Towards Prioritizing Documentation Effort

Paul W. McBurney[1], Siyuan Jiang[2], Marouane Kessentini[3],
Nicholas A. Kraft[4], Ameer Armaly[2], Wiem Mkaouer[3], and Collin McMillan[2]

[1] Department of Computer and Information Science
University of Pennsylvania, Philadelphia, PA
paulmcb@seas.upenn.edu

[2] Department of Computer Science and Engineering
University of Notre Dame, Notre Dame, IN, USA
{sjiang1, aarmaly, cmc}@nd.edu

[3] Department of Computer and Information Science
University of Michigan-Dearborn, Dearborn, MI, USA
{marouane, mmkaouer}@umich.edu

[4] ABB Corporate Research, Raleigh, NC, USA
nicholas.a.kraft@us.abb.com

✦

Abstract—Programmers need documentation to comprehend software, but they often lack the time to write it. Thus, programmers must prioritize their documentation effort to ensure that sections of code important to program comprehension are thoroughly explained. In this paper, we explore the possibility of automatically prioritizing documentation effort. We performed two user studies to evaluate the effectiveness of static source code attributes and textual analysis of source code towards prioritizing documentation effort. The first study used open-source API Libraries while the second study was conducted using closed-source industrial software from ABB. Our findings suggest that static source code attributes are poor predictors of documentation effort priority, whereas textual analysis of source code consistently performed well as a predictor of documentation effort priority.

Index Terms—code documentation, program comprehension, software maintenance.

1 INTRODUCTION

Programmers are notorious for neglecting source code documentation due to time and monetary pressures [1], [2]. These pressures often force programmers to defer documentation of source code until the final stages of development leading up to a release, or to write documentation and then neglect its maintenance [3]. Yet, the importance of documentation is widely recognized [3], [4], [5]. High-quality documentation has numerous benefits, ranging from better comprehension (implying wider impact and more rapid bug repair [6]) to speedier onboarding of new employees [7]. In short, programmers need high-quality source code documentation but lack the time to write or maintain it.

Several approaches deal with the problem of lacking documentation, for example, self-explanatory code that has less need for documentation, and modern IDEs that help programmers navigate through code so that programmers do not need to read documents to comprehend the code. However, even with these techniques, it is still time-consuming and challenging to understand a complicated large piece of code. Manually written documents provide insights from the developers that cannot be replaced by other means.

Programmers must *prioritize* their documentation effort. The sections of code that are the most important for developers to understand should be documented first. While it is not ideal to skip documentation, the reality is that not all code will always be documented to a high level of quality. Note that, in this paper, we focus on prioritizing documentation effort for code documentation. Other forms of documentation, such as design documentation and user documentation, are also important, but not the emphasis of this study. By prioritizing source code documentation, the programmers can help to ensure that time-limited efforts are dedicated to the highest-value areas of code. Programmers may intuitively "sense" what some of the high value areas of code are, but manually marking these areas is time-consuming for large projects. An automated solution is desirable – documentation is one of the top areas where programmers appreciate automation [3].

Software engineering literature would seem to suggest that *static attributes* about source code would provide useful clues for prioritization. For example, Murphy *et al.* [8] found that patterns in source code are often reflected in documentation. Inoue *et al.* [9] and Grechanik *et al.* [10] found that a developer's program comprehension benefits from seeing functions that are called frequently. Grechanik *et al.*

detected these functions by calculating the PageRank [11] of functions in a call graph. Stylos and Myers [12], [13] have shown how statically-derived function usage information can benefit documentation. Holmes [14] demonstrates how static context is useful for finding examples for documentation. Based on this and other strong literary evidence, we formed the working hypothesis that static code attributes assist documentation prioritization by highlighting which sections of code are the most important for comprehension.

Besides static source code attributes, we added a textual vector-space model (VSM) approach and a textual comparison approach. For the VSM approach, we calcualted tf/idf (term frequency/inverse document frequency) for each word in the source code. For the textual comparison approach, we used two similarity metrics to determine the similarity between the source code and the corresponding project home webpage. We computed all the attributes for the classes in three open source Java libraries and two closed source Java softwares. We also recruited programmers to read the code and manually prioritize the code for documentation. Using this collected data, we trained an artificial neural network to create a binary classifier for classifying documentation priority. The purpose of this study is to determine which set of candidate features (static source code attributes, VSM, or textual comparison) can be best used to predict for documentation priority. In this study, we consider documentation priority at the class granularity. Thus, we are providing a model for determining which Java classes humans should document first to maximize the efficiency of documentation effort.

Our results show that, surprisingly, the static source code attributes **were not** effective with average precision of 37.4%. In contrast, our textual comparison attributes and the VSM approach had much higher precision of 71.2% on average.

It should be noted that this study is both foundational and preliminary in nature. This study is foundational in that it is, to the best of our knowledge, the first study to suggest an approach for automatically prioritizing source code documentation. It is preliminary in that we are investigating candidate features for future documentation prioritization research. We do not suggest that our resulting approach is fully generalizable, nor is it mature enough for software firms to use as is reliably. Our approach currently relies largely on manual training and testing using a gold set, which must be constructed using human experts. However, we believe that our preliminary investigation is necessary before future work in this direction can be accomplished.

Specific novel contributions in this paper are:

1) An experiment determining the accuracy of using static source code attributes for documentation effort prioritization.

2) An experiment comparing static source code attributes, textual comparison attributes, and the VSM approach for documentation effort prioritization. We conducted these experiments in both an open source and an industrial context.

3) A complete implementation of the approaches for the purpose of reproducibility and further study. We make all static and textual data available for the open source projects, as well as analysis scripts and raw and pro-

cessed data.[1] Textual data of the industrial projects has been redacted due to commercial interests. All the other data of the industrial projects is available.

Roadmap In the rest of this paper, we will introduce and describe: the problem of prioritizing documentation effort (Section 2), the overview of our approach (Section 3), the gold sets (Section 4) and the features (Section 5) that we used to build binary classifiers for prioritizing documentation, the artificial neural network models and the training process (Section 6), our research questions and evaluation setting (Section 7), our quantitative results and comments made by the participants (Sections 8 and 9), the threats to validity (Section 10), discussion and future work (Section 11), related work (Section 12)and conclusion (Section 13).

2 PROBLEM

In this paper, we target the problem of prioritizing software documentation effort. Our **Research Objective** is to evaluate the effectiveness of different attributes of source code to predict whether a piece of code is important to document. To the best of our knowledge, this is the first work that seeks to do this. Developers need a way to prioritize documentation effort, as developers often write incomplete documentation, or neglect writing documentation entirely [3], [15], [16]. This results in documentation falling behind the source code. This disparity is referred to as a "documentation debt" and can result in the code becoming more difficult to use and maintain [17].

Our study has a large potential impact on how developers document source code. If developers can know what sections of source code are most important to document, their efforts can be focused there first. Effective documentation prioritization would ensure that software documentation is present on sections of code critical to understanding and maintaining the system. Additionally, this could benefit development teams with very limited resources. Finally, documentation prioritization could work well with automatic source code summarization. While automatic source code summarization does not yet match the quality of manually-written documentation [18], it could still be used to supplement manually-written documentation in lower priority sections of source code.

3 APPROACH OVERVIEW

To assess the effectiveness of different attributes of source code in predicting the importance of documenting a class, we followed four steps: obtaining "gold sets", extracting features, training models with different features, comparing the performances of different models.

We performed two user studies to create "gold sets" of important Java classes to document in the selected projects. The first study took place at the University of Notre Dame. The second study took place at ABB Corporate Research. The primary reason we performed two user studies is that we wanted to consider two different groups of programmers. The two studies are complementary. In the open-source study, we consider the perception of importance to

---

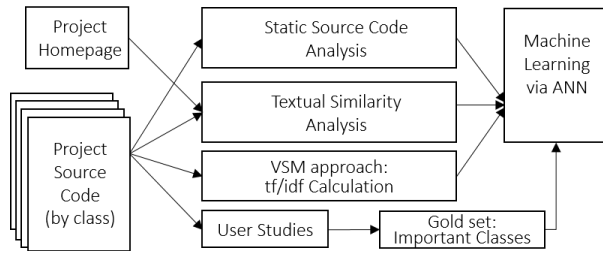1. http://www.cis.upenn.edu/~paulmcb/research/doceffort/

Fig. 1. An architecture of our approach for one Java project. First, we conducted two user studies to obtain important classes (our gold set) in each Java project. Second, we extracted static source code attributes, textual similarity attributes, tf/idf values from source code and project homepages. Finally, we built classification models via an artificial neural network (ANN) to predict important classes in a Java project.

document by non-developers to a software system. In the closed-source study, we consider the perception of importance to document by developers. We separate these two groups using separate user studies because each group may have different opinions on what in source code indicates documentation importance. Developers may be biased by sections of code they most needed to maintain, while non-developers are learning the system anew and may not have full understanding of the source code. Because of this difference, our approach may work in one study but not work in the other. So we conducted two studies to see the effectiveness of our approach in the two studies separately.

Then, we collected attributes of source code to be used as features in machine learning. We considered three types of attributes to compare their effectiveness in predicting whether a class is important to be documented. The three types are 1) static source code attributes, which are about static structures of code, 2) textual comparison attributes, which are about the text similariy between the source code of a Java class and the homepage of the corresponding project, and 3) the vector-space model (VSM) approach, which uses term frequency/inverse document frequency (tf/idf), which considers the importance of each word in a class file.

After we obtained the attributes, we used the attributes as features and trained different models with different features. We tried four different groups of the features: 1) only the static source code attributes, 2) only the textual comparison attributes, 3) only the VSM approach, and 4) both the static source code attributes and the textual comparison attributes. For each Java project and each group of the features, we built a model using the artificial neural network (ANN). The use of the ANN is justified and described in Section 6. Finally, we evaluated the models with cross validation on our gold sets. An overview of our approaches can be seen in Figure 1.

