Detecting Important Terms in Source Code for Program Comprehension

Paige Rodeghero
Clemson University
prodegh@clemson.edu

Collin McMillan
University of Notre Dame
cmc@nd.edu

Abstract

Software Engineering research has become extremely dependent on terms (words in textual data) extracted from source code. Different techniques have been proposed to extract the most "important" terms from code. These terms are typically used as input to research prototypes: the quality of the output of these prototypes will depend on the quality of the term extraction technique. At present no consensus exists about which technique predicts the best terms for code comprehension. We perform a literature review, and propose a unified prediction model based on a Naive Bayes algorithm. We evaluate our model in a field study with professional programmers, as well as a standard 10-fold synthetic study. We found our model predicts the top quartile of the most-important terms with approximately 50% precision and recall, outperforming other popular techniques. We found the predictions from our model to help programmers to the same degree as the gold set.

1. Introduction

"Terms" in source code are one of the most predominant forms of data used in software engineering research. Terms are alphanumeric strings in source code such as variable names, function names, and words from comments. Terms are not necessarily natural language words – they often include context specific abbreviations such as `blkwrt` for block write, or obscure vernacular such as `atof`. Nevertheless, it is a key assumption across many areas of software engineering that source code terms contain valuable information. Work in traceability link recovery [1], information retrieval [2], feature location [3], summarization [4], bug localization [5], quality analysis [6], clone detection [7], and even security [8], all rely on this assumption. The assumption persists even though terms are selected at the discretion of programmers, and even though the terms themselves do not affect the software's behavior [9].

The reason that the assumption persists is that programmers tend to encode high-level information in terms, that is difficult to understand from the structure of the code without the terms. For example, "airline seat reservation" is easy for human programmers to understand, but it would be quite difficult to comprehend solely by tracing data flow or badly-named function calls. This is a situation known as the "concept assignment problem", and has been studied for over two decades [10]. The idea is firmly entrenched that terms are important for both human comprehension and for software engineering research.

But some terms are more important than others. Both practitioners and researchers must prioritize some terms over others due to time constraints and information overload [11]. Terms in function signatures, for example, are widely believed to be more important, *in general*, than terms in other areas of code [12]. At the same time, popular measures such as term frequency-inverse document frequency (tf/idf) are based on the idea that terms that occur rarely and in a small number of areas are more important than terms that occur often in many areas – the term "access control" is likely more valuable than "print" [13]. Other work has suggested that important terms are ones that occur frequently in other software of the same domain [14], or are from APIs [15] (see Section 3.1 for a detailed definition of term importance).

What is missing from software engineering research is a unified model for identifying the importance of terms in source code. A variety of different models have been proposed which represent different theories about what programmers need. These models each have strengths and weaknesses, and some have been shown to be quite accurate. However, these models are often in conflict, or are accurate in some scenarios but not others.

In this paper, we propose a unified model based on a literature survey to identify 20 prominent models for the importance of terms. We implement a source code metric for each model, and use those metrics as attribute inputs for a machine learning algorithm. We use the algorithm to train our model on a dataset that closely approximates importance: terms ranked by tracking the

URI: https://hdl.handle.net/10125/60186
ISBN: 978-0-9981331-2-6

eye movements of programmers as viewed longer and more often than other terms. Our objective in using this dataset is to train the model based on the programmers' actual reading behavior. In this study, evaluation is based primarily on comprehension-specific tasks, but the hope overall is that the model can be extended in a context-insensitive manner.

We evaluated and calibrated our unified model in two ways. First, we performed a standard 10-fold study using the training/testing datasets. Second, we evaluated the output of our model for six different software programs in a cross-validation user study, in which human experts rated the output of our predictive model and actual data. In the 10-fold study, our approach predicted the top 25% of important terms with a true positive rate (e.g. recall) of 51.1%, a false positive rate of 24.5%, and a precision of 46.5%. In the cross-validation user study, our predictive model had equivalent performance for actual comprehension tasks, as using the gold set data.

