Using Heuristics to Estimate an Appropriate Number of Latent Topics in Source Code Analysis

Scott Grant, James R. Cordy, David B. Skillicorn

Queen's University
School of Computing
Kingston, Ontario

Abstract

Latent Dirichlet Allocation (LDA) is a data clustering algorithm that performs especially well for text documents. In natural-language applications it automatically finds groups of related words (called "latent topics") and clusters the documents into sets that are about the same "topic". LDA has also been applied to source code, where the documents are natural source code units such as methods or classes, and the words are the keywords, operators, and programmer-defined names in the code. The problem of determining a topic count that most appropriately describes a set of source code documents is an open problem. We address this empirically by constructing clusterings with different numbers of topics for a large number of software systems, and then use a pair of measures based on source code locality and topic model similarity to assess how well the topic structure identifies related source code units. Results suggest that the topic count required can be closely approximated using the number of software code fragments in the system. We extend these results to recommend appropriate topic counts for arbitrary software systems based on an analysis of a set of open source systems.

Keywords: source code analysis, latent dirichlet allocation, latent topic model, code clusters

1. Introduction

Latent topic models can assist in understanding software systems by considering the set of words used in the source code fragments and identifying relationships between the fragments based on vocabulary. The vocabulary used in a software system's code is large and repetitive, making it difficult to associate code fragments by simple word search alone. Thus it is helpful to look for *latent factors* that capture deeper structure. These latent factors are unobserved, but through correlation between observable attributes (such as the variety and frequency of words used in a fragment), capture deeper structure in the system being studied [1]. For example, in the social sciences, latent factors can represent abstract concepts such as intelligence, social class, power, and expectations. In economics, latent factors correspond to concepts such as quality of life, morale, and happiness [2].

In natural language, the observations taken typically relate to word frequency. Instead of measuring attributes of a country, such as the GDP or average lifespan, we measure attributes about the document, such as word frequency. These word counts are provided as input to a statistical structure called a topic model, in which a "topic" describes some relationship between

parts of the data. Recently factor analysis using topic models has also been applied to software systems, using topic models to analyze units of source code such as methods or classes as text documents [3].

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 [4, 5]. However, it is not at all clear that these same rules are appropriate for source code, and the problem of 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 paper. In this paper, we use Latent Dirichlet Allocation (LDA) [6] as a statistical model to infer an appropriate number of latent topics needed to optimize the topic distributions over a set of source code fragments. Specifically, this work attempts to simplify the parameter selection problem using observable characteristics of the software system being modelled. LDA assumes a Dirichlet distribution on document generation from latent topics, and also on word generation within a document from latent topics. We 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, and observe that optimal topic counts emerge naturally using these measures. We use these observations to infer a general rule for predicting and tuning the appropriate topic count for representing a source code system based on its external characteristics when using LDA.

2. Background

A latent variable model specifies the distribution of a set of observed variables in which some additional variables are assumed to exist and be unobservable. Whereas the observable variables have been objectively measured in some way, the unobservable variables, called latent variables or latent factors, 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, some hidden substructure is assumed to be present [1].

The main advantage of latent factor 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 may be reduced to a much smaller set, making subsequent computations more practical. Therefore prediction and clustering can be done using the latent factors rather than the collected attributes with more robust results at less cost.

Some latent-factor models also rank the latent factors in order of importance, and provide an indication of how significant each one is in the global system. This makes it possible to truncate the representation even further. For example, singular value decomposition (sometimes known as latent semantic indexing in the context of information retrieval) provides an ordered set of singular values that have a natural interpretation as the importance of each associated latent factor.

Researchers have been looking for latent factors in source code and program documentation for over a decade [7]. Although it is, in a sense, straightforward to find some latent factors, it is

substantially more difficult to interpret them as software or programmer concerns, and validation is troublesome because of the lack of obvious ground truth.

Factor analysis based on matrix decompositions, using a matrix whose rows correspond to program units, and whose columns correspond to terms, have been tried in software settings. These include:

- *Singular value decomposition* (SVD), building on work in information retrieval in natural language;
- *Latent Dirichlet allocation* (LDA), used as a topic-modelling approach in natural language; and
- *Independent component analysis* (ICA), which is commonly used for factor analysis in many settings, but has not been widely applied to natural language.

The original paper describing the use of singular value decomposition (often called 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 [7, 8] began exploring LSI's potential in software by performing a handful of clustering and classification experiments for 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 topics to retain when performing a singular value decomposition of a text corpus has been fairly subjective. Many authors propose somewhere in the range of 200 to 300 topics [7, 8], 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 [9]. Kuhn *et al.* suggest using smaller topic values, noting that a smaller number of topics may be warranted for analyzing software corpora because the document count is smaller than typical natural-language corpora [10]. However, source code documents are classes in their research, whereas for us documents are methods or functions.

Latent Dirichlet Allocation [6] is a generative statistical model in which a set of latent topics are assumed to determine the distribution of documents and terms. SVD looks for components with uncorrelated variation and works for datasets in which the attributes are of any type. LDA assumes a different statistical foundation which fits more directly onto language settings. In particular it assumes that each document has a particular distribution of topics, and that each term is chosen by first choosing its topic and then choosing a particular word from that topic. The LDA algorithm then tries to estimate the topic distribution from the term use in the documents. A topic is therefore a related set of terms which, in turn, induces relationships on the documents. With LDA, the topic count is a user-defined parameter that must be provided before the model is generated. Algorithmically, if the document-term matrix is $n \times m$, and the LDA model is asked to find t topics, the result is an $n \times t$ matrix giving the membership probability of each document in each topic, and a $t \times m$ matrix giving the membership probability of each term in each topic.

While the data reflects a bag-of-words model, the frequencies of words in documents are not independent, and order matters. LDA allows this more subtle information to be partly taken into account, for example because it reflects the fact that the presence of one word in the next slot of a document prevents other words from occupying that slot.

The first application of LDA to source code was in 2007, when Linstead *et al.* began to use LDA to visualize topic emergence over several versions of a project [11, 12]. Shortly afterwards, in 2008, Maskeri *et al.* showed its application to the extraction of business topics from source code [13]. 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.

Thomas *et al.* [14] investigated how topic models can be used to describe the evolution of source code with a study over 12 releases of a well documented system. In their study, they were able to show that topic evolution coincides with observable changes in the source code, and that topic models can be used to understand the development and design history of a project. This research led them to the *Diff* model [15], an extension of an existing evolution model, taking software corpora into consideration by noting that most documents are not altered between versions and most changes are small. In this way, the authors imply that there is a notable difference when using source code instead of natural language as input documents to a topic model.