4 GOLD SETS

This section describes two studies we conducted to obtain the gold sets. In one study, we hired graduate students to assess whether the classes of three open source projects are important to be documented. In the other study, we asked professionals in industry to assess the importance of documenting the classes of two closed source projects.

We call the first study "open source study" and the second study "closed source study".

4.1 Open Source Study

We performed a study on three open source projects in the Department of Computer Science and Engineering at the University of Notre Dame. We hired 9 graduate students as participants. Participants were paid $100 each for their participation. We asked each participant to evaluate the importance of the methods in one of the three open source projects. In the end of the study, each of the three projects was evaluated by three of our nine participants.

4.1.1 The API Libraries and the Programming Tasks

The three open source projects that were evaluted are NanoXML, JExcelAPI, and JGraphT. They are all Java API libraries. The descriptions of the projects are listed below.

NanoXML A library that allows interfacing with XML files. Website: http://nanoxml.sourceforge.net/orig/

JExcelAPI A library for interfacing with Microsoft Excel files. Website: http://jexcelapi.sourceforge.net/

JGraphT A library for building and utilizing graphs. Website: http://www.jgrapht.org/

Before evaluating an API library, the participants were asked to do a programming task involving the API library, so that they could know the API library. We chose the three programming tasks that are small enough to be feasibly completed within a reasonable time limit. The projects, as well as a brief description of the programming task, are listed in Table 1.

4.1.2 Participants

We ensured that no participant had previous experience with the API library. This was done to avoid any previous bias with the project, and ensure that each participant was required to learn the API library. Each participant had at least 2 years prior Java programming experience, with an average experience of 5 years. All participants completed the programming task or spent 70 minutes attempting it, and used the entire remainder of the 90 minute time limit to do as much of the survey as possible. Of the participants, 2 participants did not complete the programming task. One programmer failed to complete the NanoXML task, while another failed to complete the JGraphT task.

4.1.3 Evaluation Process

We asked each participant to follow the steps below to evaluate an API library. Each programmer was given 90 minutes to complete the study.

1) **Do the programming task.** The purpose of the programming task was to acquaint each participant with the API library. A maximum of 70 minutes could be spent on the programming task. After the programming task was completed, or 70 minutes were exhausted (whichever came first), programmers were asked to continue to the next step.

   In this step, programmers were given a laptop with an Ubuntu virtual machine. Programmers were required to use the provided laptop and virtual machine

TABLE 1
List of API Libraries Used in Our Open Source Study and the Programming Tasks Given to the Participants

Library	Programming Task	LOC	# of classes	# of classes evaluated
NanoXML	Populate a given data structure with the contents of an XML file.	9,932	29	21
JExcelAPI	Create a Microsoft Excel workbook with two blank worksheets. Modify the API Library's worksheet creation to print the message "This worksheet was created by JExcelAPI" on all worksheets created using JExcelAPI.	79,153	511	52
JGraphT	Build a graph using JGraphT that matches a given graph image. Perform a depth-first-search on that graph to find a particular element.	29,988	256	209

to complete the task. The virtual machine was run using Oracle VM Virtual Box inside of Windows XP. This was done to ensure a consistent start state for each participant.

In each start state, the evaluated API Library had been loaded into an Eclipse workspace. Participants had the Javadocs for the project on the desktop of the virtual machine. Participants were instructed, in person, what their programming task was (the details about each task in Table 1). Each participant was shown where to find the Javadocs for the API Library. Participants were asked to use the Javadocs when learning the system to perform the programming task. The monitor was recorded using screen capture software while the participants completed the programming task. This allowed us to see how the programmer learned the system, and where in the API library's Javadocs they looked. After the participant completed the programming task or exhausted 70 minutes, whichever came first, they were directed to the survey.

2) **Fill the survey.** Each participant is given a survey, which asks the participant to identify how important it is to document certain methods in the API library.

The survey contained every method in the API source code, but the survey would present a method at a time to the participant. In addition to the method name, participants were given a link to the location in source code of the method. Participants were then asked to rate how important they felt the method was to document for program comprehension irrespective of existing documentation. Participants could answer that they believed the method was "not important to document," "somewhat important to document," "important to document," or "very important to document."

A screenshot of the survey is shown in Figure 2. Participants were told to evaluate the importance to document a method with respect to program comprehension. The participants were told to consider how important the method was to document for the overall API library, not just in relation to the programming task.

The order of the methods was randomized. Participants spent at least 20 minutes on the survey. If the participant had completed the programming task in less

TABLE 2
The number of classes evaluated by each participant in open source studies

Library	Participant Id	# of methods evaluated	# of classes evaluated
NanoXML	Participant 1	65	17
NanoXML	Participant 2	305	21
NanoXML	Participant 3	165	21
JExcelAPI	Participant 4	433	52
JExcelAPI	Participant 5	435	52
JExcelAPI	Participant 6	61	23
JGraphT	Participant 7	135	88
JGraphT	Participant 8	623	200
JGraphT	Participant 9	274	145

than 70 minutes, they were asked to spend more time on the survey up to a total of 90 minutes completing the study. Because the survey asked programmers to score every method in an API Library, no programmer completely finished the survey before the 90 minute time limit. We kept what survey data had been collected up to that point. This upper limit was put in place to control for participant fatigue. The number of scored classes in each project is listed in Table 1. The number of classes each participant evaluated is listed in Table 2.

4.1.4 Data Collection

We assign each participant response a numeric value. For example, if a participant in our open source study rated a method as "very important to document", we assigned that user response the score of 4. "Important to document" is assigned the score 3, "Somewhat important to document" is assigned the 2, and "Not important to document" is assigned the score of 1. This is an intuitive labeling for a Likert scale. We then assign to each method the average of all participant scores. In this dataset, a method with a higher average score is considered more important to document, and therefore a higher priority. A method with a lower average score is considered less important to document, and therefore less of a priority.

After the methods are scored, we assign to each class the average of the average scores for each scored method in the class. If a method was not rated by any participant, it did not
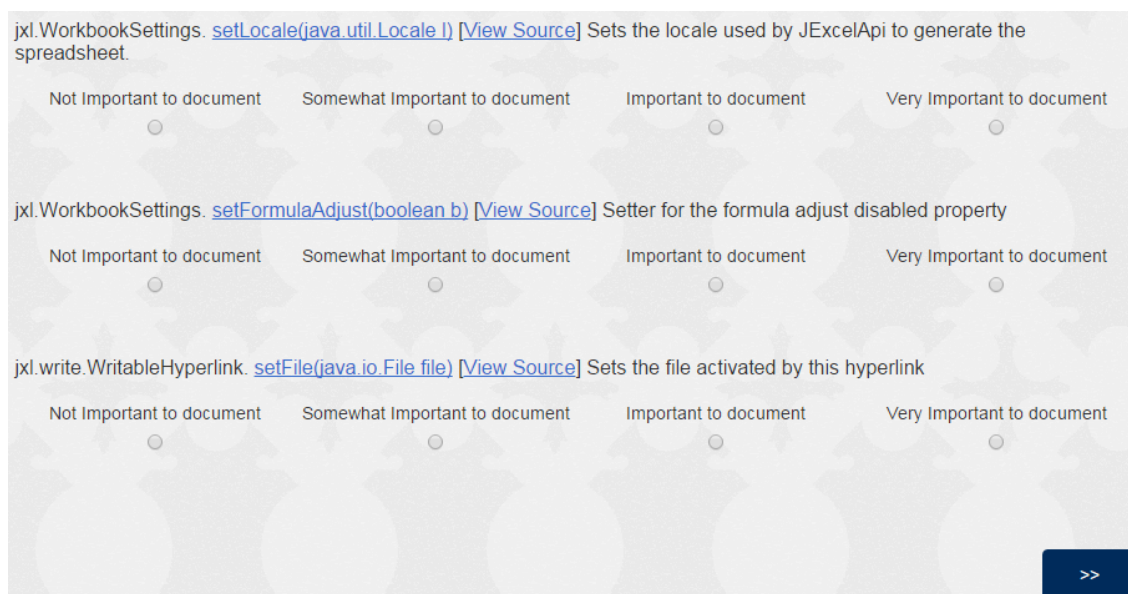
Fig. 2. A screenshot from our survey. Each page of the survey asked programmers to rate 5 questions. Only 3 are shown in this image due to space limitations. The order of the methods was randomized for each participant, and no methods were shown to the same programmer twice.

factor into its class's average. If no methods in a class were rated by participants, then that class was not included in the resulting dataset. A class with a higher average score is considered higher-priority to document, while a class with a lower average is considered lower-priority to document.

4.2 Closed Source Study

We performed a second user study at ABB Corporate Research. The closed source study focused on professional programmers evaluating proprietary source code. In the closed source study, participants were given two Java projects developed and maintained by ABB.

4.2.1 Closed Source Projects

Due to the proprietary nature of these projects, we will refer to them as Project B and Project D. These projects are both larger than 300 classes and over 2500 methods. Project B was a library component of another application, while Project D was a standalone application.

4.2.2 Participants

We had six participants in this study. All six participants were professional programmers at ABB Corporate Research. Five of the six participants evaluated Project B. None of the five participants had previous programming experience with Project B. All six participants evaluated Project D. Four of the six participants had previous programming experience with Project D.

