Mining Concepts from Code with Probabilistic Topic Models

Erik Linstead, Paul Rigor, Sushil Bajracharya, Cristina Lopes, Pierre Baldi
Donald Bren School of Information and Computer Sciences
University of California, Irvine
Irvine, CA 92697-3425 USA
{elinstea,prigor,sbajrach,lopes,pfbaldi}@ics.uci.edu

ABSTRACT

We develop and apply statistical topic models to software as a means of extracting concepts from source code. The effectiveness of the technique is demonstrated on 1,555 projects from SourceForge and Apache consisting of 113,000 files and 19 million lines of code. In addition to providing an automated, unsupervised, solution to the problem of summarizing program functionality, the approach provides a probabilistic framework with which to analyze and visualize source file similarity. Finally, we introduce an information-theoretic approach for computing tangling and scattering of extracted concepts, and present preliminary results.

Categories and Subject Descriptors

I.2.m [**Computing Methodologies**]: Artificial Intelligence—*Miscellaneous*

General Terms

Algorithms, Experimentation

Keywords

program understanding, topic models, mining software

1. INTRODUCTION

As the availability of open source software repositories continues to grow, so does the need for tools that can automatically analyze these repositories on an increasingly larger scale. The automated analysis of such repositories is important, for instance, to understand software structure, function, complexity, and evolution, as well as to identify relationships between humans and the software they produce. Tools to mine source code for functional concepts and organization are also of interest to private industry, where they can be applied to such problems as in-house code reuse and refactoring.

ASE'07, November 5–9, 2007, Atlanta, Georgia, USA.

Automated topic modeling has been successfully used in text mining and information retrieval where it has been applied to the problem of summarizing large text corpora. Recent techniques include Latent Dirichlet Allocation (LDA), which probabilistically models text documents as mixtures of latent topics, where topics correspond to key concepts presented in the corpus [2]. These more recent approaches have been found to produce better results than traditional methods such as latent semantic analysis (LSA) [3].

Despite the capabilities of LDA, its applications have been limited to traditional text corpora such as academic publications, news reports, and historical documents [8]. At the most basic level, however, a code repository can be viewed as a text corpus, where source files are analogous to documents and tokens to words. Though vocabulary, syntax, and conventions differentiate a programming language from a natural language, the tokens present in a source file are still indicative of its function (ie. its topics). In this paper we develop and apply LDA, for the first time, to extract concepts from software data and demonstrate the results on 1,555 projects from SourceForge and Apache consisting of 19 million source lines of code (SLOC). We then leverage the probabilistic framework of LDA to perform concept-analysis and visualization of software similarity, and develop an information-theoretic approach for computing concept scattering and file tangling in an automated manner.

2. METHODOLOGY

Here we first describe the code repository used for the development and empirical analysis of LDA applied to software. We then give a brief overview of the underlying mathematics of the algorithm, as well as describe our methodology and tools for adapting and applying LDA to source code.

2.1 Test Data

In addition to demonstrating the effectiveness of LDA to mine concepts from code, it is equally important to show that the method can scale to arbitrarily large projects or code repositories. Elsewhere [1], we have described Sourcerer, an extensive infrastructure that we have built in order to crawl, download, store, organize, and search large software repositories. A highly configurable crawler allows us to specify the number and types of projects desired, as well as the host databases that should be targeted (eg. SourceForge, Freshmeat) and to proceed with incremental updates in an automated fashion. Once target projects are downloaded, a depackaging module uncompresses archive files while saving useful metadata (version, url, etc) in a database.

Using our infrastructure we downloaded 1,555 projects from SourceForge and Apache. For consistency, we targeted Java projects only, although our approach is independent of programming language. Analysis shows that this test set consists of approximately 113,316 source files and over 19 million lines of code.

2.2 Latent Dirichlet Allocation

In the LDA model for text, the data consists of a set of documents. The length of each document is known and each document is treated as a bag of words. Let D be the total number of documents, W the total number of distinct words (vocabulary size), and T the total number of topics present in the documents. While non-parametric Bayesian and other methods exist to try to infer T from the data, here we assume that T is fixed (e.g. $T = 100$).