The role of topics in software development has been examined in detail by Hindle *et al.*. To gain an understanding of the current state of a project, windowed topic analysis [16] was proposed, in which the model of a software system is generated using information from a small window of time. Many topics that are local to a small portion of the project's life turn out to be relevant and interesting. Additionally, some topics recur over the life of a project. The authors later demonstrated [17] how many of these topics can be named using an automated approach.

Lukins *et al.* [18] used LDA and Latent Semantic Indexing (LSI), a well-known information retrieval technique, 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.

Requirements traceability is a challenging problem that involves describing and following a requirement in forward and backward directions [19]. 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 [20]. 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 [21] expanded the analysis to include Latent Semantic Indexing as a querying model, and determined that, when recovering links between source code and documentation, this model performed at least as well as the probabilistic or VSM methods.

Although LDA has been shown to be effective in modelling software source code, it is not as applicable to the traceability problem. In fact, when compared to simpler models such as the Vector Space Model, assuming no latent topics and simply observing similarity between tokens in each document, studies such as those performed by Oliveto *et al.* [22] indicate that the Vector Space Model outperforms LDA. This result is inconsistent, as a study by Asuncion *et al.* [23] indicates that on some systems, LDA seems to perform traceability recovery as well as or better than LSI.

LDA is widely used in the data-mining community for document classification and conceptual analysis. A corresponding lack of interest in older techniques like LSI seems to indicate

that newer approaches are producing better results, and are garnering more attention due to their successes.

It is important, although difficult, to choose appropriate parameter values for models. With that in mind, optimal parameter selection has been an active area of research. Support vector machines are statistical classifiers that depend on good parameter selection to be effective. Schölkopf *et al.* [24] introduce a new algorithm called ν-SVM that eliminates one parameter from standard models, but must also be carefully selected. Steinwart [25] builds on this work to obtain good estimates of the ν parameter. The impact of the Dirichlet hyperparameters used in LDA was examined by Wallach *et al.* [26], leading to a conclusion that LDA can be optimized by modifying the structure of these parameters to handle stop words more appropriately and improve topic consistency.

The motivation for the research described here stems from our earlier work using latent factor models to uncover relationships in source code. We experimented with Independent Component Analysis [27] as a way to identify latent structure in source code, and attempted to show how it could be used as a way to identify related blocks of information [28, 29]. 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 [30]. Although these results have been promising, we remained curious about how to choose how many latent factors are necessary to 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, which led us to investigate the relationship in its full generality.

We began by using a preliminary version of the approach developed in this paper to examine 17 small to medium sized software systems [31]. By combining a pair of heuristics and using a range of latent topics, we showed that a good topic count estimate for describing a source code corpus could be obtained. Incrementing the number of topics 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 believe to be the appropriate number of topics to use for this task. This paper expands on our work by introducing new systems, including some large ones such as the entire Linux system. We also generalize the 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.

3. 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 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 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.

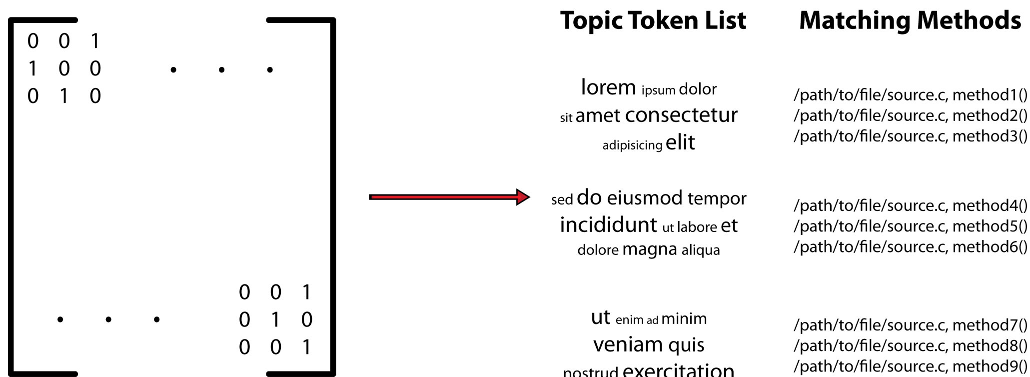

Figure 1: Converting a document-term matrix into a set of topics and related code fragments. We begin with a matrix or probabilistic representation of the original document set, abstracting away the tokens, and only considering the presence or absence of tokens. After a latent model is derived from this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

Given a source-code corpus (generally segmented into methods or functions, but any source code fragment is acceptable), a simple parse extracts the terms of interest from each document. In our case, these are programming-language keywords, and programmer-defined names. Since many programmer-defined names have internal structure, we separate such names into their component pieces if they have been built in one of the standard ways (for example, breaking at underscores) and count the entire name and all of its sub-pieces as terms. For example, the term *get_attribute* would be represented by three terms: *get*, *attribute*, and *get_attribute*.

These data are assembled into a document-term matrix, each row of which corresponds to a document (method, function), each column corresponds to a term, and where the element at (i, j) in the matrix is the frequency of occurrence of term j in document i. The row count of the matrix, n, is the document count, and the column count, m, is how many distinct terms exist across the entire corpus.

LDA is applied to this document-term matrix. Like most clustering algorithms, the desired number of topics or clusters, t, is an input to the algorithm. The result is an $n \times t$ matrix whose element at (i, j) represents the probability that document i is in topic j, and a $t \times m$ matrix whose element at (i, j) is the probability that term j participates in topic i. This is therefore a soft clustering of documents into topics, and words into topics; if necessary, this can be converted to a hard clustering by allocating each document and term to the topic in which it has the greatest probability.

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.

3.1. 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 zeros everywhere else. However, the allocation of a document to clusters will not always be so tight – there will a maximum value in its row that is used to allocate it to a cluster, but the other entries may also have substantial probabilities. In the worst case, the maximum may not be much larger than the next largest. A simple measure of cluster quality across the entire set of

documents is to compute the average probability of the winning cluster for each document. This is problematic because, while it would be 1 in a perfect clustering, it is not clear how large to expect it to be in an imperfect but realistic one.

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, to 0 for orthogonal vectors, to -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. To ensure that we consider only the nearest neighbours in the model evaluation, we would like to restrict the number of neighbours that we look at to only small values. It may be sufficient to only look at the top-ten nearest neighbours. However, in our study, we use four different values to see if varying the number of nearest neighbours has an effect on parameter estimation.

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.

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 neighbors, but they will tend to be significantly different in their term use and so far away in cosine difference.

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?

The values k_D and k_T are constants, so we consider the same number of similar code fragments for each document and topic.

