Improving Topic Model Source Code Summarization

Paul W. McBurney, Cheng Liu, Collin McMillan, and Tim Weninger
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN, USA
{pmcburne, cliu1, cmc, tweninge}@nd.edu

ABSTRACT

In this paper, we present an emerging source code summarization technique that uses topic modeling to select keywords and topics as summaries for source code. Our approach organizes the topics in source code into a hierarchy, with more general topics near the top of the hierarchy. In this way, we present the software's highest-level functionality first, before lower-level details. This is an advantage over previous approaches based on topic models, that only present groups of related keywords without a hierarchy. We conducted a preliminary user study that found our approach selects keywords and topics that the participants found to be accurate in a majority of cases.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.9 [**Software Engineering**]: Management—*Productivity*

General Terms

Algorithms, Documentation

Keywords

Source code summarization, software topic models

1. INTRODUCTION

Source Code Summarization is the task of creating a brief description of a section of source code [2, 17, 18, 24, 26]. This description explains the key functionality of the source code, such as the features the code implements or how the code is related to other parts of the software. Programmers read these descriptions to gain an understanding of the code quickly, and the descriptions are popular in many types of documentation, such as JavaDocs [15]. Unfortunately, at present nearly all of these descriptions are written manually. This manual process is a problem because a typical program may have descriptions summarizing each method, class, and file in the program. For a large program, developers need to spend a correspondingly large amount of time writing and maintaining each description by hand. Numerous studies have shown that this manual process is expensive, and that programmers take shortcuts when possible [6, 13, 8, 22].

Software engineering research has begun to yield automated source code summarization techniques [24]. One prominent strategy is to use a *topic model*. A topic model is a statistical model in which words are associated with other words based on their co-occurrence in documents. Different groups of the words called topics are assigned a probability of being associated with each documents. In software, each document is a section of code (e.g., a Java class), and the words are terms in that code (e.g., identifier names or words in comments). Code summarization techniques based on topic models are described extensively in software engineering literature [19]. But as a recent study by Panichella *et. al* points out, these techniques often "have rather low performance when applied on software data" [19].

Panicella *et. al* also point out that this low performance is due to the differences between source code and natural language. At the same time, other studies have shown that the process of reading source code is different than reading natural language, involving a different mental process for comprehension [4]. One of the key differences is that when programmers read source code summaries, they look for the highest-level functionality performed by the code, and then seek to understand how lower-level functionality supports the higher-level features [14, 12]. This mental process is akin to the process of understanding hierarchical data such as web content [29]. Specialized hierarchical topic models have been designed for this data. Unfortunately, current applications of topic models to software do not consider these specialized techniques. The low performance of standard topic modeling techniques on software may be improved by using the specialized hierarchical techniques.

In this paper, we present an emerging source code summarization technique that organizes the topics in source code into a hierarchy. At the top of the hierarchy are topics which describe the highest-level functionality, while lower-level supporting functionality are lower in the hierarchy. For example, a topic "play sound" would be a parent in the hierarchy to topics "decode mp3" and "open files". Our technique employs the HDTM algorithm described by Weninger *et. al* [29] to extract a topic hierarchy for a software system, then we display the hierarchy to programmers in a navigable web interface. We conducted a preliminary user study to explore the benefits and drawbacks of the technique, and to guide future development.

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.
ICPC '14 ERA Hyderabad, Andhra Pradesh, India
Copyright 2014 ACM 978-1-4503-2879-1/14/05 ...$15.00.

2. BACKGROUND

This section presents the problem we target and the topic modeling algorithm we use for our approach. Note that this algorithm has been evaluated elsewhere; we mention it here because it is a critical component of our work.

2.1 The Problem

The problem we target in this paper is that current topic modeling approaches in software engineering do not explicitly provide information about how the topics are related. The topics extracted by current approaches are typically lists of keywords related to one feature in the software. For example, "midi file reader." These lists can be helpful to programmers [7], but what they lack is contextual information about how different features are connected. For example, "midi file reader" may be a component of a higher-level feature such as "music player."

