related, but how strongly they are related may remain unclear. To compare operations, the <operation> elements
could be extracted, but these contain little information other than the operation name and ignore other valuable information such as type definitions. This led the authors to develop a method to gather the relevant pieces of the operation and consolidate them into self-contained operation descriptions called WSCells (pronounced “wizzles”). The previous work showed that WSCells yield much more meaningful results for clone detection. In this chapter, we experiment with whether they yield a similar improvement for semantic analysis of web services using LDA.

To analyse web service descriptions, we would like to discover these relationships in WSDL operations. However, due to the sparsity of local syntax in WSDL operation descriptions, there are not enough tokens in them to support meaningful semantic conclusions. For these reasons, we believe that utilizing topic models as a way to identify the relationships between individual web services is an ideal way to show that WSCells add important semantic structure.

7.2 Background

7.2.1 Web Service Description Language (WSDL)

A WSDL description of a web service contains an interface definition of one or more operations provided by that service. However, each definition is broken up into pieces, and linked together in a chain. The pieces are organized into 5 main sections:

Types

The types section contains type definitions for exchanging data between a client and a service using the XML Schema Language (XSL). The schema may define complexTypes, which are essentially objects that contain other elements, or simpleTypes, which may be restrictions of primitive types, enumerated types, or patterns (among
others). Other elements may be declared as these types, thereby inheriting the elements contained within.

Messages

Messages define the data elements corresponding to the input, output and faults of each operation. They contain one or more parts, which may refer to elements defined in the types section. These parts can represent parameters of the operation, but often they simply refer to an element in the types section that contains the parameters.

Port Types

The port types section contains one or more operations that make up the web service. Each of these operations may contain an <input> or <output> element depending on what kind of communication takes place (e.g. request-response, notification, etc.). Operations may also contain any number of faults, indicating potential errors that may occur. These inputs, outputs and faults refer to messages defined elsewhere in the file.

Bindings

Bindings define a message format and protocol for a port type. Often this protocol is Simple Object Access Protocol (SOAP).

Services

The services section defines a group of ports, each of which defines an endpoint and specifies an address for a binding.
For the purposes of similarity analysis, we ignore the Bindings and Services sections because they contain information specific to the service and are not likely to contain anything useful for identifying relationships.

The description of an operation begins in the <portTypes> element where operations are listed in their own <operation> element. Each of these elements contain a number of <input>, <output> or <fault> elements that correspond to a <message> element defined somewhere else in the description. These contain <part> elements that can refer to other remote elements in the <types> element. The elements in the <types> element can also contain elements that have other types associated with them, which can contain more elements, and so on. The result is that operation descriptions are split into remote pieces and intermingled in the same description. This poses a problem for analysis techniques such as LDA, because operations cannot be easily separated into units.

Figure 7.1 provides one example for the “ReserveRoom” operation of a simple hotel reservation service, which takes a “Payment” element and a “Room” element as input and sends an acknowledgement back to the client. Simple <input>, <output> and <fault> elements are expanded to include the <message> elements to which they refer. Then, each <part> element of each message is expanded with the corresponding element in the types section. Finally, each element with a complexType is expanded to include its type definition. This continues recursively until only primitive types remain.

Figure 7.2 shows two unrelated web services that are incorrectly found to be similar when analysed without contextualization, labelled with the web service provider and line number where the operation was found. The stock quote operation and author operation share a SOAP interface, but are otherwise in completely different domains. The SOAP keyword is identified by the topic model with no contextualization, and treated as important enough to link these two operation together. With
contextualization, this problem is alleviated.

Figure 7.3 shows an example of a web service that is particularly well suited to contextualization. The basic operation holds almost no information about the type of data it is designed to handle. After contextualizing it, the operation contains a full set of currency data, similar rate types, and other information about how it can be configured.

### 7.3 Method

Previous work by Martin and Cordy identified a way to leverage existing code clone detection tools to find similarities in a web service repository, and found that the poor locality of WSDL operation descriptions made it difficult to extract a set of
Figure 7.2: Two unrelated web services that are incorrectly found to be similar when analysed without contextualization.

potential clones for comparison [76]. Clone detectors assume that related blocks of code are grouped together in a continuous sequence of lines; however, this is not true for WSDL where <operation> elements will contain references to elements in other parts of the description.

To solve this problem, Martin and Cordy [76] developed a method to give context to these bare operations by injecting the referenced elements into the operation itself. These contextualized operations are called Web Service Cells, or WSCells, because they are like the cells that make up a web service. To do this, a source transformation language called TXL [25] was used. The TXL program takes a single WSDL document, extracts the base operation (the <operation> element), and inserts the referenced elements into the element that references it. So for an <input> inside the operation, the corresponding <message> is found and inserted into it; for each <part> inside the <message>, the corresponding <element> is found and inserted; and so on, until there are no more elements left. For example, consider the WSCell, shown in Figure 7.1, for a “ReserveRoom” operation of a simple hotel reservation service. This operation takes a “Payment” object and a “Room” object as input and sends an acknowledgement or a fault in return. The WSCell includes all referenced elements from each section of the WSDL description inserted into the elements that...
Before contextualization:

```xml
<wsdl:operation name="GetBRAZIBOR">
  <wsdl:input message="tns:GetBRAZIBORSapIn"/>
  <wsdl:output message="tns:GetBRAZIBORSapOut"/>
</wsdl:operation>
```

After contextualization:

```xml
<operation name="GetBRAZIBOR">
  <input message="tns:GetBRAZIBORSapIn">
...  
  <element name="BRAZIBORTypes">
    <s:restriction base="s:string">
      <s:enumeration value="Overnight"/>
      <s:enumeration value="OneYear"/>
...  
  <output message="tns:GetBRAZIBORSapOut">
  ...
  <element name="Currency" type="tns:Currencies">
  <element name="Currencies">
    <s:restriction base="s:string">
      <s:enumeration value="USD"/>
      <s:enumeration value="AED"/>
      <s:enumeration value="AFA"/>
...  
  <element name="Date" type="s:string"/>
  <element name="Value" type="s:double"/>
  <element name="Text" type="s:string"/>
  <element name="Source" type="s:string"/>
  <element name="Description" type="tns:RateDescription">
    <element name="Type" type="tns:RateTypes">
    <element name="RateTypes">
      <s:restriction base="s:string">
        <s:enumeration value="FederalFunds"/>
        <s:enumeration value="FederalFundsTargetRate"/>
        <s:enumeration value="Libor1Month"/>
        <s:enumeration value="Libor2Month"/>
...  
```

Figure 7.3: A web service that is particularly well suited to contextualization.
Martin and Cordy’s research showed that WSCells showed a clear improvement in the identification of similar operations when using clone detection, and convinced them that the contextualization of WSDL operation descriptions can make it easier to find related operations using existing tools. With that in mind, we take a similar approach and apply it to topic models to show that conceptual relationships can be identified in a similar way.

We performed a comparison of the bare <operation> elements and the contextualized operations (WSCells) to see how well LDA was able to model meaningful relationships between them. In each case, we generated a model of the data, and for each WSDL operation, used the Hellinger similarity metric described in Section 2.5 to identify the other most similar operations. We then examined the list of the most similar operations for each individual web service to see if contextualizing provided a better set of related web services. We evaluated several topic counts empirically to determine an appropriate value. For this data set, we used a fixed value of 100 topics.

Our experimental data is a set of WSDL service description files with over 500 service descriptions containing over 7,500 operations from a wide variety of domains, obtained through a web services search engine by Seekda [4]. Our goal is to show that contextualizing web service operations provides a clear improvement in the ability of a topic model to identify related web services. We will use a global analysis to visualize overall structural improvements and large-scale sets of related features, and a local analysis to directly observe how actual recommendations can be improved for individual web service operations.
7.4 Analysis

We use the visualization tools presented in Chapter 5 to examine the effects of contextualization from both a global and local perspective.

7.4.1 Global Structure

Figure 7.4 shows the top 25,000 pairs as determined by the Hellinger similarity metric. Each image represents a single model generated from the web service data. On the left (a), the basic WSDL operations have been used to generate a model. On the right (b), WSCells are used. The bare operations in (a) show some simple structure, but otherwise show sparse relationships between relatively unrelated data. By observing less noise in the Bluevis plot in (b) and directly examining the type of relationships identified in the Bluevis structure, it can be observed that the WSCells are able to capture larger groups of related web services.

Horizontal lines indicate operations that are probably from the same service or service provider, and diagonal lines indicate potentially similar operations from other sources. The visualization is mirrored horizontally, and diagonal lines will show up as an X in the image (any connection between \( a \) and \( b \) will also result in a connection from \( b \) to \( a \)). Bright white lines appear where large collections of related web services overlap, which can indicate either strong local relationships in a single provider, or a strong indication that a set of web services are cloned by another provider. Many of the similar operations identified by using the simple WSDL operations as input are meaningless, and show up due to shared tokens like \( \text{get} \) or \( \text{SOAP} \). Some clear structure is discovered by the model in both cases.