4.2.3 Evaluation Process

Unlike the process in our open source study, we did not ask the participants to do programming tasks before they fill the survey. We decided to not use a programming task due to limits of time and access. Additionally, many programmers had already had prior programming experience with one of the projects. The programmers were also familiar with the
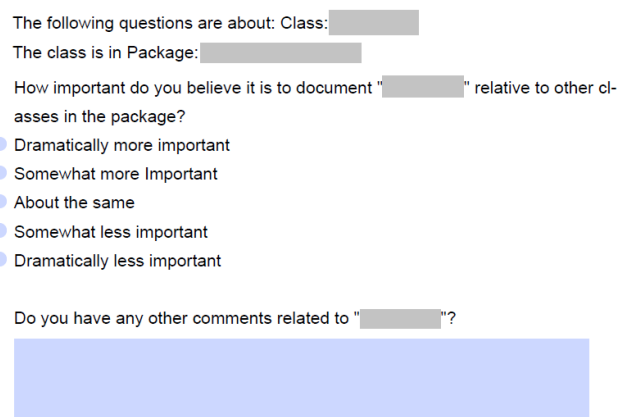

Fig. 3. A screenshot from our survey in closed source projects. Each page of the survey asked programmers to rate the importance of documenting a class. We removed the class names and Package names due to the proprietary nature.

domain and purpose of each project making a programming task unnecessary.

For each project, each participant was given 30 minutes to complete a survey, for a total of 60 minutes. This survey asked participants to evaluate the importance of Java classes to be documented in each project. We chose to analyze classes in the closed source study, rather than methods, due to limited time with the professional programmers. A screenshot of the survey is shown in Figure 3.

In the survey, participants could initially choose a package to evaluate. This was allowed so participants could evaluate packages they had more experience working with. After selecting a package, participants were given a random selection of five classes from the package to evaluate. All programmers who selected a package were initially shown the same five classes. This was to ensure we had multiple participants evaluating the same classes to evaluate the

TABLE 3
The number of classes evaluated by each participant in closed source studies

Library	Participant Id	# of classes evaluated
Project B	Participant 1	14
Project B	Participant 2	12
Project B	Participant 3	8
Project B	Participant 4	0
Project B	Participant 5	10
Project B	Participant 6	10
Project D	Participant 1	27
Project D	Participant 2	39
Project D	Participant 3	3
Project D	Participant 4	22
Project D	Participant 5	6
Project D	Participant 6	5

TABLE 4
Settings of the Open and Closed Source Studies

Setting	Open Source Study	Closed Source Study
Having programming tasks before the surveys	Yes	No
Assessed objects	Methods	Classes
Choice number for each question	4	5
Likert scale labeling	1 to 4	1 to 5
Optional comments	No	Yes

consistency of how participants view importance. After evaluating these five classes, the participant was given the option to go to a different package, or continue rating classes from the selected package.

For each class, the programmer was asked, "How important do you believe it is to document *Class* relative to other classes in the package?", where *Class* is the name of the Java class. The programmer could answer "dramatically more important," "somewhat more important," "about the same," "somewhat less important," or "dramatically less important." Programmers were instructed to consider documentation importance as it relates to program comprehension. Then, we have an optional text box for the programmer to enter any comment about the class. After 60 minutes had passed, programmers ended the study. We listed the number of classes each participant evaluated in Table 3.

4.2.4 Data Collection

We interpret the responses from participants in our closed source user study by assigning numeric values to each class based on participant responses. If a participant said a class was "dramatically more important" to document, that response was assigned a value of 5. Likewise, if a participant said a class was "dramatically less important" to document, that response was assigned a value of 1. Intermediate values were assigned corresponding numbers in turn. Each class's importance was the average of all participant evaluations of that class. If a class was not evaluated by any participant, it was not included in the dataset.

4.3 Comparison between the Two Studies

In this subsection, we compare the settings of the open and closed source studies in Sections 4.1 and 4.2. Table 4 shows the major differences between the two studies. First, because the participants in the closed source study are familiar with the evaluated projects, we did not ask the participants to complete a programming task before they took the survey.

Second, we decided to ask participants to rate methods in the open source study because the participants were not familiar with the open source projects and it was easier for the participants to rate smaller components like methods instead of classes. Because the participants in the closed source study are familiar with the projects and we had

limited time with the professional programmers, we asked them to rate classes directly.

Third, we had two different Likert scales in the studies. The difference does not affect our evaluations later, because our evaluations for the two study are independent.

Finally, we did not have comment boxes in the open source study so that the participants could rate as many methods as they can. To complement the open source study, we added comment boxes in the closed source study.

5 FEATURES

In this section, we describe three types of the features we collect for Java classes. The first type is the static source code attributes. The second type is the textual comparison attributes. And the third type is the vector-space model (VSM) approach, that is, tf/idf (Term Frequency times Inverse Document Frequency) for each word in the source code. All the attributes in the first and the second groups are listed in Table 5. We will discuss all the features in the following subsections.

5.1 Static Source Code Attributes

One set of the attributes we examine are attributes collected from *static source code analysis*. Static source code analysis is examining software by extracting characteristics from source code. This differs from dynamic source code analysis, which is examining software by executing it and observing the behavior. Binkley [19] describes static source code analysis as parsing the source code. Static source code analysis has been used in a variety of software engineering fields, including defect prediction [20], [21], [22], clone detection [23], and feature location [24] among many more applications.

In this paper, we consider three types of static source code attributes: *size* attributes, *complexity* attributes, and *object-oriented* attributes of Java code. Besides the attributes of the three types, we also consider **%Comments**, which is the number of lines that have comments divided by the total number of lines in a class.

5.1.1 Size attributes

Size attributes are static source code metrics that refer to the size of a section of source code. Many size attributes center around the number of lines of code in source code. However, there are multiple ways to measure lines of code in source code. In this paper, we consider two size attributes: **lines of**

code (LOC) and **number of statements (Statements)**. LOC is the number of lines in a Java class, including the comments but excluding the empty lines. Statements attribute is the number of executable lines of code in a Java class.

5.1.2 Complexity attributes

Complexity attributes are static metrics that measure the complexity of software. In this paper, we included 11 complexity attributes. Attribute **%Branch** is the percentage of the branch statements in a class. The branch statements includes `if`, `else`, `for`, `do`,`while`, `break`, `continue`, `switch`, `case`, `default`, `try`, `catch`, `finally`, and `throw`.

Attribute **Calls** is the number of method calls in a class. **Calls per Statement** is the number of calls in a class divided by the number of the statements in the same class. **Methods per Class** is the number of methods in a class. **Statements per Method** is the total statement count divided by the total method count in a class.

One popular complexity measurement is McCabe's cyclomatic complexity [25]. McCabe's cyclomatic complexity is based on control flow graphs. A control flow graph is a directed graph that represents statements in a section of source code. Each statement is a node. Edges represent possible control flow patterns. For a method, Cyclomatic complexity is calculated using $v(G) = e - n + 2$, where e is the number of edges in the flow graph, and n is the number of nodes in the flow graph. Cylcomatic complexity has been found to be a good predictor of effort estimation [26] and software defects [27], [20]. It should be noted, however, that some researchers disagree that cyclomatic complexity is useful for defect prediction [28], [29]. We consider attributes **Avg. Complexity**, **Max Complexity**, and **Weighted Methods per Class (WMC)** of a class, which are the arithmetic average, the max value, and the sum of all the Cyclomatic complexity values measured for all the methods in the class.

Additionally, we consider the attributes about branch depths. For a statement, we measure the depth of the nested branch the statement is in. For example, in code `if(...){ if(...){ s1; }}`, statement `s1` has the branch depth of two. Attribute **Avg. Depth** is the average branch depth of all the statements in a class. And attribute **Max Depth** is the maximum branch depth found in a class.

The last attribute is **Number of Class Fields (NOF)**, which is the number of fields of a class.

5.1.3 Object Oriented attributes

Some *Object Oriented* (OO) attributes relate to the inheritance structure of object oriented languages. Chidamber and Kemerer [30] proposed several OO metrics to analyze the structure of object oriented programs. We consider five OO attributes in this paper.

The first attribute is **Depth of Inheritance Tree (DIT)**, which refers to the number of ancestor classes a given class has. The more ancestor classes a given class has, the higher the potential complexity. Thus, a large DIT can suggest higher complexity [22]. Second, we consider attribute **Number of Sub-Classes (NSC)** (also referred to as Number of Children (NOC)) is the number of children a class has. A class with a large number of subordinate classes may need to be tested more thoroughly because of its broad impact [30].

The third attribute is **Lack of Cohesion of Methods (LCOM)**, which is a measurement of the cohesiveness of a class. This attribute is calculated with the Henderson-Sellers method [31], which is

$$LCOM_{Henderson} = (\frac{\sum_{f \in F} m(f)}{|F|} - m) \times \frac{1}{1 - m} \qquad (1)$$

where F is the set of all the fields in the class; $m(f)$ is the number of methods accessing the field f; m is the number of the methods. A low LCOM value indicates a cohesive class, and a LCOM value close to 1 suggests that the class lack cohesion.

The fourth attribute is **NORM** which is the number of the methods in a class that are overriden by its child classes. Finally, we consider attribtue **Abstract**, which is whether a class is abstract or not.

5.1.4 Extraction Method

To collect source code attributes, we use two tools: Source-Monitor and Metrics. SourceMonitor[2] is a standalone program that analyses Java source code. SourceMonitor can produce metrics related to the size of the Java source code, such as lines of code and number of statements. Additionally, SourceMonitor can calculate the McCabe cyclomatic complexity of source code. Metrics[3] is an Eclipse plug-in for performing static analysis of source code. Metrics can determine various object-orientation related metrics, such as those proposed by Chidamber and Kemerer [30].