3.2. 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

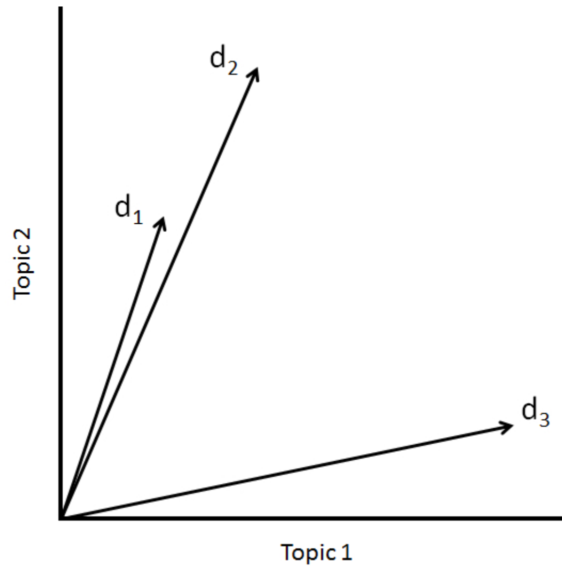

Figure 2: Nearest neighbour measure: code fragments d_1 and d_2 are qualitatively similar since their counts for topics 1 and 2 are proportional, although of different magnitudes. Their cosine distance is small. Fragment d_3 is significantly different from each of them.

```
/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 3: 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.

to locate clones [30], 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 analyzed the proximity of clones to one another in a range of open-source applications and languages [32], 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 [33]. 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 total 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.

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.

3.3. Combining Nearest Neighbours and Code Locality

Two unsupervised measures are provided for determining whether or not a source code fragment is related to another source code fragment. 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 related to it are found, and 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. 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.

3.4. Experiments

A Python script for our framework is available from the website [34]. The LDA implementation used is the freely available GibbsLDA++ package [35]. The only parameter modified in GibbsLDA++ was the topic count.

Each source code package was segmented into individual methods or functions without comments using TXL [36], and tokenized into lower-case strings. We retain all other keywords, including those that are programming-language specific. Although these keywords are likely to be found in a majority of the input data, they are similar to common words in natural language, and do not necessarily affect the results. This data was used as input for GibbsLDA++ on each run. Our Python script 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 1);
- A number of clusters, ranging from 50 up to the number of clusters 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 4 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 neighbors of any code fragment will be in the same topic, although there is, as expected,

Corpus	Location	Code Fragments
abyss (C)	http://abyss.sourceforge.net/	148
bison (C)	http://www.gnu.org/software/bison/	315
cook (C)	http://miller.emu.id.au/pmiller/software/cook/	1362
freebsd (C)	http://www.freebsd.org/	53260
gzip (C)	http://www.gzip.org/ v1.2.4	117
httpd (C)	http://httpd.apache.org/	5758
linux (C)	http://www.linux.org/ 2.6.37	256779
linuxkernel (C)	*linux*, *kernel* directory	3964
postgresql (C)	http://www.postgresql.org/	4689
weltab (C)	http://www.bauhaus-stuttgart.de/clones/	123
wget (C)	http://www.gnu.org/software/wget/	219
snns (C)	http://www.ra.cs.uni-tuebingen.de/SNNS/	2213
castle (C#)	http://www.castleproject.org/	9530
db4o (C#)	http://www.db4o.com/ v7.4	13855
linq (C#)	http://msdn.microsoft.com/	638
nant (C#)	http://nant.sourceforge.net/ v0.86 beta 1	2383
rssbandit (C#)	http://www.rssbandit.org/ v1.5.0.17	4587
django (Python)	http://www.djangoproject.com/	7084
plone (Python)	http://plone.org/	1899
zope (Python)	http://www.zope.org/	37101
derby (Java)	http://db.apache.org/derby/	32781
hadoop (Java)	http://hadoop.apache.org/	21478
heritrix (Java)	http://crawler.archive.org/	4762
jforum (Java)	http://jforum.net/	2437
jhotdraw (Java)	http://www.jhotdraw.org/	2536
ofbiz (Java)	http://ofbiz.apache.org/	14707

Table 1: A list of the source code packages used in this study, and where to obtain them.

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 5 shows, for the 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 topics lose their coherence. Note that the dependence on k_T is not monotonic.

4. Results

Table 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 whitespace, 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 that the latent factors best capture the relationships between the code fragments. Adding more topics after this

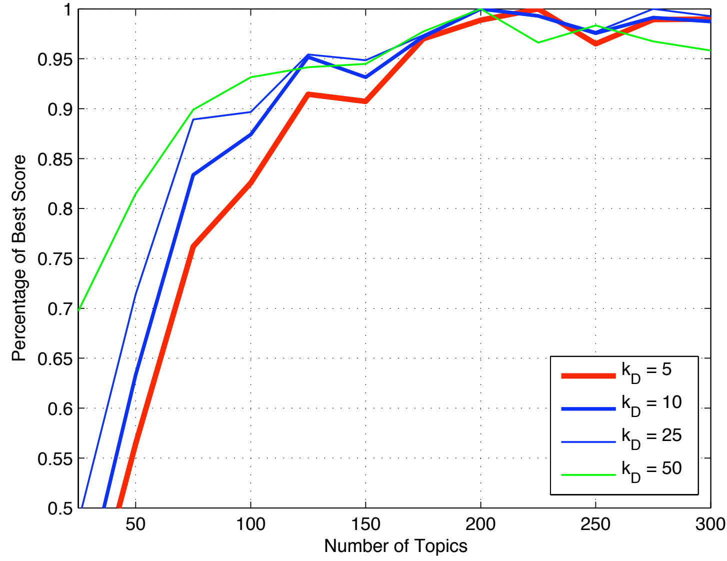

Figure 4: PostgreSQL document proximity scores. An estimate on the amount of original accuracy retained with lower topic counts can be plotted.

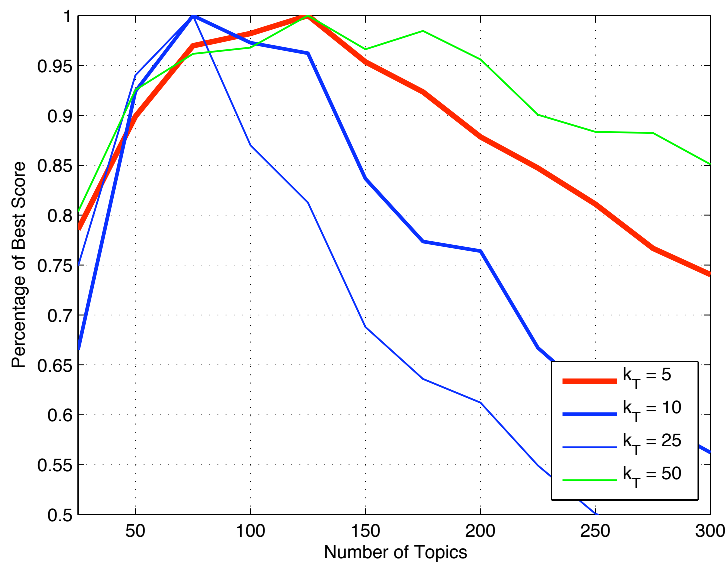

Figure 5: PostgreSQL topic proximity scores. A clear peak emerges around 75 to 125 topics, suggesting that the appropriate number of latent topics lies in this range.

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

Table 2: Source code results where a clear proximity score peak emerges.

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

Table 3: Stabilization of topic proximity scores where no document proximity score peak emerges.

point decreases the ability of the clustering to extract the latent relationships; similarly, choosing too few topics aggregates unrelated fragments too strongly.

Table 3 shows the results for systems where the topic proximity score did not show a clear peak.

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 3 lists a set of systems where clear peaks in the proximity scores were not present. Figure 6 shows the results in more details for one such system. The nearest neighbor score remains useable, 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 neighbor measure flattened, we can claim that the ideal proximity count should be around this number. This claim is strengthened by recent work in the clone detection community, demonstrating

