**Mining Business-Topics in Source Language using Latent Dirichlet Allocation**

# ABSTRACT

One of the difficulties in maintaining a large software system is the absence of documented business domain topics and correlation between these domain topics and source code. Without such a correlation, people without any prior application knowledge would find it hard to comprehend the functionality of the system. Latent Dirichlet Allocation (LDA), a statistical model, has emerged as a popular technique for discovering topics in large text document corpus. But its applicability in extracting business domain topics from source code has not been explored so far. This paper investigates LDA in the context of comprehending large software systems and proposes a human assisted approach based on LDA for extracting domain topics from source code. This method has been applied on a number of open source and proprietary systems. Preliminary results indicate that LDA is able to identify some of the domain topics and is a satisfactory starting point for further manual refinement of topics.

# Categories and Subject Descriptors

D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—Restructuring, reverse engineering, and reengineering;

**General terms**

Latent Dirichlet Allocation

**Keywords**

Maintenance, Program comprehension, LDA

# INTRODUCTION

Large legacy software systems often exist in a state of disorganization with poor or no documentation. Adding new features and fixing bugs in such a system is highly error prone and time consuming since the original authors of the

|  |
| --- |
| Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.  *ISEC’08,* February 19-22, 2008, Hyderabad, India.  Copyright 2008 ACM 978-1-59593-917-3/08/0002 ...$5.00. |

system are generally no longer available. Moreover, the people maintaining the code-base do not comprehend the functional purpose of different program elements (functions, files, classes, data structures etc.) and the roles they play to fulfill various functional services offered by the system.

When a software system is small, one can understand its functional architecture by manually browsing the source code. For large systems, practitioners often rely on program analysis techniques such as call graph, control and data flow and slicing [2].Though this is very helpful to understand the structural intricacies of the system, it helps a little to comprehend the functional intent of the system. The reason is not difficult to understand. Structural analysis techniques work on the structural information derived from a set of program source code. The structural information is at a very low level of granularity- such as files, functions, datastructures, variable usage dependencies, function calls and so on. This information hardly reveals any underlying functional purpose. For a large system, this information becomes overwhelmingly large for a maintainer to manually derive any functional intent out of it. Moreover, for a system with millions of lines of code, the memory requirement to hold the structural information and perform various analysis often becomes a bottleneck.

An important step to comprehend the functional intent of the system (or the intended functional architecture) is to identify the business topics that exist in the system, around which the high level components (or modules) have been implemented. For example consider a large banking application that deals with customers, bank accounts, credit cards, interest and so on. A maintainer without any application knowledge will find it difficult to add a new interest calculation logic. However, if it is possible to extract the business topics such as “customer”, “interest” from the source code and establish an association between “interest” with various program elements, it would be of immense help to the maintainer to identify the related functions, files, classes and data structures and carry out the required changes for interest calculation. This in turn can make a novice programmer much more productive in maintaining a system, specially when the software is in the order of millions of lines of code with little documentation.

A plausible technique to identify such topics is to derive semantic information by extracting and analyzing various important “keywords” from the source code text [13]. It is often the case that the original authors of the code leave hints to the meaning of program elements in the form of keywords in files, functions, data type names and so on. For instance, it is highly common to use keywords like “proxy”, “http”while implementing an http-proxy functionality. Similarly, for a banking application one would surely like to use a keyword like “interest” in implementing functions, classes and data structures pertaining to interest calculation.

Assuming that the meaningful keywords do exist in program elements, is it possible to semantically correlate these keywords to meaningful clusters where each meaningful cluster of keywords can be interpreted as a “Topic”? For instance, is it possible to group all “proxy” related keywords together into one cluster and identify it as“proxy”topic and The "verification" associated with the keywords in another collection that makes up the "authenticity" title? If this is the case, one can establish a connection between the “proxy” heading and the various program elements (files, tasks or data structures) related to the “proxy”.

This paper addresses the above problem and suggests a humanitarian approach according to the Latent Dirichlet Allocation (LDA) [7] for identifying topics in the source code. The LDA is well-known in the field of text classification and the identification of topics from text documents. To the best of our knowledge, This is the first attempt to use LDA in the context of source code analysis.

This paper is structured as follows: In the following section we provide a brief review of the relevant documentation for our work and the required information behind the LDA. Section 3 discusses the function of the LDA in extracting domain titles from source code and provides an interpretation of the LDA model in the source code context. A detailed description of extracting domain titles from source code using LDA is presented in section 4. We have used LDA for most open source and related programs. Section 5 presents the results obtained. The advantages and disadvantages of the LDA-based approach and other interesting developments made during our evaluation are outlined in Section 6. Finally, Section 7 discusses future research indicators and concludes the paper.