5.2 Textual Comparison Attributes

Textual analysis, with relation to source code, involves extracting semantic information from source code identifier names. Identifier names are a powerful tool for developers to communicate to program readers [32], [33]. Our previous work, McBurney, *et al.* [34], investigated the textual similarity between source code and summaries of that source code. Our work found that readers perceive summaries that are more similar to source code as more accurate than summaries that are textually dissimilar. In this paper, we consider two textual analysis attributes: Overlap and STASIS. We apply these attributes by comparing Java source code to the homepage of a project website. It should be noted that this technique is, in effect, a type of static source code attribute. However, in the scope of this paper, we consider this technique distinct from the extraction of static source code attributes. This is because the goal of this approach is to consider keyword similarity, rather than the structure of the code.

5.2.1 Overlap

Overlap is a textual similarity metric that compares the literal representation of two bodies of text. To calculate Overlap, each body of text is treated as a set of words. Overlap is equal to the percentage of shared words between these two sets divided by the the total number of unique

---

2. http://www.campwoodsw.com/sourcemonitor.html
3. http://metrics.sourceforge.net/

words in both sets combined. To calculate Overlap of text body A and text body B, we have the formula:

$$\frac{|WT(A) \bigcap WT(B)|}{|WT(A) \bigcup WT(B)|} \qquad (2)$$

where $WT(A)$ and $WT(B)$ are the set of unique words of A and B.

5.2.2 STASIS

STASIS, developed by Li *et. al* [35], is a *semantic similarity* metric. *Semantic similarity* refers to the similarity between the meanings of two bodies of text. STASIS contrasts with Overlap, as Overlap does not consider semantic information. STASIS accounts for the semantic meanings of words by leveraging WordNet [36]. WordNet [36] is a hierarchical database of English words organized into sets of synonyms, or *synsets*. In WordNet, words in the same synset are very similar, and synsets that are close to each other in the hierarchical structure are more similar. STASIS uses this to calculate the similarity of two words.

To calculate the similarity between two text bodies, given two text bodies A and B, we first get a set of all the unique words W from A and B. Then, we create a vector of the size of W, which is to represent A. Each element in the vectors corresponds to a word in W, and the initial values for all the elements are zeros. Then, for each word in A, we assign one to the corresponding element of the vector of A; for each word not in A (but in B), we calculate the similarity between the word and every word in A, and we assign the highest similarity score to the corresponding element of the vector of A. Then, we get the vector for B in a same way. Finally, the similarity between A and B is the cosine similarity of the two vectors as follows:

$$\frac{V_A \cdot V_B}{\|V_A\|\|V_B\|} \qquad (3)$$

where V_A is the vector for A and V_B is the vector for B.

STASIS can also consider word order similarity. However, we do not consider word order in this work. This is because source code word order is often unrelated to summary word order [34].

5.2.3 Extraction Method

We used words from only the main page of the project website (see Appendix A for details.)

We acknowledge that a project website may not be suitable in all circumstances. For instance, if the website is out of date, or non-existent, then it would not be useful for this approach. However, in our study, we found each project's website was suitable in our test programs. The project website homepage was selected, because all the test projects had a publicly available homepage. The homepage gives a broad overview of the purpose and features of the software.

We pre-processed each Java class file by splitting on and removing all special characters. Then, we kept two sets of words for each class file: one with camel case splitting, the other without camel case splitting. Existing comments (including JavaDoc comments) are included as part of the source file.

We calculated Overlap and STASIS for both sets of words (with or without camel case splitting, but always with splitting on special characters). STASIS is configured to not consider word order. The result of this analysis is a set of scores between zero and one, where a larger similarity score means the source code is more similar to the project website. The implementation of STASIS is accessible at http://www.cis.upenn.edu/~paulmcb/research/doceffort/stasis.py

5.3 Textual VSM and Tf/idf

Our final source code analysis approach uses a vector space model (VSM) [37] to classify Java class files as either important or not important. The purpose of this approach is to evaluate the effectiveness of textual analysis in automatic documentation prioritization without the benefit of a manually written document, such as a project website.

First, we extracted keywords from all the Java files by removing all special characters and numbers, then splitting on camel case. Additionally, we consider the class files' words both with stemming and without stemming. Second, we converted the words into a vector space model using term frequency/inverse document frequency (tf/idf), which is done by Weka's `StringToWordVector` filter. Tf/idf is an approach for determining how important a term is by considering how much the term appears in the document divided by how many documents the term appears in the corpus. In this way, a word that appears frequently in one document, and is very rare outside the document, is important to that document. So in the VSM approach, we had an attribute for each word in all the class files.

6 MODEL AND TRAINING

We built a classification model approach to identify whether or not a given Java class was important to document based on the features we defined in Section 5. We define "important" as being in the top 25% in our gold set for a Java project. While we regret that this threshold of 25% appears arbitrary, given the preliminary nature of this study, a percentage threshold is our best option. The other possibility would have been to use an absolute threshold. However, this would require making assumptions about some minimal "importance" score. Given the foundational and preliminary nature of this study, we do not feel this would be appropriate. A threshold of 25% was specifically chosen because it gave less noise than lower percentages, but was not as coarse as higher percentages.

To build the classification model, we created an artificial neural network (ANN) [38] by using Neuroph[4]. We selected ANN in our study for two reasons. First, of available classification techniques, ANN performed the best in pilot testing. Second, ANN is widely used in the current literature to address similar problems [39], [40].

We trained an ANN model for each open/closed source project and each of the four group of the features, which are 1) static source code attributes, 2) textual comparison attributes, 3) the VSM approach, and 4) a combination of 1) and 2). For each ANN model, we a performed 10-fold cross-validation for evaluation.

4. http://neuroph.sourceforge.net/

TABLE 5
List of Static Source Code Attributes and Textual Comparison Attributes in Our Approach

Type		Attribute	Description	Source
Static	Size	LOC	Lines of code (including comments, excluding empty lines)	
		Statements	Executable Lines of Code	
	Complexity	%Branch	Percentage of statements that branch	SourceMonitor
		Calls	Number of calls	
		Calls Per Statement	Number of calls / number of statements	
		Methods Per Class	Number of methods per class	
		Statements Per Method	Average of statements per method	
		Avg. Depth	Average branch depth of statements	
		Max Depth	Maximum branch depth of statements	
		Avg. Complexity	Average McCabe Cyclomatic Complexity of methods	
		Max Complexity	Maximum McCabe Cyclomatic Complexity of methods	
		WMC	Weighted methods per class	Metrics[**]
		NOF	Number of class fields	
	OO[*]	DIT	Depth of inheritance tree	
		NSC	Number of child classes	Metrics
		LCOM	Lack of cohesion of methods	
		NORM	Number of overriden methods	
		Abstract	Whether or not the class is abstract	
	Others	%Comments	Percentage of lines of code with comments	SourceMonitor
Textual Comparison		Class Appearance	Whether or not the class name appears	
		Package Appearance	Whether or not the package name appears	
		Combination Appearance	Whether or not package and class name appears	
		Overlap w/ Splitting	Literal similarity with splitting on camel case.	Scripts[***]
		Overlap w/o splitting	Literal similarity without splitting on camel case.	
		STASIS w/ splitting	Semantic similarity with splitting on camel case.	
		STASIS w/o splitting	Semantic similarity without splitting on camel case.	

[*] OO: Object Oriented attributes in Section 5.1.
[**] Metrics is an Eclipse plugin.
[***] We wrote Python scripts to calculate the textual comparison attributes. The script for calculating STASIS is accessible at http://www.cis.upenn.edu/~paulmcb/research/doceffort/stasis.py

6.1 Parameters

Parameter setting influences significantly the performance of a machine learning algorithm [41]. Within ANN, we used two hidden layers. There was a significant performance increase between 1 hidden layer and 2 hidden layers. However, increasing the number of hidden layers past 2 did not yield further improvements.

6.2 Parameter Tuning

We used a trial-and-error methodology to tune parameters, where several parameter values are tested and compared. For each couple (algorithm, software system), we use the trial and error method in order to obtain a good parameter configuration. Each ANN is executed 30 times with different configurations and then comparison between the configurations is done using the Wilcoxon test [42] with a 99% confidence level α (= 1%). The Wilcoxon signed-rank test is a non-parametric statistical hypothesis test used when comparing two related samples to verify whether their population mean-ranks differ or not. The latter verifies the null hypothesis H0 that the obtained results of two algorithms are samples from continuous distributions with equal medians. The alternative hypothesis H1 is that the results of two algorithms are not samples from continuous distributions with equal medians. The p-value of the Wilcoxon test corresponds to the probability of rejecting the null hypothesis H0 while it is true (type I error). A p-value that is less than or equal to α (≤ 0.01) means that we accept H1, and we reject H0. However, a p-value that is strictly greater than α (> 0.01) means the opposite. In this way, we could decide whether the outperformance of each algorithm

over one of each of the other algorithms (or the opposite) is statistically significant or just a random result. All the results presented in this experiment were statistically significant on 30 independent runs using the Wilcoxon rank sum test such that α (< 1%).

It is important to note that all heuristic algorithms have the same termination criterion for each experiment (same number of evaluations) in order to ensure fairness of comparisons. Based on the trial and error method, we used a maximum number of iterations of 1000 runs, the learning rate was 0.25, the number of hidden nodes was 10 and a feed-forward fully connected architecture.

6.3 Feature Selection

Due to the number of static source code attributes we considered, we performed feature selection using a Chi-Squared test to reduce the number of features. Other techniques at this time were not tried. A full study to determine the best feature selection approach is beyond the scope of this paper. Given the small number of textual comparison attributes, we thought that the risk of greater overfitting and model uncertainty outweighed the potential benefit of a reduced model (via feature selection). Furthermore, we did not perform feature selection on the combination of static source code attributes and textual comparison attributes, because this combination is compared with the textual comparison attributes, on which we did not perform feature selection. For the VSM approach, this data is more complicated so we did not perform the feature selection.