2. The Problem

The problem we target in this paper the lack of consensus among software engineering researchers about what terms in source code tend to be the most valuable for programming tasks. A multitude of software engineering techniques extract terms from source code, for a variety of purposes, including feature location, bug localization, etc. While these techniques are rich and diverse in purpose, they are related by a common need: to extract the terms from source code that represent the high-level concepts that programmers have about the code. Programmers use some terms for comprehension more than others, and tools could benefit by using the same terms. Several approaches to extracting terms have been proposed. Without evidence to support researchers in deciding how to extract these terms, the researchers must guess which approach is likely to extract the best terms.

A solution to this problem would have significant impact on software engineering research, given the quantity of work that extracts terms from source code. Since these terms tend to be the input to the research tools, the quality of the output of the tools is likely to be increased with improved term extraction from code. It is possible that some tools may need different input than others (e.g., context-specific knowledge), though even these tools are likely to benefit from a term extraction technique that more-closely matches the techniques that programmers use.

3. Background

In this section, we define "importance" of terms, in the context of the related software engineering literature.

3.1. Important Terms

To say that a term is "important" here means that that term is beneficial to programmers during program comprehension. In theory, all terms are valuable for comprehension to some degree. But in practice, some terms are much more valuable than others – to the degree that a programmer may only need to read a single term in a function to gain an understanding of the purpose of that function. A huge body of literature has been devoted to studying how this is possible. As Storey [16] points out, literary opinion has organized into a few competing theories, with different features of software highlighted as having different effects on comprehension. However, one consensus is that, over time, programmers have moved from a preference for "systemic" comprehension [17, 18] to "opportunistic" comprehension [19, 20]. In short, it means that programmers now "try to avoid program comprehension" by reading only the minimum number of terms necessary [21], whereas thirty years ago, programmers tended to read a smaller number of artifacts in greater detail [18]. While it is tempting to ascribe this trend to personal factors such as declining work ethic, it is more likely a consequence of dramatically increased software size [22], availability of software examples [15], and constant training from work force churn [23]. The reality is that terms are now more critical than ever for rapid comprehension, and programmers need to find the most important terms as quickly as possible.

3.2. Modeling Importance

Software engineering researchers have attempted to create models of the importance of terms, independent of a context. Through measurements such as tf/idf and term location, researchers have attempted to predict which terms in software are most important for any context. In the example of tf/idf, terms are considered more important for a section of code, if those terms occur more often there than in other sections [24]. In Section 4, we describe 20 metrics that have been proposed to model importance.

For validation of any model of importance, it is crucial to measure the actual importance of terms to programmers during programming tasks. There are two key strategies to measure importance. First, survey procedures may be used to ask programmers to report the terms that they subjectively feel are valuable for comprehension. Second, it is possible to record observable behaviors that are indicative of importance, such as number of times a programmer looks at a term. In this paper, we use both survey procedures and observable behaviors for validation. For our purposes, both are context-specific to program comprehension.

The observable behaviors we use for validation are the eye movements, also known as fixations and gaze times, that programmers make while comprehending code, as detailed in the next section.

3.3. Eye-tracking Data

Eye-tracking technology is starting to be more widely used in software engineering studies. Early on, Crosby *et al.* performed a study that showed reading source code is performed differently than reading prose [25]. Both Crosby *et al.* and Bednarik *et al.* also found in several studies that programmers tend to move around the document, focusing on various, specific sections of the code such as comments and outputs, rather than reading the document straight through to the end [26, 25]. In addition to naturally focusing on specific areas, some studies have shown that fixation on certain areas of code have an effect on bug detection and requirements tracing [27, 28]. These studies show that eye-tracking can be useful for guiding improvements to software engineering and program comprehension methodologies and tools.

