JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 0000; **00**:1–27
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr

Automated Feature Discovery via Sentence Selection and Source Code Summarization

Paul W. McBurney* and Cheng Liu and Collin McMillan

Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN, USA
{*pmcburne, cliu7, cmc*}*@nd.edu*

SUMMARY

Programs are, in essence, a collection of implemented features. Feature Discovery in software engineering is the task of identifying key functionalities that a program implements. Manual feature discovery can be time-consuming and expensive, leading to automatic feature discovery tools being developed. However, these approaches typically only describe features using lists of keywords, which can be difficult for readers who are not already familiar with the source code. An alternative to keyword lists is sentence selection, in which one sentence is chosen from among the sentences in a text document, to describe that document. Sentence selection has been widely studied in the context of natural language summarization, but is only beginning to be explored as a solution to feature discovery. In this paper, we compare four sentence selection strategies for the purpose of feature discovery. Two are off-the-shelf approaches, while two are adaptations we propose. We present our findings as guidelines and recommendations to designers of feature discovery tools. Copyright © 0000 John Wiley & Sons, Ltd.

Received ...

1. INTRODUCTION

Feature discovery in software engineering is the task of identifying the key functionality that a program implements [13, 31]. A "feature" is defined as a user-visible characteristic about the behavior of a program (e.g., "plays mp3 files") [6]. The notion of a feature is important because programs are often thought of as implementing sets of features: Software engineers determine what features to implement through domain analysis [19] and requirements elicitation [15]. Engineers link feature descriptions to various software artifacts through traceability [26]. Acceptance testing confirms that software implements a required set of features [3]. And regulatory requirements often dictate that specific features be implemented for safety or privacy reasons [7]. In all these areas, feature discovery is a critical task because of the need to know what features a given piece of software implements.

Unfortunately, at present feature discovery is a largely manual process. Programmers typically have three options for feature discovery [41]: First, programmers may turn to software documentation, such as requirements documents. Second, programmers may read the source code of the program, and execute the code with different inputs. Third, programmers may communicate directly with the authors of the source code. In the ideal case, the documentation will include an explicit list of features. But the ideal case is often not realistic because the feature lists are often either out-of-date or incomplete [11, 18]. The alternative of communicating with the code authors is also often not realistic, since the authors may not even be known [41]. Therefore, programmers conducting feature discovery are forced to

manually read the source code or interact with the program. This manual feature discovery process is often extremely expensive, such as in domain analysis where tens or even hundreds of programs must be processed [13, 19].

Automated techniques for feature discovery have been proposed as an alternative to the manual approach [13, 50, 8, 21, 27, 38, 49]. The most-common strategy that has been found to be effective is to use a topic model such as Latent Dirichlet Allocation (LDA) to extract groups of keywords that are associated with different functionality [12]. This strategy will produce lists of keywords ostensibly linked to different features, for example "sound mp3 wav midi" versus "save load file open". The advantage to these approaches is that links are preserved between the lists of keywords and the source code that contains those keywords. The programmer can use these links to roughly divide the program into groups of code that implement different features – some sections of code will be more-related to the "sound mp3 wav midi" feature than the "save load file open" feature.

But the disadvantage is that the groups of keywords are difficult to understand without already understanding the source code. For example, the keywords "sound mp3 wav midi" indicate that some audio functionality is implemented, but the details are obscure. It is difficult to know whether the program plays those file formats, or handles streaming them across a network, or possibly converting from one format to another. Existing automated feature discovery techniques are effective at organizing programs into categories of functionality, but are much less able to provide lists of features that are readable in isolation, without also reading the source code.

In this paper, we compare four sentence selection strategies as automated feature discovery techniques. These techniques produce readable, natural-language sentences about software. The input to the techniques is the source code documentation, in the form of Javadocs, of a program. The output is a list of features in which each feature is described by one sentence. The techniques work by using a sentence selection algorithm to extract one sentence for each of the groups of keywords extracted by a topic model. Where Javadocs documentation is available, the tools select sentences from this documentation. Two of the sentence selection algorithms we examine have been proposed and evaluated elsewhere. Two others we adapt from related tools and propose as alternatives. Further, we conduct a follow-up study to improve the results of our Overlap approach.

We found that current methods to generating natural language feature lists from software documentation currently do not meet programmer expectations. However our two proposed approaches are preferred when compared directly to existing textual summarization approaches. Additionally, we found that our overlap approach can be effective in giving programmers an idea of the purpose of a given program. Our work lays a foundation for future work in feature discovery by providing four approaches to generating natural language feature lists. Finally, the data for our work is presented in an online appendix* for reproducibility.

2. THE PROBLEM

We address the following gap in the literature regarding program comprehension: there currently exists no fully automatic approach to generate human readable natural language features lists of existing software engineering projects, that have been shown to meet quality standards sufficient for use by programmers. Understanding the features of a software project is necessary for using the project correctly [6, 41]. Manually written documentation can suffer from a number of problems. Manual documentation is time-consuming and expensive to write, leading the documentation to often be incomplete [11, 18] or outdated [14, 42]. Automatic approaches to describe features have produced short keywords that describe features in the

*`http://www.nd.edu/~pmcburne/features/`

DOI: 10.1002/smr

source code, such as "sound mp3 wave midi." However, these descriptions often lack enough specificity to help the reader understand the context of the given keywords.

Natural language sentences that describe features could provide more clear and specific understanding of the features to programmers. Currently, there are a small number fully automatic approaches that summarize Java projects at the method-level granularity using natural language sentences [45, 44, 29]. Additionally, Moreno *et al.* [36, 35] generate natural language summaries to describe Java classes by identifying class stereotypes. However, to the best of our knowledge, no such tools exist to describe the project-level granularity in natural language sentences. In this paper, we target the problem of describing the features at the system-level granularity of software tools. Natural language features lists could greatly assist programmers in feature discovery, helping to alleviate the large manual effort [13, 19] that programmers currently must put forth to understand a project. Such a tool could reduce effort required in several steps of the software engineering process, such as domain analysis [19], requirements elicitation [15], acceptance testing [3], et al. Our work is intended to be a foundation for automatically generating natural language feature lists.

3. BACKGROUND

This section examines the background of feature discovery. Additionally, this section describes relevant tools to our approach and study. These tools are LSS, LDA, and TLDR.

3.1. Feature Discovery

Feature Discovery is using available software resources, such as documentation, bug reports, and requirements documents, to identify features in a software project. Requirements documents exist to specify features in a project. However, as projects develop over time, features may be added or dropped. This results in documentation becoming outdated or incomplete for the purposes of feature discovery [11, 18]. Automatic approaches to feature discovery have emerged to address this shortcoming. Several different types of approaches have been used for feature discovery, including conditional compilation [8], text mining available documentation [13, 12, 21, 27], and using available software labels [49]. Related to feature discovery is feature location, where the source code relevant to a particular feature is searched for [50]. In the vein of feature discovery, Sridhara *et al.* presented an approach that identifies portions of source code that describe high-level actions in a given project [43]. Our work contributes to feature discovery by mining software documentation in the form of JavaDocs.

3.2. LDA

LDA (Latent Dirichlet Allocation), described by Blei *et al.* [5] is a topic modeling technique to describe a document as a set of topics. Each topic is made up of a list of keywords that can be used to describe the topic. For example, a topic with the keywords "sound mp3 wav midi" would likely encompass portions of a document that address audio files. LDA has become the most-common strategy to extract keywords in order to produce feature lists [12, 21]. Additionally, LDA has been applied to a large number of other software engineering problems including software traceability [1, 25, 37], source code analysis [16, 4], defect prediction [23], and software repository mining [47, 48].

LDA can take a set of documents, or a *corpus*, such as source code documentation for different methods, and derive topics to describe the set of documents. Each topic is made up of several probabilistic keywords, where each keyword has a different weight describing how important it is to the topic. The keywords are pulled from the individual words in the documents, with the topics being created based on co-occurrence of words. Words that occur frequently together are likely to be part of the same topic.

Initially, a user specifies number of topics, n, to be generated. LDA works by first iterating through each document. Each keyword in each document is randomly assigned to one of the n

DOI: 10.1002/smr

topics. Then, for each document d, we iterate through each word w. For each word, we compute the probability that w appears in the topic t, and that the topic t appears in document d. Word w is then assigned to a new topic where the probability of the word being in the new topic is the highest. After a large iterations over the corpus of documents, the topics will become more stable, with fewer words being "moved" around from topic to topic. At this point the topics will contain lists of commonly occurring words.

LDA has several tunable parameters that must be assessed carefully [4]. First, an LDA user must choose the number of topics to create from a corpus of documents. Each corpus must be considered individually. If the number of topics is too large, the topics become unstable, noisy, and too specific to be useful. If the number of topics is too small, key information may be lost, and incompatible concepts may be meshed into a single topic.

Two more parameters in LDA that must be tuned are α and β. α refers to the the strength of the prior belief that all the topics are uniformly distributed throughout the documents. A large α assumes that each document contains a large number of topics. A small α assumes that each document is made mostly of a small number of topics. A large α therefore encourages simple topics with a small number of important keywords [4]. β affects the number of words per topic. A smaller β results in topics containing fewer keywords. A larger β results in topics that contain multiple important keywords. These parameters will be tuned differently depending on the goal of the topic model [4]. In feature location, a larger α would be preferable, as it would encourage more topics-per-document, and increase the recall of identifying the topic within relevant sections of code [4].

3.3. LDA-GA

The LDA Genetic Algorithm (LDA-GA) is an algorithm developed by Panichella *et al.* [38] that determines a near-optimal set of LDA input parameters for a given project. The parameters LDA-GA recommends are α, β, number of topics, and number of iterations. LDA-GA does this by using genetic algorithms to randomly generate, combine, and mutate different LDA input configurations. LDA starts with a set of randomly generated parameters, which can fit a range specified by a user. For example, a user may specify that they want an α no less than .1 and no greater than .9. Additionally, users can set a minimum and maximum number of allowable topics. LDA is then run for each set of random input parameters. Each configuration is then scored on a fitness function. This fitness function scores a configuration based on the topics that configuration produces. Topics that are cohesive, where documents that most fit into the topics are most similar, are considered better. Additionally, a high separation, where documents are dissimilar to the average of documents in another topic, is preferable. The most fit LDA configurations are then used to generate the next generation. These configurations are combined with a small chance for a random mutation. After several generations, the population with stabilize around a particular near-optimal configuration of input parameters. These input parameters can then be used to run LDA on the system.