1. DATE AND RELATED WORK

Researchers have long recognized the value of linguistic knowledge as identifying words and comments in the understanding of the system. For example, Biggerstaff et al. [5] suggested the provision of domain concepts as a way to understand the system. Tonella et al. [8] suggest job titles and signatures for specific domain details. Anquetil et al. [3] suggest that the information obtained in the file name usually serves the purpose of the source code specified in the file. Wilde et al. [24] also suggested the use of language knowledge to identify the purpose of the program. Since then, linguistic data has been used in various system analysis and archiving activities such as tracing between external texts and source code [4, 16], embedded location [17, 21, 25], identifying high-level concepts [15] and so on.

More recently, language data has also been used to identify topics in source code and was later used for software collection [13] and software classification [11].

Kuhn et al. [13] use Latent Semantic Analysis (LSA)

[9] A method based on the identification of topics in the source code by skillfully combining art software such as methods, files or packages based on identifiers and comments. The way we work is different from that of Kuhn et al. in two ways. First, and most importantly, our interpretation of the "topic" is different from that of Kuhn. Kuhn translates software-based software technologies (such as methods, files, etc.) as titles while translating a set of doctrinal language words based on the identifier words and “title” comments. Another important difference lies in the method used for semantic collection. While Kuhn et al. we have adopted LSA to compile a set of software components, our method of combining language words is based on the Latent Dirichlet Allocation.

Kawaguchi et al. [11] uses language information in source code to automatically identify categories and classify open source repositories. A set of related identifiers is classified as “category”. In our case, we refer to a set of words based on identifiers as "subject"; therefore the title can certainly be considered as “category”. However, our approach to the integration of academic related terminology terminals is different from the method proposed by Kawaguchi et al. [11]. Kawaguchi et al. first it uses the LSA to find matching pairs between terms and then uses the merging algorithm to put the same words together. The LDA-based approach we have used reduces the need for two steps. Since LDA is actually a method of modeling topics, it not only finds similarities between words, and creates a collection of similar words to form a theme. Throughout this section, we provide a brief description of the LDA and its use in extracting articles from text documents.

1.1 LDA

The Latent Dirichlet Allocation (LDA) [7] is a mathematical model, in particular the subject model, which was originally used in the field of textual analysis of representative texts. The basic idea of ​​the LDA is that a document can be considered as a mixture of a limited number of titles and that every word that sounds familiar in a text can be associated with one of these titles. Given the draft document, the LDA seeks to obtain the following:

• Identifies a set of topics

• Links vocabulary to topic

• Describes a specific combination of these topics in each text in the corpus.

LDA is used to exclude topics from text. For example, Newman et al. [19] applied to the LDA for 400 articles such as "September 11 attack", "Harry Potter", "Basketball" and "Holidays" from a compilation of 330000 New York Times news articles and representing each news article as a combination of these articles. The LDA was also applied for the identification of topics in various fields. For example, LDA has been used to obtain scientific articles from paper photographs published in the national science school [10] program. McCallum et al. [18] have suggested LDA to publish articles on social networks and use them in the 250,000 Enron email group. The variance in LDA was also used by Steyvers et al. [22] analyzing 160,000 themes from a computer-assisted "citeseer" collection. Recently, Zheng et al. [6] used LDA to obtain various biological concepts from a protein-related organization

For the sake of completeness, we briefly introduce the LDA model. A thorough and complete description of the LDA model can be found in [7]. The vocabulary for describing the LDA model is as follows:

**word** A *word* is a basic unit defined to be an item from a vocabulary of size *W*.

**document** A document is a sequence of *N* words denoted by *d* = (*w*1*,*··· *,wN*) where *wn* is the *n*th word in the sequence.

**corpus** A corpus is a collection of M documents denoted by **D** = {*d*1*,*··· *,dM*}.

In the statistical natural language processing, it is common to model each document **d** as a multinomial distribution *θ*(*d*) over *T* topics, and each topic *zj,j* = 1···*T* as a multinomial distribution *φ*(*j*) over the set of words *W*. In order to discover the set of topics used and the distribution of these topics in each document in a corpus of documents **D**, we need to obtain an estimate of *φ* and *θ*. Blei et al. [7] have shown that the existing techniques of estimating *φ* and *θ* are slow to converge and propose a new model- LDA. The LDA based model assumes a prior Dirichlet distribution on *θ*, thus allowing the estimation of *φ* without requiring the estimation of *θ*.