For this study, we used Rodeghero *et al.*'s previous eye-tracking data as a basis for determining what gaze times should be considered important. In the previous study, researchers examined the source code terms and areas of code programmers focused on when reading a method for the purpose of comprehension [12]. They recruited 10 professional programmers, with approximately 13 years of experience each on average, to quickly read unfamiliar Java methods and write a short summary about each one. As the programmers scanned the source code, we had an eye-tracking system keeping record of their eye movements. The gaze times were calculated for each separate fixation, then also aggregated as a total gaze time for each term. The study determined there were both areas of code and certain terms in each method that programmers tended to focus on. Given a gaze time associated with a term, they were able to determine importance.

4. Our Approach

In this section, we describe our approach.

4.1. Key Idea

First, we parse the source code of a given project to give us all possible terms. Second, we extract all the metrics associated to each term. Third, we use a combination of the metrics to create test data that can be used for classification. Fourth, we use a classifier to put each term into metric-based categories. Fifth, using those categories, we calculate a predicted gaze time for each term. Using previous work, we can make use these gaze times for importance comparison. Finally, with that gaze time available, we then determine which terms are important on a per method basis.

4.2. Parsing the Source Code

For parsing, we do not assume that the project source files are laid out in any specific directory structure. We also do not assume that each Java source file only contains one class or only belongs to one package. Therefore, first, we parse every file in a Java project individually. We keep a log of which project, class, file, and method in which each term occurs. As each file is parsed, we capture certain properties of each class, method, line, and term contained within the file, to facilitate the metric extraction. These properties include the various amounts, sizes, positions, and names associated with each piece of the Java source code file.

4.3. Extracting the Metrics

Next, we extract the metrics from the parsed source code files and stored properties (see Table 1). Each metric has its own calculation. The most simple calculations are length, line size, relative position, and line location, as those have a direct link to the properties of each file that we captured during parsing. Compound names, abbreviated names, capitalization, assignment, variable, input parameter, and return value are all relatively simple, as well, because they only require a small additional parse of the line and/or term involved. To calculate POS, LSA, and Complexity, we used third-party libraries NLTK, GenSim, and Lizard, respectively. Signature, invocation, conditional, and nesting require additional parsing of small snippets of code within each method, so these take additional time. The metrics that took the most time and effort to collect were tf/idf and scope, which require a full examination of the entire source code project to gather all needed information for calculation. As each metric is collected, it is attached to every term, method, and class.

4.4. From Metrics to Importance

Then, we take these various metrics and combine them with the terms into a test data file that can be used for training and testing. Once the test file is constructed, we use a Naive Bayes classifier to categorize each term based on its metric values.

Using the categories now given to each term, we calculate an appropriate gaze time determined by Rodeghero *et al.*'s study's collection of professional programmer gaze times and selected important word lists [12]. This calculation was tuned by extensive calibration to make sure the addition of each metric value added the right amount of gaze time to each term.

Finally, we use a gaze time threshold, determined in Rodeghero *et al.*'s work [48], as our lower limit of importance. If a term's predicted gaze time is above this limit, it is simply given an extra label of "Important". With this label, a program or tool used to generate comments or summaries would know that this

Metric Name	Type	Definition
Assignment*	s	whether the term is being assigned a new value within this method [29]
Cyclomatic Complexity	s	cyclomatic complexity of a method [30, 31, 29]
Conditional*	s	is in a conditional statement inside the method [12]
Input Parameter*	s	includes all the input terms to the method [32]
Invocation*	s	is in an invocation or function call inside the method [12]
Line location*	s	the location in which the term appears in the method by line number [29]
LSA	s	meaning associated with the term within the context of current method [33, 34]
Multiple nesting	s	how many loops the term is located within [29]
Relative location	s	the character number the word appears in relation to the method's beginning [29]
Return Value	s	includes all the output terms from the method [32]
Scope of variable	s	the scope in which the term was created [29]
Signature	s	term is located inside the signature of the method [12]
Tf/idf	s	occurrence of a particular keyword in an individual document [35, 36, 37, 12]
Variable*	s	whether the term is a variable, as opposed to a constant, method, etc. [29]
Abbreviated name*	t	whether a term contains abbreviations or is abbreviated [6, 38]
Capitalization	t	if the keyword contains capitalization [6, 29]
Compound name*	t	how many words a term contains within itself [6, 38, 39, 40, 41]
Length	t	the total number of characters in a term [6, 29, 42]
Line Size	t	length of line, by characters, where the keyword resides in the method [43, 29]
Part of Speech*	t	the keyword's part of speech in English [6, 44, 45, 46, 47]