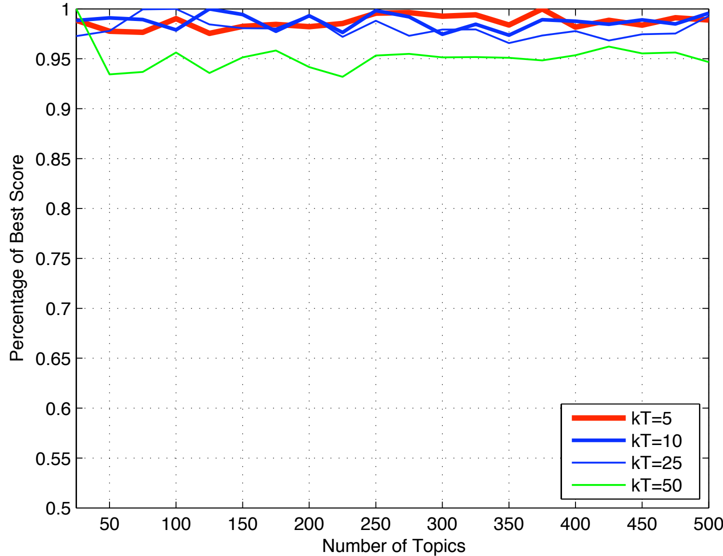

Figure 6: JHotDraw proximity scores. In the cases where there is no peak, the results are often an almost flat plot. In our sample data set, the nearest neighbour score plots always resembled Figure 4, even in the cases where no peak emerged. Graphs like this are almost exclusively found in packages with either poor or simple folder structure, or a small number of fragments.

a clear relationship between proximity in the package structure and the likelihood of finding clones [33]. Using this, together with our observation that clones often share similar semantic information and are frequently identified as semantically related in latent models [30], we believe that the proximity score is 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 neighbor 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 analyzed showed a clear peak where the proximity score was maximized, although the peak varied in magnitude and location. From Table 2 and Table 3, 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 bes 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 it determines code fragment similarity extrinsically, and so provides an objective measure against which to compare the clustering. As an

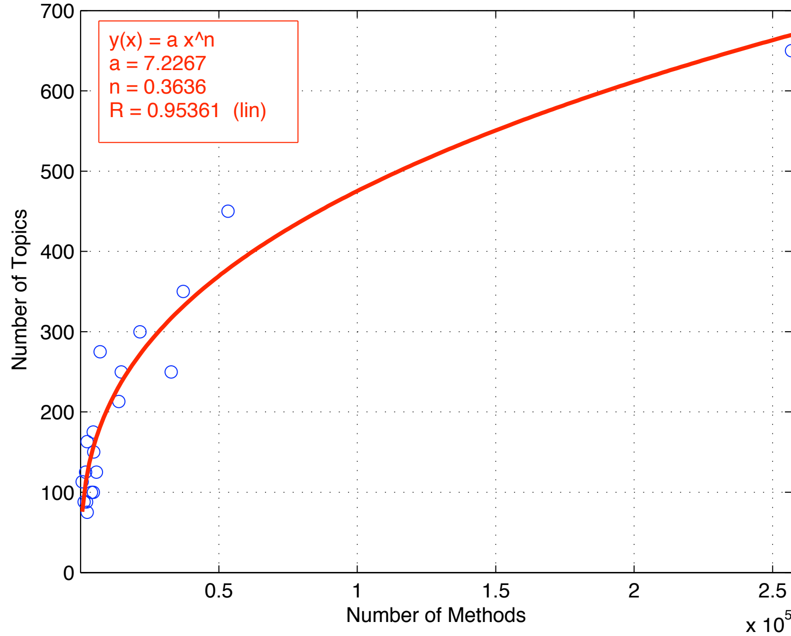

Figure 7: The curve fitted to the results. The x-axis measures the number of source code fragments. The y-axis measures the number of topics that our approach has estimated for the appropriate value.

example, consider a measure in which two source code fragments are treated as conceptually similar if they have 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 2 and Table 3 and fitting a power curve to the data. We experimented with fitting three curves based on: the number of code fragments, the counts of the number of terms, and the non-trivial lines of code in each system.

Figure 7 shows the curve fitted to the number of code fragments for all of the source-code packages we used. The fitted curve is given by:

$$t(x) = 7.2267 * x^{0.3636} \tag{1}$$

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 9 provides the curves fitted to the tokens and the lines of code.

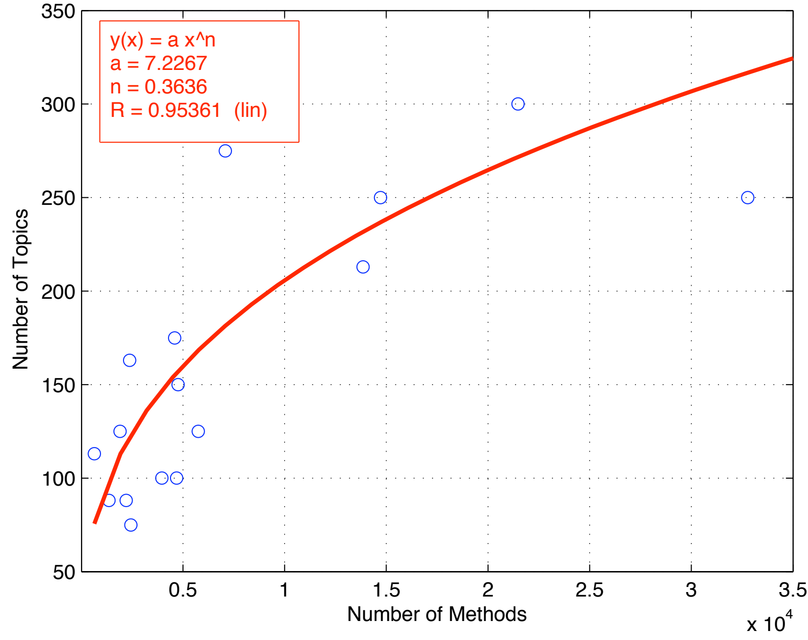

Figure 8: A close-up of the curve fitted to the method results from Figure 7. The x-axis measures the number of source code fragments. The y-axis measures the number of topics that our approach has estimated for the appropriate value.

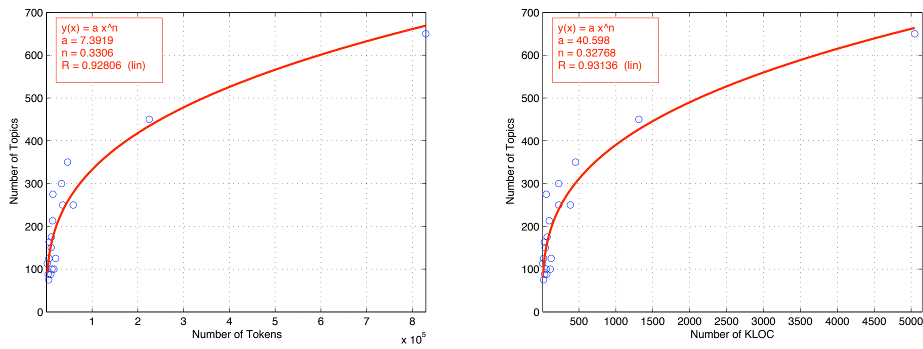

Figure 9: For completeness, we include the curves fitted to the number of tokens and lines of code obtained from our data.

Equation 1 is a specific fit to the individual cases that we reviewed in this study. We generalize this equation as follows:

$$t(x) = 7.25 * x^{0.365} \tag{2}$$

Equation 2 provides an estimate for the appropriate number of topics for source code using

Corpus	Code Fragments	Topic Peak	Eqn. 1 Estimate	Eqn. 2 Estimate
cook (C)	1362	75-100	100	101
snns (C)	2213	75-100	119	121
linuxkernel (C)	3964	100	147	149
postgresql (C)	4689	75-125	156	159
httpd (C)	5758	125	168	171
freebsd (C)	225139	450	378	385
linux (C)	829195	650	670	683
linq (C#)	638	100-125	76	77
nant (C#)	2383	150-175	122	124
rssbandit (C#)	4587	150-200	155	157
db4o (C#)	13855	200-225	232	236
jforum (Java)	2437	75	123	125
heritrix (Java)	4762	150	157	159
ofbiz (Java)	14707	250	237	241
derby (Java)	32781	250	317	323
hadoop (Java)	21478	300	272	276
plone (Python)	1899	125	112	114
django (Python)	7084	275	182	184
zope (Python)	37101	350	331	337

Table 4: A summary of the latent topic estimates using the heuristic method, Equation 1, and Equation 2.

LDA based on our observations of several dozen software systems. The estimates given by Equation 1 and Equation 2 are summarized in Table 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.* in [10]. For an $m \times n$ document-term matrix, where m is the number of documents (classes, instead of methods or functions) and n is the total number of terms 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 with the tokens and source code fragments. From this, we can approximate the equation by replacing n by m to get $(m^2)^{0.2}$, or $m^{0.4}$. The exponent 0.4 suggests that as the number of documents increases, the appropriate rate of change between their topic count and our topic count is roughly similar.

5. Analysis

5.1. Evaluating New Systems

In this section, we apply Equation 2 for one previously examined system and three new systems, and obtain the proximity scores described earlier to identify the topic peak. The values are compared to evaluate the performance of Equation 2 for new systems.

Table 5 lists four new systems used in the analysis. For *httpd*, we use the same project as earlier, but use a version of the source code that is nearly two years more recent. The *php-src* and

Corpus	Location	Code Fragments	Topic Peak	Eqn. 2
httpd (C)	http://httpd.apache.org/	4148	100-125	151
memcached (C)	http://xbmc.org/	288	30	57
php-src (C)	http://php.net/	8170	125-175	194
xbmc (C)	http://xbmc.org/	22059	275-325	279

