Estimating the Optimal Number of Latent Concepts in Source Code Analysis

Scott Grant
Queen's University
School of Computing
Kingston, Ontario
scott@cs.queensu.ca

James R. Cordy
Queen's University
School of Computing
Kingston, Ontario
cordy@cs.queensu.ca

Abstract

The optimal number of latent topics required to model the most accurate latent substructure for a source code corpus is an open question in source code analysis. Most estimates about the number of latent topics that exist in a software corpus are based on the assumption that the data is similar to natural language, but there is little empirical evidence to support this. In order to help determine the appropriate number of topics needed to accurately represent the source code, we generate a series of Latent Dirichlet Allocation models with varying topic counts. We use a heuristic to evaluate the ability of the model to identify related source code blocks, and demonstrate the consequences of choosing too few or too many latent topics.

1. Introduction

Concept and feature location techniques are designed to extract related subsets of program code in order to aid program comprehension. These location techniques whether supervised or not seek to identify related blocks of code, and aim to ease the difficult process of making sense of large code bases. This can remove a great deal of overhead when trying to understand a set of code, and can even work to prevent related methods from going unnoticed when developing an understanding of unfamiliar source code. The initial roots of concept and feature location can be traced back to program comprehension theories [8, 9, 28]. These early works attempted to determine how a programmer developed the comprehension necessary to debug, modify, or document code. From these, Biggerstaff identified the *concept assignment problem* [3, 4], and described it as the problem of discovering individual human-oriented concepts and assigning them to their implementation-oriented counterparts for a given a program.

One of the common questions asked by program comprehension relates to the number of concepts that most appropriately describes a set of source code. Statistical tests can demonstrate the amount of loss that occurs when approximating document sets by reducing the number of dimensions used to describe the data. Standard topic count metrics like the square root of the document size, or even the magic number 300, have all been proposed and used as accepted values [17]. While it is true that these are common in natural language analysis, it has not been definitively shown that source code and natural language text are similar enough to warrant the use of the same values. It is not clear whether or not source code analysis will require more or fewer topics for similar document set sizes. Having a way to verify results can demonstrate the relationship between natural language and source code, and how they map down to conceptual models.

In this paper, we use Latent Dirichlet Allocation (LDA) [5] as a statistical model to infer an appropriate number of latent topics needed to optimize the topic distributions over a set of source code methods. LDA is a generative statistical model that postulates a latent set of topics threaded through a set of documents. It assumes these documents have been generated due to the probability distribution over these topics, and that the words in the documents themselves are generated probabilistically in a similar manner. We segment a source code package into its individual methods, treat each method as a document, and generate successive LDA models with different values for the constant number of latent topics. Our metric is a simple heuristic based on location in the source code package structure that gives a quick and reasonable estimate about whether or not a pair of methods are conceptually related to one another.

The main contributions of this paper are:

- *A method for demonstrating an estimate of the optimal number of latent topics in source code.* The idea behind discovering latent topics is to find a substructure that associates the existing methods in a realistic way. Too few topics leads to over-generalization, and the inaccurate association of code blocks to one another

that are actually unrelated. Too many topics may fail to bring related sets of code together, and reduce the system to nothing better than a system with the same number of topics as there are methods.

- *A rough guideline for estimating good latent topic counts in source code based on experimental results.* It is possible to run an analysis before an experiment on each set of source code, but a goal would be to provide a reasonable way of estimating the number of topics required to describe the system without this testing. We perform our analysis over several code packages, and attempt to estimate the actual optimal number of topic counts found by experimentation. Again, we define the optimal number of topics as the topic count that best identifies pairwise relationships between conceptually related documents.

- *Verification and comparison of previous estimates for the latent topic counts in source code.* Some work has been done in this area already [16, 23], but it has unfortunately not provided many solid conclusions aside from a subjective evaluation. We demonstrate the validity of our results through a set of heuristics designed to give a reasonable estimate of the strength of each topic size in describing the conceptual substructure of the data.

This research can be used by the source code analysis and concept location communities to improve existing experiments, to evaluate the data garnered from existing results, and to choose appropriate topic counts for new experiments. A similar approach to examining the ideal latent topic count in natural language was taken by Kontostathis [15], but comparatively little analysis has been made in the software concept location community [1]. With this experiment, we hope to identify appropriate topic count values for source code data, allowing for more confident conclusions about program comprehension.