The images on their own are not enough to suggest that the appearance of structure is a definitive proof that using contextualized operations allows topic models to identify related web services. To understand what type of information is being
Figure 7.4: Bluevis shows how non-contextualized operations (a) are not able to capture large groups of web services when compared to WSCells (b).

identified, we performed an analysis of the large global features uncovered by this visualization.

We will first examine the operations diagram (a) in detail. Most of the structure is formed by small blue lines that connect across web systems. Due to the splitting of compound tokens, general terms (this may include terms like load, application, order, and log) are indications for LDA to treat those operations as related. The majority of these solitary connections are between unrelated operations that happen to share a few tokens in either their name or in the input or output messages. An example can be seen in Figure 7.2. The two operations in question deal with completely
different domains, and are only related due to the shared use of the SOAP interface. Many of the other tokens are common to all operations (wsdl, operation, message, and do not provide value for the model. This motivates our use of contextualization; with additional context in the form of tokens, a topic model is able to identify more meaningful relationships.

In the center of visualization (a) in Figure 7.4 is a large X that shows a large block of related operations found in two separate sections of the list. Although these are not strict clones of each other, they are related by subject area, they are offered by the same web service (although through a separate interface), and that they all include the “parameterOrder” element to provide additional context. The parameter list has a greater influence on the results of the model due to the introduction of many new tokens. Many elements of a WSDL operation are optional, and without contextualization, these rare instances stand out greatly amidst other operations.

At least two more groups of related web services are found below the large X previously discussed, and the one immediately below stands out as being particularly important in the model. The smaller but brighter X immediately below is a collection of web services from the same provider; the upper block is entirely related to orders, and the bottom block is entirely related to users. The crossover between the blocks is due to the inclusion of fault tags in the operations. Relatively few of the operations include this information so, like the parameter list in the previous example, these tokens are a strong indication of similarity.

While it is true that the inclusion of parameter lists and fault tags can be an indication of similarity, these data are optional, and not commonly used in the examples we examined. The naive approach of using basic operations with topic models gives some context about the web services, but in general, the results are either trivial duplication of common tokens, or web services that are found through the same provider.
The visualization in (b) using contextualized operations as input to LDA shows a significant reduction in sparse random connections. When we investigate these more closely, it appears that the majority of the information shown by the visualization contains a relevant semantic structure. Several large fan-out points appear, indicating web services that have similar operations offered by other providers. Large horizontal blocks indicate clusters of related operations from the same web service. To better understand what the hot points in the visualization actually indicate, we take a closer look.

One large set of related operations connected by diagonal lines can be seen starting near the top, running down to the bottom, and finally joining just above the center point of the image. These services were detected by the previous model, and a close examination of the left image will show some faint lines running between those areas. However, they do not stand out amidst the other functions that appear to be related, and may be easily missed. When we examined these, it turned out that they were three related collections of operations to retrieve holiday dates. Each collection was for a different geographical region, and included operations like GetHalloween, GetBoxingDay, and GetGuyFawkesNight. Even with these terms being duplicated across the three collections, the naive approach to using raw WSDL operations did not uncover this association.

Another large section of related operations can be seen in the large X in the bottom half of the visualization. The majority of these are found in a cluster of operations that provide bank reference rates for lending. They have names like GetEURIBOR, GetBRAZIBOR, and GetMOSIBID, and intuitively share similarities. However, the operations are terse, and the input and output messages simply append SoapIn and SoapOut, as in the earlier examples from Figure 7.2. After contextualization, a portion of the new WSCell can be seen in Figure 7.3. A great deal of additional information about the applicable currencies, related rate types, and other financial
information is explicitly added to the description. These keywords are likely to be shared with other financial web services, and a topic model like LDA will be able to use these tokens to detect similarities between them.

Although the majority of related operations in this block are calls to get specific related rates (GetSIBOR, GetLIBOR, and so on), there are additional relationships detected from other areas of the collection. This indicates that LDA is able to identify web services from other providers when the topic model is able to draw on the explicit data added by contextualization; LDA is not able to make these connections with the raw WSDL operations. The goal of web service discovery across systems using topic models is made more achievable through contextualization.

### 7.4.2 Local Similarity

In this study, we focused on the list of the top related operations, with the assumption that a user who was interested in identifying related operations would want to see this data. We examine the local similarities using POCO, as described in Section 5.2.

Figure 7.5 shows the top results for a single operation (GetFathersDay, from the USHolidayDates service). This particular method is clustered in among the holiday web services previously discussed in Section 7.4.1. Even the clusters along the sides at the endpoints of the X can be seen in this view, where several dense areas of related operations that deal with holidays in different services are identified by the model. These operations all share a related naming convention, which would intuitively lead to the assumption that the bare WSDL operations would also be able to identify these related methods. In fact, when examining the top results for the same operation without contextualization, the list includes GetLogo and GetEURIBOR, two web services with no meaningful relationship to GetFathersDay. With a small number of meaningful tokens, the non-contextualized results simply do not have enough context
Figure 7.5: Using POCO for individual operation similarity (Adapted from [76]).

to allow LDA to form meaningful conceptual structure, and the results cannot be used for web service discovery in the way that WSCells can.

Figure 7.1 shows an examination of the results of contextualization for a single operation. In this example, the GetCEO operation from the xinsider web service was used. The upper table (a) shows the most related operations as discovered by the topic model with no contextualization. Only one of the operations seems plausible at first glance (the alternate GetCEO operation in xinsider, starting at line 1603). The lower table (b) shows the most related operations for GetCEO as discovered by
### (a) Similar WSDL Operations

<table>
<thead>
<tr>
<th>Operation Name</th>
<th>Service Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>getDVDSHops</td>
<td>seawiseservice (1340)</td>
</tr>
<tr>
<td>Book</td>
<td>K4TAirSell (240)</td>
</tr>
<tr>
<td>GetWebsites</td>
<td>KYWOrgData (584)</td>
</tr>
<tr>
<td>GetWeatherReport</td>
<td>usweather (55)</td>
</tr>
<tr>
<td>GetCEO</td>
<td>xinsider (1603)</td>
</tr>
<tr>
<td>GetEaster</td>
<td>USHolidayDates (934)</td>
</tr>
<tr>
<td>GetSports</td>
<td>livescoresservice (7293)</td>
</tr>
<tr>
<td>GetTURKIBOR</td>
<td>xinterbanks (5993)</td>
</tr>
<tr>
<td>GetBookTitles</td>
<td>BibleWebService (192)</td>
</tr>
<tr>
<td>GetData</td>
<td>DataParam (296)</td>
</tr>
</tbody>
</table>

### (b) Similar WSCells

<table>
<thead>
<tr>
<th>Operation Name</th>
<th>Service Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetIssuerOwnerships</td>
<td>xinsider (1700)</td>
</tr>
<tr>
<td>GetDirectors</td>
<td>xinsider (1593)</td>
</tr>
<tr>
<td>GetDirectors</td>
<td>xinsider (1685)</td>
</tr>
<tr>
<td>GetInsiderTransactions</td>
<td>xtibco (2474)</td>
</tr>
<tr>
<td>GetInsiders</td>
<td>xtibco (2387)</td>
</tr>
<tr>
<td>GetOfficerCompensations</td>
<td>xcompensation (509)</td>
</tr>
<tr>
<td>GetIssuerTransactions</td>
<td>xinsider (1643)</td>
</tr>
<tr>
<td>GetIssuerOwnership</td>
<td>xinsider (1608)</td>
</tr>
<tr>
<td>GetIssuerOwnership</td>
<td>xinsider (1710)</td>
</tr>
<tr>
<td>GetRoster</td>
<td>xinsider (1680)</td>
</tr>
</tbody>
</table>

Table 7.1: Similar operations before (upper) and after (lower) contextualizing the topic model with contextualization, which are more useful while still managing to span several related web services.

To show that this type of improvement is common, we expand our view to look at the most similar operation for other examples. In Figure 7.1, we show how the set of most similar operations is also greatly improved. This is an expansion of the kind of data we see from Figure 7.2. In some cases, such as the GetReservations operation, the improvement is clear. With non-contextualized operations, LDA suggests that the most similar operation is GetSOFIBOR. With WSCells, LDA instead
suggests GetRoomAvailabilityForDay. Some other cases are not as clear, as with the GetIssueData operation. The non-contextualized suggestion is word_cloud, and the WSCells suggestion is GetFlightData.

The data from Figure 7.1 helps to show that this approach can be used for identifying specific semantically related operations. We address a concern with similar operations being found in the same service in the contextualized example as a threat to validity, and note that filtering out operations from the same service still results in markedly better results than bare WSDL operations alone.

Figure 7.2 gives a side-by-side comparison of the most relevant operation as determined by LDA for the bare WSDL operations and the contextualized operations. The left column gives the operation, the service (WSDL file), and the starting line number, given to remove ambiguity if two web services share the same name. These ten examples are typical of the majority of operations affected by contextualizing.