Table 5: A summary of the latent topic estimates for new systems using the heuristic method and Equation 2.

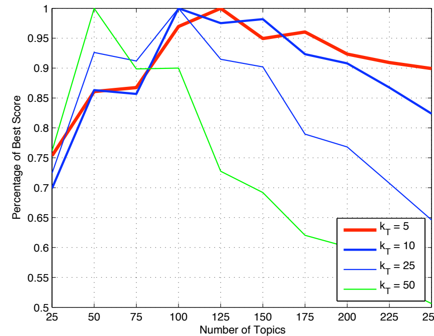

Figure 10: Topic distribution plots for a more recent version of httpd that was not used in determining the appropriate topic count equation. In this system, the function count is 4148, and Equation 2 suggests an appropriate topic count of 151.

xbmc systems are open-source projects of a non-trivial size. The *php-src* project includes the full repository of the PHP language interpreter. The *xbmc* project is an open source software media player and entertainment hub for digital media. The *memcached* project is relatively small open-source project that acts as a distributed memory object caching system used to lighten database load. Each of the projects was processed using the approach detailed in Section 3.4, and a range of models with differing topic counts was generated.

Each of the topic peaks observed and listed in Table 5 is close to the estimated value given by Equation 2. As the only other previously observed project, *httpd* is comparable to the earlier estimate given in Table 2. The newer version of the source code contains fewer functions, and exhibits a topic peak similar to the earlier version but slightly lower by approximately the same amount. The *memcached*, *php-src*, and *xbmc* projects have estimates that are reasonably close to the observed topic peaks.

The topic peaks for the three new systems are provided. Figure 10 shows the topic peak for *httpd*. Figure 11 shows the topic peaks for *memcached*, *php-src*, and *xbmc*.

5.2. *Optimizing for Co-maintenance*

To show the practical benefits of the proposed heuristic, we apply the approach to the problem of idenfifying co-maintenance relationships in source code. Co-maintenance is an observable property of software systems under source control in which source code fragments are modified together in some time frame. For example, implementing a feature may involve many individual changes across several methods or files. A changelist groups these individual changes together [37]. Each changelist in the history of the project can be considered to be a set of modified code fragments. A topic model that captures accurate information about the co-maintenance

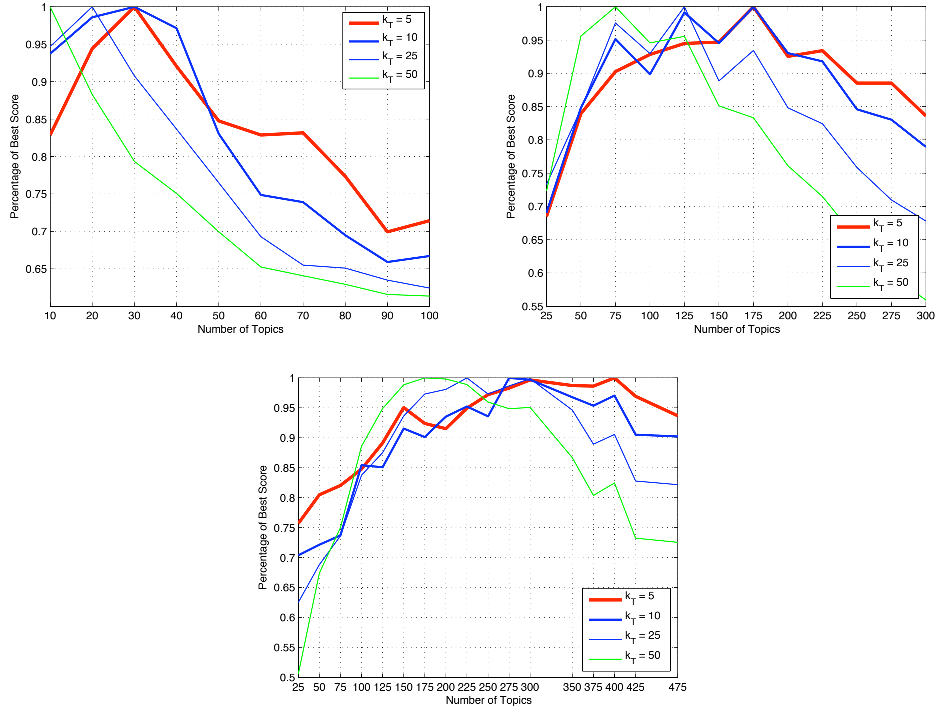

Figure 11: Topic distribution plots for *memcached*, *php-src*, and *xbmc*, three new systems not used in determining the appropriate topic count equation. The *memcached* system contains 288 functions, and Equation 2 suggests an appropriate topic count of 57. The *php-src* system contains 8170 functions, and Equation 2 suggests an appropriate topic count of 194. The *xbmc* system contains 22059 functions, and Equation 2 suggests an appropriate topic count of 279.

relationships of source code should cluster well within changelists, since developers often submit changelists modifying code fragments that are related to one another. As in Section 5.1, we apply Equation 2 for one previously examined system and two new systems to show that the estimated topic count produces clusters that demonstrate co-maintenance relationships and are reasonably sized and well distributed.

In our earlier research [38, 39], we explored the relationship between topic models and co-maintenance history by introducing a visualization that compares conceptual cohesion within changelists. This view of the project history can give insight about the semantic architecture of the code, and identify a number of patterns that characterize particular kinds of maintenance tasks. One strong motivation for this research is the ability to use a topic model, such as LDA, to identify co-maintenance relationships without the need to mine the revision history. For example, in projects that have no revision history, topic models can still be used to extract co-maintenance relationships.

Each visualization is generated as an interactive HTML page, based on the input from a source code repository and some secondary source, such as a topic model. The rows of the display are the list of relevant changelists over the history of the project, starting at the oldest, and progressing forward towards the newest revisions at the bottom. Each column in the display

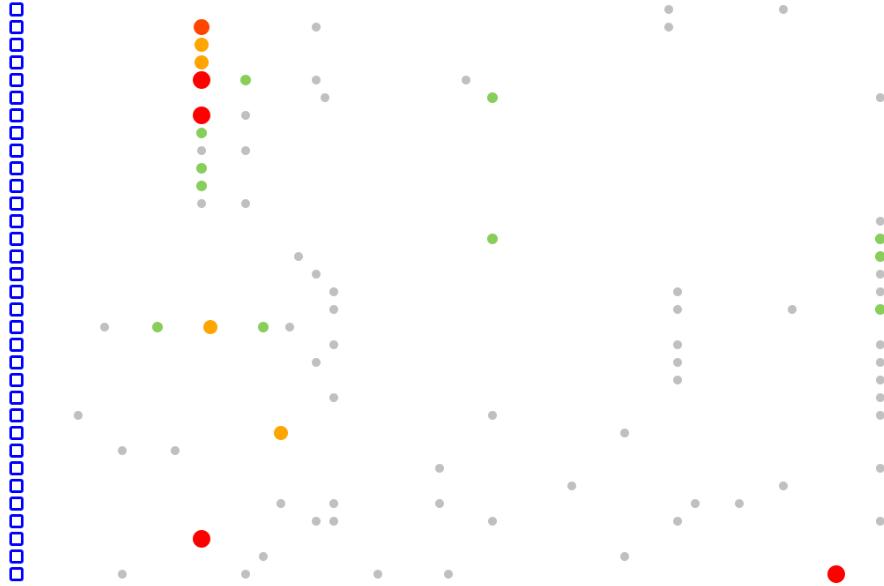

Figure 12: A full view of the visualization for approximately 35 consecutive changelists in httpd's history. Each blue square on the left defines a horizontal row corresponding to a changelist. Each column corresponds to a topic, and each coloured circle represents some number of modified code fragments from that topic. The size and colour of each circle shows how many code fragments from that topic were modified in this particular changelist.

corresponds to a of the concepts described by the model. The circles of varying size and colour on each row are an indication that some code fragments were modified in that changelist. The horizontal location of the circle indicates the concept in which the modifications were made.

Figure 12 provides a demonstration of this visualization. We plot a portion of the change history for the Apache httpd webserver 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 this demonstration, we use the largest red circles to indicate five or more code fragments that are associated with a concept, ranging down to a single small grey circle for one code fragment.