To clarify some terminology, we define some of the terms used in this paper. In concept location, the latent topics uncovered by information retrieval techniques are often considered concepts, and we use the terms *concept* and *latent topic* interchangeably. Also, information retrieval models discuss the relationships between documents in a corpus of text. For our purposes, the *documents* are source code methods, and the *corpus of text* is the total source code of the application. We also use those terms interchangeably, and any reference to a document can be considered to refer to a source code method. Information retrieval methods consider the *terms* used in documents, and for our purposes, terms are *tokens*. We consider the optimal number of concepts to be the one that best relates the documents to one another through the latent substructure of the model. If a certain topic value for the model gives the highest probability that two conceptually related documents are related in the model, and other topic values tend to give worse results overall, the original topic value is optimal.

2. Background

A latent variable model specifies the distribution of a set of random variables in which some additional variables are assumed to exist and be unobservable. The observable variables are referred to as manifest variables, and have been directly measured in some way. The unobservable variables are called latent variables, and are inferred somehow from the manifest variables. Latent variable models differ from traditional statistical models only in the sense that in addition to the observed data, we assume some hidden substructure to be present [2].

In the social sciences, latent variables are used to represent highly abstract concepts like intelligence, social class, power, and expectations. Economics uses the theory of latent variables aggressively, and considers concepts like quality of life, morale, and happiness as theoretical values that are hidden inside real data [6]. The analysis of source code and program documentation has been looking at latent variables for over a decade, and although there have been promising results, no real specific latent variables have been consistently identified. The reason for this absence is the difficulty in characterising a latent variable; coming up with an appropriate description for a concept that is strictly based on the presence or absence of tokens may be too vague, and the subtleties of the concept may be overlooked. For example, a concept that arises with the presence of the term "memory" may be initially judged as a set of memory management methods, but this interpretation is difficult to prove without a full audit of the results.

Latent variable models are primarily used for two main reasons. First, models derived from a large set of data may be too big to process in any meaningful way. Using a latent variable model to extract latent variables as new components can act as a dimensionality reduction technique, which can transform a large matrix into a smaller close representation of the data. Many of these latent models can even provide a value for the accuracy maintained in the new representation. For example, in the Singular Value Decomposition used in Latent Semantic Indexing, the singular values of the diagonal Σ matrix correspond to the importance of each dimension in describing the original data set. Second, extracting the latent variables can help to detect structure in the relationships between the manifest variables. Identifying correlations in this way can demonstrate information about the original data that may not have been immediately clear.

The fundamental premise behind a latent variable model

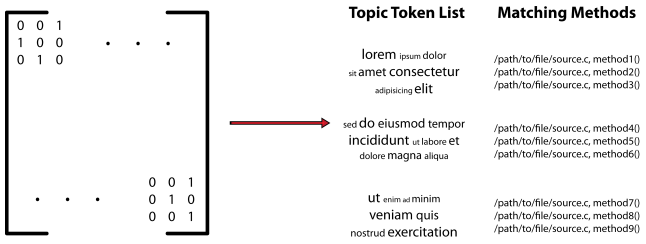

Figure 1. Converting a term-document matrix into a set of topics and related methods. We begin with a matrix or probabilistic representation of the original document set, abstracting away the actual tokens, and only considering the presence or absence of tokens. After a latent model is derived from this data, the collection of topics can be examined, and the most relevant documents for each topic can be identified.

is that there is some covariation observed among the manifest variables that can be explained by a mathematical relationship between them, and that these relationships can be extracted as latent variables. Figure 1 provides a brief overview of the process of moving from a term-document matrix into the desired topic list with related methods.

From a source code corpus (generally segmented into methods, but any source code block is acceptable), we can generate a term-document matrix M in which the rows relate to individual terms or tokens in the code, and the columns correspond to the segmented source methods. The value in position M_{ij} indicates the importance or presence of the ith term in document j. This term-document matrix is taken as input into a latent variable model, which identifies statistical relationships in the data. These relationships are often treated as latent concepts that relate relevant documents to one another. Additional techniques exist to determine which documents are most relevant given a latent variable, and the most relevant documents may be treated as the best fitting documents for a given concept. If, for example, it happened that a concept was extracted relating to the methods of the code dealing with the logging subsystem, we would expect the best fitting documents for that topic to all be fundamentally related to logging in some way.