3.4. LSS

LSS (Lightweight Semantic Similarity) is a metric to determine the similarity two small pieces of text developed by Croft *et al* [9]. LSS was designed to compare two short text items regardless of whether or not they have sentence structure. Specifically, LSS was originally used to compare captions of photographs to determine whether two pictures were on a similar topic. LSS determines the similarity between two pieces of text by considering word *semantics*. *Semantics*, in this case, refers to considering the word's meaning, rather than only considering its literal text.

For example, according to LSS, a sentence "This method plays a sound file" would be semantically very similar to "This function plays an audio item." Even though the literal words in each sentence are not the same, the semantics, or meanings, of these words are nearly identical. These two sentences would be more similar to each other than a sentence like "This

DOI: 10.1002/smr

method calculates the square root of the input," which has nearly no semantic similarity other than referring to the existence of a method.

LSS judges semantic similarity using WordNet [34]. Wordnet is a hierarchical data structure used to classify words and determine similarity between multiple words. Words are classified into synsets. A synset is a set of synonymous words. A synset can, itself, contain multiple synsets. This forms a hierarchy with more general words at the top of the hierarchy, and more specific words lower in the hierarchy. For example "music" would likely be below "sound" in the Wordnet hierarchy, as "music" refers to a particular subset of "sounds." Similarity between words is determined by the distance between each words location in WordNet. Additionally, the depth of two synsets deepest common ancestor is directly proportional to the similarity of two words. This is because terms deeper in Wordnet are more specific, while keywords higher are more general.

LSS compares two sets of text, A and B in 2 basic steps. First, LSS pre-processes the text items. A and B are stripped of all special characters and punctuation. Duplicate tokens within A and B are also removed. During pre-processing, LSS uses Wordnet to identify the synsets for each words between the two text items being compared.

Second, LSS constructs C, which is a concatenation of the terms in A and B. C is then used to create similarity matrix $\mathbf{C}$. Each item $\mathbf{C}_{i,j}$ is equal to the similarity between items i and j, found using WordNet. Using A as an example, we create a weighted term vector $\mathbf{A}$. Each element $\mathbf{A}_i$ is set to the sum of all elements in A compared to C_i, using the similarity matrix $\mathbf{C}$. The same process is repeated for B to produce the weighted term vector $\mathbf{B}$. The similarity between A and B is then defined as the cosine similarity between vectors $\mathbf{A}$ and $\mathbf{B}$. The output will be between 0 and 1, with 1 being an identical match, and 0 being completely semantically dissimilar. In practice, a similarity of 0 is rarely, if ever, found.

3.5. TextRank

TextRank is a graph-based model for text processing created by Mihalcea, et al [33]. TextRank is an unsupervised method that can be used for either keyword or sentence extraction. In this paper, we focus on TextRank's sentence extraction tool. We used an implementation of the TextRank algorithm created by David Adamo [†]. This implementation returns a 100-word summary of the body of text selected from the highest rated sentences.

TextRank works in three basic steps. First, we extract all sentences from a block of text. Each sentence becomes a vertex in a graph. The edges in the graph are then weighted by the similarity. This results in a highly connected graph. A weight of 1.0 between two vertices would mean two identical sentences. The larger the weight of the edge, the more similar the sentences. Edge weight can be determined by any type of similarity metric. In this project, we use the Levenshtein distance [20] to calculate the edge weights. This was selected over more common approaches such as using cosine similarity due to a somewhat better performance in internal preliminary experimentation. This experimentation was done using a selection of Java projects we have used in software studies [29, 28]. These projects include NanoXML, Jedit, and JHotDraw. The final step is to run PageRank to score each vertice. The vertices are then sorted by their PageRank score, and the top scoring vertice sentences are used as the feature list.

3.6. TLDR

TLDR[‡] (Too Long Didn't Read) is a plug-in extension available for various web browsers, including the Google Chrome [§] web browser [¶]. TLDR takes as input a webpage, such as

[†] `https://github.com/davidadamojr`
[‡] `http://tldr.io/browser-extension`
[§] `http://chrome.google.com`
[¶] As of 16 April 2015, the TLDR is no longer functional. TLDR's website and support channels have all been taken down. The app can still be downloaded, but does not function

Prepared using smrauth.cls
DOI: 10.1002/smr

a news article. TLDR uses the PlexiNLP [‖] API, previously Stremor, as a natural language processing tool to automatically generate summaries. TLDR automatically selects sentences from a given web page to return as a summary. Users can customize how large they want the summary to be. Users can choose to have a short summary containing just a few sentences, or larger summaries that contain more of the source material, such as 25%, 50%, or 75% of the original source material. In small summaries, TLDR attempts to isolate the most important sentences from the article. For larger summaries, TLDR attempts to filter out extraneous sentences that do not provide core understanding. TLDR often looks for repeated words, and leverages document format in order to determine important words and phrases. In this study, we use TLDR's "summary" feature, which returns a summary anywhere from 3 to 7 sentences long of the body of text.

4. APPROACH

This section details how we use the supporting technologies described in Section 3 to generate the summaries and feature lists evaluated in our user study. An overview of our approach is illustrated in Figure 1. This section first describes how we collect and preprocess documentation from a Java project. Additionally, we describe the four summarization approaches used in our user study. The two feature list tools, Overlap and LSS, that use LDA are described in Section 4.2. The two textual analysis summarization tools are described in Section 4.3. Both of these approaches, as well as TextRank and TLDR, require the source code to have Javadocs comments. Programs that have very limited or no documentation cannot be effectively analyzed using this approach.

4.1. Documentation Processing

In this section, we will describe how we extract and preprocess Java documentation in order to create feature lists. For consistency, we use the same document extraction process for both LDA-based approaches and textual analysis tools.

Our extraction process is illustrated in Figure 1. Initially, we extract all comments and JavaDocs from the source code of a Java project. All special characters are removed from the extracted text except for periods, question marks, and exclamation points. The exception is made for these characters because they are sentence delimiters. The text is then split into sentences using these delimiters. We do not always split on periods. If, for example, a period is between two numbers with no whitespace, it is likely a version number delimiter. Additionally, each method's Javadocs are separated by linebreaks. This means if a particular Javadoc section does not end with punctuation, as was often the case, we treat the end of that Javadoc as the end of a sentence. In-line comments, which rarely have punctuation at the end to denote the end of a sentence, have a period added. Additionally, if a Javadoc contains 2 or more consecutive linebreaks, we treat those line breaks as the end of a sentence even if there is no punctuation before the line breaks. We cannot treat individual line breaks in Javadocs as the end of sentences, as linebreaks are often added to ensure the entirity of the documentation stays "on screen." After separating sentences, we processed the words in the sentences as follows. First, we split all words on camelcasing. This means identifier names such as "loadFile" become "load file." Finally, we remove Javadocs keywords such as "@param." This is done because they do not add to the sentences semantic information when looked at from the project level granularity.

This forms our list of sentences that are used by the LDA-based approaches and the textual analysis tools to select sentences to include in the feature list. The list of sentences are also used by LDA to create the topic model for the given Java project. Before the sentences are

[‖] `http://www.plexinlp.com/index.html`

DOI: 10.1002/smr

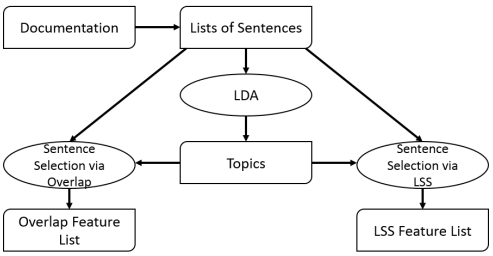

Figure 1. This flowchart shows how we extract a feature list from source code documentation. First, we extract sentences from the documenation. We then create topics from those sentences using LDA. Finally, for each topic, we select a sentence using either Overlap or LSS to create a feature list. The feature is made up of a list of sentences that are most similar to the topics generated by LDA.

used by LDA, we perform some additional text modification. First, we removed a list of *stop words*. *Stop Words* are words that add little semantic information. We considered two types of stop words. We removed English stop words such as "the", "an", etc. Next, we removed Java stop words. These include common keywords in Java such as "return", "if", etc. These words are removed from the sentences before running LDA to ensure we specifically focus on keywords that would contain meaningful semantic information.

4.2. LDA-Based Approaches

This section will explain how we use LDA in our Overlap and LSS feature list generation tools. We will explain how we configure and use LDA to create topic models from sentence lists. We will then explain how we use these topics models to generated a feature list using Overlap and LSS.

4.2.1. Configuring LDA For our approach, we used the parallel C++ implementation of LDA (PLDA) created by Liu *et al* [22]. PLDA produces an output topic model of a specified number of topics. We chose to produce 10 topics for all Java projects. While this number may be too large for simpler projects, we found by inspection that it created a balance of coverage and conciseness for most projects. Before we run PLDA, we remove several stop words, such as "the", "a", "and", et al. We then give the term frequencies of each Java class in a project to PLDA as input. We set the values of the parameters α and β to .1 and .01 respectively. We chose α to be small because we believed that each class (which were treated as documents) would contain a small number of topics, and often only 1. We chose β to be very small because we believed most keywords would only apply to at most one topic, due to the large size of some projects' documentation. Our topic model was created after 2500 iterations, of which 1500 were "burn-in" iterations. The output topic model is then used in our Overlap and LSS approach to select sentences from the sentence list.