7 EVALUATION SETTING

In this section, we describe the setting of our evaluation, in which we evaluate the models with different candidate features (Section 6). Our evaluation is to determine whether static source analysis can be used to automatically detect sections of source code with a high documentation priority. The classification models predict whether an input Java class is important. We use the granularity of class, because we found methods to be too noisy. Additionally, using class was the best option to ensure the broadest coverage given the limitations of time in our study.

We evaluate the classification models by assessing the effectiveness and the ranking relevance of each model. First, we compare different models to determine the effectiveness of different features in predicting documentation priority. If the model with some candidate features performs better in our study, we argue that the features in this model are better for predicting documentation priority.

Second, we obtained a ranking from each model and evaluated the ranking relevance for each model. Although our ANN models are binary classifiers, given a Java class, the models can output a value between 0 and 1. These values indicate the importance of the classes, so a ranking of the classes can be generated by comparing the values.

7.1 Research Questions

The goal of this evaluation is to determine the effectiveness of source code attributes in predicting documentation effort priority. We ask the following research questions to assess the effectiveness and the ranking relevance of each model.

Assessing Effectiveness. To assess effectiveness, we have the following three research questions:

RQ_1 To what degree do static source code attributes differ in prioritization effectiveness with textual comparison attributes?

RQ_2 To what degree do the combination of static source code attributes and textual comparison attributes differ in prioritization effectiveness with only textual comparison attributes?

RQ_3 To what degree do static source code attributes and/or textual comparison attributes differ in prioritization effectiveness with textual VSM?

In these research questions, *prioritization effectiveness* refers to whether important Java classes are correctly predicted by a given classification model. We measure effectiveness by considering the precision and recall of our classification models with respect to classes that are deemed important. The purpose of RQ_1 is to determine whether a classification model using only static source code attributes or a classification model using only textual comparison attributes performs better. The purpose of RQ_2 is to determine if static source code attributes add meaningful information to textual comparison attributes. If static and textual attributes in combination perform better than just textual attributes, it would suggest that collecting and learning on a combination of static and textual attributes would be an ideal approach. The purpose of RQ_3 is to compare the results of the classifier based on a textual VSM approach with

those from the combination of static source code attributes and textual comparison information.

Ranking Relevance. To assess the ranking relevance of our models, we ask the following three research questions:

RQ_4 To what degree do static source code attributes differ in ranking relevance with textual comparison attributes?

RQ_5 To what degree do static source code attributes and textual comparison attributes differ in ranking relevance with only textual comparison attributes?

RQ_6 To what degree do static source code attributes and/or textual comparison attributes differ in ranking relevance with only textual VSM?

In these research questions, *ranking relevance* refers to the accuracy, precision, and recall of the top $k\%$ classes. Accuracy refers to the percentage of classes for which we make at least one correct recommendation in the top $k\%$ ranked classes (labeled as **Accuracy@k**). Precision refers to the percentage of classes in the top $k\%$ ranked-list generated by our models that are in the top k% most important classes in our gold set (labeled as **Precision@k**). Recall refers to the percentage of classes in the top $k\%$ most important classes in our gold set that are in the top $k\%$ ranked-list generated by our models (**labeled as Recall@k**). These research questions are important to consider, as they will allow us to see the trade-off between accuracy and precision when k increases. We note that we did not use AUROC as one of our measures, because our goal is not to pick a definitive winner. Instead, as our study is foundational and preliminary, we are trying to understand the relative quality and performance characteristics for our models. So we used precision, recall, and accuracy to understand the relative performance and potential tradeoffs.

7.2 Reproducibility

To ensure reproducibility, we have made anonymous survey and research data from our open source study available via an online appendix:

http://www.cis.upenn.edu/~paulmcb/research/doceffort/

Additionally, we have made the virtual machines for the starting state of our programming tasks available upon request. Because of the proprietary nature of our closed source study, we cannot make that data publicly available.

8 RESULTS

In this section, we discuss the results of predicting documentation priority on data collected from our models. We used 10-fold cross validation to evaluate the models. Figure 4 summarizes our findings regarding the correctness of the results generated by the models with the four groups of the features. The VSM approach correctly recommended the important classes for documentation with more than 80% of precision and recall on the five systems. This approach performed the best overall. The average precision and recall of the classification models using the textual comparison attributes is around 75%. The textual comparison attributes performed very closely with the VSM approach. Surprisingly, the static source code attributes resulted in the lowest precision and recall score with an average for both lower
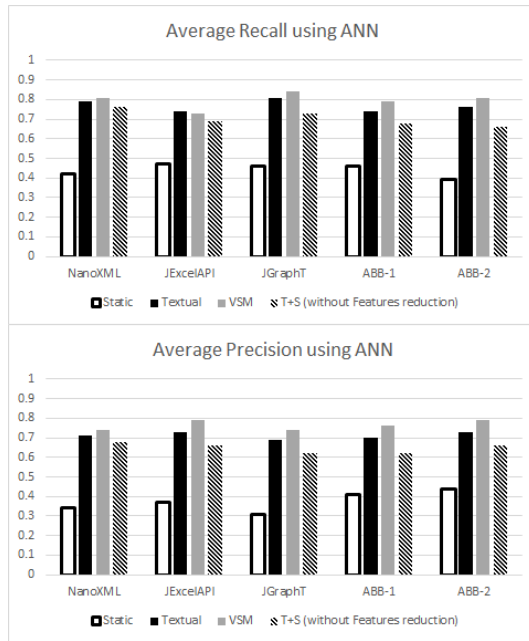
Fig. 4. These two column graphs show the average recall (top) and the average precision (bottom) for each approach over a 10-fold cross validation using ANN. The two textual approaches consistently outperform the static and static+textual approach. ABB-1 refers to the closed source Project B, while ABB-2 refers to Project D.
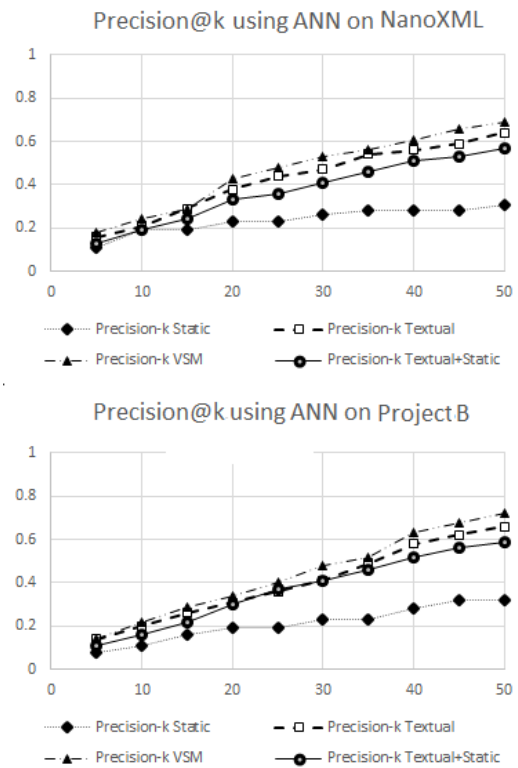

Fig. 5. These two line graphs show the precision where the top k percent of classes ranked by our user study is considered important to document. The graphs represent learning on NanoXML (top) and the closed-source Project B (bottom). We limit showing only two graphs for this project, however other graphs exhibited similar behavior and are available in our online appendix.

than 50%. The performance of the hybrid textual comparison and static source code attributes was lower than the VSM approach and the textual comparison attributes, but higher than the static source code attributes. An interesting observation is that the performance of the VSM approach in terms of precision and recall appears to be independent of the size of the system and the number of expected classes.

Using these results, we can answer RQ_1. We have found that using static source code attributes as features for a predictive model resulted in substantially lower precision and recall than using either textual comparision attributes or the VSM approach. Further, we can answer RQ_2 and RQ_3 by noting that the combination of the static source code attributes and the textual comparison attributes was outperformed by textual comparision attributes only and the VSM approach.

Figures 5 through 7 are used to evaluate the quality of the class importance ranking proposed by the different models. To this end, we present the Precision@k, Recall@k and Accuracy@k results for the models, with k ranging from 5% to 50% of top classes in the ranking. In this paper, we show six representative graphs. However, the other graphs are available in our online appendix (see Section 7.2). Based on all the measures, the VSM approach achieves the best results on all five projects. For example, on the JGraphT our VSM approach achieved Accuracy@k of 92%, 88%, 84%, and 79% for k = 10, 20, 30, 40 and 50 respectively. The same observation is valid also for the precision@k and recall@k results. That is to say that most of important classes are located in the top of the list of recommended classes for documentation using the VSM approach. Thus, the ranking may speed-up the process of locating important classes to document by programmers. Using this information, we

answer RQ_4 and RQ_5 by noting that the static code source code attributes, with or without the textual comparison attributes, resulted in poorer documentation effort prioritization than the textual comparison attributes and the VSM approach. The answer to RQ_6 is that the VSM approach outperformed all other attributes over our data. The implications of these answers are discussed in Section 11.

9 COMMENTS FROM EVALUATORS

In this section, we discuss the findings from the comments made by the participants in the closed source study. The participants in the closed source study were asked to provide commentsp justifying the level of importance they assigned to each Java class. We chose to add optional comment boxes in our closed source study due to the participants in our closed source study being professional programmers with, on average, more programming experience than those in our open source study. Further, the open-source study was much longer in terms of time, and we were concerned about fatigues. Due to the fact that the closed source study used proprietary source code, names of classes, methods, and variables are redacted. Additionally, we cannot provide any code examples.