Table 1. Our 20 "Importance" Metrics. Asterisks indicate metrics that remained after we determined the best configuration of metrics (see Section 6.1). In the column "Type," 's' stands for 'structural' and 't' for 'textual'.

term needs to be included to maximize a programmer's understanding of the method. Additionally, we separate terms into quartiles, of roughly even size, in order to answer our research questions.

5. Synthetic Study Design

This section describes the synthetic study we perform of our approach. The study is "synthetic" because it evaluates how well our approach predicts the importance of terms as determined by eye tracking data.

5.1. Research Questions

The objective of this study is to determine how to configure our approach to have the highest possible prediction quality, using a verifiable and reproducible procedure. By "configure our approach", we mean which metrics to include as input – some have higher predictive power than others. Since our approach is based on a machine learning algorithm, the Research Questions (RQ) we ask use quartiles for even categories, as well as correspond to a standard 10-fold cross-validation procedure for these algorithms:

RQ_1 Which **configuration** of metrics has the highest quality of predictions, in terms of quartiles of important terms?

RQ_2 Which **class** of metrics has the highest quality of predictions, in terms of quartiles of important terms?

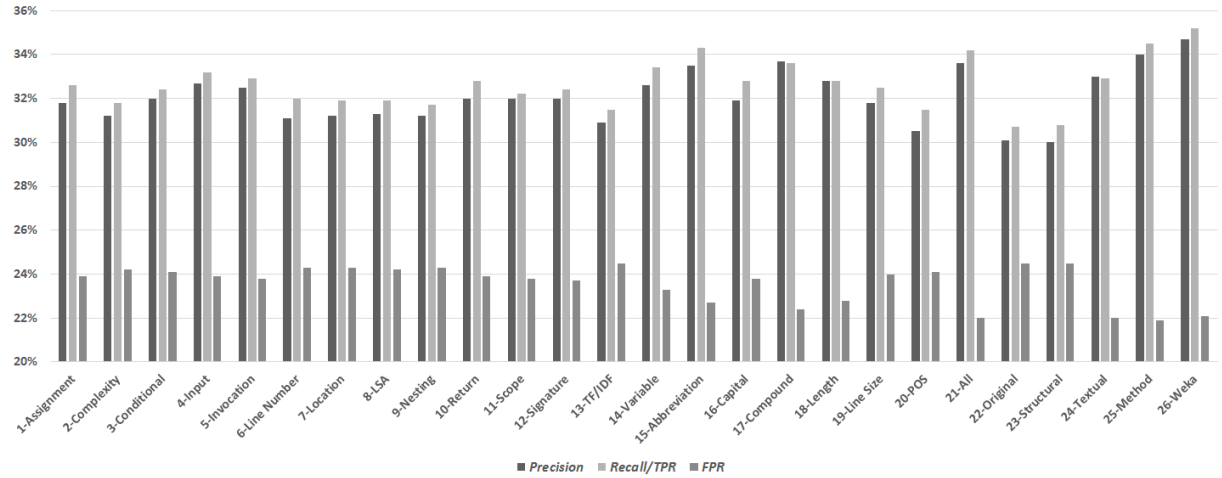
The rationale behind RQ_1 is that several configurations of our approach are possible, in terms of metrics to use as input. Ideally, a minimum set of metrics would be used to achieve the highest possible prediction quality. We choose to evaluate against quartiles of important terms because a quartile approach guarantees balanced data, which will avoid bias from adding cost-sensitive modifications.