7.5 Summary

Topic models are an automated way for effectively identifying conceptually related pieces of data. Although web service languages are not natively good input sources for these techniques due to their sparse descriptions, contextualizing the operations can give a topic model an ideal amount of information to derive these conceptual relationships. Demonstrating that this is true can be done through the use of local and global similarity visualizations such as POCO and Bluevis.

While this chapter dealt with the use of topic models to improve the source data, in the next chapter, input data is used to improve the information captured in a topic model. Specifically, information about the architectural structure is used in conjunction with similarity queries made to an LDA model to identify an appropriate number of topics for the model.
Table 7.2: Side-by-side comparison of the most relevant operation as determined by LDA for the bare WSDL operations and the contextualized operations (Adapted from [76])

<table>
<thead>
<tr>
<th>Operation</th>
<th>Most Similar WSCell</th>
<th>Most Similar WSDL Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListFinancials xfinancials (2508)</td>
<td>GetFinancialServicesItemList xfinancials (2548)</td>
<td>LanguagesList Articles (432)</td>
</tr>
<tr>
<td>ExportShipsAndCategories Export (319)</td>
<td>ExportItineraryAndSteps Export (324)</td>
<td>Search xscreener (1055)</td>
</tr>
<tr>
<td>GetIssueData xemerging (718)</td>
<td>GetFlightData FastTrack (438)</td>
<td>word_cloud taporwareServices (434)</td>
</tr>
<tr>
<td>GetWeatherReport usweather (41)</td>
<td>GetWeather globalweather (76)</td>
<td>GetIndices xquotes (1813)</td>
</tr>
<tr>
<td>GetAIDIBOR xinterbanks (5455)</td>
<td>GetTRLIBOR xinterbanks (5425)</td>
<td>GetCarriers blackbox (9303)</td>
</tr>
<tr>
<td>searchByIdentifier icontest (1240)</td>
<td>searchByNameAndAddress icontest (1225)</td>
<td>GetLastSecurityHeadlines xreleases (594)</td>
</tr>
<tr>
<td>ToolsAndHardwareBox WSAmazonBox (734)</td>
<td>KitchenAndHousewareBox WSAmazonBox (699)</td>
<td>ListRenditions DocMan (1350)</td>
</tr>
<tr>
<td>GetReservations holidayguide (467)</td>
<td>GetRoomAvailabilityForDay holidayguide (455)</td>
<td>GetSOFIBOR xinterbanks (5597)</td>
</tr>
<tr>
<td>GetOtherProductInfo blackbox (9207)</td>
<td>NextOtherProductPortion blackbox (9203)</td>
<td>GetParkingInfo blackbox (9251)</td>
</tr>
<tr>
<td>GetAllSplitsByExchange xglobalhistorical (1836)</td>
<td>GetAllCashDividendsByExchange xglobalhistorical (1851)</td>
<td>GetTeamLoyalties2 livescoresservice (6749)</td>
</tr>
</tbody>
</table>
Chapter 8

Tuning Latent Models

The previous chapters show approaches for identifying the nature of similarity and the quality of the relationships found by topic models. In this chapter, we consider parameter selection for generating models based on source code. In particular, a critical question in understanding the latent topics uncovered by factor analysis of text documents is how many topics should be sought. If too few are used, largely independent factors are forced together and separate concepts may be clustered as one. If too many are used, strongly related documents will be artificially partitioned, and single concepts may be split across topics, making them difficult to see. Rules for choosing an appropriate topic count to represent a corpus of natural language documents have been empirically determined [41, 43]. However, it is not at all clear that these same rules are appropriate for source code, and determining a topic count that most appropriately describes a set of source code documents is an open problem.

This problem is the subject of this chapter. In this chapter, we use Latent Dirichlet Allocation and experiment with a range of topic counts, using a heuristic based on a combination of cosine similarity and physical proximity in the source code as a measure of likely relationship. We observe that optimal topic counts emerge naturally using these measures, and use these observations to infer a general rule for predicting
the appropriate topic count for representing a source code system based on its external characteristics.

As shown in the previous chapter, models that more accurately capture relationships between the source code documents are much more valuable. This suggests that investing time in preprocessing and parameter selection is necessary to generate appropriate models of the data. With this in mind, we use information about the software system when tuning parameter values to build a model that can be used to identify co-maintenance relationships.

8.1 Background

The decision about how many topics to retain has been fairly subjective. For natural language, many authors propose somewhere in the range of 200 to 300 singular values using LSI [69, 67], and a recent study showed “islands of stability” around 300 to 500 topics for document sets in the millions, with performance degrading outside of that range [18]. Kuhn et al. suggests using a value of \((m \times n)^{0.2}\), noting that a smaller number of topics may be warranted for analysing software corpora because document count is fewer than typical natural-language corpora [53]. However, in their research, source code documents are classes, whereas for us documents are methods or functions, and so our topic counts will necessarily differ from theirs.

By using a pair of heuristics and a range of latent topics, we show that a good topic count estimate for describing a source code corpus can be obtained. Incrementing the topic count and testing the relevance within the model of the source code fragments associated with each topic showed a clear performance peak, and therefore what we predict to be the appropriate number of topics to use for this task. We generalize this approach and derive a simple equation that can be used to estimate appropriate topic counts for arbitrary software systems. Our derived results are similar to previously
observed empirical evidence.

8.2 Approach

The goal of this research is to identify a method for estimating the appropriate number of latent factors for a software system, preferably as a function of easily computed, large-scale properties. Previous chapters have focused on determining whether or not these latent topic models were able to identify relationships in source code documents, but did not focus on how many topics represent the data most appropriately. We do this by, first, clustering the methods of each software system into varying numbers of latent topics using LDA, and then assessing the appropriateness of each of the resulting models.

Since we have no ground truth for the “right” number of clusters, we must also develop measures to capture the intuitive idea of the “appropriate” number of clusters. We design two measures that estimate the relationships between code fragments. The first measures the extent to which the LDA clustering is internally consistent; the second the extent to which its clusters fit with external structure of the system being studied.

We want to assess, for each topic count choice, how well the LDA clustering has succeeded in placing similar code pieces in the same topic. To do this we use two measures.

8.2.1 Cosine similarity of co-clustered code

If the LDA clustering were perfect, then each row of the document-topic matrix would contain a single 1 in the column to which the document associated with that row was allocated, and zeroes everywhere else. However, the allocation of a document to
clusters will not always be so tight – there will be a maximum value in its row that is used to allocate it to a cluster, but the other clusters may also have substantial probabilities. In the worst case, the maximum may not be much larger than the next largest.

Instead, we measure the quality of the clustering by computing the cosine similarity, in the document-topic matrix, for pairs of documents. Each row of the matrix consists of the membership probabilities of each document in each of the clusters. Similarity of a pair of records could be measured by the Euclidean distance between their rows, but it is better instead to consider each row as a vector, and measure similarity of pairs of records by the angle between these two vectors – vectors that “point” in the same direction are similar, even though one might be much longer than the other. In other words, two records are similar if their probability vectors have similar patterns of high and low values, regardless of their absolute magnitudes.

The computation for cosine similarity is given by

$$\cos \theta = \frac{d_1 \cdot d_2}{||d_1||d_2||}$$

and its value ranges between +1 for vectors that point in the same direction, 0 for orthogonal vectors, and −1 for vectors that point in opposite directions.

As the topic count increases, it becomes inherently more likely that similar source-code fragments lie in the same topic, so this measure is broadly increasing as a function of the topic count. The rate at which it increases is the property of interest.

Most systems will have some outlier fragments that do not fit well into any of the clusters, so the measure may not reach 1. For example, code fragments like a main() method will have many neighbours, but they will be significantly different in their term use and so far away in cosine difference.

For each document, we consider the nearest neighbours, the code fragments that
Figure 8.1: Nearest neighbour measure. Code fragments $d_1$ and $d_2$ are qualitatively similar since their counts for topics 1 and 2 are proportional, although of different magnitudes. Their cosine distance is small. Fragment $d_3$ is significantly different from both of them.

are closest in the vector space by cosine similarity. These neighbours are determined by the model to be similar, and we would expect that similar neighbours to be those that perform similar functions or are likely to be co-maintained. To see if varying the number of nearest neighbours has an effect on parameter estimation, we look at four different values. In our study, we use $k_D$ to refer to the nearest neighbours for a document, and use values from the range 5, 10, 25, and 50.