Latent Dirichlet Allocation [5] is a generative statistical model of a document set in which a set of latent topics are assumed to determine the distribution of documents and tokens. It expands on other latent models like Latent Semantic Indexing by resting the model on a sounder statistical foundation, and making assumptions about the probability distributions that appear to provide more accurate conclusions. It is important to reiterate that although we use LDA as a latent model in this experiment, and the specific topic counts are tied to the performance of this particular model, any latent topic model can be substituted with this technique, and can be expected to provide similar results.

The motivation for this research stems from our earlier work using latent models to uncover relationships in source code. We examined Independent Component Analysis [10] as a way to identify latent substructure in source code, and attempted to demonstrate how it could be used as a way to identify related blocks of information [13, 14]. We also used latent models to identify clones, and showed that some variance in the syntactic data can be resolved through the latent semantic information identified by these models [12]. Although these results have been very promising, we were always curious as to how to choose the number of latent concepts that would best fit the data. Our attempts to test different values often led to subjective evaluations, or an inability to discover if one value was better than the others, and so we resolved to attempt to identify the appropriate topic count value for each source code corpus.

3. Approach

3.1. Overview

The goal of this research is to identify a method for obtaining an estimate of the optimal number of latent topics for source code analysis. Previous research has relied on determining whether or not these latent topic models were able to identify relationships in source code documents, but did not focus on the number of topics that best represents the data. We present two metrics that can be used as heuristics to estimate relationships in documents, and then combine the two, in order to use the size of the intersection of the document sets from the two heuristics as an estimate to which we can compare an LDA model's performance in concept location for a given topic count.

In the first heuristic, we use the vector space model [27] as a way of identifying related documents. If a document is represented by a vector of size k in a model with k latent topics, the most conceptually relevant documents to it are its nearest neighbours by cosine distance in the vector space. In the second heuristic, we identify a heuristic based on source code locality, and justify its use as a metric for conceptual relevance. We explain how the two metrics are combined to produce two scores on the overall ability of the model to represent the conceptual relationships in the source code, and iteratively increase the topic count in order to identify the optimal number of latent topics for a given source code corpus.

In the vector space model, each document is modelled as a vector in some high dimensional space, with each axis of the dimensional space representing some aspect of the fre-

quency of a term. Some information about the original set is lost, including the ordering of the documents, and the ordering of terms within each document. However, this mapping from documents to vectors allows for a new way to quantify the similarity between each other.

3.2. The Vector Space Neighbours

In order to accurately estimate the conceptual relationships obtained by the topic distributions over documents, we treat each document topic distribution as a vector, and use the cosine distance between two vectors as a measure of their conceptual similarity. Our vector space representation uses the topic distribution over each document, and so each document in a model with *k* topics is represented by a vector of *k* real numbers ranging from 0 to 1.

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

$$\cos\theta = \frac{d_1 \cdot d_2}{|d_1||d_2|} \tag{1}$$

For the purposes of this experiment, we consider document A to be conceptually related to document B if document A is one of the *n* nearest neighbours to document B. Our value for *n* is a variable measured in this experiment, and is discussed in further detail later; generally, *n* is a small number relative to the size of the document set. It is important to note that the fact that this relationship is not commutative, and if document A is conceptually related to document B, it is not necessarily true that document B is related to document A. This is best considered in the case of outlier methods like *main()*, who must by definition have some nearest neighbours, even if they are very far away in the vector space. We use *main()* as an outlier example due to the fact that it often makes calls to a large set of conceptually dissimilar helper methods, but does not rank highly in any particular conceptual topic itself.

Heuristic 1: *A method is likely to be conceptually related to its nearest neighbours in the vector space representation of the source code methods.*

We justify heuristic 1 by noting that this has been an assumption in latent variable models since the inception of the Vector Space Model [27], and has continued on in the concept location and program comprehension community [20, 21]. In practice, the nearest neighbour and clustering approaches are a common way to find related artifacts in a set, and we follow this approach. The basic premise of these models is to identify relationships between documents, and the vector space representation allows a way to determine how related the documents are in the conceptual space represented by the new basis.