4.2.2. Overlap Using the list of sentences extracted from the documentation and the LDA topic model, we create a feature list for a given Java project. We do this by finding the "best" sentence for each topic in the LDA topic model. The "best" sentence is the sentence which scores highest using the Overlap metric. The process by which we create a feature list using Overlap is illustrated in Figure 2.

To calculate overlap between a topic and a sentence, we compare the ten most important words in the given topic to the words in the sentence. First, we remove all the stop words from the sentence that we remove from the LDA model. Then, we define the list of the top ten topic keywords as **A**. Additionally, we define the list of words in the given sentence as **B**. Using these defintions, overlap is calculated using the following formula:

DOI: 10.1002/smr

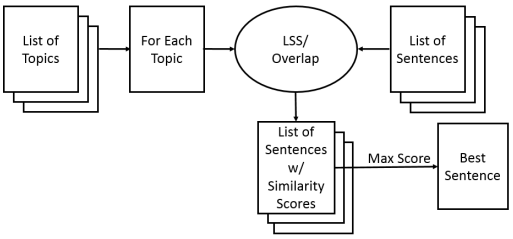

Figure 2. This flowchart shows how we generate a feature list using a list of topics from the LDA topic Model and the list of sentences. This process is typically repeated, for this study, 10 times, once for each topic. The sentence selected for each topic is the one that scores highest with respect to the given similarity metric, either Overlap or LSS.

$$Overlap(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{1}$$

The more words that are shared by the topic and sentence, the more similar the topic and the sentence are. Note that we do not use stemming in our selection of words. If not including stemming results in a decreased performance, we would expect the LSS approach to outperform Overlap. This is because words sharing a common root are usually in the same, or nearly the same, synset in WordNet. For each topic in the LDA topic model, we select the sentence from the sentence list that has the highest overlap score. Because our LDA topic model is typically 10 topics, this will produce 10 sentences. These 10 sentences are used as the feature list.

4.2.3. LSS The process of selecting sentences using LSS to form a feature list is nearly the same as Overlap. We use the same LDA topic model as Overlap for LSS, as well as the same sentence list. The only difference is that, instead of using the Overlap metric to select the best sentences, we use the LSS metric. This will produce a set of sentences with one sentence for each topic. The selected sentence is the most semantically similar sentence to the topics. This allows for sentences with words that are similar in meaning to the topic keywords, even when the words are literally different, to be selected as the best sentence.

4.3. Textual Analysis Tools

Our textual analysis tools, TextRank and TLDR, take in the list of sentences and output a summary, which we use as the feature list for the given Java project. Our textual analysis tools do not make use of LDA. TextRank is described in Section 3.5, and TLDR is described in Section3.6. We use an implementation of TextRank created by David Adamo.[**]. TextRank creates a summary of the sentence list. TLDR was run by opening each sentence list as a text file in the Google Chrome web browser[††]. The TLDR plug-in generated a summary for the page, which we use as the feature list.

5. EVALUATION

In this section, we outline our methodology for evaluating the four approaches for automatic project summarization. We evaluate our two approaches that generate features lists, Overlap and LSS, alongside our approaches for textual summarization, TextRank and TLDR. We define our research questions and describe how we answer the research questions.

[**]`https://github.com/davidadamojr/TextRank`
[††]Google Chrome Version 37.0.2062.103 m

DOI: 10.1002/smr

5.1. Research Questions

In this paper, we conduct a user study to determine how the four project summarization tools perform. We seek to answer the following research questions:

RQ_1 Which automatic project summarization approach do programmers find to be the most accurate, and to what degree?

RQ_2 Which automatic project summarization approach do programmers find to be the most content adequate, and to what degree?

RQ_3 Which automatic project summarization approach do programmers find to be the most concise, and to what degree?

RQ_4 Which automatic project summarization approach do programmers find to be the most readable, and to what degree?

RQ_5 When asked to compare two automatic project summarization approaches directly, which automatic project summarization approach do programmers prefer most?

The rational for RQ_1, RQ_2, and RQ_3 is to find the quality of our project summary approaches. Accuracy, content adequacy, and conciseness are metrics that have been used to measure quality in previous automatic summarization studies [45, 29]. A summary that is accurate will not provide a programmer with false information about a project. A summary that is content adequate includes all or most of the important features within a project. A summary that is concise does not contain trivial information that is unnecessary to understanding the project. The rationale for RQ_4 is to determine how easily programmers can read the output. Given that all four of the approaches rely on sentence selection, it is important to know if combining sentences results in confusion. Possible sources of loss of readability could be having disjointed sentences or pronoun disagreement. The rationale for RQ_5 is to see, when comparing two generated summaries directly, which approach's summaries programmers prefer.

5.2. Data Collection

Initially, we selected 50 Java projects a large repository of SourceForge* Java projects. These projects represent a wide variety of purposes and include a text editor, a music player, and an XML parsing tool, among many other purposes. Six of the projects selected were from our previous work in automatic source code summarization [29]. The other 44 projects were randomly selected from a repository of SourceForge Java projects. Selections were limited to projects with raw source code available that also had a project website. This limitation was added so, in our user study, programmers could quickly get an expert summary of what a given project's purpose is. Four of the 50 of the projects were removed from this list due to a lack of source code documentation.

Of the remaining projects, another 10 were removed because they were too large for our implementation of TextRank to summarize. Our TextRank implementation could generally handle up $\tilde{2}$ million characters from extracted sentences before having memory problems. This mean TextRank could not be used on larger projects. Our other approaches could handle all the projects, although LSS took several days to generate sentences for the largest processes due to WordNet. By comparison, Overlap took several seconds to generate sentences for the same projects. Another 4 projects were then removed because their entire source code documentation was less than 20 unique sentences. This was seen as too small to be useful for generating automatic summarizations from the documentation. In total, we summarized 32 projects using all 4 approaches. These projects and their summaries are available in the online appendix (see Section 5.6).

*`http://www.sourceforge.net`

DOI: 10.1002/smr

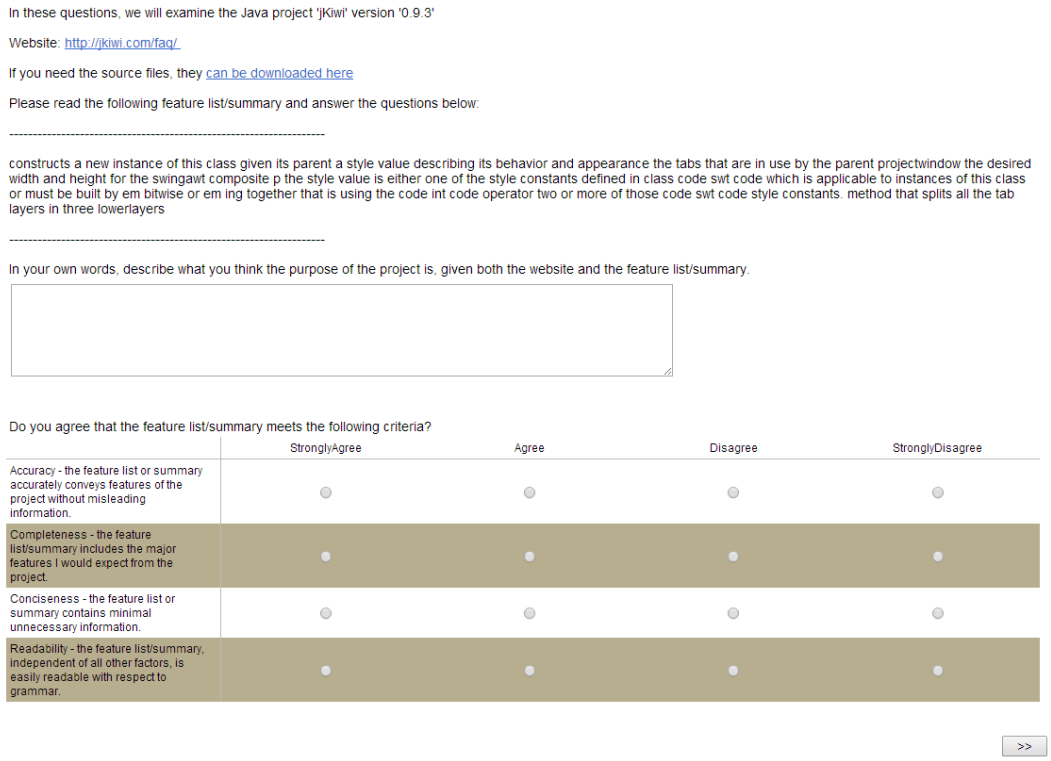

Figure 3. The is a screen capture of an evaluation question from the survey. In this question, the participant is asked to evaluated a TextRank generated summary of the documentation for the JKiwi project. The participant is asked to evaluate the summary by answering one essay question and 4 multiple choice questions.

5.3. User Survey

This section describes our user study survey. The remaining participant was an information technologies professional employed by the University of Notre Dame. Participants were initially vetted to ensure they had a computer science or programming background. The participants took the survey online and were asked to work alone. Each participant was asked 5 evaluation questions and 10 comparison questions, in that order. Both of these question types are described below. The participants could skip a question at their own discretion. Reasons for skipping a question could be not understanding a project well enough to rate a summary. Additionally, participants could end the survey at any time. Surveys results where participants took less than 2 minutes per question on average were discarded on the basis that the participant may not have given due diligence to answer the questions honestly.