Our work has the potential to impact many areas of program comprehension research. Topic models are widely-used in software engineering [16, 19]. Unfortunately, the main weakness to topic modeling approaches is that there is no agreed-upon technique for labeling the topics so that those topics can be understood in context [28, 16]. In other words, currently a programmer can read topics to determine what features are implemented in software, but it is difficult for the programmer to determine how those features interact, or which features are the most important. Our work has the potential to improve these current approaches by providing a way for programmers to gain this knowledge via a hierarchy of topics.

2.2 HDTM Topic Modeling Algorithm

Given a graph of documents (vertices) and links (directed edges) $G = V, E$ and a root vertex $r \in V$, the Hierarchical Document Topic Model (HDTM) uses a non-parametric Bayesian graphical model to generate a **hierarchy** of the documents. The details of this algorithm can be found in [29]; we describe a general overview of the algorithm here. HDTM defines a document hierarchy to be a directed graph such that each vertex/document has one and only parent document except the root vertex/document, which has no parent. Because HDTM takes as input any directed graph – where vertices often have several adjacent edges – the overarching goal of HDTM is to find the single-best parent for each vertex. In other words, hierarchy-edges are selected from the set of existing edges; edges are never created.

HDTM samples hierarchy-edges using the random walk with restart (RWR aka personalized PageRank) Stochastic method. RWR starts at the root vertex and stochastically 'walks' the graph picking a random number x at each successive vertex. When RWR is 'unlucky', *i.e*, $x < \gamma$, the walker 'restarts' at the root node and stochastically walks again. In this way each vertex in the graph is assigned a probability of being reached via some adjacent edge, *i.e*, from some parent-vertex. These probabilities are be updated in the term sampling step.

HDTM requires that each vertex represents some multinomial distribution. For our purposes we represent each vertex as a multi-set, or bag, of words corresponding to the text in a function's comment section. Conceptually speaking, every time that the random walker chooses an edge it probabilistically chooses to carry some words with it, and every time the walker arrives at a new vertex it probabilistically deposits some of the words that its carrying. The act of carrying and depositing terms changes the multinomial distribution of the vertex, thereby changing the probability that the walker will visit again.

So HDTM operates in two steps: 1) By walking the graph based on the global restart probability γ and each vertex's multinomial distribution, and 2) by updating the multinomial distribution when the walker deposits terms and picks terms to carry forward. Step 1 and 2 alternate using a Markov chain Monte Carlo algorithm called Gibbs sampling. Several thousand Gibbs-iterations are usually required before a hierarchy emerges.

The resulting hierarchy describes the most probable parent for each vertex, but more importantly the distribution of terms inside each vertex will have changed such that vertices higher in the hierarchy contain more general terms, and vertices lower in the hierarchy will contain terms more specific to their location. All vertices at all levels will contain terms that are the most descriptive of their respective subtrees. Thus, HDTM creates not only a document-hierarchy, but also a topic-hierarchy as well.

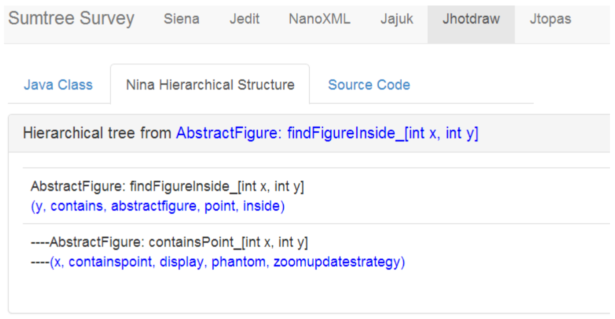
3. OUR APPROACH

This section describes our approach to summarization. Generally speaking, our approach works in four steps: 1) we represent a software project as a call graph, 2) we process the call graph to prepare it for a topic model, 3) we use the HDTM algorithm with this processed call graph, and 4) we display the hierarchical structure of the topics in a web interface. The following details the procedure:

In order to create topic model of a Java project, we first need to represent that Java project as a directed graph. Defining a Java project as a graph requires defining the graph's *nodes* and *edges*. The *nodes* of the graph are Java methods within a project. The *edges* connect the nodes to one another in a meaningful way. In our graph, the edges are defined by the method calls within a Java project. Each edge originates from the calling method, and terminates at the method called. In this way, we can represent any given Java project as a directed graph.