In addition to the per-document nearest neighbours, we are also concerned with the documents that are most strongly related to a topic. For example, for topic $t$, what are the $k_T$ source code fragments that have the highest probability of being generated from that topic? Or, to put it another way, what are the $k_T$ source code fragments that best represent this topic?
CHAPTER 8. TUNING LATENT MODELS

/httpd-2.3.8/modules/aaa/mod_auth_basic.c
/httpd-2.3.8/modules/aaa/mod_auth_digest.c
...
/httpd-2.3.8/server/util_xml.c

Figure 8.2: Proximity measure. Two code fragments are likely to be conceptually related if they are found in the same file or folder. The source code files `mod_auth_basic.c` and `mod_auth_digest.c` are more likely to be conceptually related than `mod_auth_basic.c` and `util_xml.c`, or `mod_auth_digest.c` and `util_xml.c`, as they lie closer together in the package structure.

8.2.2 Code Locality

Our second measure is based on proximity in the source code structure. We assume that two code fragments are likely to be related if they are contained in the same region of the source-code tree. The underlying assumption is that developers structure code in a relatively ordered way, and so conceptually related code fragments will often be found together. Depending on the particular structure and size of the code, we use either files or directories as the relevant units of the source-code tree.

Some evidence exists to justify the assumption that developer choices of source-code tree organization reflect similarity of code. In our earlier research using information retrieval methods to locate clones [34], the results suggested a correlation between clones and latent topics. While it is not true that all clones are conceptually related, they are more likely to be conceptually related than not.

Roy and Cordy have analysed the proximity of clones to one another in a range of open-source applications and languages [97], and have observed that, in most cases, the majority of clones are found in the same source file or folder. Most larger systems exhibit this effect, but it does not always hold for all applications and languages. In particular, in recent work on cloning in Python, clones were observed to be more distributed across the application structure than in other languages [96]. Even in the worst case, where clones are evenly distributed across the application, the justification
only needs to rely on the fact that clones are more likely to be found near one another in the source-code tree than two random source code fragments.

We decided to use code locality instead of clones as a measure for two main reasons. Code locality provides a larger set of comparisons to make, as code clone pairs make up a small subset of the number of code fragment pairs. Also, source-code clones are likely to score highly in document-term models because, by definition, they already have a large percentage of tokens in common.

8.2.3 Combining Nearest Neighbours and Code Locality

Two unsupervised measures are provided for determining similarity between pairs of source code fragments. We combine the two measures to allow for an unsupervised method for evaluating the ability of a latent model to describe the latent relationships that we would like to uncover in the document set.

For each document, the $k_D$ nearest neighbours are determined using the cosine distance. From these $k_D$ neighbours, the fraction of the documents that exhibit code proximity is calculated. The average score over all documents is calculated, and referred to as the document proximity measure; the average number of source code fragments that exhibit code locality in the top $k_D$ nearest neighbours for each function in the software system.

For each topic, the $k_T$ source code fragments that are most strongly similar to it are found. These can be considered the documents that best represent the information uncovered by this latent topic. From these $k_T$ fragments, the fraction of the documents that exhibit code proximity is calculated by comparing each code fragment to the others and taking the count of pairs that are found in the same location in the file structure. The average score over all topics is calculated, and referred to as the topic proximity measure; the average number of source code fragments that exhibit
code locality in the top $k_T$ functions relevant to each topic in the latent model. For example, for $k_T = 5$ and four of the documents are in the same directory, the topic proximity score is $\binom{4}{2}$, or 6.

The nearest-neighbour score for the latent model describes how well the winner-take-all strategy for allocating records to clusters matches a more generic similarity measure. The proximity score describes how well the clustering generated from internal evidence of the source code agrees with the implicit external evidence that drives the arrangement of the source-code tree.

### 8.2.4 Experiments

Each source code package was segmented into individual methods or functions without comments using TXL [25], and tokenized into lower-case strings. This data was used as input for GibbsLDA++ on each run. Our Python script [33] was used to automate runs with varying numbers of topics and to calculate the measure scores for each clustering. The experiments involve a large parameter sweep over:

- A range of systems of varying sizes and purposes, and written in a range of languages (Table 8.1);
- A number of clusters, ranging from 50 up to the number where the topic proximity score begins to decrease in large systems (although we use a topic step of 25 in medium systems and 5 in small systems);
- Ranges of $k_D$ and $k_T$ of 5, 10, 25, and 50.

We show two examples of these measures, for the C functions of the source code of the PostgreSQL database system. Figure 8.3 shows how the document proximity scores vary as the topic count increases. As the topic count increases, it becomes
increasingly likely that the $k_D$ nearest neighbours of any code fragment will be in the same topic, although there is, as expected, a dependence on how large $k_D$ is. Notice that, for large $k_D = 50$ the curve turns down as the number of clusters increases, because the size of some clusters drops below 50.

Figure 8.4 shows, for same corpus, the topic proximity scores as the topic count increases. The maximum values are found at around 75 to 125 clusters, depending on the value of $k_T$, after which the topic proximity score begins to plummet, and the
8.3 Results

Table 8.2 provides a summary of our results. Each source-code corpus (set of methods/functions) is listed by name and description. The number of individual tokens (words) in the whole corpus is provided, and the number of code fragments (functions/methods) given as documents is listed to provide an idea of the size of the system. The LOC (lines of code) metric for our purposes is restricted to the non-trivial, non-empty lines. We strip out comments and white-space, and focus on the lines with interesting code. The cluster peak indicates the point at which the topic proximity score reaches a plateau, and so where the topic proximity measure suggests
Figure 8.4: PostgreSQL topic proximity scores. A clear peak emerges around 75 to 125 topics, suggesting that the appropriate number of latent topics lies in this range. That the latent factors best capture the relationships between the code fragments. Adding more topics after this point decreases the ability of the clustering to extract the latent relationships; similarly, choosing too few topics aggregates unrelated fragments too strongly.

In smaller systems, and those that use methodologies such as aspect-oriented programming, the locality of source code in files and folders may be reduced. Unsurprisingly, the measure does not perform as well for such systems. Table 8.3 lists a set of systems where clear peaks in the proximity scores were not present. Figure 8.5 shows the results in more details for one such system. The document proximity score remains usable, and from this, we can estimate how well the model identifies the latent structure. For example, if we know that, in our other examples, the proximity score peak was typically found just before the point at which the nearest neighbour
Table 8.2: Source code results where a clear proximity score peak emerges

<table>
<thead>
<tr>
<th>Corpus</th>
<th>Tokens</th>
<th>Code Fragments</th>
<th>LOC</th>
<th>Topic Peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>cook (C)</td>
<td>3992</td>
<td>1362</td>
<td>35k</td>
<td>75-100</td>
</tr>
<tr>
<td>snns (C)</td>
<td>9625</td>
<td>2213</td>
<td>63k</td>
<td>75-100</td>
</tr>
<tr>
<td>linuxkernel (C)</td>
<td>12379</td>
<td>3964</td>
<td>57k</td>
<td>100</td>
</tr>
<tr>
<td>postgresql (C)</td>
<td>16700</td>
<td>4689</td>
<td>111k</td>
<td>75-125</td>
</tr>
<tr>
<td>httppd (C)</td>
<td>20488</td>
<td>5758</td>
<td>124k</td>
<td>125</td>
</tr>
<tr>
<td>freebsd (C)</td>
<td>225139</td>
<td>53260</td>
<td>1311k</td>
<td>450</td>
</tr>
<tr>
<td>linux (C)</td>
<td>829195</td>
<td>256779</td>
<td>5050k</td>
<td>650</td>
</tr>
<tr>
<td>linq (C#)</td>
<td>1993</td>
<td>638</td>
<td>8k</td>
<td>100-125</td>
</tr>
<tr>
<td>nant (C#)</td>
<td>6133</td>
<td>2383</td>
<td>30k</td>
<td>150-175</td>
</tr>
<tr>
<td>rssbandit (C#)</td>
<td>10871</td>
<td>4587</td>
<td>68k</td>
<td>150-200</td>
</tr>
<tr>
<td>db4o (C#)</td>
<td>13658</td>
<td>13855</td>
<td>97k</td>
<td>200-225</td>
</tr>
<tr>
<td>jforum (Java)</td>
<td>5295</td>
<td>2437</td>
<td>21k</td>
<td>75</td>
</tr>
<tr>
<td>heritrix (Java)</td>
<td>10374</td>
<td>4762</td>
<td>43k</td>
<td>150</td>
</tr>
<tr>
<td>ofbiz (Java)</td>
<td>35931</td>
<td>14707</td>
<td>227k</td>
<td>250</td>
</tr>
<tr>
<td>derby (Java)</td>
<td>58623</td>
<td>32781</td>
<td>383k</td>
<td>250</td>
</tr>
<tr>
<td>hadoop (Java)</td>
<td>33265</td>
<td>21478</td>
<td>225k</td>
<td>300</td>
</tr>
<tr>
<td>plone (Python)</td>
<td>5590</td>
<td>1899</td>
<td>20k</td>
<td>125</td>
</tr>
<tr>
<td>django (Python)</td>
<td>14160</td>
<td>7084</td>
<td>55k</td>
<td>275</td>
</tr>
<tr>
<td>zope (Python)</td>
<td>46553</td>
<td>37101</td>
<td>453k</td>
<td>350</td>
</tr>
</tbody>
</table>

Table 8.3: Stabilization of topic proximity scores where no document proximity score peak emerges.