5.3.1. Evaluation Questions Each participant is first asked to answer 5 evaluation questions. An example of an evaluation question is shown in Figure 3. Each evaluation question selects a random project and one of the four automatically generated summaries or feature lists generated for the project. The participant is given the name of the project, a version number, the project website, a link to download the source code for that version of the project, and a summary. Each participant is asked to, in their own words, state what they believe the purpose of the project is. The participant is then asked four multiple choice questions. The questions ask the participants if they agree that the given summary or feature list is accurate, complete, concise, and readable. These question respectively correspond with RQ_1, RQ_2, RQ_3 , and RQ_4. The participant can choose between "Strongly Agree," "Agree," "Disagree," and

DOI: 10.1002/smr

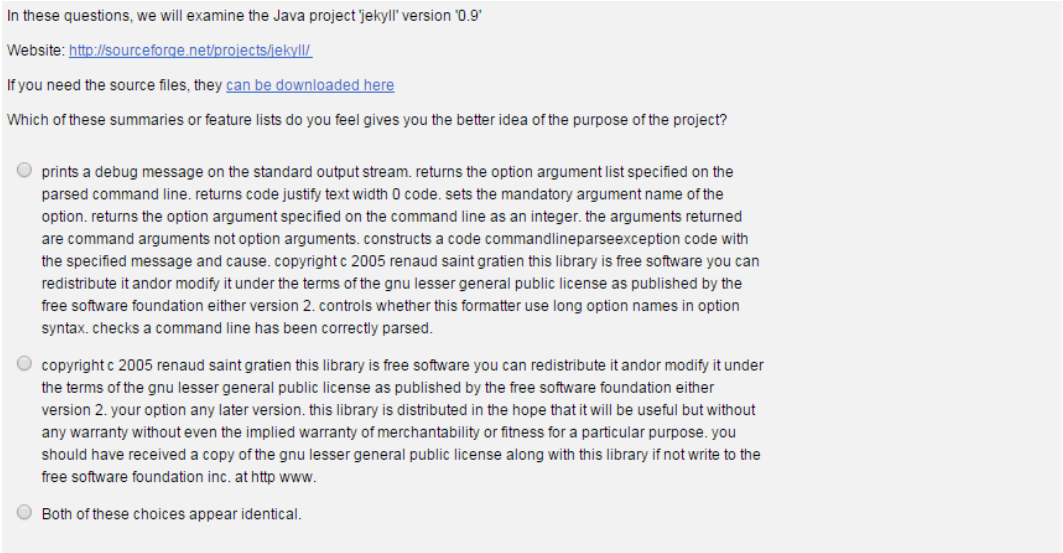

Figure 4. The is a screen capture of a comparison question from the survey. In this question, the participant is asked to choose which of two summaries the feel is better for describing the NanoXML project. The participant is asked to evaluate the summary by answering a single multipe choice question.

"Strongly Disagree." After answering all the questions, the participant may move on to the next question by clicking a button in the bottom right of the page.

5.3.2. Comparison Questions After the evaluation questions, each participant is asked 10 comparison questions. An example of a comparison question is shown in Figure 4. Each comparison question randomly selects a project, then randomly selects 2 summaries or features lists generated for the project. As in the evaluation questions, the participants are given the name of the project, a version number, the project website, a link to download the source code for that version of the project. Unlike in the evaluation question, participants are not asked to summarize the project in their own words. This was done to avoid fatigue on the part of the participant. The participants were simply asked to choose, via radio button, which summary or feature list they felt was better. Additionally, the participants are given a third option, "Both of these choices appear identical." This option was included because, for some projects, Overlap and LSS generated very similar summaries. After answering, participants could advance to the next question by clicking a button in the bottom right of the page.

5.4. Participants

Our user study was conducted by 13 participants. Participants were compensated $30 USD for their time spent evaluating our tool. Of the 13 participants, one survey response was thrown out for incompleteness, leaving us with 12 participants. Eleven of these participants were graduate students in the Department of Computer Science and Engineering at the University of Notre Dame. The remaining participant was an information technology professional. We informed participants when they were recruited that the participants should be comfortable reading and understanding the Java programming language. However, out study did not require the participants to read the Java source code, nor did it require them to complete any programming task.

Prepared using smrauth.cls
DOI: 10.1002/smr

5.5. Statistical Test

In this study, we will use the Mann-Whitney statistical test [24] to answer our research questions. Mann-Whitney is selected because we cannot guarantee our results will be normally distributed. Additionally our results are unpaired due to the random nature of the question selection. When performing statistical tests comparing four groups of responses, where each set of data represents one of the four approaches, we use the following procedure: 1) We sort each set of data by the mean of the score being tested. 2) The set of data with the best mean is given rank 1. 3) The set of data with the next highest mean is compared to the first group using Mann-Whitney. If there is no statistically significant difference, the second set of data is also given rank 1. Otherwise, if there is a statistically significant difference, the second set of data is given rank 2. For each remaining population, we compare to all samples in the highest current rank. This means, for example, in the case where both of the first two populations are rank 1, we compare the third set of data to both the first and the second using Mann-Whitney. Thus, it is possible that all four sets of data will have the rank 1, meaning no statistically significant difference exists between all four sets of data.

5.6. Reproducibility

For reproducibility, we have posted the list of projects we used in this study to an online appendix [†]. We have additionally included our anonymized survey data in the appendix.

5.7. Threats to Validity

As with all software engineering studies, a source of threat to validity are the projects selected for study. We cannot guarantee that a similar study that used different projects will produce the same results. We mitigate this threat by studying 32 projects with a wide variety of purposes. We avoid the threat of author selection bias by randomly selecting the projects. As a result of the large number of projects we consider, however, we can expect documentation quality to vary. Some projects we consider are poorly documented. This decision was intentional, to account for a possible weaknesses of each approach. However, this threat should be mitigated by the fact that we use all four approaches for each project. Poor documentation, specifically a lack of sufficient Javadocs, that would hurt one approach would almost certainly hurt the other approaches as well, as all approaches relied on leveraging Javadocs to generated sentence lists.

Additionally, our set of participants could be a source of threat to validity. As stated in Section 5.4, we sought participants from a variety of backgrounds to mitigate this threat. However, we cannot guarantee a similar study conducted with different participants would produce the same results. Nearly all of our study participants were graduate students, with one professional programmer. We felt comfortable that these participants would be capable of understanding the high-level concepts of each project. Because we only had one professional programmer, we cannot meaningfully differentiate between the behavior of student participants and professional participants.

The approaches in our study can produce different numbers of sentences. While LSS and Overlap typically produce 10 sentences, this is not always the case in TextRank and TLDR. This can threaten the validity of our completeness and conciseness results. We chose not to mitigate this threat by setting a fixed number of sentences to generate. This was because reducing the number of topics with LSS and Overlap would significantly decrease the usefulness of the LDA model generated. We chose not to expand TextRank, as it usually led to repeating the same sentences that were in the smaller summary. This would mean TextRank would significantly decrease in conciseness without improving in completeness. TLDR did not allow us to modify the size.

[†] `http://www.nd.edu/~pmcburne/features/`

DOI: 10.1002/smr

Table I. Average scores by approach

Approach	Accuracy	Completeness	Conciseness	Readability
Overlap	1.93	1.80	2.00	2.47
LSS	1.94	1.69	1.69	2.00
TextRank	1.62	1.62	1.87	1.81
TLDR	1.92	1.92	1.85	2.23
Overall	1.85	1.75	1.85	2.12

Another source of threat to validity comes from only studying Java projects that have an associated website. These websites provide an explanation of the source code authors' intended purposes for the project. This could bias a participant where they rate one automatically generated summary higher based on that summary being more similar to the summary provided by the authors. This choice was made so that study participants could quickly see what a given Java project they are unfamiliar with. We believe this condition was necessary, as asking the participants to understand a project by reading documentation and studying the source code would have resulted in a large amount of fatigue, forcing our study to limit itself to a very small number of questions per participant. This also would have likely meant less understanding of the projects by the participants, resulting in less useful results.

A threat to validity unique to TextRank and TLDR summaries comes from the formatting of the documentation. TextRank and TLDR both can leverage the format of their input text. Specifically, TextRank and TLDR consider paragraph breaks. As we described in Section 4.3, we add paragraph breaks after each method summary in our documentation parsing. This is done because we consider each method summary to be it's own encapsulated piece of information. We cannot guarantee that a different formatting decision would result in the same results. However, we believe based upon preliminary analysis that this difference, if any, would be limited.

6. RESULTS

In this section, we present the results of our user study. We answer the five research questions defined in Section . The average scores for the Evaluation Questions on each approach on each metric can be seen in Table I. These averages are taken from the Evaluation Questions of our survey. We quantified the user answers of "Strongly Agree", "Agree", "Disagree", and "Strongly Disagree" as 4, 3, 2, and 1 respectively. In this way, larger numbers represent better averages. Overall is the average of all 4 approaches collectively.

6.1. RQ_1 Accuracy

Our study found that there was no statistically significant difference between any of the four approaches with respect to accuracy. Our study found that on average participants disagreed that the generated feature lists accurately depicted the given Java project, regardless of the approach. However, TextRank did perform a fair amount worse than the other three approaches.

LSS performed the best on accuracy, with an average of 3.06. Overlap and TLDR performed very similarly, with averages of 3.07 and 3.08 respectively. TextRank scored somewhat worse, with an average accuracy score of 3.38, which is near the borderline between "Disagree" and "Strongly Disagree." For TextRank, participants "Strongly Disagreed" that the given feature list was accurate in a majority of cases. According to our statistical tests in Table II, none of the approaches are significantly better or worse than the others. Thus, while TextRank does have a notably worse mean than the other approaches, it does not meet the threshold of significance.

Prepared using smrauth.cls
DOI: 10.1002/smr

Table II. Mann-Whitney Tests for statistical significance. U, U_{expt}, and U_{vari} are Mann-Whitney derived values use to calculate the decision value p. If p ¡ .05, the two populations are statistically distinct, and the rank of the population with the lower mean is incremented.