However, in our study, we found that in many Java projects, the generated graph is disconnected. This means that there does not exist a single method that is connected either directly or indirectly to all other methods via the directed call graph. This presents a problem to topic modeling, as in an unconnected graph, it is impossible to create a single hierarchical structure that contains every method in a given project. To rectify this, for each Java project, we create a *phantom* node. This phantom node is connected to every other node in the directed graph. While the phantom node does not represent an actual method in a project, it can be understood as though it were a method that *calls* every other method in the project and is itself *called by* every other method in the project. The phantom node ensures that the directed graph representing a Java project is fully connected.

Using this generated graph, we can use the HDTM algorithm described by Weninger *et. al* to create a topic model. To do this, we define each node in the graph, meaning each method in a Java project, as a document in our topic model. The topics within each document are simply the list of keywords in the code of each method. We chose not to include words from author comments within our topics. This is be-



Hierarchical tree from AbstractFigure: findFigureInside_[int x, int y]

AbstractFigure: findFigureInside_[int x, int y]
(y, contains, abstractfigure, point, inside)

----AbstractFigure: containsPoint_[int x, int y]
----(x, containspoint, display, phantom, zoomupdatestrategy)

Figure 1: Example display for the `findFigureInside` method in jhotdraw. The child method `containsPoint` is called by `findFigureInside`. `findFigureInside` was considered the most suitable parent node of `containsPoint` by HDTM.

cause we seek to develop an automatic summarization tool that does not rely manually written documentation. We then run using HDTM with $\gamma = .75$. The value was chosen via experimentation, as an examination to determine the best possible *gamma* value is beyond the scope of this pilot study. Additionally, the phantom node is selected as the top node on our tree. This was done to ensure consistency, as the artificially created phantom node is the only node certain to exist regardless the input Java project.

HDTM creates a hierarchical tree structure from our Java project. The hierarchical tree structure originates from our phantom node. While each node, other than the phantom node, still represents a method in the input Java project, the topics in these nodes are no longer keywords from just that method. As a result of HDTM, each node contains an ordered list of topics selected from that method and all descendants of that method. This allows common keywords of child methods to be visible in the parent method's topics, even if that particular keyword does not appear in the parent method. In this way, we create a hierarchy of topics, with nodes that contain common and general terms higher in the tree. By contrast, nodes containing specific methods that do not provide broad understanding of the project will be lower in the tree.

3.1 Example

To illustrate how the hierarchical tree structure is used to find keywords, we present an example taken from our pilot study (see Section 4). As shown in Figure 1, the method **`findFigureInside`** in the jHotdraw class `AbstractFigure` is the parent of the method `containsPoint`. This shows that, according to HDTM **`findFigureInside`** is the most similar method that calls `containsPoint` in jHotDraw.

The keywords from `containsPoint`, namely *contains*, *point*, and *inside* appear in the top five keywords in **`findFigureInside`**. Each of these keywords help communicate the purpose of the method `findFigureInside`, which, according to author comments, returns a figure that *contains* the given *point inside* of it. Because **`findFigureInside`** relies on a call from `containsPoint`, and both methods contain three topics, the listed topics are higher in the hierarchical tree, meaning they are general to both methods. This also means that while none of those three topics appear in the list of top five topics in `containsPoint`, `containsPoint` still can be considered related to those topics because it is a child in a branch below those topics.

4. PRELIMINARY EVALUATION

We conducted a pilot study to evaluate our approach. We recruited three participants for the study, all graduate students with an average of 5 years experience in Java programming.

In this pilot study, we gave the users with the interface shown in Figure 1. The user would see the top 5 topic keywords for a given method, as well as the hierarchy of all child methods and topics. We asked the participants to explain in their own words what they believe the method does without looking at the source code. After that, we had the participants examine the source code and then summarize the method in their own words. Based upon their summarization, we asked the participants three multiple choice questions.