Among the comments, we found a disagreement over the importance of document interfaces. Several study participants said that implementation should be documented.
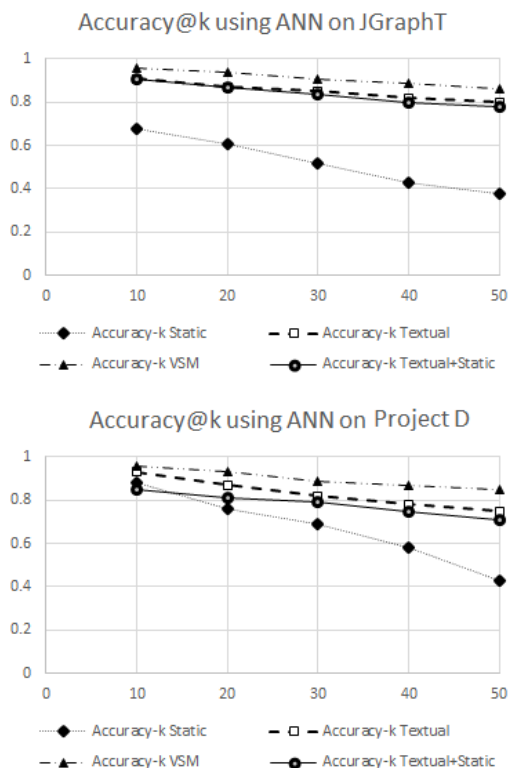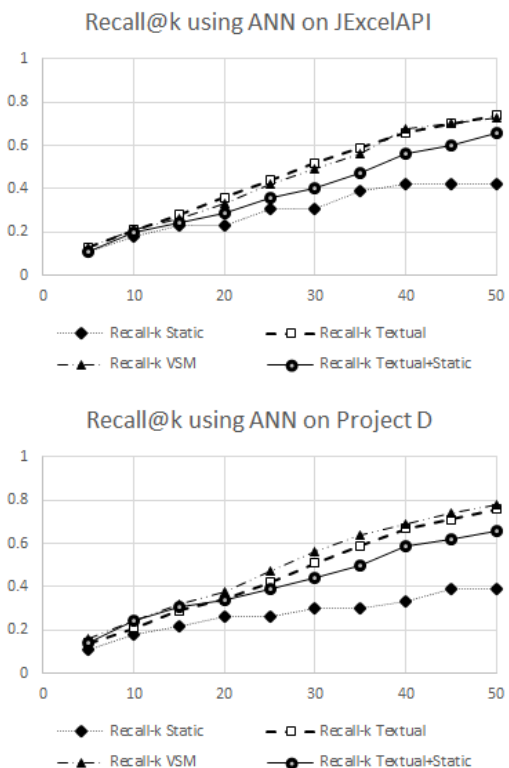
Fig. 6. These two line graphs show the recall where the top *k* percent of classes ranked by our user study is considered important to document. The graph represents learning on the closed-source JExcelAPI (top) and the closed-source Project D (bottom). We limit showing only two graphs for this project, however other graphs exhibited similar behavior and are available in our online appendix.

Fig. 7. These two line graphs show the accuracy where the top *k* percent of classes ranked by our user study is considered important to document. The top graph represents the open source project JGraphT, and the bottom graph represents the closed-source Project D. We only show these two graphs, however other graphs exhibited similar behavior and are available in our online appendix.

A sample of related comments from different participants follow:

- "Interfaces typically need pretty good documentation so you know what the expectations are."
- "I like to have interfaces well-documented..."
- "This class is an interface- so I think it's very important to document it..."

However, one participant would note "interface" in the comments when a class was an interface, and rate the class as "Somewhat less important to document" relative to other classes in the package. This dissent was less common, but it notes a disagreement in opinion on what should be documented in a professional environment.

Several participants state that when a class contained methods that were always or almost always simple or obvious, it was less important to document. A sample of comments follows:

- "Pretty straightforward implementation of a known pattern- with some adaptations...less important to document."
- "Seems straightforward- so less documentation needed."
- "Simple getters and setters ... documentation is somewhat redundant."

This is interesting, because simplicity, especially in the form of "simple getters and setters", should be apparent in metrics such as average McCabe Cyclomatic Complex-ity [25] of methods in the class. However, our machine learning suggests that static metrics, including average method complexity per class, are not high-quality predictors of documentation importance. This is complicated further by several comments reference complex methods and control flow as important to document:

- "Great method documentation- but control flow sometimes needs better documentation"
- "*Name of a method* in the class has a significant amount of undocumented logic and control code."
- "...more complex methods may need more documentation."

It is very surprising that participants, in their comments, suggest that source code complexity should indicate documentation importance. However, this directly conflicts with our quantitative findings. We do not have a clear explanation for this contradiction. It is possible that programmers attribute complexity to documentation importance out of habit or training, but ultimately judge importance for program comprehension based on the high-level goals of a given class. This possibility will need to be the subject of a more focused future study. Due to the proprietary nature of the source code, we cannot show the source code that generated these comments, nor can we discuss it in any detail.

Several participants made reference to the identifiers, or names, of methods and variables in the class.

- "Relatively little logic- and the method names are fairly explanatory."
- "It's always good to document classes that hold data- which is what this class seems to be. However- they don't need a ton of documentation because they are usually easy-to-understand (e.g. I understand what the field *name of a variable* likely contains because of its name)."
- "Because abbreviations are used...it is difficult to understand this class."

The importance of identifiers has been noted by Deissenbock and Pizka [43] as well as by Lawrie *et al.* [44], [32]. Bulter *et al.* [45], [46] have connected flaws in naming conventions with poorer code quality. The comments by our participants seem to show agreement with the literature. Additionally, these comments help to validate that usefulness of our textual based machine learning approaches at predicting documentation importance.

10 THREATS TO VALIDITY

As with any software engineering user study, a key threat to validity lies in the selected participants. All programmers bring their own experience and biases into any study. To mitigate this threat in our open source study, we ensured no programmer had prior experience with their assigned API library. Further, the participants come from several different research backgrounds, giving us a diversity of views. We felt that it was important that no participant have prior experience, as we wanted to monitor how a programmer learns a new API library. However, being unfamiliar with a system can create a bias from ignorance. To mitigate this threat, in our closed-source study, the participants were familiar with the projects. Additionally, several participants had programming experience on one project.

Another source of threat to validity comes from studying exclusively API libraries in our open source study. We chose to study API libraries specifically, as these were easily adaptable to programming tasks. Because the participants were not experienced with their given API libraries before the study, we had to keep the tasks small enough to be feasibly completed in the allotted time. Extending the time to perform more advanced tasks would have resulted in a fatigue factor that we wanted to avoid, as the survey completed after the programming task was the primary means of data collection. Using different API libraries, or using Java programs other than API libraries, may result in different conclusions.

A third threat of validity emerges from having different levels of granularity in our studies. Due to time limitations, we were unable to have the developers in our closed-source study evaluate projects at the method level, and instead limited the evaluation to class level. We note that while this introduces a bias, our results in Section 8 were remarkably consistent in both settings. We believe that the results being so consistent actually strengthens the generality of our approach. Thus, it could be also considered as an advantage of the approach since programmers may have different preferences as to whether to document at the class or method level.

A fourth threat to validity comes from the use of a programming task to acquaint the participants with the system in our open source study. The programming task given could bias the participants towards thinking certain methods or classes are disproportionately more important because the given task required the programmer to use it. The decision to use a programming task was made because we felt that there is a greater threat to validity by asking programmers unfamiliar with a given library to determine what in the library is important. By giving a small programming task, we ensure that programmers have become at least somewhat familiar with the system they are evaluating. To mitigate this threat in our overall paper, our closed-source study has no programming task, and relies upon professional experience with a given piece of software, or a familiarity with its purpose, to avoid biasing the participants. This can come with its own threats to validity, as programmers may be biased by previous experience working on specific sections of source code.

A fifth threat to validity is that some classes/methods were evaluated by only one participant (both in open source study and closed study). Therefore, there is no consensus on the importance of these classes/methods. This is a design decision we made in our studies, because we want to maximize the number of the evaluated classes/methods.

A sixth threat to validity comes from the Likert scale we used in the surveys. Likert scales produce ordinal data, but we assigned integer values to the scales and computed average of the values, which may introduce a bias to the results [47], [48], [49]. We computed averages in two cases. In the first case, we used the average as a proxy for consensus when several participants rated the same component. In the second case (which occurred in only the open source study), we computed the average of the methods' rates as the rate of a class. We note that we used the rating to partition the classes into two groups: top 25% and bottom 75%. We then claim that classes in the top 25% are more important to document than the classes in the bottom 75%. Specifically, we do not claim that any particular class in the top 25% is more important to document than any other class in the top 25%. Although this use of averages results in a threat to validity, we believe the threat to be minor. In particular, we acknowledge that there may be little difference in importance among the classes in the vicinity of the 25% threshold. To be safe, we included ties at the 25% to mitigate our threat.

The parameter tuning of the model building algorithm in our experiments creates an internal threat in that we need to evaluate its impact on the obtained results in our future work. In our current experiments, we used a trial-and-error strategy on over 30 runs for each system. Another internal threat is related to the possible correlation between some features used in our machine learning model. We are planning to use a dimensionality reduction technique to possibly reduce the number of features of our model. Finally, we considered each of the different features with equal importance. However, some attributes are possibly more important than others when ranking documentation. Our plan for future work is to empirically evaluate the importance of the used metrics to find the best weights to assign for the features using a learning-to-rank technique.

11 DISCUSSION

Our paper explores the idea of prioritizing documentation effort using static source code analysis and textual evaluation. Surprisingly, we found that static source code attributes are a poor predictor of documentation effort priority. This is surprising given that static source code analysis has been used to find "important" sections of source code for program comprehension [9], [10] and for finding source code examples for documentation [14]. Given how poorly static source code attributes performed across all five source code projects in our study, we are pessimistic of their usefulness in this field.