<table>
<thead>
<tr>
<th>Corpus</th>
<th>Tokens</th>
<th>Code Fragments</th>
<th>LOC</th>
<th>Topic Peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip (C)</td>
<td>940</td>
<td>117</td>
<td>4k</td>
<td>5-10</td>
</tr>
<tr>
<td>abyss (C)</td>
<td>641</td>
<td>148</td>
<td>2k</td>
<td>10-15</td>
</tr>
<tr>
<td>weltab (C)</td>
<td>736</td>
<td>123</td>
<td>10k</td>
<td>10-15</td>
</tr>
<tr>
<td>wget (C)</td>
<td>1520</td>
<td>219</td>
<td>7k</td>
<td>20-25</td>
</tr>
<tr>
<td>bison (C)</td>
<td>2024</td>
<td>315</td>
<td>9k</td>
<td>20-25</td>
</tr>
<tr>
<td>castle (C#)</td>
<td>14779</td>
<td>9530</td>
<td>88k</td>
<td>175-225</td>
</tr>
<tr>
<td>jhotdraw (Java)</td>
<td>3133</td>
<td>2536</td>
<td>16k</td>
<td>100-200</td>
</tr>
</tbody>
</table>

measure flattened, we can claim that the ideal proximity count should be around this number. This smoothing effect is similar to the noise reduction that occurs when the least relevant latent topics are stripped from the data set, and the primary relationships are retained [70]. This claim is also strengthened by recent work in the
clone detection community, demonstrating a clear relationship between proximity in the package structure and the likelihood of finding clones [96]. Using this, together with our observation that clones often share similar semantic information and are frequently identified as semantically related in latent models [34], we consider the proximity score a reasonable measure.

Thus the strategy for using the two measures is this: if the proximity score shows a clear peak, then the location of this peak is taken to indicate the appropriate number of clusters. When it does not show a peak, then the value for which the nearest neighbour score flattens is taken to be the appropriate number of clusters. Uncertainties about the point of flattening can be resolved by comparing the system to others of similar size that do exhibit a peak.

A majority of the source code datasets that we analysed showed a clear peak where the proximity score was maximized, although the peak varied in magnitude and location. Table 8.2 and Table 8.3 show that the appropriate number of topics is often slightly less than the commonly assumed default of 300 dimensions, for source-code packages of twenty thousand code fragments or less. The data hints at a fractional power growth in the best number of latent factors as the document count increases. As noted earlier, the topic counts also seem to be language dependent, so that it is not always true that code written in C would use the same number of topics for its best fitting model as code written in Python or C#.

The idea of the proximity score is not tied to only using proximity in the source-code tree. Other mechanisms that measure conceptual relationships between source-code fragments would serve equally well. The essential feature is that proximity determines code fragment similarity extrinsically, and so provides an objective measure against which to compare the clustering. As an example, consider a measure in which two source code fragments are treated as conceptually similar if they have
Figure 8.5: JHotDraw proximity scores. In the cases where there is no peak, the results are often an almost flat plot. In our sample data set, the nearest neighbour score plots always resembled Figure 8.3, even in the cases where no peak emerged. Graphs like this are almost exclusively found in packages with either poor or simple folder structure, or a small number of fragments.

shared a version control change list. A large number of false positives will be identified by this measure, and it may not cover the entire source code base. However, because this captures some of the conceptual similarities in the code, it can be used as a replacement for proximity score, and could therefore be used to evaluate the performance of a latent factor model.

To estimate appropriate topic counts for future research, we fitted a curve to the predicted best number of clusters across our results. This was done in Matlab, by taking the results from Table 8.2 and Table 8.3 and fitting a power curve to the data. We experimented with fitting three curves based on the code fragment count, the number of terms, and the non-trivial lines of code in each system.
Figure 8.6 shows the curve fitted to code fragments for all of the source-code packages we used. For a source-code system with \( x \) code fragments, \( t(x) \) provides an approximation of the appropriate number of topics to model the data. This suggests that, when systems are large enough, there is considerable latitude in choosing the right number of clusters, as the curve is flat over large size ranges. For smaller systems, the appropriate number of clusters is sensitive to system size, and more care is needed. Figure 8.8 provides the curves fitted to the tokens and the lines of code. We generalize this equation as follows:

\[
t(x) = 7.25 \times x^{0.365}
\] (8.1)

Equation 8.1 provides an estimate for the appropriate number of topics for source code using LDA based on our observations of several dozen software systems. The estimates given by Equation 8.1 are summarized in Table 8.4 along with the suggested topics of the measure-based approach. As expected, the equations suggest too few clusters for small systems, but are otherwise remarkably robust across a range of system sizes, and for systems written in different languages.

This equation can be compared to an estimate given by Kuhn et al. [53]. For an \( m \times n \) document-term matrix, where \( m \) is documents (classes, instead of methods or functions) and \( n \) is the term count over all documents, the authors suggest using a value of \((m \times n)^{0.2}\). While this exponent is less than the value obtained by our study, there are some strong similarities between the two equations. In the software systems we examined, the average term-to-document ratio suggests a linear relationship between the tokens and source code fragments. From this, we can approximate the equation by substituting \( n \) with \( m \) to get \((m^2)^{0.2}\), or \( m^{0.4}\). The exponent 0.4 suggests that as the document count increases, the appropriate rate of change between their topic count and our topic count is roughly similar.
8.4 Summary

We have presented a method for estimating the appropriate number of topics in a latent model by defining and using two measures. In many cases, this method identifies a clear peak in the proximity score, that we assume predicts the ideal number of latent topics needed to extract a hidden substructure that relates source-code fragments to one another.

These measures provide a way to estimate the most appropriate number of topics needed to describe a source-code corpus. This estimate can be used as a starting point for future studies using LDA to provide a reasonable suggestion for the number
Figure 8.7: A close-up of the curve fitted to the method results from Figure 8.6. The x-axis measures the source code fragment count. The y-axis measures how many topics our approach has estimated for the appropriate value.

Figure 8.8: Curves fitted to the token count and lines of code obtained from our data.
of topics that models conceptual structure appropriately. Our study used Latent Dirichlet Allocation, but the method is model-independent, and would be well suited to future studies with other techniques like Latent Semantic Indexing.

In the next chapter, we use the visualizations and techniques shown earlier in this thesis to explore using Independent Component Analysis as a software topic model.
Chapter 9

Topic Models Using ICA

In Chapter 2, statistical latent variable models for factor analysis were introduced. Applying these models to source code, as discussed in Chapter 3, can provide an unsupervised method for identifying related source code fragments. This chapter uses a model called Independent Component Analysis (ICA) discussed in Chapter 6 in relation to software clone detection as a topic model to identify related code fragments. As a blind-signal-separation technique, ICA identifies statistically independent signals in text, as opposed to ones that are just decorrelated. This restriction results in a topic model that identifies smaller subtle relationships in documents while still uncovering similar global structure to other models like LDA.

In this chapter, we show how ICA can be used to perform unsupervised source code topic identification, and compare it to LDA. To compare the clustering results between ICA and LDA, the revision history visualization introduced in Chapter 4 and the Bluevis global visualization introduced in Chapter 5 are used.

In the context of the thesis, this chapter uses the visualizations developed earlier in the thesis to compare and evaluate two different topic models. We examine how the co-maintenance history is captured using each model, and show that ICA is able to capture a similar but subtly different set of information.
9.1 Approach

Independent Component Analysis, discussed in Section 2.6, is a blind-signal-separation technique commonly applied to signal and image data. ICA is a blind approach because no prior knowledge about the source data is assumed. ICA can be considered as a linear transform of the original source data, and given by the equation $X = AS$. An original data matrix $X$ is factored 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 $k \times n$.

To treat ICA as a topic model, the $A$ and $S$ matrices are used to identify the topics and the relevance of the modelled documents to those topics. Each of the independent components in $S$ is treated as a statistically independent topic, while the weight of each signal in a document indicates the importance or magnitude of that topic.

The mixing matrix is used to determine a given signal’s relevance to each of the source code fragments. If there are $k$ signals extracted using ICA, each source code fragment’s row in the mixing matrix will have $k$ columns corresponding to each signal’s significance. The highest value can be interpreted as indicating the strongest presence of that signal, or the most relevant topic to that code fragment. As a topic model, the extracted signals are discussed as topics.

9.2 Implementation

The input source data is separated into individual methods or functions. From here, a list of the unique tokens used across the source code is generated, and a document-term matrix is created. Rows of this matrix correspond to functions, and values in
CHAPTER 9. TOPIC MODELS USING ICA

```
SIGNALS = 100;
[COEFF, SCORE, latent] = princomp(zscore(ica), 'econ');
[icasig, A, W] = fastica(SCORE(:, 1:SIGNALS)');

terms = W * COEFF(:, 1:SIGNALS)';
```

Figure 9.1: Matlab commands to obtain signals and relevant terms.

the columns are the number of observations of each token in the given code fragment.