The model assumes that each topic t is associated with a multinomial distribution $\phi_{\bullet t}$ over words w, and each document d is associated with a multinomial distribution $\theta_{\bullet d}$ over topics. More precisely, the parameters are given by two matrices: a $T \times D$ matrix $\Theta = (\theta_{td})$ of document-topic distributions, and a $W \times T$ matrix $\Phi = (\phi_{wt})$ of topic-word distributions. Given a document d containing N_d words, for each word the corresponding $\theta_{\bullet d}$ is sampled to derive a topic t, and subsequently the corresponding $\phi_{\bullet t}$ is sampled to derive a word w. A fully Bayesian model is derived by putting symmetric Dirichlet priors with hyperparameters α and β over the distributions $\theta_{\bullet d}$ and $\phi_{\bullet t}$. For instance, the prior on $\theta_{\bullet d}$ is given by

$$D_\alpha(\theta_{\bullet d}) = \frac{\Gamma(T\alpha)}{(\Gamma(\alpha))^T} \prod_{t=1}^{T} \theta_{td}^{\alpha-1}$$

and similarly for $\phi_{\bullet t}$.

The probability of a document can then be obtained in a straightforward manner by integrating the likelihood over parameters ϕ and θ and their Dirichlet distributions. The posterior can be sampled efficiently using Markov Chain Monte Carlo Methods (Gibbs sampling) and the Θ and Φ parameter matrices can be estimated by maximum a posteriori (MAP) or mean posterior estimate (MPE) methods.

Once the model has been formalized, applying LDA to software requires tools to pre-process source code and convert it into compatible representations for the algorithm, required as input parameters. Of these parameters, the most important is the word-document matrix, which represents the occurrence of words in individual documents.

To produce the word-document matrix for our input data we have developed a comprehensive tokenization tool tuned to the Java programming language. This tokenizer includes language-specific heuristics that follow the commonly practiced naming conventions. For example, the Java class name "QuickSort" will generate the words "quick" and "sort". All punctuation is ignored.

As an important step in processing source files our tool removes commonly occurring stop words. For the purposes of our mining task we augment a standard list of stop words used for the English language (e.g. and, the, but, etc) to include the names of all classes from the Java SDK (eg. ArrayList, HashMap, etc). This is done to specifically avoid extracting common topics relating to the Java collections framework, thus focusing the analysis on what the code is doing rather than how it is doing it.

For our test data the tokenizer yields a vocabulary of 411,561 distinct words. The result is then a 411,561 x 113,316 word-document matrix such that entry [i,j] corresponds to the number of times word i occurs in document j.

LDA is run on the input matrix with additional input parameters specifying that 100 topics should be extracted from the code (a number determined experimentally by varying the number of topics between 20 and 250 and analyzing the results for interpretability). The number of iterations, i, to run the algorithm is determined empirically by analyzing results for i ranging from 500 to several thousands. The results presented in the next section are derived using 3,000 iterations, which were found to produce interpretable topics in a reasonable amount of time. In total the process of parsing and topic modeling requires several days to run to completion on a single Sun SunFire X2200 M2 Server, using two dual-core AMD Opteron processors with 8GB of RAM.

As output the algorithm produces a document-topic matrix specifying the number of times a document was assigned to each of the 100 topics extracted from the code. The topics themselves are defined by distributions over words.

3. RESULTS

A representative subset of 7 topics extracted via LDA modeling on our test data is given in Table 1. Each topic is defined by several words associated with the topic concept. For example, the words for topic 4 reasonably describe a concept related to multi-threaded programming. Here we limit the number of words to 5 due to space constraints, but any number, subject to the size of the vocabulary, can be chosen in practice. The probability of word assignment to the topic is also given.

Examining the topic column of the table it is clear that various functional domains are represented. Topic 1 clearly corresponds to database programming, topic 2 to file processing, topic 3 to networks, topic 4 to multi-threading, topic 5 to event listeners, and topic 6 to java server pages and web programming. Topic 7, logging, is particularly interesting as it corresponds to a common cross-cutting concern from aspect-oriented programming. It is worth reiterating that these topics are determined automatically by LDA, and require no manual annotation to the source code.

While space prevents us from listing all topics extracted from each dataset, several trends reveal themselves when all results are considered. Though the majority of topics can be intuitively mapped to their corresponding domains, some topics are too noisy to be able to associate any functional description to them. For example, one topic extracted from our data consists of Spanish words unrelated to software engineering which seem to represent the subset of source files with comments in Spanish. Other topics appear to be very project specific, and while they may indeed describe a function of code, they are not easily understood by those who are only casually familiar with the software artifacts in the codebase. In general noise appears to diminish as repository size grows. Noise can be controlled to some degree with tuning the number of topics to be extracted.