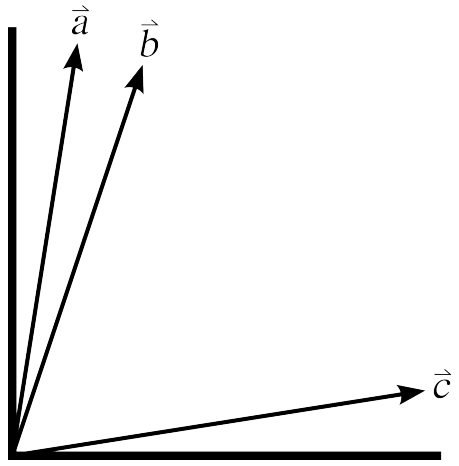

Figure 2. Heuristic 1: A method is likely to be conceptually related to its nearest neighbours in the vector space representation of the source code methods. Vectors a and b are more likely to be conceptually related than a and c or b and c, as they lie very close together in the vector space.

```
/my_code/foo/bar/file_a.c
/my_code/foo/bar/file_b.c
...
/my_code/bash/file_c.c
```

Figure 3. Heuristic 2: Two methods are likely to be conceptually related if they are found in the same file or folder. The source code files file_a.c and file_b.c are more likely to be conceptually related than file_a.c and file_c.c or file_b.c and file_c.c, as they lie very close together in the package structure.

3.3. The Package Structure Heuristic

Our second metric for automatically predicting conceptual similarity involves co-location in the source code structure. In the first case, we assume that a method is relevant

```
For k in the range of topic counts to evaluate:
    Generate an LDA model Mk with k topics
    For each document d in Mk:
        Get the m nearest neighbours by cosine distance
        Calculate the number of documents in m that lie in the same file/folder
        Call this the overall nearest neighbour score for Mk
    For each topic t in Mk:
        Get the n documents that have the highest probability of belonging to this topic
        Calculate the number of conceptually related documents for each document in n
        Call the sum of these relationships the overall topic score for Mk
```

Figure 4. Pseudocode for our process.

if it is contained in the same source code file as the source method. We assume that developers structure code in a relatively ordered way, and that in general, conceptually related methods will be found together. In the second case, we expand that assumption, and assume that one source code method is relevant to another if it is contained in the same directory. Depending on the particular structure and size of the code, we use one of these metrics as the second heuristic.

Heuristic 2: *Two methods are likely to be conceptually related if they are found in the same file or folder.*

We justify heuristic 2 by two methods. First, in our earlier research in using IR methods as a way to locate clones [12], the results pointed to a correlation between clones and concepts. While it is not true that all clones are conceptually related, they are more likely to be conceptually related than not. Second, Roy & Cordy have analyzed the proximity of clones to one another in a range of open source applications and languages [26], and have observed that in most cases the majority of clones are found in the same source file or folder. It should be noted however that while most systems exhibit this effect, it does not hold for all applications and languages. In particular, in their recent work on cloning in Python, clones were observed to be more distributed across the application structure than in other languages [25]. Even in the worst case where clones can be fairly 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 package structure than two random source code fragments.

3.4. Combining the Results

With the information outlined previously, we have outlined two unsupervised heuristics for determining whether or not a document is conceptually related to another document. We combine the two, in order 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. First, we choose a starting value, called k, for the number of latent topics we want to fit to the data. We then generate an LDA model with k latent topics, and examine the topic distributions for each document; the document's distribution over topics is its vector space representation. It is important to note that the k parameter is one value in a range of options that we want to test against. For example, we may choose k in [25, 50, 75, 100] for one system, to identify which of the four topic values fits the data best.

For each document, the n nearest neighbours are determined using the cosine distance. The number of documents in the nearest neighbours that are conceptually related using the code proximity heuristic is taken as a document score, and the average score over all documents is the *overall nearest neighbour score* for the latent model with k topics.

For each topic, the m documents with the highest probability of fitting this topic are identified, and these can be considered the documents that best represent the information uncovered by this latent topic. For each of the m documents that best match a topic, we determine how many of the other m documents are conceptually related using the code proximity heuristic, and take the sum of the scores as the *overall topic score* for the latent model with k topics.

These two results provide us with a way to estimate the evaluation of a topic model with respect to its ability to uncover the latent topics giving the best way of identifying conceptually related documents. The overall nearest neighbour score for the latent model describes how well the nearest neighbours of a document in the vector space relate conceptually. The overall topic score for the latent model describes how well the topics in the model are able to match up related sets of documents together with one another.