LDA assumes a generative process for creating a document [7] as presented below.

1. choose *N* ∼ *Poisson*(*ξ*) : Select the number of words

*N*

1. *θ* ∼ *Dir*(*α*) : Select *θ* from the dirichlet distribution parameterized by *α*.
2. For each *wn* ∈ **w** do
   1. Choose a topic *zn* ∼ *Multinomial*(*θ*)
   2. Choose a word *wn* from *p*(*wn*|*zn,β*), a multinomial probability *φzn*

In this model, various distributions namely, the set of topics, topic distribution for each of the documents and word probabilities for each of the topics are in general intractable for exact inference [7]. Hence a wide variety of approximate algorithms are considered for LDA. These algorithms attempt to maximize likelihood of the corpus given the model. A few algorithms have been proposed for fitting the LDA model to a text corpus such as variational Bayes [7], expectation propagation [14], and Gibbs sampling [10].

# APPLYING LDA TO SOURCE CODE

Given that LDA has been successfully applied to large corpus of text data (as discussed in Section 2.1), it is interesting to explore i) how applicable it is in the context of source code ii) how effective the technique is in identifying business topics in a large software system. To apply LDA in source code, we consider a software system to be a collection of source code files and the software system is associated with a set of business domain concepts (or topics). For instance, the Apache web server implements functionality associated with http-proxy, authentication, server, caching and so on. Similarly, a database server like Postgresql implements functionality related to storage management. Moreover, there exists a many-many relationship between these topics like authentication, storage management and the source code files that implement these topics. Thus a source code file can be thought of as a mixture of these domain topics.

Applying LDA to the source code now reduces to mapping source code entities of a software system to the LDA model, described in Table 1. Given this mapping, applica-

|  |  |
| --- | --- |
| LDA Model | Source Code Entities |
| word | We define domain specific keywords extracted from names of program elements such as functions, files, data structures and comments to be the vocabulary set with cardinality *V* . A word *w* is an item from this vocabulary. |
| document | A source code file becomes a document in LDA parlance. For our purpose, we represent a document  **f***d* = (*w*1*,w*2*,...,wN*)  to be a sequence of *N* domain specific keywords. |
| corpus | The software system  S = {**f**1*,***f**2*,...,***f***M*}  having *M* source code files forms the corpus. |

**Table 1: Mapping LDA to Source Code**

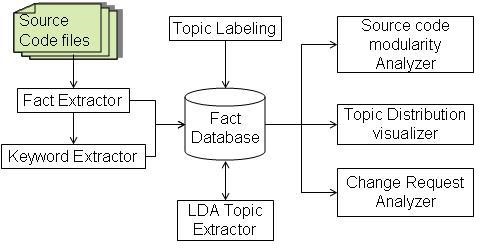
tion of LDA to source code corpus is not difficult. Given a software system consisting of a set of source code files, domain related words are extracted from each of the source code files. Using this, a source code file-word matrix is constructed where source code files form the rows, domain words form the columns and each cell represents the weighted occurrence of the word, representing the column, in the source code file representing the row. This source code file-word matrix is provided as input to LDA. The result of LDA is a set of topics and a distribution of these topics in each source code file. A topic is a collection of domain words along with the importance of each word to the topic represented as a

numeric fraction.

# IMPLEMENTATION

We have implemented a tool for static analysis of code. Figure 1 shows a part of this tool that specifically deals with topic extraction, topic location identification[5, 24], visualization of the topic distribution and a modularity analysis based on domain topics.

The main input of LDA based topic extraction is a documentword matrix **wd**[*w,fd*] = *η* where *η* is a value indicating the importance of the word *w* in the file *fd*. We will shortly describe an approach to compute *η* based on the number and the place of occurrences of *w* in *fd*. Our current implementation uses the Gibbs sampling method [10] that uses a markov chain monte carlo method to converge to the target



**Figure 1: Tool Block diagram**

distributions in an iterative manner. The detailed description of this method is not in scope of this paper.

Input parameters.

Our tool, based on the LDA process, takes two parameters α and β (as described in paragraph 2.1.1) and creates φ word distribution over titles and θ distribution of titles over documents. In addition to the parameters α and β the tool also requires the value of multiplication. Remember that the LDA defines a topic as the distribution of opportunities across all policies in the corpus. Define a user limit inani that is used to identify the most important words in each topic.