The clustering of co-maintained source code inside changelists can be used to evaluate the effectiveness of a topic model for software maintenance. For example, in a topic model that captures information about the co-maintenance of software methods, each changelist can be expected to modify a small number of topics in the model. In the visualization shown in Figure 12, this would be represented by many large red circles and relatively few small grey circles. If the topic model does not capture co-maintenance, each changelist would modify a large number of topics, as co-maintained code fragments would not be clustered together in the model.

We consider the size of clustered code fragments by topic inside of changelists when evaluating the quality of the topic distribution with respect to the co-maintenance history of a software project. In the simplest case, it would seem ideal to have code fragments from a single topic modified in each changelist. Each changelist would be relevant to a single conceptual area, and all of the source code would be related within a topic. However, this is not an acceptable performance metric; a topic model generated with one topic would suggest the positive result that each changelist modifies a single topic, when the model is actually capturing nothing of value. To

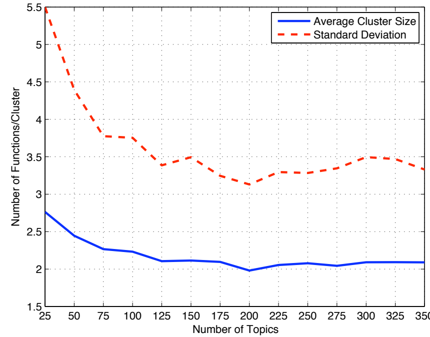

Figure 13: Average cluster size within changelists for the *php-src* system, designated by circles in Figure 12. For small topic counts, the average cluster size will be extremely large, and for large topic counts, the average cluster size and standard deviation will be very close to zero as each code fragment will be associated with a different topic. The interesting point is the dip in the average size and standard deviation, seen near 200 topics; this indicates the point where clusters at the extreme sizes are minimized, and arguably where the model best represents co-maintenance relationships.

mitigate this problem, we consider the average size of all clusters (for example, the average size of each of the clusters represented by a coloured circle in Figure 12), and the standard deviation of the set of clusters. This information shows that in addition to a good amount of clustering within changelists, there is still a reasonable distribution of code fragments over topics.

Figure 13 shows a graph of the average cluster size within changelists for *php-src*. These clusters are designated by circles in Figure 12, and range from approximately 17,000 to 23,000 individual clusters depending on topic count over the entire project history. For small topic counts, the average cluster size will be extremely large, due to code fragments being necessarily allocated to the same topic, even if they are unrelated. For large topic counts, the average cluster size and standard deviation will be very close to zero, as models will approach the point at which each code fragment will be associated with its own topic. The interesting point is the dip in the average size and standard deviation, indicating the point where clusters at the extreme sizes are minimized. This valley in the standard deviation suggests that unrelated code fragments are not being over-associated with topics, and that related code fragments are not forced into separate topics due to a high choice for the topic count parameter.

6. Threats to Validity

Using measures defined pragmatically 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 we believe that using measures like this is a valid first step towards verifying these methods. Measures that provide more accurate results may exist, and new datasets can be created. We discuss our continuing research towards producing data in this vein in the conclusions section of the paper.

Our approach also relies heavily on the ability of LDA to model the data well, and in the topic distribution as a way of describing documents. Although LDA is known to give good results in the natural-language setting, it has not yet been convincingly demonstrated that it outperforms

competing methods such as singular value decomposition or independent component analysis when applied to source code. In natural-language settings, the use of words as terms is intuitive; in software settings, there are significant choices about which syntactic entities should be treated as terms: keywords, operators, variable and method names (and in what subforms), white space, and comments. Encapsulation also means that names that are implicitly present in a particular context, such as a method, are not syntactically visible, but play important roles at run-time. On the face of it, including such names as terms should improve clustering.

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 2 must be considered an estimate.