The purpose of RQ_2 is to determine whether the textual or structural metrics reach the quality of the best configuration. A simplified approach could be created based on only textual or structural metrics, since the effort could be focused on extracting only one type of data. In addition, since software engineering research often contrasts textual and structural data [49], it may benefit the research community to know how these types of data affect the importance of terms.

5.2. Methodology

This subsection describes the methodology we followed to answer the research questions above. We followed the approach outlined in Section 4 to calculate the 20 metrics (see Table 1) from the parsed Java source code. We then created 26 configurations of the metrics. Each configuration is a different grouping of the metrics. While there are over a million possible configurations, we chose these 26 based on specific questions, such as "Would it still be useful to make predictions without entire projects?" or "Do textual or structural properties provide better results?". These 26 configurations break down as follows: 1-20) one for each metric by itself,

Figure 1. The overall accuracy measures for each configuration.



21) one for all metrics, 22) one for metrics related to our previous eye-tracking study (i.e. Signature, Conditional, and Invocation), 23) one for structural metrics, 24) one for textual metrics, 25) one for metrics that only require the method itself and not the whole project (e.g. not LSA), and 26) a configuration automatically produced by a feature selection algorithm [1] (the metrics marked by an asterisk in Table 1). Next, for each of these configurations, we trained a machine learning algorithm to create a model for predicting importance, as described in Section 4.4. The entire process for all six projects took less than four hours. With each of these prediction models, we were then able to answer our research questions. To answer RQ_1, for every metric group, we predicted the quartiles and compared the predictions to the actual quartile categorization. We performed a similar comparison to answer RQ_2.

5.3. Accuracy Measures

We use three main accuracy measures for determining the effectiveness of our metric configurations. Precision is the proportion of the predicted positive cases that were correctly placed into the appropriate quartile, as calculated using the following equation:

$$\frac{Correctly\ Predicted\ Positive}{All\ Predicted\ Positive}$$

Recall/True Positive Rate (TPR) is the proportion of positive cases that were correctly identified, as calculated using the following equation:

$$\frac{Correctly\ Predicted\ Positive}{All\ Actual\ Positive}$$

False Positive Rate (FPR) is the proportion of negative cases that were incorrectly classified as positive, as calculated using the following equation:

$$\frac{Incorrectly\ Predicted\ Positive}{All\ Actual\ Negative}$$

The main accuracy measure we focus on is recall, but all three measures are shown for each metric configuration in Figure 1.

5.4. Threats to Validity

The first main threat to this study is the threats carried over from the eye-tracking study, including participant bias, the gaze data collection, the line limitations, and the restriction of using Java methods only. Also, since our data is originates from the same system that this study is being tested against, there is the possibility of some over-fitting during the machine learning process. The second main threat to validity in this study is the metric calculations. Since the calculations for each metric were specifically written for our parsed data, there may be a better way to represent some metrics. We hope this threat is minimized from the use of several different literature bases from which these metrics were derived.

6. Synthetic Study Results

In this section, we present our results to each research question. We also present the accuracy measures for all metric configurations in Figure 1 to complement these results.

6.1. RQ_1: Best Metric Configuration

From our analysis, we found configuration 26 (the automatic feature selection, see Section 5.2) to produce the most accurate predictions. This group of metrics led to an overall recall of 35.2%. It also has a recall of 51.1% for the top quartile and a recall of 30.5% for the bottom quartile. The metrics in this configuration are marked by an asterisk in Table 1. The next most accurate metric configuration for overall quartile prediction is the method only configuration, with an overall recall of 34.5%. The worst metric configuration for overall

[1] http://weka.sourceforge.net/doc.dev/weka/classifiers/meta/AttributeSelectedClassifier.html

prediction is the configuration of only our eye-tracking paper metrics, with an overall recall of 30.7%.