1.1 Keyword removal

In order to build an LDA word document matrix, it is very important to find domain-related words in the source code. The methods used by the LDA in text-based texts look at each word in the text for this purpose. However, unlike plain text document the source code file is an edited text and it is definitely not wrong to assume that each word in the file will be related to the domain. First, a large percentage of words in source code files form the syntax of a programming language, such as, if, class, return, period and so on. In addition, domain keywords are often included within identifiers such as subtitles and identifiers that need to be properly separated to exclude appropriate sub-keywords. Given this view we suggest the following steps to extract keywords from source code files:

1. Release of Truth.

2. Separation of identification and commentary by key words.

3. Incorporate keywords into their general communities.

4. Keyword filtering to eliminate keywords that do not reflect any business concept.

True Release.

True extracting is the process of analyzing source code texts and extracting interesting meta data such as files, tasks, performance dependencies, data structure etc. We used the source navigator [1] to extract the facts from the source code.

Identification classification.

Separation of clauses into logical subwords is important because unlike the natural language text where each word is independent and found in a dictionary, source code identification words are usually not individual words but logical sequence of characters and acronyms separated by a specific letter or a naming convention. For example, a function called "add auth details" to httpd-2.0.53 source code creates three meaningful subdivisions of the letters "add", "auth" and "info" determined by "".

It is the vocabulary we use for the syllables that sound “keyword”. Each identifier is divided into groups of keywords. In order to make this distinction it is important to know how the identification words are separated. There are many programs such as underscore, hyphen, or capital letters (camel) as in "getLoanDetails". We have implemented a system that separates adjectives in Perl.

Stopping.

Often keywords in the source code are used in both singular and plural. For example "loans" and "loans". In our analysis we do not need to take them as two separate keywords. We therefore combine all keywords by inserting them into their default root. Also, It is a common practice [23, 12] to establish words in order to improve the results of analysis. We used Porter’s stemming algorithm [20] to insert all the words into common roots.

Sorting.

Not all keywords are subject to domain names. For example keywords such as “find” and “set” are very common and a stop list is started to sort such words.

1.2 File keyword map

After finding a collection of different keywords for the program provided by the software, we now calculate the wd matrix. For this purpose, we include the approximate number of the occurrence of the name w file fd as follows:

1. It defines heuristic weight

λ: {lt} → ℵ

which provides the total value defined by the user in the “locationtype” lt. "Location type" lt defines the component of the source code file such as file name, username, parameter valid names, comments, data structure name and so on where the keyword is found. The weight given to the name obtained from the job name will differ from the name obtained from the name of the data structure.

2. The wd [w, fd] value of the word w from the fd file is calculated from the estimated total frequency of each type of event in the source code fd file. That is, wd [w, fd] = Xλ (lti) × ν (w, fd, lti)

lti

where ν (w, fd, lti) defines the frequency of the word w in the local area lti of the source file fd.

To illustrate wd value calculations consider the following code snippets from the OrderDetails.java file.

the OrderDetails community section uses Java.io.Serializable {

Private String orderId; Private user String; Private String orderDate; private floatValue layout; String Private Order Status;

GetOrderStatus () public String {

return (order Status);

} ...

...

}As discussed in section 4.1, identifier names are split to get meaningful domain words and importance factor calculated for each of the words. One such word that is extracted from the above code snippet is“Order”which occurs in comments and names of different type of identifiers such as in class name, attribute name and method name. These different types of sources for words constitute our set of location types *lt*. Generally, in an object oriented system, classes represent domain objects and their names are more likely to yield domain words that are important for that class. Hence, *λ*(*class*) generally is assigned higher value by domain experts than *λ*(*attribute*). Let us assume that in this particular example *λ*(*class*) equals 2, *λ*(*attribute*) equals 1 and *λ*(*method*) equals 1. The importance factor of the word “Order” in the above code snippet as calculated according to the formula given above is 7.

**wd**[*Order,OrderDetails.java*] = 2 ∗ 1 + 1 ∗ 4 + 1 ∗ 1 = 7

Similarly, weighted occurrence is calculated for other words such as “details”, “user” and “status”.

## Topic labeling

LDA could not satisfactorily derive a human understandable label for an identified topic. In most of the cases, the terms from which a label can be derived are abbreviations of business concepts or acronyms. As a result it becomes hard to create a meaningful label for a topic automatically. In the current version of the tool, identified topics have been labeled manually.