3.5. Implementation

We present experimental results comparing the nearest-neighbour scores of our models for varying topic counts. The Python script for our experimental framework is available from `http://research.cs.queensu.ca/home/scott/scam10/`, and the LDA implementa-

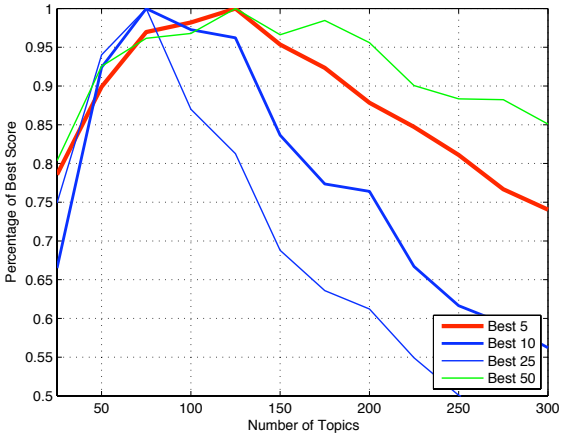

Figure 5. PostgreSQL overall topic scores. A clear peak emerges around 75 to 125 topics, suggesting that the optimal number of latent topics would sit in this range.

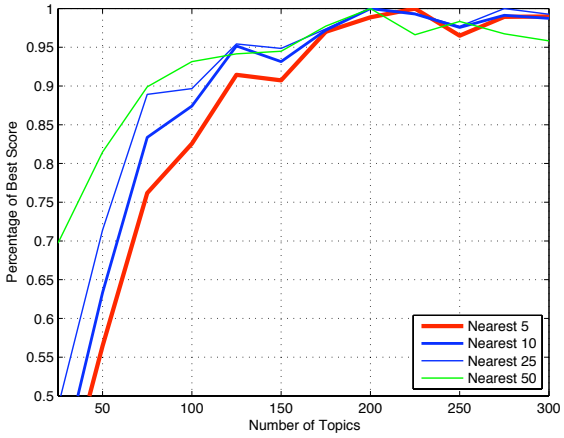

Figure 6. PostgreSQL overall nearest neighbour scores. An estimate on the amount of original accuracy retained with lower topic counts can be plotted.

tion we have used is the freely available GibbsLDA++ package [24]. The only parameter we sought to modify in GibbsLDA++ was the topic count, and so the tool was run using the defaults for variables like the α and β hyperparameters.

Each source code package was segmented into individual methods without comments using TXL [11], and tokenized into lower-case strings. This new data was used as input for GibbsLDA++ on each successive run. Our Python script was used to handle automating successive runs of latent topic sizes and calculating the overall scores for each model.

For each value k in the range of latent topic counts we wish to consider, we take the following action. We generate an LDA model of the data given the requirement that it limit the model to k latent topics. Each document has its vector space nearest neighbours calculated for a set of m values, and the average number of documents that pass both heuristics over the entire corpus is saved off. We used m = 5, 10, 25, 50 to see if varying the number of nearest neighbours affected the final evaluation of the model. This means that if 4 of the 5 nearest neighbours in the vector space were also found in the same folder in the package file structure, the document in question received a score of 80% for m = 5. For each topic in the model, the top m candidates that had the highest likelihood of being generated from that particular topic were examined. Again, we used the same set of values for m, in order to evaluate the model scope over larger numbers of documents. The number of conceptually related documents were calculated in each set of m documents for each topic, and the average was taken as the overall topic score for the model with k latent topics. An overview of the process in pseudocode is provided in Figure 4.

The *overall topic score* for a model is the clearest way to demonstrate the optimal number of topics for a source code corpus. When plotting the score for models that have various values of k, a clear peak emerges, identifying the spot in which topics contain the greatest number of methods that are co-located in the source code. Figure 5 shows an example plot for the PostgreSQL source code corpus. The maximum values are found somewhere around 75 to 125 topics, after which the overall topic score begins to plummet, and the topics lose their conceptual coherence.

The *overall nearest neighbour score* for a model demonstrates the amount of abstraction that occurs when a model with k topics is produced from the data. Figure 6 shows an example plot for the PostgreSQL source code corpus. Lower or higher values are not as important as evaluation scores for the entire model, but rather this number acts as a way to visualize the "smoothing" effect of a latent variable model when considering the information loss conceded by dimensionality reduction.

It is important to note that the asymptote of the overall nearest neighbour score graph is effectively the overall nearest neighbour score for a model with document vectors left in their original term-document matrix form. As the number of latent topics grows, the vectors lose the "smoothing" effect, and approach the original form. By comparing the peak of the overall topic score with the corresponding overall nearest neighbour score, we can show that the source code retains approximately 80% to 90% of the original information.