In addition to concept extraction, LDA provides an intuitive solution for computing the similarity between two source files by comparing their respective distributions over topics using the document-topic matrix. Several metrics are possible for this purpose, but one of the most natural measures is provided by the symmetrized Kullback-Leibler (KL)

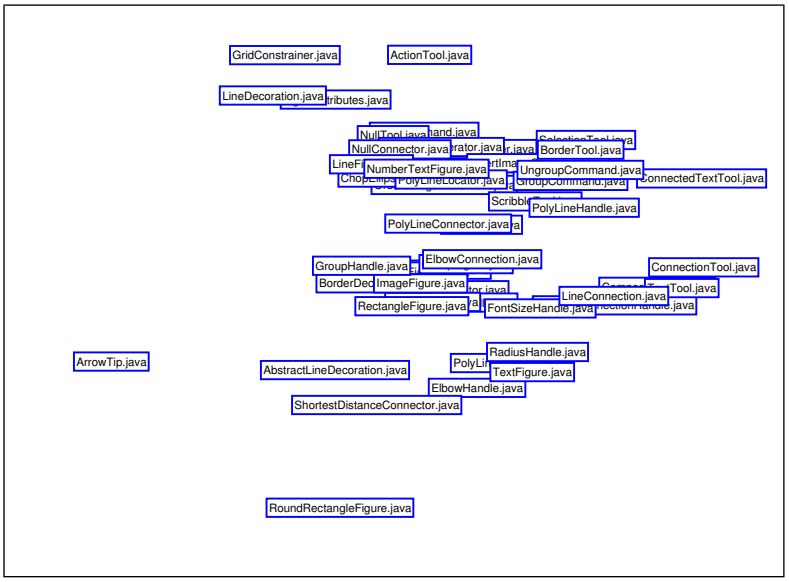

Figure 1: Subset of JHotDraw Files Clustered by KL Divergence

divergence. The resulting similarities can then be visualized using multidimensional scaling (MDS), providing a convenient and concise way to quickly summarize file similarity and project organization. Figure 1 presents one such MDS for a subset of 50 files from the JHotDraw project. The boxes represent individual files, and are arranged such that files with similar topic distributions are nearest one another. For example, by examining the figure one easily see that the files UnGroupCommand.java and GroupCommand.java are very similar, while files such as ArrowTip.java and ConnectionTool.java have little in common in terms of functional concepts. A key advantage of this approach is that topics are derived from actual source files, and thus provide a realistic and accurate basis for comparison.

Two other important distributions that can be retrieved from LDA are the distribution of topics across documents, and the distribution of documents across topics. For each word, the topic-word distribution can be normalized to derive the corresponding word-topic distribution over topics $\phi^*_{w\bullet} = (\phi_{wt} / \sum_t \phi_{wt})$. By averaging the word-topic distributions of all the words contained in a document d, we can derive the document-topic distribution $\psi_{d\bullet} = \sum_{w \in d} \phi^*_{w\bullet} / N_d$ corresponding to the distribution over topics associated with the document d. Likewise, for a given topic t, we can derive the corresponding distribution $\psi^*_{\bullet t} = (\psi_{dt} / \sum_d \psi_{dt})$ over documents. The entropies of these distributions

$$H(t) = H(\psi^*_{\bullet t}) \quad \text{and} \quad H(d) = H(\psi_{d\bullet})$$

provide an automated and novel way to precisely formalize and measure topic scattering and document tangling respectively. Scattering and tangling are two fundamental concepts of software design, put forward in intuitive form by AOP[4], which had proved to be hard to quantify and measure in previous practice. Measurements of scattering and tangling provide valuable information to software architects, for instance in terms of changing the modularity of the code (refactoring), although application of these concepts is beyond the scope of this paper.