# CASE STUDIES

We have tested our approach on a number of open source and proprietary systems. In the rest of this section we discuss the results obtained using some of the topics as examples.

## Topic Extraction for Apache

We extracted 30 topics for Apache. For the sake of brevity we list only two topics, namely “SSL” and “Logging”. Table 1(a) lists the top keywords for topic “SSL” and their corresponding probability of occurrence when a random keyword is generated from the topic “SSL”.

Our tool is able to extract not just the domain topics, but also infrastructure-level topics and cross cutting topics. For instance, “logging” is a topic that cuts across files and modules. Our tool, based on LDA, is able to cluster together all logging related keywords together as shown in table 1(b) that lists the top keywords for topic “Logging” and their corresponding probability values.

(a) Topic labeled as SSL (b) Topic labeled as Logging

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Keyword | Probability |  | Keyword | Probability |
| ssl | 0.373722 | log | 0.141733 |
| expr | 0.042501 | request | .036017 |
| init | 0.033207 | mod | 0.0311 |
| engine | 0.026447 | config | 0.029871 |
| var | 0.022222 | name | 0.023725 |
| ctx | 0.023067 | headers | 0.021266 |
| ptemp | 0.017153 | autoindex | 0.020037 |
| mctx | 0.013773 | format | 0.017578 |
| lookup | 0.012083 | cmd | 0.01512 |
| modssl | 0.011238 | header | 0.013891 |
| ca | 0.009548 | add | 0.012661 |

**Table 2: Sample Topics extracted from Apache source code**

## Topic Extraction For Petstore

In order to investigate the effect of naming on topic extraction results we considered Petstore, a J2EE blueprint implementation by Sun Microsystems. Being a reference J2EE implementation, it has followed good java naming conventions and a large number of identifiers have meaningful names.

(a) Topic labeled as Con- (b) Topic labeled as Adtact Information dress Information

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Keyword | Probability | | info | 0.418520 | | contact | 0.295719 | | email | 0.050116 | | address | 0.040159 | | family | 0.040159 | | given | 0.036840 | | telephone | 0.026884 | | by | 0.000332 | | |  |  | | --- | --- | | Keyword | Probability | | address | 0.398992 | | street | 0.105818 | | city | 0.055428 | | code | 0.055428 | | country | 0.055428 | | zip | 0.055428 | | name1 | 0.050847 | | state | 0.046267 | | name2 | 0.046267 | | end | 0.005039 | | add | 0.009548 | |

**Table 3: Sample Topics extracted from petstore source code**

As shown in table 2(a) we are able to successfully group all “contact information” related terms together. However, what is more significant in this example is that the top keywords “info”, “contact” are meaningful and indicative of the probable name of the topic. For example if we concatenate these two keywords into “info contact” it can be considered as a valid label for the “contact information” topic.

Similarly, even in the case of “address information” topic, shown in table 2(b), the concatenation of the top keywords “address” and “street” can be used to label the “address information”topic. It can be observed from the sample topics extracted that good naming convention yields more meaningful names thereby simplifying the process of labeling the topics.

## Synonymy and Polysemy resolution

One of the key factors in extracting coherent topics and grouping semantically related keywords together is the ability of the algorithm employed to resolve synonymy-different words having the same meaning. We have observed that our tool is able to satisfactorily resolve synonymy to a good extent since LDA models topics in a file and words in a topic using multinomial probability distributions. For instance consider the topic labeled as “transaction” in PostgreSQL shown in table 5.3. LDA has identified that “transaction” and“xact”are synonymous and grouped them together in a single cluster as shown below.

|  |  |
| --- | --- |
| Keyword | Probability |
| transaction | 0.149284 |
| namespace | 090856 |
| commit | 0.035349 |
| xact | 0.035349 |
| visible | 0.029506 |
| current | 0.029506 |
| abort | 0.026585 |
| names | 0.026585 |
| command | 0.023663 |
| start | 0.020742 |
| path | 0.017821 |

**Table 4: Transaction and Xact Synonymy resolution by LDA**

What’s more interesting is the fact that our tool has been able to resolve polysemy-same words having different meaning in source code. A polyseme can appear in multiple domain topics depending on the context. The reason for our tool to be able to identify polyseme is not difficult to understand. Note that LDA models a topic as a distribution of terms; therefore it is perfectly valid for a term to appear in two topics with different probability values. Furthermore, LDA tries to infer a topic for a given term with the knowledge of the context of the word, i.e. the document where the word is appearing. For instance, in Linux-kernel source code we observed that the term “volume” has been used in the context of sound control as well as in the context of file systems. LDA is able to differentiate between these different uses of the term and has grouped the same term in different topics.