RQ	H	Approach	n	mean	Std. dev.	U	U_{expt}	U_{vari}	p	Decision	Rank
RQ_1	H_1	LSS	16	1.94	0.929	108.5	112	506	0.894	Not Reject	1
		Overlap	14	1.93	0.73						1
	H_2	LSS+Overlap	30	1.93	0.828	189	195	1256	0.877	Not Reject	N/A
		TLDR	13	1.92	0.954						1
	H_3	LSS+Overlap+TLDR	43	1.93	0.856	271	344	3016	0.187	Not Reject	N/A
		TextRank	16	1.62	0.885						1
RQ_2	H_4	TLDR	13	1.92	0.954	96	97.5	383	0.959	Not Reject	1
		Overlap	15	1.80	0.561						1
	H_5	Overlap+TLDR	28	1.86	0.756	178	224	1407	0.225	Not Reject	N/A
		LSS	16	1.69	1.01						1
	H_6	Overlap+TLDR+LSS	44	1.79	0.851	295.5	352	3011	0.307	Not Reject	N/A
		TextRank	16	1.62	0.957						1
RQ_3	H_7	Overlap	15	2.00	0.756	105.5	120	567	0.556	Not Reject	1
		TextRank	16	1.87	1.09						1]
	H_8	Overlap+TextRank	31	1.93	0.929	185	202	1338	0.662	Not Reject	N/A
		TLDR	13	1.85	1.07						1
	H_9	Overlap+TextRank+TLDR	44	1.91	0.96	318	352	3133	0.55	Not Reject	N/A
		LSS	16	1.69	0.704						1
RQ_4	H_10	Overlap	15	1.47	0.64	81	97.5	419	0.434	Not Reject	1
		TLDR	13	2.23	1.09						1
	H_11	Overlap+TLDR	28	2.36	0.87	171	224	1510	0.177	Not Reject	N/A
		LSS	16	2.00	0.894						1
	H_12	Overlap+TLDR+LSS	44	2.23	0.886	253	352	3263	0.086	Not Reject	N/A
		TextRank	16	1.81	1.11						1

DOI: 10.1002/smr

When looking at all four approaches as a whole, none of the approaches performed particularly well. Across all four approaches, participants only "Strongly Agreed" that the given feature list was accurate 2 times: once for LSS, and once for TLDR. In 42.4% of cases, the participants "Strongly Disagreed" the given feature list was accurate. Possible reasons for the overall poor performance are discussed in section 7.

6.2. RQ_2 Completeness

Our study found that there was no statistically significant difference between any of the four approaches with respect to content adequacy, or "completeness." We found that participants "Disagreed" or "Strongly Disagreed" that the given feature list was complete in a large majority of cases. Discussion as to why these approaches lack completeness can be found in Section 7.

TLDR had the best average at 3.08, which is slightly worse than "Disagree." However, in TLDR, participants "Agreed" or "Strongly agreed" the feature list was complete in 23.1% of cases. Overlap had the second best average at 3.20. However, no participant "Strongly Agreed" that an Overlap feature list was complete. LSS and TextRank had average scores of 3.31 and 3.38 respectively. According to our statistical tests in Table II. We found no statistically significant differences across all four approaches.

Once again, all four approaches on average rate worse than "Disagree." Overall, participants only "Strongly Agree" or "Agreed" the feature list was complete 15% of the time across all approaches. The plurality of responses, 46.7%, "Strongly Disagree" that the feature lists were complete. This, along with the individual results, implies that these automatic approaches do not provide a good coverage of sentences that explain the diverse uses of the projects. This discussion is expanded on in the next section.

6.3. RQ_3 Conciseness

Our study found that there was no statistically significant difference between any of the four approaches with respect to conciseness. Overlap performed the best with respect to conciseness with an average of exactly 3.0. This means that for the best approach, users on average still "Disagreed" that the automatically generated feature lists were concise.

As stated, Overlap created what are participants perceived as the most concise feature lists on average. TextRank and TLDR had average scores of 3.13 and 3.15 respectively. However, participants "Strongly Disagreed" with a majority, 56.3% of TextRank generated feature lists they were presented with. LSS performed the worse with conciseness with an average of 3.31. In LSS, participants "Disagreed" or "Strongly Disagree" that the presented feature list was concise 87.5% of the time. According to our statistical tests in Table II, there was no statistically significant difference between the approaches.

6.4. RQ_4 Readability

Our study found that there was no statistically significant difference between any of the four approaches, although TextRank is a borderline case. Overlap provided the most readable feature lists on average, while TextRank provided the least readable feature lists.

On the whole, participants found overlap feature lists were the most readable, with an average score of 2.53. In a majority of cases, 53.5% participants "Agreed" that the features lists were readable. However, no participants "Strongly Agreed" that any Overlap feature lists were readable. TLDR averaged the second best readability, although a majority of time, 61.5% participants said they "Disagreed" or "Strongly Disagreed". LSS had an average readability score of 3.0 exactly, meaning on average participants "Disagreed" that the feature lists generated by LSS were readable. TextRank had the least readable feature lists. Participants "Strongly Disagreed" a majority of the time, 56.3%, that TextRank summaries were readable. According to our statistical test, there was no statistical difference between any of the approaches. However, TextRank, with a p-value of .086, was only just above the .05 threshold of significance.

DOI: 10.1002/smr

Table III. This table shows the breakdown of wins and losses when two items are compared.

Breakdown of Wins and Losses

Winner	Loser			
	Overlap	LSS	TextRank	TLDR
Overlap	-	7	4	15
LSS	1	-	7	10
TextRank	2	8	-	3
TLDR	3	4	4	-

Table IV. Normalized win, loss, and tie percentages for each approach.

Approach	Percentages		
	Win	Loss	Tie
Overlap	47.3%	10.9%	41.8%
LSS	46.7%	31.7%	21.7%
TextRank	24.1%	27.8%	48.1%
TLDR	16.9%	43.1%	40.0%

6.5. RQ_5 Comparison

Despite the mixed results of RQ_1 through RQ_4, when asked to compare directly, participants favor feature lists generated by LDA-based approaches Overlap and LSS to textual summarization-based approaches TextRank and TLDR. When asked to compare two feature lists directly, Overlap was selected as the better choice more often than any other approach. Additionally, Overlap lost the least, where losing is defined as the other approach's feature list being selected as the better of the two. By contrast, TLDR won the least and lost the most.

Table III shows the breakdown of wins and losses for comparisons of approaches. For example, in Table III where the winner is Overlap and the loser is LSS, we have the number 7. This means there were 7 instances where, when the participant was given a feature list generated by Overlap and a feature list generated by LSS, the participant chose the Overlap feature list 7 times. Participants only selected the LSS summary over the Overlap summary 1 time. For space, ties are not illustrated in this table. In this table, Overlap wins against every other approach more than it loses. TLDR appears to perform the worst, losing very frequently to Overlap and LSS. However, TLDR still wins against TextRank more often than it loses.

The winning, losing, and tieing percentages for the comparison question are shown in Table IV. Here we can see that Overlap has the largest winning percentage and the smallest losing percentage of all approaches. LSS has the second highest winning percentage, but also the second highest losing percentage. The very low tie percentage for LSS implies that participants either found the LSS feature list effective, or very poor. Possible reasons for this are discussed in Section 7. TextRank and TLDR had much lower winning percentages than Overlap and LSS. TLDR also has the highest losing percentage by over 10%. These tests suggest that the LDA-based approaches tend to beat the textual summarization-based approaches when compared directly.

7. DISCUSSION

.

This section discusses the impact of the results of our study. We additionally discuss the wider impact of our study in the field of feature discovery.

Our results for RQ_1, RQ_2, and RQ_3 show that, in general, study participants found all automatically generated summarization tools to perform to a level not meeting sufficient

DOI: 10.1002/smr

quality. Participants, on average, disagreed on all four of the sentence selection tools for automatic feature list generation across accuracy, completeness, and conciseness.

Overall, our findings suggest that sentence selection techniques are as yet insufficient for feature discovery. This is a key finding because sentence selection is currently in use in software engineering research tools [32] – our study suggests that additional innovation is needed for these tools to enter industrial use. However, sentence selection has been effective for other forms of text such as news articles. It is interesting to note that while our evaluation question results showed no statistical difference between the four approaches, our comparison question results suggest that participants clearly favored features lists created by LDA-models over textual analysis models. A reason for this difference could be that software documentation is generally made up of disjoint method and class summaries. Rather than a cohesive order of events like a news story, each piece of documentation is much more independent. While LDA focuses on overarching topics, the textual analysis tools are trying to determine each sentences importance by leveraging structure and order. However, software documentation is not strictly dependent on order, since Java methods can be in any order in the source code. Further, the structure of software documentation is very different than the structure of a news article. We believe future work might benefit from trying to find more general sentences in software documentation that discuss method interactions. Focusing on highly interactive methods proved beneficial in prior automatic summarization work [29].

Overlap, LSS, and TLDR all had very close reliability averages. TextRank was not statistically worse, though did have the worst average. Of these approaches, Overlap always had the fastest run-time. Overlap simply uses set operations, which have very low time complexity. TLDR requires communication with a server, though rarely takes more than a few minutes on large projects. On large projects we examined, such as jQuantlib*, running LSS and TextRank could take a full day. These long run-times were mostly due to these approaches needing to interact with WordNet. Interactions included search and path-finding, which on a large network of words can be time-consuming. TextRank and LSS were the largest limiting factor in projects we could select because of their long run-times and large memory requirements. However, even with all the added semantic understanding that using WordNet can allow, LSS and TextRank do not outperform Overlap in terms of feature list quality. On the evaluation questions, Overlap did not perform significantly better or worse than either TextRank for LSS. In the comparison questions, Overlap had a larger win percentage and smaller loss percentage than both TextRank and TLDR. This suggests that, given the current state of sentence selection, it may be better to use simpler metrics like Overlap for large projects.

The automated approaches, in general, performed better with respect to readability than other areas. This is likely because all these approaches select sentences from a pool of sentences written by human experts. While the other metrics rely on these sentences working together to communicate a larger goal, a feature list can still be readable even if the sentences don't interact with each other. Readability appears to result from the grammatical correctness of the sentences being selected rather than their overall cohesiveness.

A common problem with the LSS approach was that at times LSS would select very large sentences. Because of how LSS compares two bodies of text, large sentences often get artificially inflated, as a large number of words means at least one word is more likely to be semantically similar. We encountered this in previous work [30]. When this did occur, Conciseness scores and Accuracy scores were usually lower. Often, this resulted in run-on sentences in documentation being inflated over short sentences that were also related to the topic.