Table 2 presents tangling results over 100 topics for a small selection of source files from Eclipse 3.0. As one would expect, source files implementing general project features have high tangling, whereas source files corresponding to small or specialized features have low tangling. For example, a source file implementing features for general workspace management has an entropic tangling of 5.96 bits, while a document corresponding to a specific exception has only 3.30 bits of entropy, with 6.64 bits being the maximum entropy for 100 topics. Similar observations can be made with respect to the scattering of topics, with general concepts such as text manipulation occurring in a large number of source files, and specific concepts such as unit testing occurring in a much more concentrated number of files. While space does not permit the display of the results of these calculations, here we simply note that these entropies can be calculated automatically at different levels of granularity and that, for instance, large entropic scattering can reliably identify crosscutting software engineering concerns.

4. RELATED WORK

The notion of modeling source code with topics is not a new one. For instance, [9] explores code topic classification using support vector machines, but the technique is significantly different from our own. Firstly, a training set must be manually partitioned into categories based on project metadata. Topics (consisting of commonly occurring keywords) are extracted for each category, and used to form features on which to train the model. The training and testing set comprise only 100 and 30 projects respectively. The approach in [7] uses latent semantic analysis to locate concepts in code. The goal is to enhance software maintainability by easily identifying related pieces of code in a software product that work together to provide a specific functionality (concept), but may be described with different keywords (synonyms, etc). In this sense the work shares some of our goals, but does not consider the problem of automatically extracting topic distributions from arbitrary amounts of source code. The work in [5] makes progress in this area, this time using

Table 1: Selected Topics with Word Probabilities

Topic Number	Topic Words With Probabilities
1	sql 0.10167 database 0.05753 update 0.03423 jdbc 0.02837 connection 0.01899
2	file 0.15861 path 0.15815 dir 0.05695 directory 0.04789 filename 0.02962
3	server 0.10314 client 0.06729 host 0.05388 address 0.03657 port 0.03569
4	current 0.07450 pool 0.03590 run 0.02940 thread 0.02889 start 0.02751
5	listener 0.18784 event 0.11507 change 0.08566 remove 0.03827 fire 0.02781
6	tag 0.17629 page 0.14592 jsp 0.05015 jspx 0.03705 body 0.03597
7	log 0.26697 debug 0.13044 logger 0.11477 level 0.06333 logging 0.03249

LSA to cluster related software artifacts. However, new approaches for defining topic scattering and document tangling are not considered. Recently, [6] has applied author-topic models to source code to extract developer contributions. Finally, several attempts have been made to precisely define topic scattering or document tangling, for instance using fan-in/fan-out considerations, none of which so far could be automatically applied to large repositories.

5. CONCLUSIONS

Here we have leveraged the effectiveness of Latent Dirichlet Allocation to automatically extract functional concepts, in the form of topics, from source code. The methods developed are applicable at multiple scales, from single projects to Internet-scale repositories. Results indicate that the algorithm produces reasonable and interpretable topics and document-topic assignments. The probabilistic relationships between topics and documents that emerge from the model naturally provide an information-theoretic basis to define and compare program similarity, topic scattering, and document tangling with potential applications in software engineering ranging from code reuse to software refactoring.

Table 2: Tangling for Selected Eclipse 3.0 Files

File Name	Entropy (bits)
IWorkSpace.java	5.9884
WorkSpace.java	5.9636
Platform.java	5.9534
ProblemHover.java	3.49
AnnotationHover.java	3.40
AntSecurityException.java	3.30

6. ACKNOWLEDGMENTS

Work in part supported by National Science Foundation MRI grant EIA-0321390 and a Microsoft Faculty Research Award to PB, as well as National Science Foundation grant CCF-0725370 to CL and PB.

7. REFERENCES

[1] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In *OOPSLA Companion*, pages 681–682, 2006.

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

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

[4] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[5] A. Kuhn, S. Ducasse, and T. Girba. Semantic clustering: Identifying topics in source code. *Information and Software Technology (to appear)*, 2006.

[6] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi. Mining eclipse developer contributions via author-topic models. *MSR 2007: Proceedings of the Fourth International Workshop on Mining Software Repositories*, 0:30, 2007.

[7] 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.

[8] D. Newman and S. Block. Probabilistic topic decomposition of an eighteenth-century american newspaper. *J. Am. Soc. Inf. Sci. Technol.*, 57(6):753–767, 2006.

[9] S. Ugurel, R. Krovetz, and C. L. Giles. What's the code?: automatic classification of source code archives. In *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 632–638, New York, NY, USA, 2002. ACM Press.