# DISCUSSION

In this section we discuss various factors that impact the results obtained. Subsequently we will discuss benefits and limitations of our approach.

## Effect of number of Topics

Our approach for topic extraction accepts the number of topics to be extracted as an input from the user. We have observed that varying the number of topics has a significant impact on polysemy resolution. For instance, consider the example of polysemy resolution of the keyword“volume” in Linux-kernel, discussed in subsection 5.3. We have conducted our experiment on Linux-kernel source code twice. In both the times we have kept all the parameters, namely *α*, *β* the number of iterations and the cut-off threshold Ψ same except for the number of topics. In the first experiment the number of topics *T* was set to 50 and in the second experiment *T* was set to 60. In both these experiments, of the total topics extracted two topics were“sound”related topics and one topic for“file systems”. Table 6.1 lists the probabilities of keyword“volume”in“sound”and“file systems”topic for both the experiments.

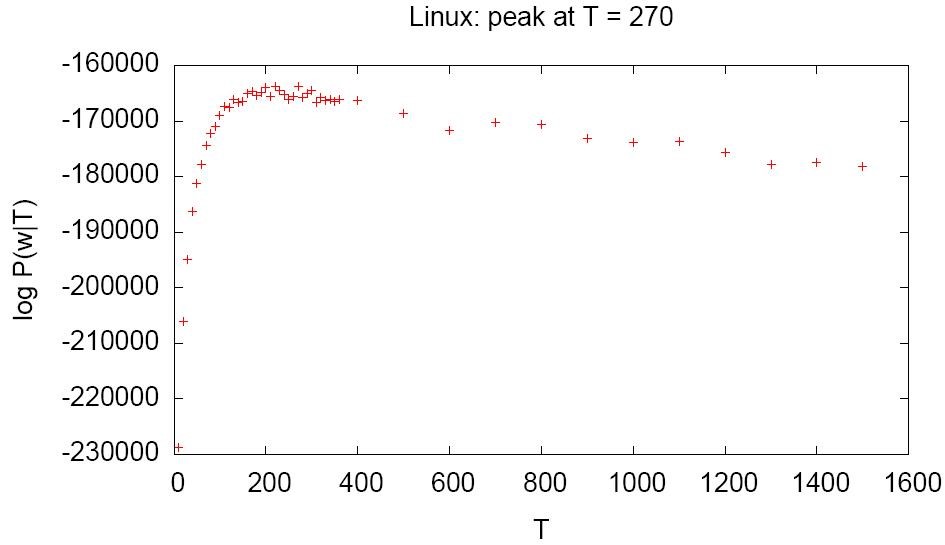
|  |  |  |
| --- | --- | --- |
| Topic type | ’Volume’ probability for Experiment1 with T=50 | ’Volume’ probability for Experiment2 with T=60 |
| Sound topic 1 | 0.024 | 0.032 |
| Sound topic 2 | 0.009 | 0.009 |
| file systems topic | *<* 0.0002 | 0.004 |

**Table 5: Effect of number of topics on polysemy resolution in Linux-kernel**

In our experiments we have used a value of the threshold Ψ to be 0*.*001 for determining whether a keyword belongs to a topic or not. If the probability of a keyword associated with a topic is less than 0*.*001 then we do not consider that keyword as indicator of that topic. In view of this, it can be observed from the table 6.1 that in experiment 1 the keyword “volume” has a probability of less than 0*.*0002 for topic “file systems”. Hence “volume” is associated with only the two sound related topics and not with the“file systems” topic. However, in experiment 2 when the number of topics was increased to 60, the probability of “volume” for topic “file systems” is 0*.*004. This probability is greater than our threshold 0.001 and hence “volume” is also considered as an indicator for “file systems” topic apart from the sound related topics. The polysemy in the keyword “volume” is revealed only in the second experiment with 60 topics.