4. Results

4.1. Overview

Figure 7 provides a summary of our results when testing several source code packages. Each source code corpus is listed by name and description. The number of individual tokens in the corpus is provided, and the number of methods, given as documents, is listed to provide an idea of the size of the data set. The topic peak indicates the point at which the overall topic score reaches a plateau, and where heuristic 2 would suggest that the latent model best captures the relationships between the documents. Adding more topics after this point decreases the ability of the latent model to extract the latent relationships, and similarly, too few topics blurs the data too strongly.

4.2. Evaluation

A significant number of the source code datasets that we reviewed showed a clear peak in the overall topic count. The peak varied in magnitude and location, but when it was present, it appeared to be an indication on the topic count that best represented the number of latent topics necessary to optimize the second heuristic.

What can be observed from Figure 7 and Figure 8 is that the optimal number of topics is often slightly less than the typical default of 300 dimensions for source code packages of ten thousand methods or less. The data hints at a logarithmic growth in the number of latent topics as the document count increases. As noted earlier, the topic counts also seem to be relevant to the language, and it is not necessarily true that code written in C would require the same number of topics in its optimal model as code written in Python or C#. It is not clear whether or not a clear formula can be derived from this sample set, but in our experiments, for source code in a given language, the topic peaks did not diverge from one another too strongly.

There is a very real concern in assuming the co-location of source code that must be addressed. Our heuristics assume a structure that allows for enough nearest neighbours to register in the files and folders of the program structure. With smaller systems, and with methodologies like aspect-oriented programming, our heuristics do not accurately interpret the cross-cutting concepts found in the code. Small systems suffer noticeably, and in Figure 8, a set of systems that did not provide clear peaks in the overall topic scores are presented. A sample of this data can be visualized in Figure 9, in which no clear peak emerges. In lieu of this information, we can still analyse the overall nearest neighbour score for the point at which things begin to stabilize, and from this, we can make some estimates on how well the model detects the latent substructure. For example, if we know that in our other examples, the overall topic score peak was often found just before the point at which the overall nearest neighbour stabilizing point was reached, we can claim that the ideal topic count for this latent topic model should not exceed this stabilizing point. Our heuristic claim is strengthened by recent work in the clone detection community, demonstrating a clear relationship between proximity in the package structure and the likelihood of clones [25]. Using this information along with our observation that clones often share similar semantic information, and are frequently identified as semantically related in latent models [12], we believe that heuristic 2 is a reasonable metric.

Given that heuristic 2 does not work for all source code packages, it should be noted that this overall technique of comparing the vector space nearest neighbours with any other metric is a feasible way to estimate the overall topic score.

The fact that we use source code locality in heuristic 2 is not binding to the overall method of combining approaches in order to identify the overall topic peak. It is simply a means to estimating conceptual relationships between source code methods in an unsupervised way. All that our method requires in order to identify such a peak is a heuristic that is correlated to the actual conceptual similarity observed in the code. Given this correlation, a source code package large enough to support a latent topic model will see the heuristic identify enough true positive and negative results to identify the overall topic peak. As an example, consider a heuristic in which two source code methods are treated as conceptually similar if they have shared a version control change list. There will undoubtedly be a large number of false positives with this heuristic, and it is not guaranteed to be comprehensive over the entire source code base. If it is true that it captures some of the conceptual similarities in the code, it could be used as a replacement for heuristic 2, and could therefore be used to evaluate the performance of a latent topic model.

4.3. Summary of Results

We have presented a method for estimating the optimal number of topics in a latent model by using two heuristics. In many cases, this method identifies a clear peak in the overall topic score, and can be posited to predict the ideal number of latent topics needed to extract a hidden substructure that related documents to one another.

The optimal topic counts for small to medium sized source code packages appears to lie slightly below the standard count of 300. More importantly perhaps, the optimal topic counts tend to be achieved when the data is smoothed out to remove approximately 10% to 20% of the original raw information. This smoothing effect is related to the

Corpus	Location	Tokens	Methods	Topic Peak
cook (C)	http://miller.emu.id.au/pmiller/software/cook/	3992	1362	75-100
httpd (C)	http://httpd.apache.org/	20488	5758	125
linuxkernel (C)	http://www.linux.org/ kernel directory, v2.6.24.2	12379	3964	100
postgresql (C)	http://www.postgresql.org/	16700	4689	75-125
snns (C)	http://www.ra.cs.uni-tuebingen.de/SNNS/	9625	2213	75-100
db4o (C#)	http://www.db4o.com/ v7.4	13658	13855	200-225
linq (C#)	http://msdn.microsoft.com/en-us/netframework/aa904594.aspx	1993	638	100-125
nant (C#)	http://nant.sourceforge.net/ v0.86 beta 1	6133	2383	150-175
rssbandit (C#)	http://www.rssbandit.org/ v1.5.0.17	10871	4587	150-200
django (Python)	http://www.djangoproject.com/	14160	7084	275

Figure 7. Source code results where a clear overall topic score peak emerges.