6.2. RQ_2: Best Metric Class

From our analysis, we found Textual metrics to produce more accurate predictions than Structural metrics. Textual has an overall recall of 32.9%. It also has a recall of 48.3% for the top quartile and a recall of 37.9% for the bottom quartile. However, Structural has an overall recall of 30.8%, a top recall of 49.6%, and a bottom recall of 25.4%.

6.3. Summary of Results

From our synthetic design results, we derive two key interpretations. First, we note that the highest performance occurred when using a subset of the 20 metrics we studied. Using this subset, we were able to reproduce the gold set to approximately 50% top precision and recall, meaning that we found about half of the top quartile of terms. In a ranked list of the terms in a typical method with 16 terms, at worst we would predict two correct answers in positions 3 and 4 in the list. A second interpretation is that textual attributes had higher predictive power than structural attributes. Note, even the metric tf/idf, which is popular in software engineering work, achieved 37.9% top precision and 39.8% recall of the top quartile, considerably lower than the best-performing configuration.

7. Empirical Study Design

This section describes the design of our empirical study. The purpose of this study is to evaluate the quality of the predictions from our approach in terms of subjective human expert opinion.

7.1. Research Questions

The objective of this study is to determine the predictive quality of our approach in terms of unbiased, expert human opinion. In other words, we ask *do our predictions actually represent terms that are important to programmers?* In addition, we aim to compare the terms that are predicted to be important, to the terms that are known to be important from the gold set of eye gazes. We pose the follow specific research questions:

RQ_3 What is the difficulty of comprehension tasks when given the **top** quartile of **predicted** terms, versus the **top** quartile of **actual** terms?

RQ_4 What is the difficulty of comprehension tasks when given the **top** quartile of **predicted** terms, versus the **bottom** quartile of **predicted** terms?

RQ_5 What is the difficulty of comprehension tasks when given the **top** quartile of **actual** terms, versus the **bottom** quartile of **actual** terms?

In the context of these questions, the "predicted" terms are the terms that our approach predicts to be in e.g. the top quartile. The "actual" terms are the terms from the gold set in the synthetic study – they are the terms that are known to have the highest gaze time from a previous experiment [12]. This gold set is only meant to represent a strong basis for our comparison and does not represent the best set of important terms overall.

The rationale behind RQ_3 is to determine how well the predicted terms approximate the actual terms for usefulness during program comprehension tasks. We know from the synthetic study that the top quartile of predicted terms are not quite the same as the top quartile of actual terms. RQ_3 explores whether the inaccuracy of the predictions significantly degrades the usefulness of the predictions for comprehension tasks.

RQ_4 compares the top quartile of terms predicted to be important, to the bottom quartile of terms predicted to be important. If our approach predicts terms that are useful for comprehension, programmers should have less difficulty comprehending when given the top quartile of terms, than the bottom quartile. On the other hand, if no difference in difficulty exists between top and bottom quartiles, then it is evidence that the predictions do not assist programmers during comprehension.

Likewise, RQ_5 seeks to measure the difference between top and bottom quartiles of the actual terms. While we may expect a difference in difficulty based on findings from related work [12], it is useful to know the degree of this difference, for comparison to the predictions. If the degree of difference is not significant between top and bottom actual terms, to top and bottom predicted terms, then it is evidence that our approach reasonably approximates the actual terms.

7.2. Methodology

This subsection describes the methodology we followed to answer the research questions above. We recruited professional programmers to do program comprehension tasks. In each "comprehension task", a programmer read the source code of one Java method and wrote an English description of the functionality of that code. Having the programmers attempt to comprehend completely unfamiliar source code is a known evaluation method called the "program model" [16]. A programmer could spend as much time on each task as desired, for 90 minutes. We chose this format primarily in order to maintain consistency with the data collected in earlier experiments [12].