Figure 9.1 shows the Matlab commands used to obtain the independent signals and the descriptive terms for each topic. The document-term matrix is normalized using the standardized z-score, where it is centered and scaled such that the columns of the matrix are zero mean and have a standard deviation of 1. ICA is applied to the normalized matrix using FastICA [46] to obtain the mixing and un-mixing matrices and the recovered signals.

Dimensionality reduction using principal component analysis or singular value decomposition is performed to explicitly define the signal count to reduce the memory footprint. ICA is a computationally expensive operation, which forces analysis to be limited to a relatively small number of signals.

Signals are described by terms in a similar way to the topics in LDA. Dimensionality reduction using PCA produces the principal component coefficients and the component scores for the documents. After the component scores in documents are used to generate the independent signals, the un-mixing matrix $W$ is used to transform the principal components into the ICA space. The columns with the greatest values in each row of $\texttt{terms}$ from Figure 9.1 identify the most significant terms for each topic. By analogy, the principal component coefficients represent the cluttered data that must be unmixed using the matrix identified by ICA to retrieve the characteristic terms for each independent topic.
9.3 Results

We first examine *cook*, a package developed in C spanning nearly 80K SLOC. The program consists of 1362 individual functions, preprocessed to retain tokens greater than three characters in length that appear multiple times. 3990 tokens were extracted, of which 3081 appeared more than once. PCA is used to reduce the signal count to 10. We use *cook* as an example because it has been used previously at the International Workshop on Detection of Software Clones [104], and has a clear structure from which we can check the accuracy of our results.

9.3.1 Topic Tokens

Figure 9.1 shows the token list corresponding to the 10 extracted topics from *cook*. These values represent the tokens that can be interpreted as being most relevant to the given topic. They are often, but not always, indicative of tokens that are used in functions that best fit this topic. For example, Topic 3 contains tokens like *string ty*, *str text*, *str free*, and *char*. When the function list that best matches this topic is examined, the top function names include *get prefix list*, *builtin expr parse lex*, and *os pathname*, which are all involved in manipulating strings.

While it is true that the top scoring tokens in a topic can often help identify the topic that has been discovered, it is not always the case. Topic 1, which is discussed in more detail below, is not immediately recognizable as a topic from the token list given in Figure 9.1. Some tokens may hint at what the topic is, but the set of tokens appears general. The combination of tokens is what identifies a function as fitting with a given topic, and not merely the presence of the high scoring tokens.

Figure 9.2 shows the token value distribution for Topic 1, which discovered a set of interpret functions that shared many similarities above. Some clearly defined peaks stand out. These correspond to tokens that are good indicators that a given function
### Table 9.1: ICA Topic Tokens for *cook*.
relates to the topic represented by this signal. Many of the token scores lie around zero, and do not provide much explanatory value about the topic.

9.3.2 Topic Functions

Figure 9.2 provides the top ten scoring functions for Topic 1. All functions deal with the interpreter in some way. One particularly interesting feature about the functions that are discovered is that they are not local to a single file, but rather span a specific subset of the codebase. Each of the functions that score highest is found within the *builtin* folder, and they share a great deal of similarity, from number and type of parameters, to their purpose. The functions share a similar topic, and can be extracted as a group.

Figure 9.3 shows the function scores for Topic 1. Our function extraction works in an orderly depth-first way, so data about functions that are close in the directory tree structure will be placed in rows that are close together in the matrix. The large bump in scores labelled A correlates to the *builtin* folder, referenced above. The primary functions are closely located in the code, which makes sense semantically.
CHAPTER 9. TOPIC MODELS USING ICA

Topic 1: Highest Scoring Functions

<table>
<thead>
<tr>
<th>Function Definition</th>
<th>File</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int match_interpret (result, args, pp, ocp)</td>
<td>cook/builtin/match.c</td>
<td>227.80</td>
</tr>
<tr>
<td>static int fromto_interpret (result, args, pp, ocp)</td>
<td>cook/builtin/match.c</td>
<td>227.52</td>
</tr>
<tr>
<td>static int interpret (result, args, pp, ocp)</td>
<td>cook/builtin/execute.c</td>
<td>223.57</td>
</tr>
<tr>
<td>static int interpret (result, args, pp, ocp)</td>
<td>cook/builtin/filter_out.c</td>
<td>222.45</td>
</tr>
<tr>
<td>static int match_mask_interpret (result, args, pp, ocp)</td>
<td>cook/builtin/match.c</td>
<td>221.87</td>
</tr>
<tr>
<td>static int script (result, args, pp, ocp)</td>
<td>cook/builtin/write.c</td>
<td>213.39</td>
</tr>
<tr>
<td>static int prepost_interpret (result, args, pp, ocp)</td>
<td>cook/builtin/text.c</td>
<td>212.23</td>
</tr>
<tr>
<td>static int interpret (result, args, pp, ocp)</td>
<td>cook/builtin/options.c</td>
<td>211.06</td>
</tr>
<tr>
<td>static int in_interpret (result, args, pp, ocp)</td>
<td>cook/builtin/boolean.c</td>
<td>210.70</td>
</tr>
<tr>
<td>static int or_interpret (result, args, pp, ocp)</td>
<td>cook/builtin/boolean.c</td>
<td>210.06</td>
</tr>
</tbody>
</table>

Table 9.2: Topic 1: Highest Scoring Functions. All of the functions most relevant to this topic are related to the interpreter in some way.

What is clear from the function list given in Table 9.2 is that the functions given are related in some way. In fact, by looking at the difference between the functions, we often get a rough gauge on the similarity of the functions as well. The only differences are a few single token changes, and a small difference in the assignment to the variable s. In the cases we have examined so far, functions that receive the same score are often Type 1 clones, and functions that score closely are often Type 2 or Type 3 clones. This is an unintended, but fortuitous side-effect of the technique, discussed in Chapter 6.

Topic 10 contained a wider range of values, and did not appear to define a single
9.4 Comparison to Latent Dirichlet Allocation

To compare ICA’s topics to the ones found by LDA, we return to the httpd software system used in Chapter 4. We keep the topic count small enough to examine the full set of topics for comparison, and look at 10 topics.

Figure 9.4 shows a sorted list of the number of documents assigned to each topic for LDA and ICA. Assigning a document to a topic is done by identifying the topic with the highest probability in LDA, or the dominant signal in ICA. As seen in the figure, there are no empty LDA topics; each topic has at least one document that has
been assigned to it. This is not true with ICA, and with httpd, approximately 10% of the ICA topics have no documents assigned to them. This indicates that there are some topics in ICA that have relatively low relevance to all documents, and are never the dominant topic in a document.

9.4.1 Revision History

Figures 9.5 and 9.6 show a snapshot of the revision history for httpd using LDA and ICA respectively.

The columns in the visualizations from Figures 9.5 and 9.6 have been ordered from left to right by the total number of methods in each column of the visualization. Specifically, a column’s weight in the sorting is based on the number of times any of the methods associated with that topic are modified in the program history. From
this, we can see that at a high level, each of the topic models is clustering in a similar way.

9.4.2 Topic Tokens

Table 9.3 provides token lists for the 10 topics identified from httpd using Latent Dirichlet Allocation. In nearly all cases, the tokens are small and general, and are often token fragments extracted from larger camel-case or underscored tokens like ServerSupportFunction or dav_method_bind. Program language keywords, including null, return, and char, are found in several topics. Type identifiers like char and apr_status_t are common, as are functions that operate on variables, like len.

Table 9.4 provides token lists for the 10 topics identified from httpd using Independent Component Analysis. In contrast to the token lists found in Table 9.3, the important tokens from the ICA topics are more specific, including larger compound
tokens. For example, topic 8 from LDA uses terms like *dav, err, resource, data*, and *lock*. ICA topic 10 is similar, and while it also begins with *resource, dav, and err* as the LDA topic does, it includes other terms like *dav.handle.err* and *dav.get.resource*.

LDA topic 3 is similar to ICA topics 8 and 9. In LDA, the important tokens include *apr, bucket, and brigade*. With ICA, the topics include *brigade* and *bucket* along with a collection of very specific tokens like *apr.bucket.brigade, apr.brigade.insert_tail*, and *apr.bucket.is_eos*. Also, LDA topic 8 is similar to ICA topic 10, with shared tokens including *dav, err, resource*. These also lead to compound tokens in the ICA topic including *dav.handle.err* and *dav.get.resource*.