The use of source code locality as a heuristic for evaluating the quality of topic clusters is imperfect, and in many cases, locality does not always correlate to conceptual similarity. Poorly structured systems will pose problems for this approach. In the majority of systems that we have examined, and in particular those of a non-trivial size, there was enough structure in the system to use this approach. However, we believe that most topics tend to occur within modular boundaries. This is supported by our recent research [39] described briefly in Section 5, in which we performed a detailed analysis of changes by topic over time using the revision control history of several projects. Based on this analysis, we believe that most topics have at least some locality with respect to modules. One surprising result from this work was a significant lack of evidence for cross-cutting topics. Traditional aspects, such as logging functions, were not identified as individual topics when using LDA. Instead, the logging components of individual subsystems tended to be grouped together with other similar parts of that same subsystem. In general, we have not observed traditional aspects showing up as individual topics in a model. In systems where the source code locality heuristic is not appropriate, it can be replaced by another approach. For example, instead of using a boolean locality heuristic between two source code fragments, it is possible to use a boolean value indicating whether the two source code fragments have been co-modified in the past.

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 proximity scores. While we are unsure whether comments should be included or not, in the context of this study we have remained consistent by omitting comments throughout.

This research has focused on optimizing LDA's topic count parameter. We made no attempt to optimize for the alpha and beta parameters used to shape the document-topic and topic-word distributions.

As mentioned earlier, the values k_D and k_T are constants over the entire data set. Considering the same number of similar code fragments for each document and topic may not be appropriate due to the underlying Dirichlet distributions. In the case of topics, for example, it is possible for a topic to be generated that no document is most associated with. Some documents are strong outliers, having few or no interesting relationships with other code fragments in the system. Varying the values k_D and k_T may allow for the heuristics to describe the system more appropriately.

7. Conclusions and Future Work

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 results can be generalized to give appropriate topic counts for arbitrary software systems. In one case study, we show how the co-maintenance relationships in a software system can be used to demonstrate that the topic clusters are reasonable.

The appropriate topic counts for small to medium sized source code packages appear to lie below the standard count of 300 for software systems of around twenty thousand code fragments or less. More importantly perhaps, the appropriate topic counts tend to be achieved when the data is smoothed out to remove 10% to 20% of the original raw information. 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 [21].

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 of topics that models conceptual structure appropriately. Although our study used Latent Dirichlet Allocation, we believe the method is model-independent, and could be well suited to future studies with other techniques like Latent Semantic Indexing.

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.

8. Acknowledgements

This work is supported in part by the Natural Sciences and Engineering Research Council of Canada, by the Ontario Graduate Scholarship Program, and by an the IBM Canada Centre for Advanced Studies.

References

[1] D. J. Bartholomew, M. Knott, Latent variable models and factor analysis, Oxford University Press Inc., Arnold, 2nd edn., ISBN 0-340-69243-x, 1999.

[2] K. A. Bollen, Structural Equations with Latent Variables, Wiley-Interscience, New York, NY, USA, ISBN 0471011711, 1989.

[3] B. Dit, M. Revelle, M. Gethers, D. Poshyvanyk, Feature location in source code: a taxonomy and survey, Journal of Software Maintenance and Evolution: Research and Practice (2011) n/a–n/aISSN 1532-0618.

[4] T. L. Griffiths, M. Steyvers, Finding scientific topics, Proceedings of the National Academy of Sciences of the United States of America 101 (2004) 5228–5235.

[5] G. Heinrich, Parameter estimation for text analysis, Tech. Rep., 2004.

[6] D. M. Blei, A. Y. Ng, M. I. Jordan, Latent Dirichlet allocation, The Journal of Machine Learning Research 3 (2003) 993–1022, ISSN 1533-7928.

[7] J. I. Maletic, N. Valluri, Automatic Software Clustering via Latent Semantic Analysis, in: Proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE '99), IEEE Computer Society, Washington, DC, USA, ISBN 0-7695-0415-9, 251, 1999.

[8] J. I. Maletic, 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), IEEE Computer Society, Washington, DC, USA, 46, 2000.

[9] R. B. Bradford, An empirical study of required dimensionality for large-scale latent semantic indexing applications, in: Proceeding of the 17th ACM Conference on Information and Knowledge Management (CIKM '08), ACM, New York, NY, USA, ISBN 978-1-59593-991-3, 153–162, 2008.

[10] A. Kuhn, S. Ducasse, T. Gírba, Semantic clustering: Identifying topics in source code, Information and Software Technology 49 (3) (2007) 230–243, ISSN 0950-5849.

[11] E. Linstead, C. Lopes, 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), IEEE Computer Society, Washington, DC, USA, ISBN 978-0-7695-3495-4, 813–818, 2008.