However, a key difference from earlier experiments is that we show the Java method to the programmer twice. First, we show the method with only one quartile of the terms visible. We obfuscated the other three quartiles using a standard term-replacement technique (replace the terms with non-meaningful strings such as `xxxx`) [50]. For example, a Java method with 20 terms

would have about five terms visible, and about 15 terms obfuscated. The visible terms correspond to e.g. the top quartile of predicted terms.

We asked the programmers to write an English summary of the method after reading the code the first time (with terms obfuscated). We also asked the programmers to rate their perceived difficulty of the task on a Likert scale from -2 to 2 where -2 corresponded to "Very Difficult", -1 to "Somewhat Difficult", 0 to "Neither Difficult nor Easy", 1 to "Somewhat Easy", and 2 to "Very Easy".

Then, we showed the programmers the same method again, except with all terms visible. We then gave the programmer the option to modify his or her English summary if he or she felt that the new information benefits comprehension. The programmer also had the option to modify his or her perceived difficulty.

We use both the perceived difficulty and the English descriptions to answer the research questions. The perceived difficulty ratings are quantitative measurements of subjective human expert opinion. We compute the difference in difficulty from the first reading of the Java methods (with only one quartile shown) to the second reading (with all terms shown). For example, if a programmer reads a method and marks a difficulty of -1, and then reads the method the second time and remarks the difficulty as 2, then the difference in difficulty is 3.

During the study, we gave the programmers the comprehension tasks in a random order: the programmers saw random Java methods with either 1) the top quartile of predicted terms, 2) top quartile of actual terms, 3) bottom quartile of predicted terms, or 4) bottom quartile of actual terms. Note that, although randomly chosen, we did attempt to provide the same number of dataset/quartile pairs to each participant for statistical purposes. We then aggregate the differences in difficulty for each of these four groups, and contrast them. For example, to answer RQ_1 we compare the difference in difficulty for the top quartile of predicted terms to the difference in difficulty for the top quartile of actual terms. Finally, we use a statistical hypothesis test to determine if the difference of the means in these groups is statistically significant. Note that we used a random order to minimize bias and to ensure that the programmers did not know if they were reading a top or bottom quartile. We also did not tell the participants that there was the possibility of receiving the different quartiles on different methods.

7.3. Participants

The participants of this study were all professional programmers. There were 11 participants in total. Four of the programmers came from the University of Notre Dame, while the other participants came from various industry companies, including IBM, Alden Systems, and Uber Technologies. The participants' years of professional programming experience range from 1 to 16, with an average of 6.85.

7.4. Subject Applications

The subject applications in this study are the six open source Java projects mentioned in Section 3.3. These projects were chosen because of their constant improvement and use, as well as their wide variety of methods. In total, 50 methods were chosen from amongst these six projects to be displayed to the participants for purposes of summarization. Each of these methods had around 22 LOC, ensuring that each method would take roughly the same amount of time to summarize and would match the limitations of the original eye-tracking study. All of these methods also have no accompanying comments.

7.5. Statistical Tests

We compared differences in difficulty using the Wilcoxon signed-rank test [51]. This test is non parametric and paired, and does not assume a normal distribution. It is suitable for our study because we compare patterns paired for each method and because our data may not be normally distributed.

7.6. Threats to Validity

The main threats to validity in this study are 1) the participants, and 2) the subject applications. Since the ratings of difficulty are subjective opinions of human programmers, we cannot guarantee that a different set of programmers would not produce a different outcome in the study. Various human factors such as fatigue, bias, experience level, and temperament affect both the subjective ratings and the English descriptions. These ratings and descriptions are likely to change considerably across programmers. We attempted to mitigate this by recruiting professionals from a wide range of backgrounds, including different industrial affiliations. We showed the programmers the methods in a random order, to prevent a bias introduced from the programmers learning from one or another Java method.

A second threat to validity is our choice of subject applications. We attempted to mitigate this threat by using a wide range of Java projects, and by selecting Java methods at random from these projects.