### 9.4.3 Topic Functions

Table 9.5 provides the top ten scoring functions for the comparable LDA Topic 3 and ICA Topic 9. These two topics share some descriptive tokens, and additionally share
CHAPTER 9. TOPIC MODELS USING ICA

<table>
<thead>
<tr>
<th>Topic</th>
<th>Tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>apr abts mutex thread void int child status test suite</td>
</tr>
<tr>
<td>2</td>
<td>int case char break else size buf len match for</td>
</tr>
<tr>
<td>3</td>
<td>apr bucket ctxx brigade len table filter status return headers</td>
</tr>
<tr>
<td>4</td>
<td>apr time return sock socket str addr pollset ssl context</td>
</tr>
<tr>
<td>5</td>
<td>key apr hash null cache attr set dbm node void</td>
</tr>
<tr>
<td>6</td>
<td>char return const ptr xml case next name tok end</td>
</tr>
<tr>
<td>7</td>
<td>aplog log error server ssl mark aplog_mark ap_log_error null cache rec</td>
</tr>
<tr>
<td>8</td>
<td>dav null return err resource error data lock pool const</td>
</tr>
<tr>
<td>9</td>
<td>apr file pool success status apr_success return apr_status_t declare new</td>
</tr>
<tr>
<td>10</td>
<td>cmd null config conf char const pool dir return module</td>
</tr>
</tbody>
</table>

Table 9.3: 10 topics extracted from httpd using LDA. LDA tokens are general in most cases, including token fragments extracted from larger compound tokens.

similarity in the functions that are most relevant to each topic.

To place the tokens and functions from Table 9.5 in context, a brief description of the APR bucket brigade system is necessary. As described in the APR (Apache Portable Runtime) documentation [1], buckets are containers of data, and brigades are containers of buckets held in a ring structure. A bucket brigade is a doubly-linked list of buckets. A filter is applied to data that is sent or received by the server. Each time [an output] filter is invoked, it is passed a bucket brigade, containing a sequence of buckets which represent both data content and meta-data.

LDA topic 3 is described by general terms dealing with the APR bucket brigade implementation. ICA topic 8 is also described by terms dealing with this implementation, but instead by specific compound tokens.

Figure 9.7 shows the scores over documents in httpd for ICA topics 8 and 9. ICA topic 9 consists of mostly positive values, and has very strong weighting towards the APR bucket brigade functions. ICA topic 8 has similar values at the most positive range of the signal, but is strongly negative.

The similarity between the strongest tokens and functions in ICA topics 8 and 9 (Figure 9.7) is surprising at first, because of the expectation that signals will be
### Table 9.4: 10 topics extracted from httpd using ICA

<table>
<thead>
<tr>
<th>Topic</th>
<th>Tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>esc firstbyte brackets esc_p reqbyte repeat_type recno zerofirstbyte zeroreqbyte tempcode</td>
</tr>
<tr>
<td>2</td>
<td>filepath notaboveroot apr_filepath notaboveroot notrelative apr_filepath notrelative last_walk apr_lnk now_merged sec_idx sec_ent</td>
</tr>
<tr>
<td>3</td>
<td>dlg showwindow box g_h idc getdlgitem srestart sstop sstart hdc</td>
</tr>
<tr>
<td>4</td>
<td>offsets notempty bptr rre true_study_size use_size_offsets true_size use_offsets get_options byteflip</td>
</tr>
<tr>
<td>5</td>
<td>entity xml decl xml_error_no_memory notation declelementtype prologstate poolfinish poolstorestring handlerarg</td>
</tr>
<tr>
<td>6</td>
<td>nomatch match_match match_isgroup isgroup end_subject match_nomatch prop_chartype rrc minimize offset_top</td>
</tr>
<tr>
<td>7</td>
<td>serversupportfunction hse log_unsuooprted cchmatchingpath cch lpszpath error_invalid_parameter apr_set_os_error unsupported support</td>
</tr>
<tr>
<td>8</td>
<td>brigade apr_brigade_insert_tail apr_bucket_brigade apr_brigade_create bucket apr_bucket_tail apr_bucket_is_eos ap_pass_brigade apr_block_read</td>
</tr>
<tr>
<td>9</td>
<td>brigade apr_brigade_insert_tail bucket_alloc bucket_tail apr_brigade_create apr_bucket_brigade alloc ap_pass_brigade apr_bucket_is_eos</td>
</tr>
<tr>
<td>10</td>
<td>resource dav err dav_handle_err dav_push_error validate dav_validate_parent dav_validate_request dav_get_resource dav_auto_version_info</td>
</tr>
</tbody>
</table>

Like LDA, the important tokens from ICA topics are more specific compound tokens. However, by plotting the two signals together, it can be seen that the two are only similar at the most significant functions. ICA topics hold information at positive and negative ends of the signal.

Table 9.6 shows the tokens and functions at the negative end of the signal. While other topic models like LDA would suggest that lower values indicate less relevance to a topic, ICA is able to identify related functions using these values. In this case, the functions deal with the httpd Multi-Processing Modules, which are responsible for binding to network ports on the machine, accepting requests, and dispatching children to handle the requests [1]. It may be the case that ICA topics have a positive or negative orientation when looking at the relevant tokens and functions.
Table 9.5: Topic functions identified from httpd. LDA and ICA are able to identify similar topics, while being radically different approaches to topic modelling.

### 9.4.4 Bluevis

Figure 9.8 provides Bluevis plots for LDA and ICA using httpd. Although there are some strong similarities between the two plots, ICA is different in two ways. First, the ICA plot is slightly noisier, while retaining much of the structure found by LDA. Second, ICA is able to identify some relationships very strongly that are
not discovered by LDA. For example, much of the new information that shows up in the Bluevis plot relates to the memory management processes in \textit{httpd}. There is a very strong white band near the bottom the plot that consists of the \textit{apr/threadproc} systems for the \textit{beos}, \textit{netware}, \textit{os2}, \textit{unix}, and \textit{win32} platforms. ICA identifies these similar functions across platforms as being strongly related, and these relationships show up strongly in the Bluevis plot.

Figure 9.7: ICA topics 8 and 9 for \textit{httpd}. These topics have similar tokens at the positive end of the signal, but differ strongly in the negative values.
Figure 9.8: LDA and ICA Bluevis plots for *httpd*.
CHAPTER 9. TOPIC MODELS USING ICA

ICA Topic 8 (negative)

<table>
<thead>
<tr>
<th>Function</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>graceful</td>
<td>ap_server_conf</td>
</tr>
<tr>
<td>gracef</td>
<td>ap_my_generation</td>
</tr>
<tr>
<td>is_graceful_shutdown</td>
<td>pending</td>
</tr>
<tr>
<td>ap_mpm_run</td>
<td>prefork.c</td>
</tr>
<tr>
<td>ap_mpm_run</td>
<td>worker.c</td>
</tr>
<tr>
<td>ap_mpm_run</td>
<td>event.c</td>
</tr>
<tr>
<td>ap_mpm_run</td>
<td>beos.c</td>
</tr>
<tr>
<td>status_handler</td>
<td>mod_status.c</td>
</tr>
<tr>
<td>child_main</td>
<td>child.c</td>
</tr>
<tr>
<td>master_main</td>
<td>mpm_winnt.c</td>
</tr>
<tr>
<td>perform_idle_server_maintenance</td>
<td>event.c</td>
</tr>
<tr>
<td>perform_idle_server_maintenance</td>
<td>worker.c</td>
</tr>
<tr>
<td>ap_mpm_child_main</td>
<td>mpmt_os2_child.c</td>
</tr>
</tbody>
</table>

Table 9.6: ICA Topic 8 functions (negative) from httpd. It appears that negative values in ICA topics hold information in addition to the positive values.

9.5 Summary

By using Independent Component Analysis to focus on the statistically independent signals, it is possible to identify strongly-related threads of information used in the code. Our analysis has so far shown that the identification of topics using ICA identifies related functions that are not limited to proximity in the code, and at the most basic level, requires little preprocessing.

At some level, ICA is similar to LDA. The revision history visualizations show how all three approaches identify relationships in the software system. In addition, Bluevis shows how the overall structure between the two indicates that many of the same relationships between code fragments are identified in both ICA and LDA. However, ICA topics are not probabilities but weights, and positive and negative values are important for determining the characteristics of a topic. By using the visualizations introduced in this thesis, we have shown how ICA can be used as a software topic model.
Chapter 10

Conclusion

In this thesis, we demonstrated that latent topic models can be shown to identify co-maintenance relationships even with no supervision through a combination of visualization and software analysis. We introduced three distinct visualizations to explore how latent topic models captured co-maintenance and clustered related code fragments inside of observed changelists. Information about a software system was used to build better models for specific use-cases, and latent models were compared to one another using the set of tools developed for this task. With this research, we attempt to improve the quality of topic models generated using source code as input.

Chapter 1 provided a summary of the thesis, and an overview of the approach and layout. In Chapter 2, a summary of latent variables and latent variable models was provided, including descriptions of several common topic models based on latent variable models. Chapter 3 provided a survey of previous work using topic models for concept location and program comprehension tasks.