Q$_1$ How accurately do you feel the given keywords describe the method? Very (accurately, somewhat accurately, somewhat inaccurately, very inaccurately)

Q$_2$ How much do you agree with this statement: "I understand the method's overall purpose in this project."? (Strongly agree, somewhat agree, somewhat disagree, strongly disagree)

Q$_3$ If you were to select five keywords to describe the method, you would have used...(All of the same keywords, most of the same keywords, only few of the same keywords, none of the same keywords)

The rationale for Q$_1$ is to determine if the participant was able to predict what the method would do based upon the keywords and hierarchy given. If participants find the keywords to be accurate, it implies that the keywords depict the purpose of the method to the user. The rationale for Q$_2$ is to determine if our approach does convey meaningful contextual information. If participants largely agree with the statement, it implies our approach successfully communicates how a method fits into a project. The rationale of Q$_3$ is to determine if the participants agree with our keyword selection. If they agree with all or most of our keywords, then this implies topic modeling can result in the mostly the same keywords being selected as the more expensive and time consuming process of using a human expert.

Our participants were given methods from a randomly selected group of methods used in our prior work [21]. The projects studied are nanoXML, jTopas, Siena, jHotdraw, jajuk, and jEdit. Two participant answered questions for all 15 methods given, while one participant only had time to answer questions for 9 methods. In total, this gives us 39 answers to each question.

4.1 Quantitative Results

For Q$_1$, our participants felt the keywords were at least "somewhat accurate" 76.9% of the time, with a plurality of 43.6% of the time the answer being "very accurate." Participants selected "very inaccurate" only 12.8% of the time. For Q$_2$, participants selected that they "strongly agreed" that they understood the method's purpose a majority of the time: 53.8% of the time. The participants "somewhat disagreed" or "strongly disagreed" 17.9% of the time. For Q$_3$, participants most frequently said they would use "most of the same keywords" 43.6% of the time, and "all of the same keywords" 33.3% of the time. Only once in all 39 responses did a participant say they would use none of the same keywords, which is equal to 2.6% of the time.

In summary, our quantitative results are promising but point to needed improvement. They suggest that keyword selection using topic modeling can be accurate and can help a reader of the source code understand a methods purpose. Additionally, readers agree with most of the keywords selected 76.9% of the time, which implies that our rapid automated approach can very frequently produce results that human experts agree with.

4.2 Qualitative Results

Throughout the study, we asked participants to provide general comments for each method they summarized. Most of the comments referred to specific keyword selections. Participants suggested certain selected keywords should not have been included, and suggested other keywords. For example, "buffer keyword should be included" and "As a keyword, properties does not make sense to me."

There were some cases where the keywords' purposes were not understood until the source code context was seen. For example, one comment on the `createElement()` function in `XMLElement` class in NanoXML read, "From the keywords, it was clear that the method would generate an XML element, but it was not clear the method would generate an empty element." While there were negative comments on particular details, on the whole, participants liked our interface, with one participant saying "I think the interface is great."

4.3 Threats to Validity

One source of a threat to external validity. An interface expert was nearby when the participants took the exam. This was to ensure if the participants were not at any point confused by the web interface, the expert could help answer any questions. The expert did not at any point answer any questions about the keywords or Java source code participants were asked to summarize. We felt a participant not knowing how to correctly interact and interpret the interface would be a larger threat to validity than the possible bias created by an experts presence. The expert did not interact with the user other than to briefly explain the interface, nor did the expert monitor the participants' progress in the survey.

5. RELATED WORK

Source code summarization is a relatively new technology with relatively little discussion in the literature. Haiduc *et al.* described one approach based on a Vector Space Model (VSM), in which a summary comprised of the n keywords with the highest term-frequency/inverse-document-frequency scores [10]. This approach has been independently confirmed to be effective [5, 7]. Most recently, Rodeghero *et al.* improved the algorithm by using data from an eye-tracking study of professional programmers [21]. Sridhara *et al.* created different approaches, notably using sentence templates for Java methods [25] and extraction of comments for method parameters [26]. Source code summarization is also related to earlier techniques in natural language summarization. Spärck-Jones [23] surveys numerous text summarization techniques, and broadly categorizes them as either "extractive", by generating summaries from the content inside a document [3, 9, 11, 20, 27], or "abstractive", by generating summaries based on external context or related documents [23]. Finally, topic modeling is widely used in software engineering [1, 19] as mentioned in Section 1.