Corpus	Location	Tokens	Methods	Topic Peak
abyss (C)	http://abyss.sourceforge.net/	641	148	10-15
bison (C)	http://www.gnu.org/software/bison/	2024	315	20-25
gzip (C)	http://www.gzip.org/ v1.2.4	940	117	5-10
weltab (C)	http://www.bauhaus-stuttgart.de/clones/	736	123	10-15
wget (C)	http://www.gnu.org/software/wget/	1520	219	20-25
castle (C#)	http://www.castleproject.org/	14779	9530	175-225
jhotdraw (Java)	http://www.jhotdraw.org/	3133	2536	100-200

Figure 8. Stabilization of overall nearest neighbour scores where no overall topic score peak emerges.

noise reduction that occurs when the least relevant latent topics are stripped from the data set, and the primary relationships are retained [22].

5. Related Work

The original paper describing the use of Latent Semantic Indexing (LSI) in program comprehension was tremendously influential, and led to a great deal of further research in the area. Maletic, Valluri, and Marcus [20, 21] began exploring LSI's potential in software by performing a handful of clustering and classification experiments against source code and documentation, and sought to determine LSI's ability to cluster groups of related code together. The early tests were promising, and suggested that even without a grammar or solutions to the problems of polysemy and synonymy, LSI could be used to support some aspects of the program understanding process.

The 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 [20, 21], and a recent study demonstrated "islands of stability" around 300 to 500 dimensions for documents sets in the millions, with degrading performance outside of that range [7]. Kuhn et al. suggest using a value of $(m \times n)^{0.2}$, and suggest that a smaller number of dimensions is warranted as the number of documents in their data set is smaller than most natural language corpora [16]. The results in this study and in the research provided above seem to suggest that this equation may be slightly below the optimal number of latent topics needed to identify the latent substructure in source code, although they treat classes as documents where we use methods.

The first application of LDA to source code was in 2007, when Linstead et al. began to use LDA to visualize the emergence of topics over several versions of a project [18, 19]. Shortly afterwards, in 2008, Maskeri et al. demonstrated its application to the extraction of business topics from source code [23]. Preliminary results indicated that some valid clustering was occurring, that topics were being identified, and interestingly that the number of topics for a large scale software system like Linux appeared to be just under 300.

Although LDA is relatively new, it is among the biggest topics researched in the data mining community with respect to document classification and conceptual analysis. A corresponding lack of interest in older techniques like LSI seems to indicate that newer approaches are producing better results, and are garnering more attention due to their suc-

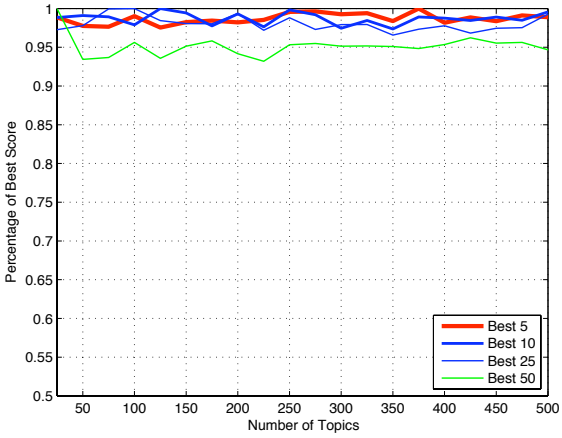

Figure 9. JHotDraw Overall Topic Scores. In the cases where no overall topic score emerges, the results are often a simple flat plot. It is important to note that in our sample data set, the overall nearest neighbour score plots always resembled Figure 6, even in the cases where no peak emerged. This indicates the loss of information as the number of topics decreases.

cesses. The concept location community would certainly benefit from further study of these techniques.

6. Threats to Validity

The use of heuristics as an evaluation method is inherently problematic, and may draw criticism. That said, there is no good set of data to evaluate the conceptual relevance scores. Concept location has lacked an unsupervised way of giving empirical evidence for the results, and we believe that the use of heuristics like this is a valid first step towards the verification of these methods. Certainly there may be better heuristics that provide more accurate results, and new datasets can be created. We discuss our continuing research towards producing data in this vein in the future work section of this paper.