Some of the projects we investigated were poorly documented. Some projects have more unique files than they have unique sentences in Javadocs. Our results, investigated on a project by project basis, appear to suggest that projects that have a very small number of unique sentences in the Javadocs perform poorly across all 4 approaches. However, this only affects a small number of the projects we studied. The difficulty is that comment quality is not always

*`http://www.jquantlib.org/en/latest/`

DOI: 10.1002/smr

related to the number of unique sentences. While clearly having very little documentation is problematic, that does not mean large projects consistently perform better. Conciseness is an important component of documentation quality [46, 45, 29], thus more sentences to choose from is not better if those sentences are not meaningful to the high-level purpose of the project. Noteworthy examples of sentences that do not serve a high-level purpose including authorship statements, licensing agreements, and GUI-related information. Future approaches could seek to remove these sentences from the base of considered sentences as a pre-processing step. The large projects in our approach were often noisy. Some of these large projects performed just as poorly as the projects with more Java classes than unique sentences. This suggests that having a diverse pool of sentences to choose from may be necessary, but it is certainly not sufficient to our approach working well.

The goal of our approaches to feature discovery is to acquaint programmers unfamiliar with a given Java project with the purpose of that project. However, Completeness can be difficult for inexperienced users to judge. This is because a programmer unfamiliar with a given system will likely early learn enough about a system to understand its general purpose in order to respond to our survey. Completeness, however, is about a given feature list noting every task a system can complete or be used to complete. In order to properly evaluate completeness, future studies will likely require the assistance of the developers of the project being evaluated.

8. FOLLOW-UP EVALUATION

We conducted a brief follow-up study in an attempt to improve our Overlap approach. In this section we discuss the method of evaluation for our follow-up study. Our follow-up study has the following goals 1) To perform a focused study on well-documented projects to prevent our results from being affected by poor-documentation. 2) To see if Overlap's performance can be improved by using LDA-GA to determine near-optimal input parameters for LDA. 3) To examine how effectively individual sentences convey high-level understanding of a project's purpose. Specifically, we ask the following research questions:

RQ_6 To what degree do sentences, selected from a well-documented Java Project using Overlap, provide understanding of the purpose of the project?

RQ_7 To what degree are individual sentences, selected from a well-documented Java Project using Overlap, relevant to the overall purpose of the project?

8.1. Data Collection

In this section, we describe how we collected and prepared data for our follow-up study. In this study, we focus on extracting sentences from documentation of three projects: jEdit [†], a text editting tool designed for programmers, Jajuk [‡], a music and audio file organization tool and player, and jHotDraw [§], a 2D graphics framework. We included information describing the size of these projects in Table V. We selected these projects because we have previously used them in automatic source code summarization research [29], and we have found these projects to be particularly well documented. Additionally, these tools have an easily understood purpose. Nearly all computer programmers will have used some kind of similar program in the past. We extracted sentences from these projects in the same fashion that we extracted sentences for the first study, as described in Section 4.1.

[†] `http://sourceforge.net/projects/jedit/`
[‡] `http://sourceforge.net/projects/jajuk/?source=directory`
[§] `http://sourceforge.net/projects/jhotdraw/?source=directory`

DOI: 10.1002/smr

Table V. The three Java projects used in our evaluation. KLOC reported with all comments removed. All projects are open-source.

	Methods	KLOC	Java Files
Jajuk	5921	70	544
JEdit	7161	117	555
JHotdraw	5263	31	466

LDA Input Parameters

Min Gibbs Iterations: 10
Max Gibbs Iterations: 100
Min Alpha: 0.1
Max Alpha: 1
Min Beta: 0.1
Max Beta: 1
Minimum Topics: 5
Maximum Topics: 20

Genetic Algorithm Parameters

Max Iterations: 100
Run: 5
Permutation Rate: 0.1
Population: 30
Elitism: 0.05
Seed: 5

Figure 5. This lists the input parameters for LDA-GA used in our study.

Table VI. This table lists the near-optimal input parameters suggested by LDA-GA.

Project	α	β	#Topics	#Iterations
jEdit	0.5906	0.5108	20	48
Jajuk	0.3315	0.7145	20	55
jHotDraw	0.8102	0.7785	20	53

8.2. Configuring LDA-GA

In this section, we discuss how we configured LDA-GA to select topics from the source code and documentation. Dr. Panichella both furnished an implementation of LDA-GA and provided assistance in configuring it for our study. For each of our three projects, we gave LDA-GA the source code for each project and the extracted sentences. LDA-GA then calculated near-optimal parameters for us to use in LDA. Using these parameters as specified by LDA-GA, we generate a topic model for the project with LDA. For reproducibility, we have included the input parameters for LDA-GA in Figure 5. These parameters were decided with assistance from Dr. Panichella, the author of the system. For the purpose of feature discovery, we limited ourselves to 20 topics maximum, as we believed if we had more topics than this, our feature list would be too long and would result in both very low-level topics and a fatigue effect in study participants. Table VI lists the input configurations suggested by LDA-GA for each project.

8.3. Selecting Sentences

In this section, we describe our procedure for selecting sentences to form our feature list from the documentation for our follow up study. We used Overlap (Section 4.2.2) to find the best

Prepared using smrauth.cls
DOI: 10.1002/smr

Table VII. Questions asked about the feature list in the first section of the follow-up study. Q_1, Q_2, and Q_3 were multiple choice. Participants could respond with "Strongly Disagree", "Somewhat Disagree", "Somewhat Agree", or "Strongly Agree". Q_4 was a required open ended question. Q_5 allowed participants to provide any additional comments, and was optional.

Q_1	I understand how the list of documentation items fits into the program's stated purpose on its website.
Q_2	I believe that if given this list of documentation items, I could predict the program's intended purpose.
Q_3	The list of documentation contains unnecessary items that distract from the program's intended purpose.
Q_4	If there were no website, and you only had the list of documentation items, what would you assume the purpose of the program was? Please be detailed.
Q_5	(Optional) Please note any additional comments here.

sentences for the top 10 keywords. We chose to use the top 10 keywords based on internal experimentation. Often if we used fewer keywords, sentences were often too vague, or only contained 1 keyword from the topic. More keywords resulted in noisy and repetitious keywords in topics. We also looked at threshold approaches, where we considered all keywords in a topic that scored above some n-value. However, often this resulted in some topics with only 1 or sometimes zero topics, while other topics has 15 or more keywords larger than the threshold. Because of the output of LDA-GA, all projects generated 20 sentences for their feature list. After generating our list of features, we removed any duplicate sentences from the resulting list. This affected only jHotDraw, which was reduced to 16 unique sentences. Jajuk and Jedit had 20 unique sentences. We chose to remove duplicate sentences without replacing them, as we wanted each topic to be affiliated only with its most similar sentence.

8.4. User Study

In order to evaluate the quality of our feature list sentences, we conducted a brief follow-up user study. Many of the participants from our first study, as well as some new participants, we asked to score our feature lists. The study was broken into three sections.

In the first section, programmers were presented with the sentences selected from jEdit. Participants were asked the questions in Table VII. In our follow-up, we specifically want to know if the given list of documentation items gives the programmers an idea of what the system's overall purpose is. We crafted these questions to determine this.

In the second and third section, we asked participants to evaluate features lists of Jajuk and jHotDraw, respectively. However, rather than rate the feature lists as a whole, we asked readers to rate each individual sentence in terms of relevance to the intended task. Programmers could rate a sentence as "Very Relevant", "Somewhat Relevant", "Somewhat Irrelevant" and "Very Irrelevant." The programmers were only asked to rate each sentence based on relevance, as completeness for one sentence would be meaningless to ask, and conciseness would be hard to judge in some cases. We asked participants to rate each sentence as we had concerns in our initial study that fatigue would cause participants to only read the first few sentences rather than the entire feature list. By examining each feature individually, we ensure the participants have to consider each sentence rather than giving in to fatigue and only skimming a larger list. This was important, as we also we asked participants Q_4 and Q_5 from Table VII. If we did not ensure all participants read each sentence in the summary, the results of Q_4 would be unreliable.

DOI: 10.1002/smr

8.5. Participants

Our study had a total of 12 participants. All 12 participants were graduate students in the Computer Science and Engineering Department at the University of Notre Dame. The participants were recruited via in-department e-mail communications. Students were offered $10 to participate in the study. Five of the twelve participants also participated in our initial study. The remaining seven did not take the initial study. Participants were informed that an understanding of basic programming and Java conventions was required. One participant only completed the first section of the study of the study. The remaining participants completed the entire study.

8.6. Reproducibility

For reproducibility, we have included anonymized study results as well as the feature lists generated for each project in an online appendix ¶.

8.7. Threats to Validity

As with any software engineering study, the participants in our study are a potential threat to validity. Given that this study was fully graduate students with a computer programming background, we believe they were qualified and that their responses deserve merit. We believe that this is especially true given that there was no programming task required, and that the task was overall relatively simple. Still, we cannot guarantee that a different group of professional programmers would research the same conclusions.

The selection of projects we study is another source of threat to validity. The projects we selected in this follow-up study were selected because they were well documented. We selected high-quality documented projects because we wanted to mitigate the threat that our results were being "brought down" by projects that were poorly documented. These projects have previously been used in source code summarization studies [29]. However, this also means we acknowledge that our approach often fails when dealing with poorly documented code. Our approach cannot work on completely undocumented code.

Another threat to validity comes from the fixed ordering of projects in our approach. We chose to fix the order of our approach in order to maximize feedback on specific projects. Our initial study was broad to the point that patterns found in individual projects with regards to the efficacy of techniques were lost in the larger study. To counter this, our follow-up study focuses on depth by collecting a more data on few projects. The fixed order ensured that the same questions were answered about each project, giving us the most possible information to work with in regards to depth. That said, the fixed order could introduce biases that limit the generalizability our results.