Interestingly, our textual comparison attributes performed consistently well over all five projects. This is interesting because each attribute relies on a project website built for the project. However, the size of the project websites, as well as their level of detail, varied dramatically. To see such consistently positive results given this limitation shows promise for textual analysis in documentation prioritization. This is supported by the performance of our VSM approach, which also performed well. Our intuition for these positive results is that the perceived importance of documentation is related to high-level concepts of the projects, which are the contents of the project websites. Thus, it is our belief that future research into documentation effort prioritization should continue to explore textual analysis.

11.1 Future Work

Future work may consider other static source code attributes that we did not investigate, such as Halstead complexity metrics [50]. Our intuition is that other internal static source code attributes would at best improve only marginally over the results in this paper. However, other forms of static analysis, such as historical changes or external attributes may prove useful.

Our open source projects' homepages list the main features of the projects. The lists may be the reason that our textual comparison attributes work better than the static source code attributes. Future work may consider calculating textual comparison attributes based on only the words of the main features instead of all the text in the homepages.

Based on the qualitative results in Section 9, several other attributes may be considered in the future work of documentation prioritization. The first candidate is the quality of the identifiers, which may have an impact on the documentation prioritization. The second candidate is the role of the class in the system [51], [52], [53], [54].

The next step towards predicting high documentation effort priority areas of source code would be to research cross-project learning. If cross-project learning is feasible, then theoretically researchers could create a corpus of existing projects with "gold sets" of high documentation effort priority classes. This corpus could be used with machine learning to provide a ranking of classes in a project by documentation effort priority. This would ensure developers could spend sufficient effort documenting important sections of source code. Lower priority sections of source code could have their documentation supplemented by automatic summarization approaches, such as our previous work [55] which generates natural language summarizations of the context of Java methods.

Our intuition is that textual comparison attributes, collected by comparing to existing documentation, is the most likely candidate for cross-project learning. The reasons for this belief stem from the consistent performance of our textual comparison metrics as well as their applicability. While the VSM approach performed well, the words that indicate importance in one project are often completely unrelated to words that indicate importance in another project. This prevents the VSM approach from working on a corpus of projects. Static source code attributes would be applicable to external projects, but the consistently poor performance in our study suggests that learning using such attributes would not be effective.

Finally, our current approach helps programmers locate important classes to document. But our approach does not give the programmers suggestions about where and what to document in a class file. Future work includes looking for important methods and fields to document in a class.

12 RELATED WORK

The work in this paper is related to *source code summarization*. *Source code summarization* approaches can automatically generate descriptions of a software system. These approaches aid program comprehension by selecting important information from source code to present to program end-users [56], [57]. Haiduc *et al.* [56] use text retrieval techniques, specifically Latent Semantic Indexing (LSI) [58], to select the most relevant terms in Java methods. Further work by Haiduc *et al.* [57] also showed that a Vector Space Model (VSM) could also be used to automatically generate extractive summaries. De Lucia *et al.* [59] and Eddy *et al.* [60] found that the "simpler" VSM based approach could outperform LSI and Pachinko Allocation Model (PAM) [61] at generating extractive summaries. Rodeghero *et al.* [62] would later modify Haiduc *et al.*'s VSM approach based on the results of a programmer eye tracking study. Recently, researchers have developed tools that generate natural language summaries of sections source code, such as Java methods and classes [55], [63], [64], [65]. These approaches could be used to summarize low documentation effort priority sections of source code automatically. However, natural-language summaries generated by state-of-the-art source code summarization approaches are not yet of the same quality as manually-written expert summaries [18]. Because of this lower quality, we argue high documentation effort priority sections of source code should be documented manually.

Recently, research has been carried out to mine software version repositories [66], [67] using machine learning techniques. One of the important challenges addressed is to detect and interpret groups of software entities that change together. Soetens *et al.* [68] proposed an approach to detect (reconstruct) refactorings that are applied between two software versions based on the change history. The scope of this contribution is different than the one proposed in this paper, to the best our knowledge our work is the first study to prioritize software documentation effort using mining approaches. Ratzinger et al. [69] used change history

mining to predict the likelihood of a class to be refactored in the next two months using machine learning techniques. In their prediction models, they do not distinguish different types of refactorings; they only assess the fact that developers try to improve the design. In addition, data extraction from development history/repository is very well covered. Research has been carried out to detect and interpret groups of software entities that change together. These co-change relationships have been used for different purposes. Zimmermann *et al.* [67] have used historical changes to point developers to possible places that need change. In addition historical common code changes are used to cluster software artifacts [70], to predict source code changes by mining change history [71], to identify hidden architectural dependencies [72], or to use them as change predictors [73]. The closest work to our contribution is the study proposed by Ouni *et al.* [74] to prioritize refactoring opportunities for software maintenance using multi-objective optimization.

Profiling is a dynamic program analysis methodology where different attributes of code, such as memory size, frequency of instruction or method calls, etc. Most commonly, profiling is used to optimize program execution. However, profiling has been used in a number of software engineering tasks. Reps *et al.* [75] and Chilimbi *et al.* [76] used profiling to detect and locate software defects. Lutz *et al.* [77] used profiling to detect when safety critical systems are likely to fail due inappropriate instruction calls, memory limitations, etc. Profiling can identify, in practice, which methods are called most frequently in common execution chains [78]. Using this information, we could get a set of dynamic program analysis metrics. These metrics would serve as another candidate feature set that could predict for documentation prioritization.

13 CONCLUSION

In this paper, we researched whether static source code attributes and textual information in source code could be used to automatically prioritize documentation effort. To the best of our knowledge, this is the first work to conduct a study in order to evaluate documentation effort prioritization. We conducted two user studies to determine what sections of source code are most important to document. We surprisingly found that common static source code attributes are not good predictors of documentation effort. However, we found that textual analysis of source code and existing documents can be effective predictors of documentation effort priority.

REFERENCES

[1] 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

[2] 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

[3] A. Forward and T. C. Lethbridge, "The relevance of software documentation, tools and technologies: a survey," in *Proceedings of the 2002 ACM symposium on Document engineering*, ser. DocEng '02. New York, NY, USA: ACM, 2002, pp. 26–33. [Online]. Available: http://doi.acm.org/10.1145/585058.585065

[4] T. C. Lethbridge, J. Singer, and A. Forward, "How software engineers use documentation: The state of the practice," *IEEE Softw.*, vol. 20, no. 6, pp. 35–39, Nov. 2003. [Online]. Available: http://dx.doi.org/10.1109/MS.2003.1241364

[5] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 255–265. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337254

[6] I. R. Katz and J. R. Anderson, "Debugging: An analysis of bug-location strategies," *Hum.-Comput. Interact.*, vol. 3, no. 4, pp. 351–399, Dec. 1987. [Online]. Available: http://dx.doi.org/10.1207/s15327051hci0304_2

[7] S. Bugde, N. Nagappan, S. Rajamani, and G. Ramalingam, "Global software servicing: Observational experiences at microsoft," in *Proceedings of the 2008 IEEE International Conference on Global Software Engineering*, ser. ICGSE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 182–191. [Online]. Available: http://dx.doi.org/10.1109/ICGSE.2008.18

[8] G. C. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: Bridging the gap between source and high-level models," *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. 4, pp. 18–28, 1995.

[9] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Component rank: relative significance rank for software component search," in *Proceedings of the 25th International Conference on Software Engineering*, ser. ICSE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 14–24. [Online]. Available: http://dl.acm.org/citation.cfm?id=776816.776819

[10] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A search engine for finding highly relevant applications," in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, vol. 1. IEEE, 2010, pp. 475–484.

[11] A. N. Langville and C. D. Meyer, *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton, NJ, USA: Princeton University Press, 2006.

[12] J. Stylos and B. A. Myers, "Mica: A web-search tool for finding api components and examples," in *Proceedings of the Visual Languages and Human-Centric Computing*, ser. VLHCC '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 195–202. [Online]. Available: http://dx.doi.org/10.1109/VLHCC.2006.32

[13] J. Stylos, B. A. Myers, and Z. Yang, "Jadeite: improving api documentation using usage information," in *CHI '09 Extended Abstracts on Human Factors in Computing Systems*, ser. CHI EA '09. New York, NY, USA: ACM, 2009, pp. 4429–4434. [Online]. Available: http://doi.acm.org/10.1145/1520340.1520678

[14] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *Proceedings of the 27th international conference on Software engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 117–125. [Online]. Available: http://doi.acm.org/10.1145/1062455.1062491

[15] 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

[16] Y. Shmerlin, I. Hadar, D. Kliger, and H. Makabee, "To document or not to document? an exploratory study on developers motivation to document code," in *Advanced Information Systems Engineering Workshops*, ser. Lecture Notes in Business Information Processing, A. Persson and J. Stirna, Eds. Springer International Publishing, 2015, vol. 215, pp. 100–106. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-19243-7_10

[17] E. Tom, A. Aurum, and R. Vidgen, "An exploration of technical debt," *J. Syst. Softw.*, vol. 86, no. 6, pp. 1498–1516, Jun. 2013. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2012.12.052

[18] P. W. McBurney and C. McMillan, ""automatic source code summarization of context for java methods"," in *IEEE Transactions of Software Engineering (to appear)*, 2015.

[19] D. Binkley, "Source code analysis: A road map," in *2007 Future of Software Engineering*, ser. FOSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 104–119. [Online]. Available: http://dx.doi.org/10.1109/FOSE.2007.27

[20] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *Software Engineering, IEEE Transactions on*, vol. 33, no. 1, pp. 2–13, Jan 2007.