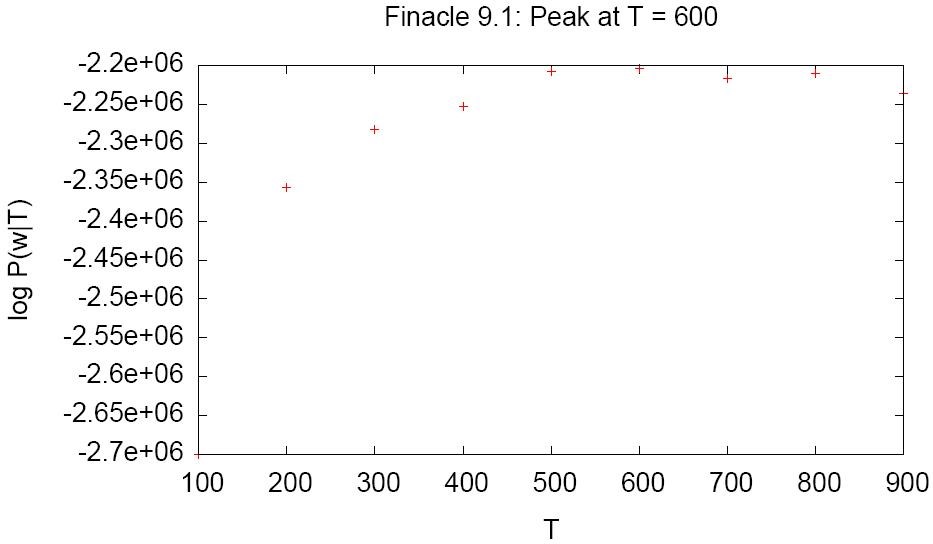
## Discovering optimal number of Topics

The problem of identifying optimal number of topics is not specific to source code alone. Topic extraction from text documents faces a very similar problem. Griffiths and Steyvers recommend trying different numbers of topics T and suggest using the *maximum likelihood* method on *P*(*w*|*T*) [10]. Applying this technique to extract topics from Linux suggests that the optimal number of topics in the case of Linux is 270 as shown in figure 2.



**Figure 2: Inferring optimum number of topics for Linux**

However, automatically inferring the number of topics by maximizing the likelihood is not without problems.



**Figure 3: log-likelihood graph for our proprietary system**

We applied our tool to extract topics for a very large proprietary business application having multi-million lines of C code. The number of topics predicted using the likelihood method was much larger than what the architects and domain experts of the proprietary system considered reasonable. As shown in figure 3, likelihood peaked at *T* = 600 suggesting that the optimal number of topics for our system was 600. However, the architects and domain experts felt that the resonable number of topics to be around 100.

## Effect of *α* and *β*

As discussed in section 2.1.1, the LDA model accepts two parameters *α* and *β*. *α* controls the division of documents into topics and *β* controls division of topics into words. Larger values of *β* yield coarser topics and larger values of *α* yields coarser distribution of document into topics. Hence the right value of *α* and *β* is needed to derive good quality topics and assignment of topics to documents.

Some of the implementations of LDA estimate these values on-the-fly while other implementations rely on the user to provide appropriate values. In our implementation the values of *α* and *β* needs to be provided by the user.

## Human Intervention

Even though LDA based topic extraction presented in this paper is automatic and unsupervised, we believe that human intervention is necessary in a number of aspects to achieve results of acceptable quality. In this subsection we point out areas of our method which would be helped by human intervention.

*Expert delineated Keyword Filtering:.*

Both keyword extraction and subsequent filtering has impact on the quality of the results obtained. The vocabulary of source code is much smaller than that of natural language text corpus and hence the effect of missing or incorrect terms is much stronger. Also, in the context of keyword extraction from identifiers in the program, we have observed that not all identifiers are equally good indicators of the business topics. For this purpose we have introduced the weighing scheme *λ* as described in Section 4.2 where an expert, for instance, can assign more weight to file names over say comments in the program. Human intervention can also improve the filtering of keywords by identifying infrastructure and domain specific stop words. For instance “EJB”, “get”, “set” are some of the common keywords which needs to be filtered out.

*Number of Topics:.*

As discussed in section 6.2, the log-likelihood method for estimating the number of topics is not always appropriate and in a number of cases the number of topics is better supplied by domain experts and architects of the system. During our experiments we have observed that one needs to try different number of topics and repeat the topic extraction process to get a set of topics of acceptable quality.

*Topic validation and labeling:.*

In our experience topic extraction has been an iterative process. Topics extracted initially are evaluated and based on the results keyword extraction and filtering heuristics are updated and the *α* and *β* parameters varied to extract better topics. It is difficult to automatically evaluate the quality of topics obtained. We needed a domain expert who can manually examine the cluster of terms and check if it truly represents a domain topic. Moreover, when a domain topic has been identified labeling has to be done manually.