Chapter 4 used the topic clusters found by several topic models to identify a relationship with the co-maintenance history of a software project by mining the revision control history. A visualization was provided to show how topics are modified over time. Chapter 5 introduced two visualizations to view a local per-document view
of topic similarity and a global system-wide view. Chapter 6 looked at how the vector space model applies to software code clone discovery, a related research area.

In Chapter 7, we applied the visualizations developed earlier to examine how contextualization can improve the quality of topic models for domain-specific languages. Chapter 8 used knowledge about the architecture of a software system to find appropriate values for model parameters, including the topic count to use. We applied a latent variable model previously unused in program comprehension called Independent Component Analysis in Chapter 9, and showed how it is able to identify similar code fragments in software.

Individual latent topics extracted from source code are generally difficult to categorize. As collections of these latent topics, topic models can be hard to understand. This thesis has addressed this problem using several approaches including visualization, and attached the evaluation to software maintenance, an observable part of a software project. The thesis shows how maintenance history can be analysed using source control revision logs to show that topics relate co-maintained code fragments together.

10.1 Contributions

Topic visualization over changelists: The changelist is the atomic instance of change in a software project’s revision history, consisting of a list of modified files changed at the same time. We use these changelists with topic models to show how topics are changed over the revision history of a software project, and show how related code fragments are often clustered in changelists. This visualization motivates the thesis by showing that topic models are capturing information about code fragments that may be modified together, even when the topic model has no information about the previous revision history.
CHAPTER 10. CONCLUSION

Global and local similarity visualizations: A topic model extracts information about the way that documents relate to unobserved latent variables. We observe these relationships in different ways. Similarity at a global level is used to identify the nature of the most significant relationships identified by the model. This global similarity can also be used to compare two different topic models. Similarity at a local document level can be used to observe the most similar code fragments directly, and has the closest analogy to a tool used by a developer in software maintenance.

Relationship between topic models and clone detection: Although topic models are not designed to perform clone detection, each technique uses a token-based comparison to identify similarity between code fragments. Pairwise similarity measures in topic models, including the cosine distance for vector representation of documents, can be used as a fuzzy evaluation when determining if a pair of code fragments are clones.

Exploring similarity between web services: Web service descriptions authored in WSDL, a domain-specific language, are difficult for humans to understand. We applied topic models to similarity detection across web services using contextualization to explicitly include information that was implicitly referenced. The visualizations introduced in this thesis are used to show that contextualization improves topic model performance, and allows web services to be clustered better when compared to non-contextualized service descriptions.

Identifying an appropriate topic count for software: We use source code locality to identify an appropriate topic count to ensure that topic models capture relevant information about a software system. The locality heuristic ensures that a model is built with some knowledge about the architecture of a software project. By using a collection of open source systems, we derive an equation that is used to estimate an appropriate topic count for arbitrary software systems.
Introducing Independent Component Analysis as a topic model: Independent component analysis has been used as a blind signal separation technique, but has not yet seen application as a topic model. We use the extracted signals as topics, and use the un-mixing matrix to derive the related tokens and code fragments for each topic.

10.2 Threats to Validity

We have not judged the quality of the code submissions in the revision history of most projects. The projects we have examined so far have been stable open-source software packages, and we have not encountered any surprising or unusual transactions. Nevertheless, we expect that the majority of check-ins by experienced programmers are of a reasonable quality to use in this study.

Using pragmatically defined measures as an evaluation method is inherently problematic. That said, there is no good set of data to provide ground truth about conceptual relevance of one software fragment to another. Concept location has lacked a way of giving empirical evidence for the results, and using measures like the ones introduced in Chapter 8 appear to be valid first steps towards verifying these methods. Measures that provide more accurate results may exist, and further datasets can be created.

Although we have studied a number of open-source systems covering several different programming languages, we recognize a threat to external validity that must be addressed. Our survey of systems is not comprehensive, and any value obtained from Equation 8.1 must be considered an estimate.

One concern with Bluevis is the fact that the code fragments are not ordered in any significantly meaningful way. Ordering by architecture is reasonable, but does not indicate that neighbours in the ordering are necessarily related. In addition, it is
difficult to make firm conclusions about the information that Bluevis provides. For example, Bluevis cannot reliably say with certainty that a model is good or bad, or that two models are different. As an exploratory visualization, it can only be used to help understand a model, and not as a proof that a model is correct.

10.3 Future Work

Additional work with Independent Component Analysis: Our initial work with ICA was purely exploratory. It was not clear how ICA could be used with software in a way that was different from LDA. Now that we have used the model with some larger systems, and applied our visualizations to understand what ICA does differently from other topic models, we plan to perform more analysis.

It appears that ICA is better than LDA at finding small distinct features in a software system, while performing slightly worse overall at general clustering over the entire system. We would like to understand what feature characteristics lead to better detection using ICA, and if improved normalization in the pre-processing of the document-term matrix can also improve the final results of the model.

Software maintenance experiment: The conclusions made in this thesis lead to a natural experiment; how well do topic models predict future maintenance in software systems? We have already developed a tool to synchronize to a specified point in the revision history, to generate a model from the source, and to examine the successive changelists. However, some issues with the evaluation must be addressed. For example, the standard precision and recall metrics are not necessarily appropriate because it is uncommon for a single code fragment to be modified with each other related fragment.

Appropriate values for other parameters: We have used the results from Chapter 8 to identify appropriate topic count values for Latent Dirichlet Allocation.
In the study, we did not modify the alpha and beta hyperparameters. We would like to understand how modifying these values affects the utility of the model. In addition, we are interested in how well this approach generalizes to parameters in other models.
Bibliography


[59] Erik Linstead, Paul Rigor, Sushil Bajracharya, Cristina Lopes, and Pierre Baldi. Mining concepts from code with probabilistic topic models. In *Proceedings of the*


Appendix A

Mathematical Techniques

A brief overview of some basic mathematical background is necessary to accurately describe the methods used in this thesis. 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 non-singular matrix, and for a vector $\mathbf{x}$ and a matrix $A$ that defines a transformation, we can write $A\mathbf{x}$ to say that $A$ acts on $\mathbf{x}$ by left multiplication to produce a new vector called $A\mathbf{x}$ [55].

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

$$A\mathbf{x} = \lambda \mathbf{x} \quad (A.1)$$

As seen in Equation A.1, $\mathbf{x}$ does not have its direction changed after having the linear transformation defined in $A$ applied, but only a potential change in its magnitude. If $\mathbf{x}$ is a non-trivial, or non-zero solution of $A\mathbf{x} = \lambda \mathbf{x}$, $\mathbf{x}$ is referred to as
an eigenvector of $A$, and $\lambda$ is an eigenvalue of $A$ that corresponds to $x$. 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 $x$, the mean of the elements $\bar{x}$ is the sum of the values in the matrix divided by the number of elements. If $\bar{x}$ 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 vary with one another in some way, the covariance of the two variables will be non-zero. For two vectors $x$ and $y$ of length $n$, the covariance can be calculated using Equation A.2.

$$\text{cov}(x, y) = \frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y}) \quad (A.2)$$

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 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.
A.1 Principal Component Analysis

Principal Component Analysis (PCA) is a technique related to factor analysis (see Chapter 2.1.3), and linearly transforms an original set of observed 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.

PCA is commonly used as a method to identify sets of related information in data 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 measure on the overall importance of each axis. The axis with the greatest eigenvalue contains 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 each axis that is below a certain relevancy threshold. The goal is to uncover the most meaningful basis to express a set of data.

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 by the magnitude of the eigenvalues corresponding to those eigenvectors.

A.2 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 $\lambda_1$, then the corresponding
APPENDIX A. MATHEMATICAL TECHNIQUES

150

eigenvector $\mathbf{x}_1$ indicates the direction in which the stretching effect is greatest [55]. If we assume that $\mathbf{x}$ is a unit vector, then $||\mathbf{x}|| = 1$, and from Equation A.1,

$$||A\mathbf{x}|| = |\lambda|$$  \hspace{1cm} (A.3)

From Equation A.3, finding the vector $\mathbf{x}$ that maximizes $||A\mathbf{x}||$ with the constraint that $||\mathbf{x}|| = 1$ results in finding the eigenvector aligned along the direction that $A$ stretches most in its transformation. The absolute value $\lambda_1$ is the largest singular value of $A$, and is the length of the vector $A\mathbf{x}$. 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:

$$A = U \Sigma V^T$$  \hspace{1cm} (A.4)

In the decomposition in Equation A.4, $U$ is an $m \times n$ matrix with columns $u_1..n$ comprised of the left singular vectors, $\Sigma$ 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 $\Sigma$ 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 $\Sigma$ 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 often done to avoid processing extremely large matrices. As the diagonal elements of $\Sigma$ 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 $\Sigma$ 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.

$$A^r = \sum_{k=1}^{r} u_k s_k v_k^T$$  \hspace{1cm} (A.5)

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

$$A_k = U_k \Sigma_k V_k^T$$  \hspace{1cm} (A.6)