[21] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: current results, limitations, new approaches," *Automated Software Engineering*, vol. 17, no. 4, pp. 375–407, 2010. [Online]. Available: http://dx.doi.org/10.1007/s10515-010-0069-5

[22] R. Subramanyam and M. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: implications for software defects," *Software Engineering, IEEE Transactions on*, vol. 29, no. 4, pp. 297–310, April 2003.

[23] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, May 2009. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2009.02.007

[24] 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. [Online]. Available: http://dx.doi.org/10.1002/smr.567

[25] T. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, vol. SE-2, no. 4, pp. 308–320, Dec 1976.

[26] A. De Lucia, M. Di Penta, S. Stefanucci, and G. Venturi, "Early effort estimation of massive maintenance processes," in *Software Maintenance, 2002. Proceedings. International Conference on*, 2002, pp. 234–237.

[27] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 452–461. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134349

[28] N. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *Software Engineering, IEEE Transactions on*, vol. 26, no. 8, pp. 797–814, Aug 2000.

[29] M. Shepperd and D. Ince, "A critique of three metrics," *Journal of Systems and Software*, vol. 26, no. 3, pp. 197 – 210, 1994. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0164121294900116

[30] S. R. Chidamber and C. F. Kemerer, "Towards a metrics suite for object oriented design," *SIGPLAN Not.*, vol. 26, no. 11, pp. 197–211, Nov. 1991. [Online]. Available: http://doi.acm.org/10.1145/118014.117970

[31] B. Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, 1996.

[32] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "Effective identifier names for comprehension and memory," *Innovations in Systems and Software Engineering*, vol. 3, no. 4, pp. 303–318, 2007. [Online]. Available: http://dx.doi.org/10.1007/s11334-007-0031-2

[33] A. A. Takang, P. A. Grubb, and R. D. Macredie, "The Effects of Comments and Identifier Names on Program Comprehensibility: An Experimental Study," *Journal of Programming Languages*, vol. 4, no. 3, pp. 143–167, 1996.

[34] P. McBurney and C. McMillan, "An empirical study of the textual similarity between source code and source code summaries,"

[35] Y. Li, D. McLean, Z. A. Bandar, J. D. O'Shea, and K. Crockett, "Sentence similarity based on semantic nets and corpus statistics," *IEEE Transactions on Knowledge and Data Engineering*, vol. 18, no. 8, pp. 1138–1150, 2006.

[36] G. A. Miller, "Wordnet: A lexical database for english," *Commun. ACM*, vol. 38, no. 11, pp. 39–41, Nov. 1995. [Online]. Available: http://doi.acm.org/10.1145/219717.219748

[37] G. Salton, A. Wong, and C. S. Yang, "A vector space model for automatic indexing," *Commun. ACM*, vol. 18, no. 11, pp. 613–620, Nov. 1975. [Online]. Available: http://doi.acm.org/10.1145/361219.361220

[38] M. T. Hagan, H. B. Demuth, and M. Beale, *Neural Network Design*. Boston, MA, USA: PWS Publishing Co., 1996.

[39] G. R. Finnie, G. E. Wittig, and J.-M. Desharnais, "A comparison of software effort estimation techniques: Using function points with neural networks, case-based reasoning and regression models," *J. Syst. Softw.*, vol. 39, no. 3, pp. 281–289, Dec. 1997. [Online]. Available: http://dx.doi.org/10.1016/S0164-1212(97)00055-1

[40] C. López-Martín and A. Abran, "Neural networks for predicting the duration of new software projects," *J. Syst. Softw.*, vol. 101, no. C, pp. 127–135, Mar. 2015. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2014.12.002

[41] A. Arcuri and G. Fraser, "Parameter tuning or default values? an empirical investigation in search-based software engineering," *Empirical Software Engineering*, vol. 18, no. 3, pp. 594–623, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10664-013-9249-9

[42] F. Wilcoxon, *Individual Comparisons by Ranking Methods*, ser. Bobbs-Merrill Reprint Series in the Social Sciences, S541. Bobbs-Merrill, College Division.

[43] F. Deissenbock and M. Pizka, "Concise and consistent naming [software system identifier naming]," in *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, May 2005, pp. 97–106.

[44] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a name? a study of identifiers," in *In 14th International Conference on Program Comprehension*. IEEE Computer Society, 2006, pp. 3–12.

[45] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Relating identifier naming flaws and code quality: An empirical study," in *Reverse Engineering, 2009. WCRE '09. 16th Working Conference on*, Oct 2009, pp. 31–35.

[46] ——, "Exploring the influence of identifier names on code quality: An empirical study," in *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, March 2010, pp. 156–165.

[47] G. Norman, "Likert scales, levels of measurement and the "laws" of statistics," *Advances in Health Sciences Education*, vol. 15, no. 5, pp. 625–632, 2010. [Online]. Available: http://dx.doi.org/10.1007/s10459-010-9222-y

[48] G. M. Sullivan and A. R. Artino Jr, "Analyzing and interpreting data from likert-type scales," *Journal of graduate medical education*, vol. 5, no. 4, pp. 541–542, 2013.

[49] B. Lantz, "Equidistance of likert-type scales and validation of inferential methods using experiments and simulations," *The Electronic Journal of Business Research Methods*, vol. 11, no. 1, pp. 16–28, 2013.

[50] M. H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA: Elsevier Science Inc., 1977.

[51] N. Dragan, M. L. Collard, and J. I. Maletic, "Reverse engineering method stereotypes," in *2006 22nd IEEE International Conference on Software Maintenance*, Sept 2006, pp. 24–34.

[52] ——, "Automatic identification of class stereotypes," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, Sept 2010, pp. 1–10.

[53] L. Moreno and A. Marcus, "Jstereocode: Automatically identifying method and class stereotypes in java code," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 358–361.

[54] J. Y. Gil and I. Maman, "Micro patterns in java code," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 97–116.

[55] 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

[56] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, vol. 2, May 2010, pp. 223–226.

[57] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *Proceedings of the 2010 17th Working Conference on Reverse Engineering*, ser. WCRE '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 35–44. [Online]. Available: http://dx.doi.org/10.1109/WCRE.2010.13

[58] T. K. Landauer, P. W. Foltz, and D. Laham, "An introduction to latent semantic analysis," *Discourse processes*, vol. 25, no. 2-3, pp. 259–284, 1998.

[59] 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.

[60] B. Eddy, J. Robinson, N. Kraft, and J. Carver, "Evaluating source code summarization techniques: Replication and expansion," in *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, May 2013, pp. 13–22.

[61] W. Li and A. McCallum, "Pachinko allocation: Dag-structured mixture models of topic correlations," in *Proceedings of the 23rd International Conference on Machine Learning*, ser. ICML '06. New York, NY, USA: ACM, 2006, pp. 577–584. [Online]. Available: http://doi.acm.org/10.1145/1143844.1143917

[62] 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

[63] L. Moreno, J. Aponte, S. Giriprasad, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *Proceedings of the 21st International Conference on Program Comprehension*, ser. ICPC '13, 2013.

[64] 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.

[65] 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

[66] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll, "Predicting source code changes by mining change history," *Software Engineering, IEEE Transactions on*, vol. 30, no. 9, pp. 574–586, Sept 2004.

[67] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *Software Engineering, IEEE Transactions on*, vol. 31, no. 6, pp. 429–445, June 2005.

[68] Q. Soetens, J. Perez, and S. Demeyer, "An initial investigation into change-based reconstruction of floss-refactorings," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, Sept 2013, pp. 384–387.

[69] J. Ratzinger, T. Sigmund, P. Vorburger, and H. Gall, "Mining software evolution to predict refactoring," in *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, Sept 2007, pp. 354–363.

[70] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 190–. [Online]. Available: http://dl.acm.org/citation.cfm?id=850947.853338

[71] T. Gîrba, S. Ducasse, A. Kuhn, R. Marinescu, and R. Daniel, "Using concept analysis to detect co-change patterns," in *Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting*, ser. IWPSE '07. New York, NY, USA: ACM, 2007, pp. 83–89. [Online]. Available: http://doi.acm.org/10.1145/1294948.1294970

[72] A. E. Hassan and R. C. Holt, "Predicting change propagation in software systems," in *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ser. ICSM '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 284–293. [Online]. Available: http://dl.acm.org/citation.cfm?id=1018431.1021436

[73] D. Beyer and A. Noack, "Clustering software artifacts based on frequent common changes," in *Proceedings of the 13th International Workshop on Program Comprehension*, ser. IWPC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 259–268. [Online]. Available: http://dx.doi.org/10.1109/WPC.2005.12

[74] A. Ouni, M. Kessentini, S. Bechikh, and H. Sahraoui, "Prioritizing code-smells correction tasks using chemical reaction optimization," *Software Quality Control*, vol. 23, no. 2, pp. 323–361, Jun. 2015. [Online]. Available: http://dx.doi.org/10.1007/s11219-014-9233-7

[75] T. Reps, T. Ball, M. Das, and J. Larus, *Software Engineering — ESEC/FSE'97: 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering Zurich, Switzerland, September 22–25, 1997 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, ch. The use of program profiling for software maintenance with applications to the year 2000 problem, pp. 432–449. [Online]. Available: http://dx.doi.org/10.1007/3-540-63531-9_29

[76] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani, "Holmes: Effective statistical debugging via efficient path profiling," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 34–44. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070506

[77] R. R. Lutz, "Software engineering for safety: A roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 213–226. [Online]. Available: http://doi.acm.org/10.1145/336512.336556

[78] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," in *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, ser. SIGPLAN '82. New York, NY, USA: ACM, 1982, pp. 120–126. [Online]. Available: http://doi.acm.org/10.1145/800230.806987