[12] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, 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), ACM, New York, NY, USA, ISBN 978-1-59593-882-4, 461–464, 2007.

[13] G. Maskeri, S. Sarkar, 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), ACM, New York, NY, USA, ISBN 978-1-59593-917-3, 113–120, 2008.

[14] S. W. Thomas, B. Adams, A. E. Hassan, D. Blostein, Validating the Use of Topic Models for Software Evolution, in: Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation, SCAM '10, IEEE Computer Society, Washington, DC, USA, ISBN 978-0-7695-4178-5, 55–64, 2010.

[15] S. W. Thomas, B. Adams, A. E. Hassan, D. Blostein, Modeling the evolution of topics in source code histories, in: Proceedings of the 8th Working Conference on Mining Software Repositories, MSR '11, ACM, New York, NY, USA, ISBN 978-1-4503-0574-7, 173–182, 2011.

[16] A. Hindle, M. Godfrey, R. Holt, What's hot and what's not: Windowed developer topic analysis, in: Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM '09), ISSN 1063-6773, 339–348, 2009.

[17] A. Hindle, N. A. Ernst, M. W. Godfrey, J. Mylopoulos, Automated topic naming to support cross-project analysis of software maintenance activities, in: Proceedings of the 8th Working Conference on Mining Software Repositories, MSR '11, ACM, New York, NY, USA, ISBN 978-1-4503-0574-7, 163–172, 2011.

[18] S. K. Lukins, N. A. Kraft, L. H. Etzkorn, Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation, in: Proceedings of the 2008 15th Working Conference on Reverse Engineering, IEEE Computer Society, Washington, DC, USA, ISBN 978-0-7695-3429-9, 155–164, 2008.

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

[20] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, E. Merlo, Recovering Traceability Links between Code and Documentation, IEEE Transactions on Software Engineering 28 (10) (2002) 970–983, ISSN 0098-5589, doi: \bibinfo{doi}{http://dx.doi.org/10.1109/TSE.2002.1041053}.

[21] A. Marcus, 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), IEEE Computer Society, Washington, DC, USA, ISBN 0-7695-1877-X, 125–135, 2003.

[22] R. Oliveto, M. Gethers, D. Poshyvanyk, A. De Lucia, On the Equivalence of Information Retrieval Methods for Automated Traceability Link Recovery, in: Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension, ICPC '10, IEEE Computer Society, Washington, DC, USA, 68–71, 2010.

[23] H. U. Asuncion, A. U. Asuncion, R. N. Taylor, Software traceability with topic modeling, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10, ACM, New York, NY, USA, ISBN 978-1-60558-719-6, 95–104, 2010.

[24] B. Schölkopf, A. J. Smola, R. C. Williamson, P. L. Bartlett, New Support Vector Algorithms, Neural Computation 12 (5) (2000) 1207–1245, ISSN 0899-7667.

[25] I. Steinwart, On the optimal parameter choice for ν-support vector machines, IEEE Transactions on Pattern Analysis and Machine Intelligence 25 (2003) 1274–1284, ISSN 0162-8828.

[26] H. Wallach, D. Mimno, A. McCallum, Rethinking LDA: Why Priors Matter, in: Y. Bengio, D. Schuurmans, J. Lafferty, C. K. I. Williams, A. Culotta (Eds.), Advances in Neural Information Processing Systems 22, 1973–1981, 2009.

[27] P. Comon, Independent Component Analysis, A new concept?, Signal Processing 36 (3) (1994) 287–314.

[28] S. Grant, D. B. Skillicorn, J. R. Cordy, Topic Detection Using Independent Component Analysis, in: Proceedings of the 2008 Workshop on Link Analysis, Counterterrorism and Security (LACTS '08), 23–28, 2008.

[29] S. Grant, J. R. Cordy, D. B. Skillicorn, Automated Concept Location Using Independent Component Analysis, in: Proceedings of the 15th Working Conference on Reverse Engineering (WCRE '08), 138–142, 2008.

[30] S. Grant, J. R. Cordy, Vector Space Analysis of Software Clones, in: 17th IEEE International Conference on Program Comprehension (ICPC '09), 233–237, 2009.

[31] S. Grant, J. R. Cordy, Estimating the Optimal Number of Latent Concepts in Source Code Analysis, in: Proceedings of the 10th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '10), IEEE Computer Society, Timişoara, Romania, 2010.

[32] C. K. Roy, 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 22 (3) (2010) 165–189.

[33] C. K. Roy, J. R. Cordy, Are scripting languages really different?, in: Proceedings of the 4th International Workshop on Software Clones (IWSC '10), IWSC '10, ACM, New York, NY, USA, ISBN 978-1-60558-980-0, 17–24, 2010.

[34] S. Grant, Topic Estimate Source Code, `http://research.cs.queensu.ca/home/scott/scam10/`, 2010.

[35] X.-H. Phan, 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`, ????

[36] J. R. Cordy, The TXL source transformation language, Science of Computer Programming 61 (3) (2006) 190–210, ISSN 0167-6423.

[37] A. Hassan, R. Holt, Studying the evolution of software systems using evolutionary code extractors, in: Software Evolution, 2004. Proceedings. 7th International Workshop on Principles of, ISSN 15504077, 76 – 81, doi: \bibinfo{doi}{10.1109/IWPSE.2004.1334771}, 2004.

[38] S. Grant, J. R. Cordy, D. B. Skillicorn, Reverse Engineering Co-maintenance Relationships Using Conceptual Analysis of Source Code, in: Proceedings of the 18th Working Conference on Reverse Engineering (WCRE '11), 87–91, 2011.

[39] S. Grant, J. R. Cordy, D. B. Skillicorn, Using Topic Models to Support Software Maintenance, in: Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR '12), 403–408, 2012.