Our approach also relies heavily on the ability of LDA to model the data appropriately, and in the topic distribution as a way of describing documents. The latent topic counts given in this research may not be directly applicable to all information retrieval techniques, and care should be taken when using other methods such as Latent Semantic Indexing or Independent Component Analysis. We do not believe that the numbers should be taken at face value, and instead point to the method as a way of evaluating performance in experiments using source code as a document corpus. The evaluation method described in this paper can be used with other latent class models, and may assist other researchers when setting up experiments.

7. Conclusions and Future Work

By using a pair of reasonable heuristics together with one another, and varying the number of latent topics extracted from a statistical model, we have demonstrated a way to determine a good estimate for the best number of concepts needed to describe a source code corpus. Our experiment used Latent Dirichlet Allocation, but our experimental method is model-independent, and would be well suited to future experiments with other techniques like Latent Semantic Indexing. By incrementing the number of topics and testing the relevance of methods associated with each topic, we show a clear peak in the ability of the model to identify related groups of code.

A clear point that seems to arise is that the number of topics used when modelling source code in a latent topic model must be carefully considered. In our example with the PostgreSQL code, choosing 200 topics instead of 100 may result in a model that only identifies a fraction of the desirable latent substructure. The choice of the number of latent topics for the model may be more important than previously considered, and choosing arbitrary values may sacrifice a significant amount of accuracy.

The value of retaining comments during the preprocessing and partitioning stage is unclear. Our process currently strips all comments from the source code before it is segmented into documents, and while this is a fairly common process in concept location, the exact value of comments is not known. We plan to perform additional tests to see how this affects our overall topic scores.

In order for us to reliably gauge the effectiveness of our own concept location techniques against the results of other methods, we need better data. That data does not exist now, and current evaluation techniques are not provably effective, although reasonable heuristics provide good estimates. With the aid of researchers in the area, along with a range of interested parties with some coding experience, we believe that a set of pairwise relevance comparisons can be collected, in order to generate test data to verify the results being collected from existing IR techniques. We have authored a tool to collect information about how users interpret conceptual similarity, and are generating a data set to use for evaluating results.

8. Acknowledgement

This work is supported in part by the Natural Sciences and Engineering Research Council of Canada, by the On-

tario Graduate Scholarship Program, and by an IBM CAS faculty award.

References

[1] P. F. Baldi, C. V. Lopes, E. J. Linstead, and S. K. Bajracharya. A theory of aspects as latent topics. *ACM SIGPLAN Notices*, 43(10):543–562, 2008.

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

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

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

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

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

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

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

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

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

[11] J. R. Cordy. The txl source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

[12] S. Grant and J. R. Cordy. Vector space analysis of software clones. In *17th IEEE International Conference on Program Comprehension (ICPC '09)*, pages 233–237, May 2009.

[13] S. Grant, J. R. Cordy, and D. B. Skillicorn. Automated concept location using independent component analysis. In *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE '08)*, pages 138–142, October 2008.

[14] S. Grant, D. B. Skillicorn, and J. R. Cordy. Topic detection using independent component analysis. In *Proceedings of the 2008 Workshop on Link Analysis, Counterterrorism and Security (LACTS '08)*, pages 23–28, April 2008.

[15] A. Kontostathis. Essential Dimensions of Latent Semantic Indexing (LSI). In *Proceedings of the 40th Annual Hawaii International Conference on System Sciences (HICSS '07)*, page 73, Washington, DC, USA, 2007. IEEE Computer Society.

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

[17] T. K. Landauer and S. T. Dumais. A solution to plato's problem: The latent semantic analysis theory of the acquisition, induction, and representation of knowledge. *Psychological Review*, 104:211–240, 1997.

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

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

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

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

[22] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, pages 125–135, Washington, DC, USA, 2003. IEEE Computer Society.

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

[24] X.-H. Phan and C.-T. Nguyen. GibbsLDA++, A C/C++ Implementation of Latent Dirichlet Allocation (LDA) using Gibbs Sampling for Parameter Estimation and Inference. `http://gibbslda.sourceforge.net`.

[25] C. K. Roy and J. R. Cordy. Are scripting languages really different? In *4th International Workshop on Software Clones (IWSC '10)*, May 2010.

[26] C. K. Roy and J. R. Cordy. Near-miss function clones in open source software: An empirical study. *Journal of Software Maintenance and Evolution, Special Issue on WCRE '08*, 2010.

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

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