6. CONCLUSION

We have presented an approach to source code summarization using topic modeling. Initial study implies our approach is a viable source code summarization methodology.

7. REFERENCES

[1] P. F. Baldi, C. V. Lopes, E. J. Linstead, and S. K. Bajracharya. A theory of aspects as latent topics. *OOPSLA '08*

[2] H. Burden and R. Heldal. Natural language generation from class diagrams. In MoDeVVa '11. Proceedings.

[3] F. Chen, K. Han, and G. Chen. An approach to sentence-selection-based text summarization. *TENCON '02.*

[4] M. E. Crosby and J. Stelovsky. How do we read algorithms? a case study. *Computer*, 23(1):24–35, Jan. 1990.

[5] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella. Using ir methods for labeling source code artifacts: Is it worthwhile? In *ICPC '12. Proceedings*

[6] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira. A study of the documentation essential to software maintenance. In *SIGDOC '05, Proceedings.*

[7] B. Eddy, J. Robinson, N. Kraft, and J. Carver. Evaluating source code summarization techniques: Replication and expansion. In ICPC '13, Proceedings.

[8] B. Fluri, M. Wursch, and H. C. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In WCRE '07, Proceedings.

[9] J. Goldstein, M. Kantrowitz, V. Mittal, and J. Carbonell. Summarizing text documents: Sentence selection and evaluation metrics. In *SIGIR '99, Proceedings.*

[10] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. On the use of automated text summarization techniques for summarizing source code. In *WCRE '10, Proceedings.*

[11] S. Haiduc, J. Aponte, and A. Marcus, Supporting program comprehension with source code summarization ICSE '10

[12] R. Holmes, R. J. Walker. Systematizing pragmatic software reuse. *ACM TOSEM.*, 21(4):20:1–20:44, Feb. 2013.

[13] M. Kajko-Mattsson. A survey of documentation practice within corrective maintenance. *ESE.* Jan. 2005.

[14] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE TSE.* Dec. 2006.

[15] D. Kramer. Api documentation from source code comments: a case study of javadoc. In *SIGDOC '99, Proceedings.*

[16] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi. Mining concepts from code with probabilistic topic models. In *ASE '07, Proceedings.*

[17] S. Mani, R. Catherine, V. S. Sinha, and A. Dubey. Ausum: approach for unsupervised bug report summarization. *FSE '12*

[18] L. Moreno, J. Aponte, S. Giriprasad, A. Marcus, L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *ICPC '13, Proceedings.*

[19] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *ICSE '13, Proceedings.*

[20] D. R. Radev, H. Jing, and M. Budzikowska. Centroid-based summarization of multiple documents: Sentence extraction, utility-based evaluation, and user studies. In *NAACL-ANLP-AutoSum '00, Proceedings.*

[21] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello. Improving automated source code summarization via an eye-tracking study of programmers. In *ICSE '14, Proceedings.* To appear.

[22] J. Sillito, G. C. Murphy, K. De Volder. Asking and answering questions during a programming change task. *IEEE TSE*

[23] K. Spärck Jones. Automatic summarising: The state of the art. *Inf. Process. Manage.*

[24] G. Sridhara. *Automatic Generation of Descriptive Summary Comments for Methods in Object-oriented Programs.* PhD thesis, University of Delaware, Jan. 2012.

[25] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *ASE '10, Proceedings.*

[26] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Generating parameter comments and integrating with method summaries. In *ICPC '11, Proceedings.*

[27] D. Wang, S. Zhu, T. Li, and Y. Gong. Comparative document summarization via discriminative sentence selection. *ACM Trans. Knowl. Discov. Data*

[28] T. Wang, G. Yin, X. Li, and H. Wang. Labeled topic detection of open source software from mining mass textual project profiles. In *SoftwareMining '12, Proceedings.*

[29] T. Weninger, Y. Bisk, and J. Han. Document-topic hierarchies from document graphs. In *CIKM '12, Proceedings.*