Additionally, there is a significant threat to validity of bias related to Q_4. Specifically, programmers are asked to view the website before answering Q_4, which can result is significant bias in their answers. We have the programmer view the website, specifically in section two and three, to rate each extracted sentence. Because there were a 20 and 16 sentences to rate in Jajuk and jHotDraw, respectively, we were concerned that without requiring participants to answer a question on each sentence, participants would only read the first few sentences before assuming purpose. We are concerned that this was the case in our initial study. This would be problematic, as the order of the sentences is an arbitrary result of the generated topics.

9. FOLLOW-UP STUDY RESULTS

In this section, we present the results of our follow-up user study. The implications of these results will be discussed in Section 10.

¶ `http://www.nd.edu/~pmcburne/features/`

DOI: 10.1002/smr

Table VIII. Section 1 responses to multiple choice questions.

Question	Strongly Disagree	Somewhat Disagree	Somewhat Agree	Strongly Agree
Q_1	1	3	8	0
Q_2	1	4	7	0
Q_3	1	3	3	5

9.1. Section 1 Results

Section 1 of our survey is used to address RQ_6. In Section 1 of our survey, participants were given a feature list for jEdit constructed using topics generated by LDA-GA. These topics were used to select sentences using our Overlap sentence selection approach. The majority of participants, 8 of 12, "Somewhat Agreed" that they understood how the list of sentences from documentation fit into the program's stated purpose, however, none "Strongly Agreed". One participant "Strongly Disagreed". This participant, in Q_4 stated that the documentation list was difficult to understand, and if they had to guess, they "would assume that this program creates a dockable window user interface that contains a toolbar." Seven of the 12 participants "Somewhat Agreed" that they could predict the program's intended purpose from the documentation.

However, on Q_3, 8 of the 12 participants either "Somewhat Agreed" or "Strongly Agreeed" that the list of documentation contained unnecessary items that distract from the program's intended purpose. Because we had 20 topics generated for jEdit, this could be the result of selecting sentences for topics that do not communicate high-enough level ideas. These sentences could distract or confuse from the project's intended purpose. In the case of jEdit, several of the output sentences referred to things attributed to GUI elements, which many participants noted in Q_4 is not something they would intuitively expect in a text editor. Overall, we answer RQ_6 by noting that we did improve over our wider survey. However, conciseness in particular is still a significant concern.

9.2. Section 2 and 3 Results

Table IX. Aggregate responses to multiple choice questions for Section 2 and 3. In each Section, participants were asked to rate the relevance of each sentence generated by LDA-GA topics using Overlap sentence selection.

Project	Very Irrelevant	Somewhat Irrelevant	Somewhat Relevant	Very Relevant
Jajuk	75	44	67	34
jHotDraw	29	46	57	43

In the remaining two sections of our study, participants were asked to rate the relevance of sentences selected from documentation. For Jajuk, the results indicate that many of the sentences participants believed the sentence to be "Very Irrelevant." Across all sentences, this was the most common answer. Of the 20 sentences, 12 received more "Very Irrelevant" or "Somewhat irrelevant" scores from participants than "Somewhat Relevant" or "Very Relevant". Of the remain 8 sentences, which received more "Relevant" ratings, only 2 received more "Very Relevant" scores than "Somewhat Relevant." However, despite the lower performance, when asked what the participants would assume the purpose of the program was if they only had to go off the features list, 7 of the 11 participants mentioned said they would consider the program a music or media player of some form. These participants specifically mentioned that sentences containing words such as "album", "artist", and "track" indicate this functionality. Additionally, one of the sentences, extracted from a test program within the software, makes reference to the band "Red Hot Chili Peppers." Of the remaining

DOI: 10.1002/smr

4 participants, 2 said they would think the list described some user interface tool. Another participant said they believed this was a data wizard used to construct a screen. The remaining participant said they would not be able to figure out any intent based on the list.

The quantitative results for jHotDraw were better. It should be noted for jHotDraw there were 4 duplicate sentences that were removed, and as such each participant only rated 16 sentences. In this case, the majority of ratings by participants for jHotDraw were either "Somewhat" or "Very Relevant". Ten of the 16 sentences received more "Relevant" ratings than "Irrelevant". Of the 10 sentences rated "Somewhat" or "Very Relevant", 3 were rated more frequently as " Very Relevant". Of the 11 participants, 10 said that if given the feature list, they would believe this project would be a drawing tool of some kind, albeit one of the participants said "but still with very low confidence, because almost all GUI applications have something for drawing and some visual views." It is worth mentioning that the title jHotDraw includes the word draw in it, which could influence how our participants answered Q_4. The remaining participant said the program would be a GUI editing or designing tool. In these sections, we answer RQ_6 with cautious optimism that our approach can provide some high-level understanding. However, in RQ_7 we find that there are still a large number of irrelevant sentences being included in our summaries.

10. FOLLOW-UP STUDY DISCUSSION

In this section, we discuss the findings of our follow-up user study. Overall, we believe the results of this follow-up study show that there is some promise to automatic feature discovery via sentence selection. While our initial study had, overall, poor results, our follow-up study shows that sentence selection can work for the purpose of generally informing a programmer what the purpose of a given project is on well documented projects.

A common point of confusion among study participants was that the features often referred to GUI elements. Many participants said these documentation items often led them to believe a program allowed you design or edit GUIs. In future work, it may be worth exploring reducing the number of topics that refer to graphic and interface elements.

It should be noted that for every project, LDA-GA suggested 20 topics. This was our maximum allowable number of topics based on our LDA-GA parameters. We set 20 as the maximum out of concern that the summaries would become too large for an effective human study, as well as the topics would become too specific and low level to clearly indicate a projects overall purpose. When we removed the upper-bound of 20 topics, all three projects were generating in excess of 60 topics. As we discussed in the prior paragraph, it's likely that GUI topics could be better combined to reduce their confusion impact on the summary output without affecting the overall message. This would reduce the number of topics more naturally then setting a fixed upper-bound number of topics.

We do not believe any of the automatic approaches presented in this paper would outperform human experts at this time. A concern is that while feature discovery may perform better on well-documented projects, it is reasonable to believe that if a project is well documented, it likely has a website that lists the features written by a human expert. However, from our own experience, we have found projects with expired websites and no clear means to contact the developers. If the project were additionally poorly documented, our approach as is would not perform adequately. We believe that future work could focus on combining automatic natural language documentation and summarization approaches, such as Sridhara *et al.* [45] or our own previous work, McBurney *et al.* [28] with automatic feature discovery, which could allow for poorly-documented projects or projects lacking any documentation to generate a feature list.

While this work lays a foundation for project summarization using feature discovery by source code analysis, future work could extract sentences from other sources beyond source code and embedded JavaDocs. These other sources of documentation could include change logs, developer communications, bug reports, etc. These sources have been used to summarize

DOI: 10.1002/smr

individual sections of source code [39, 51], and we believe sentences extracted from these sources could be used to describe features of an entire project. While it is beyond the scope of this foundational work, future work will want to mine these sources when available.

11. RELATED WORK

This section will cover related work in automatic source code summarization. This paper is related to source code summarization in that we are attempting to provide overall understanding of a software project. Source code summarization is a growing field in the literature. Several of these approaches rely on information retrieval (IR) techniques. Haiduc *et al.* presented an approach to summarizing source code methods using a Vector Space Model [17]. This approach summarized individual source code methods by providing keywords that describe the method, similar to how LDA produces topics in the form of keywords in feature discovery [12, 21]. This work was later modified by Rodeghero *et al.* where keyword selection was weighted base on source code context [40]. In this approach different source code contexts were considered more important due to the results of an eye-tracking study. This paper demonstrated that programmers preferred when keywords were selected when considerations for source code context were made. These approaches focus on individual method summarization. Our goal is to summarize projects at a higher level.

More recently, two source code summarization approaches use natural language summarization. Sridhara *et al.* create an approach that summarize Java methods by identifying important statements in a given method using IR techniques [44]. These statements are then translated into natural language using templates to create natural language summaries of the method [45]. Our later work expanded on the idea of natural language summaries by creating natural language summaries of Java methods using a method's contextual information [29]. Contextual information was derived using a call graph where methods with higher PageRank scores were considered more important to describing other methods. This work differs from this current work in that both of these approaches describe only individual methods. These approaches currently do not allow for project-wide summarization. Additionally, Panichella *et al.* used developer communications, including bug reports and mailings lists, for Lucene and Eclipse to automatically generate descriptions of methods [39]. Wong *et al.* mine comments from StackOverflow and other programming Q&A sites to generate source code summaries [51]. Related to this, Zhang *et al* mine discussions on API libraries to identify problematic features to bring them to the attention of the developers [52]. In the future, developer discussions may be used to supplement documentation in our open feature extraction approaches. This would be particularly useful when documentation is lacking or limited in the source code itself.

Another approach to source code summarization is topic-modeling. As discussed in Section 3.1, LDA is a commonly used tool in software topic modeling [12, 21]. De Lucia *et al.* demonstrated that techniques such as LDA produce artifact labelling similar to human programmer labeling [10]. Panichella *et al.* presented LDA-GA, a genetic algorithm searching tool to derive a near optimal LDA configuration for modeling a given software project [38]. Baldi *et al.* adapted latent topics for Aspect-Oriented Programming [2]. In our own recent work, we represented Java projects as a tree-like hierarchy of topic, where more general concepts would be higher in the tree, and less general topics would be lower in the tree [28]. By presenting methods in a hierarchy with associated topics, programmers felt they understood better how the given method fit into the overall project. This work, alongside the growing source code summarization field, lead us to believe that project-wide natural language summarization is possible with further work in the field.

DOI: 10.1002/smr

12. CONCLUSION

In this paper, we explored four automatic feature list generation tools for Java projects. Each of these tools selected sentences from existing documentation to form a feature list. Our evaluation found that there is still much work to be done in this area, as these preliminary techniques did not perform well in our evaluation. While LDA can find topics that help programmers understand projects better [12, 21], it is not always possible to describe these topics as natural language sentences extracted from JavaDocs. Existing work has shown that it is possible to produce quality automatic summarizations of Java methods. The problem of reliably automatically summarizing projects with natural language remains unsolved, especially for projects that are poorly documented. However, our evaluation lays the groundwork for future work in the area of automatic source code summarization and presents four baseline techniques to compare to. Additionally, we show that there is some promise that automatically generated natural language feature lists can give programmers a general idea of what the purpose of a given software project is.