# CONCLUSION AND FUTURE WORK

In this paper we have investigated the effectiveness of the LDA in the context of the comprehension program and suggested a LDA-based approach to subject-coding the source code. A review of several open sources and affiliate programs has shown that our tool is able to adequately extract some but not all domain titles. We also saw the need to include something personal in order to improve the quality of published articles.

Another disadvantage of the LDA is that it does not find correlations between published titles or identifying topics at a different level of granularity. As part of our future work we plan to investigate ways of extracting topics at different levels of granularity and to identify the different relationships between them. we intend to compare the alternative methods of publication with the LSA with the LDA-based approach presented here. Finally, We believe that the proposed LDA-based approach to the issuance of a business title is promising and ensures continuous investigation and validation.

# REFERENCES

1. Source navigator 5.4.1. http://sourcenav.sourceforge.net, 2003.
2. P. Anderson and M. Zarins. The codesurfer software understanding platform. In *IWPC*, pages 147–148. IEEE Computer Society, 2005.
3. N. Anquetil and T. C. Lethbridge. Recovering software architecture from the names of source files. *Journal of Software Maintenance: Research and Practice*, 11:201–221, 1999.
4. G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Transactions in Software Engineering*, 28(10):970–983, 2002.
5. T. J. Biggerstaff, B. G. Mitbander, and D. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–83, May 1994.
6. Z. Bin, M. David, and L. Xinghua. Identifying biological concepts from a protein-related corpus with a probabilistic topic model. *BMC Bioinformatics*, 7, 2006.
7. D. Blei, A. Ng, and M. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.
8. B. Caprile and P. Tonella. Nomen est omen:

Analyzing the language of function identifiers. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, 1999.

1. S. Deerwester, S. T. Dumais, G. W. Furnas, and T. K. Landauer. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41:391–407, 1990.
2. T. Griffiths and M. Steyvers. Finding scientific topics. In *Proceedings of the National Academy of Sciences*, pages 5228–5235, 2004.
3. S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue. MUDABlue: An automatic categorization system for open source repositories. In *APSEC*, pages 184–193. IEEE Computer Society, 2004.
4. A. Kuhn. Semantic clustering: Making use of linguistic information to reveal concepts in source code. Master’s thesis, University of Bern, 2006.
5. A. Kuhn, S. Ducasse, and T. Gˆırba. Semantic clustering: Identifying topics in source code. *IST*, 2006. To appear.
6. J. Lafferty and T. Minka. Expectation-propagation for the generative aspect model. In *Proceedings of the 18th Conference on Uncertainty in Artificial Intelligence*, 2002.
7. A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE 2001)*, pages 107–114, Nov. 2001.
8. A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *International Conference on Software Engineering*, pages 125–134. IEEE Computer Society Press, may 2003.
9. A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 214–223, Nov. 2004.
10. A. McCallum, A. Corrada-Emmanuel, and X. Wang. Topic and role discovery in social networks. In L. P. Kaelbling and A. Saffiotti, editors, *IJCAI*, pages 786–791. Professional Book Center, 2005.
11. D. Newman, C. Chemudugunta, P. Smyth, and M. Steyvers. Analyzing entities and topics in news articles using statistical topic models. In *Lecture Notes on Computer Science*. Springer-Verlag, 2006.
12. M. F. Porter. *An algorithm for suffix stripping*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.
13. D. Poshyvanyk and A. Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *ICPC*, pages 37–48. IEEE Computer Society, 2007.
14. M. Steyvers, P. Smyth, M. Rosen-Zvi, and T. L. Griffiths. Probabilistic author-topic models for information discovery. In W. Kim, R. Kohavi, J. Gehrke, and W. DuMouchel, editors, *KDD*, pages 306–315. ACM, 2004.
15. S. Ugurel, R. Krovetz, C. L. Giles, D. M. Pennock, E. J. Glover, and H. Zha. What’s the code? automatic classification of source code archives. In *Proceedings of the eigth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 632–638, 2002.
16. N. Wilde, M. Buckellew, H. Page, V. Rajlich, and L. Pounds. A comparison of methods for locating features in legacy software. *Journal of Systems and Software*, 65(2):105–114, 2003.
17. W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. Sniafl: Towards a static noninteractive approach to feature location. *ACM Transactions on Software Engineering and Methodology*, 15(2):195–226, April 2006.