ACKNOWLEDGMENT

The authors would like to thank the participants in our evaluation for their careful efforts. Additionally, the authors would like to thank Dr. Annibale Panichella for both furnishing an implementation of LDA-GA and providing assistance in configuring it's parameters.

REFERENCES

1. H. U. Asuncion, A. U. Asuncion, and R. N. Taylor, "Software traceability with topic modeling," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 95–104. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806817
2. P. F. Baldi, C. V. Lopes, E. J. Linstead, and S. K. Bajracharya, "A theory of aspects as latent topics," *SIGPLAN Not.*, vol. 43, no. 10, pp. 543–562, Oct. 2008. [Online]. Available: http://doi.acm.org/10.1145/1449955.1449807
3. K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.
4. D. Binkley, D. Heinz, D. Lawrie, and J. Overfelt, "Understanding lda in source code analysis," in *Proceedings of the 22Nd International Conference on Program Comprehension*, ser. ICPC 2014. New York, NY, USA: ACM, 2014, pp. 26–36. [Online]. Available: http://doi.acm.org/10.1145/2597008.2597150
5. D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, Mar. 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=944919.944937
6. A. Classen, P. Heymans, and P.-Y. Schobbens, "Whats in a feature: A requirements engineering perspective," in *Fundamental Approaches to Software Engineering*. Springer, 2008, pp. 16–30.
7. J. Cleland-Huang, A. Czauderna, M. Gibiec, and J. Emenecker, "A machine learning approach for tracing regulatory codes to product specific requirements," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 155–164. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806825
8. M. V. Couto, M. T. Valente, and E. Figueiredo, "Extracting software product lines: A case study using conditional compilation," in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*. IEEE, 2011, pp. 191–200.
9. D. Croft, S. Coupland, J. Shell, and S. Brown, "A fast and efficient semantic short text similarity metric," in *Computational Intelligence (UKCI), 2013 13th UK Workshop on*, Sept 2013, pp. 221–227.
10. 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 *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, June 2012, pp. 193–202.
11. S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, ser. SIGDOC '05. New York, NY, USA: ACM, 2005, pp. 68–75. [Online]. Available: http://doi.acm.org/10.1145/1085313.1085331
12. B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.
13. H. Dumitru, M. Gibiec, N. Hariri, J. Cleland-Huang, B. Mobasher, C. Castro-Herrera, and M. Mirakhorli, "On-demand feature recommendations derived from mining public product descriptions," in *Software Engineering (ICSE), 2011 33rd International Conference on*, May 2011, pp. 181–190.

DOI: 10.1002/smr

14. B. Fluri, M. Wursch, and H. C. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes," in *Proceedings of the 14th Working Conference on Reverse Engineering*, ser. WCRE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 70–79. [Online]. Available: http://dx.doi.org/10.1109/WCRE.2007.21
15. J. A. Goguen and C. Linde, "Techniques for requirements elicitation," *Requirements Engineering*, vol. 93, pp. 152–164, 1993.
16. S. Grant, J. R. Cordy, and D. B. Skillicorn, "Using heuristics to estimate an appropriate number of latent topics in source code analysis," *Science of Computer Programming*, vol. 78, no. 9, pp. 1663 – 1678, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642313000762
17. S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *Reverse Engineering (WCRE), 2010 17th Working Conference on*, Oct 2010, pp. 35–44.
18. M. Kajko-Mattsson, "A survey of documentation practice within corrective maintenance," *Empirical Softw. Engg.*, vol. 10, no. 1, pp. 31–55, Jan. 2005. [Online]. Available: http://dx.doi.org/10.1023/B:LIDA.0000048322.42751.ca
19. K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," DTIC Document, Tech. Rep., 1990.
20. V. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," *Soviet Physics Doklady*, vol. 10, p. 707, 1966.
21. E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi, "Mining concepts from code with probabilistic topic models," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. New York, NY, USA: ACM, 2007, pp. 461–464. [Online]. Available: http://doi.acm.org/10.1145/1321631.1321709
22. Z. Liu, Y. Zhang, E. Y. Chang, and M. Sun, "Plda+: Parallel latent dirichlet allocation with data placement and pipeline processing," *ACM Transactions on Intelligent Systems and Technology, special issue on Large Scale Machine Learning*, 2011, software available at http://code.google.com/p/plda.
23. S. Lukins, N. Kraft, and L. Etzkorn, "Source code retrieval for bug localization using latent dirichlet allocation," in *Reverse Engineering, 2008. WCRE '08. 15th Working Conference on*, Oct 2008, pp. 155–164.
24. H. Mann and D. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The Annals of Mathematical Statistics*, pp. 50–60, 1947.
25. A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Proceedings of the 25th International Conference on Software Engineering*, ser. ICSE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 125–135. [Online]. Available: http://dl.acm.org/citation.cfm?id=776816.776832
26. A. Marcus, J. I. Maletic, and A. Sergeyev, "Recovery of traceability links between software documentation and source code," *International Journal of Software Engineering and Knowledge Engineering*, vol. 15, no. 05, pp. 811–836, 2005.
27. G. Maskeri, S. Sarkar, and K. Heafield, "Mining business topics in source code using latent dirichlet allocation," in *Proceedings of the 1st India Software Engineering Conference*, ser. ISEC '08. New York, NY, USA: ACM, 2008, pp. 113–120. [Online]. Available: http://doi.acm.org/10.1145/1342211.1342234
28. P. W. McBurney, C. Liu, C. McMillan, and T. Weninger, "Improving topic model source code summarization," in *Proceedings of the 22Nd International Conference on Program Comprehension*, ser. ICPC 2014. New York, NY, USA: ACM, 2014, pp. 291–294. [Online]. Available: http://doi.acm.org/10.1145/2597008.2597793
29. P. W. McBurney and C. McMillan, "Automatic documentation generation via source code summarization of method context," in *Proceedings of the 22Nd International Conference on Program Comprehension*, ser. ICPC 2014. New York, NY, USA: ACM, 2014, pp. 279–290. [Online]. Available: http://doi.acm.org/10.1145/2597008.2597149
30. P. McBurney and C. McMillan, "An empirical study of the textual similarity between source code and source code summaries," *Empirical Software Engineering*, pp. 1–26, 2014. [Online]. Available: http://dx.doi.org/10.1007/s10664-014-9344-6
31. C. McMillan, N. Hariri, D. Poshyvanyk, J. Cleland-Huang, and B. Mobasher, "Recommending source code for use in rapid software prototypes," in *Software Engineering (ICSE), 2012 34th International Conference on*, June 2012, pp. 848–858.
32. ——, "Recommending source code for use in rapid software prototypes," in *Proceedings of the 2012 International Conference on Software Engineering*. IEEE Press, 2012, pp. 848–858.
33. R. Mihalcea and P. Tarau, "Textrank: Bringing order into text," in *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing , EMNLP 2004, A meeting of SIGDAT, a Special Interest Group of the ACL, held in conjunction with ACL 2004, 25-26 July 2004, Barcelona, Spain*, 2004, pp. 404–411. [Online]. Available: http://www.aclweb.org/anthology/W04-3252
34. G. A. Miller, "Wordnet: A lexical database for english," *COMMUNICATIONS OF THE ACM*, vol. 38, pp. 39–41, 1995.
35. L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, May 2013, pp. 23–32.
36. L. Moreno, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Jsummarizer: An automatic generator of natural language summaries for java classes," in *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, May 2013, pp. 230–232.
37. R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "On the equivalence of information retrieval methods for automated traceability link recovery," in *Program Comprehension (ICPC), 2010 IEEE 18th*

DOI: 10.1002/smr

International Conference on, June 2010, pp. 68–71.
38. 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 *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 522–531. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486857
39. S. Panichella, J. Aponte, M. Di Penta, A. Marcus, and G. Canfora, "Mining source code descriptions from developer communications," in *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, June 2012, pp. 63–72.
40. 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 *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 390–401. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568247
41. T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 255–265. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337254
42. J. Sillito, G. C. Murphy, and K. De Volder, "Asking and answering questions during a programming change task," *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 434–451, Jul. 2008. [Online]. Available: http://dx.doi.org/10.1109/TSE.2008.26
43. G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *Software Engineering (ICSE), 2011 33rd International Conference on*, May 2011, pp. 101–110.
44. G. Sridhara, "Automatic generation of descriptive summary comments for methods in object-oriented programs," Ph.D. dissertation, Newark, DE, USA, 2012, aAI3499878.
45. G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 43–52. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859006
46. D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," in *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, May 2013, pp. 83–92.
47. S. Thomas, "Mining software repositories using topic models," in *Software Engineering (ICSE), 2011 33rd International Conference on*, May 2011, pp. 1138–1139.
48. K. Tian, M. Revelle, and D. Poshyvanyk, "Using latent dirichlet allocation for automatic categorization of software," in *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, May 2009, pp. 163–166.
49. T. Wang, G. Yin, X. Li, and H. Wang, "Labeled topic detection of open source software from mining mass textual project profiles," in *Proceedings of the First International Workshop on Software Mining*, ser. SoftwareMining '12. New York, NY, USA: ACM, 2012, pp. 17–24. [Online]. Available: http://doi.acm.org/10.1145/2384416.2384419
50. N. Wilde and M. C. Scully, "Software reconnaissance: mapping program features to code," *Journal of Software Maintenance: Research and Practice*, vol. 7, no. 1, pp. 49–62, 1995.
51. E. Wong, J. Yang, and L. Tan, "Autocomment: Mining question and answer sites for automatic comment generation," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, Nov 2013, pp. 562–567.
52. Y. Zhang and D. Hou, "Extracting problematic api features from forum discussions," in *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, May 2013, pp. 142–151.

DOI: 10.1002/smr