8. Empirical Study Results

In this section, we present our results to each research question, along with examples of the answers.

8.1. RQ_3: Top Predicted vs. Top Actual

We did not find a statistically significant difference between the difficulty of the program comprehension

Table 2. Statistical summary of the results for RQ$_3$, RQ$_4$, and RQ$_5$. Wilcoxon test values are U, U_{expt}, and U_{vari}. Decision criteria is p. A "Sample" is one programmer for one method.

RQ	H	Dataset	Quartile	Samples	$\tilde{x}$	Variance	U	U_{expt}	U_{vari}	p
RQ_3	H_1	Predicted	Top	43	1.093	0.991	6	7.5	11.25	0.0766
		Actual	Top	43	1.116	0.819				
RQ_4	H_2	Predicted	Top	43	1.093	0.991	0	162.5	1187.5	<1e-4
		Predicted	Bottom	43	1.767	1.849				
RQ_5	H_3	Actual	Top	43	1.116	0.819	0	150	1080	<1e-4
		Actual	Bottom	43	1.791	2.122				

tasks with the predicted top quartile of terms versus the actual top quartile of terms displayed. We posed H_1:

H_n The difficulty of comprehension tasks when given the **top** quartile of **predicted** terms versus when given the **top** quartile of **actual** terms is not statistically different.

Using the Wilcoxon signed-rank test, we were unable to reject the null hypothesis (see Table 2). These results indicate that programmers have no more difficulty summarizing source code with the top predicted terms than with the top actual terms.

An illustrative example of a summary written for each of these datasets from the same method:

Top Predicted: "increases the play count"

Top Actual: "updates the index of a certain value within an array"

Both datasets appear to have provided similar information to the programmer. These summaries reveal enough to understand that a value is updated within the method.

8.2. RQ_4: Top Predicted vs. Bottom Predicted

We found statistically significant evidence that programmers had less difficulty with the predicted top quartile of terms displayed than with the predicted bottom quartile of terms displayed. We posed H_2:

H_n The difficulty of comprehension tasks when given the **top** quartile of **predicted** terms versus when given the **bottom** quartile of **predicted** terms is not statistically different.

Using the Wilcoxon signed-rank test, we rejected the null hypothesis (see Table 2). These results indicate that programmers have less difficulty summarizing source code with the top predicted terms than with the bottom predicted terms.

An illustrative example of a summary written for each of these datasets from the same method:

Top Predicted: "This function sets a keyword handler in a thread safe fashion"

Bottom Predicted: "Sets handler for something"

It appears that the top dataset produced more information to the programmers. The top summary knows that the handler is for a keyword, while the bottom does not.

8.3. RQ_5: Top Actual vs. Bottom Actual

We found statistically significant evidence that programmers had less difficulty with the actual top quartile of terms displayed than with the actual bottom quartile of terms displayed. We then posed H_3:

H_n The difference in difficulty of comprehension tasks when given the **top** quartile of **actual** terms versus when given the **bottom** quartile of **actual** terms is not statistically different.

Using the Wilcoxon signed-rank test, we rejected the null hypothesis (see Table 2). These results indicate that programmers have less difficulty summarizing source code with the top actual terms than with the bottom actual terms.

An illustrative example of a summary written for each of these datasets from the same method:

Top Actual: "This sets newline characters and x to escaped versions. So newlines become just a string; etc."

Bottom Actual: "Don't know what it's for"

This demonstrates how the top dataset helped the programmer understand much more information with the top quartile than the bottom. The top summary has a mostly complete grasp on the method behavior, while the bottom cannot tell anything about it.

8.4. Summary of Results

Our key finding is that there was no statistically-significant difference between the difficulty of the program comprehension task when using the top predicted terms, versus the top actual terms. In other words, the predicted terms helped the programmers to the same degree as the actual terms.

This is a meaningful finding because it indicates that our prediction model's performance is equivalent to a gold set for program comprehension tasks. In addition, in RQ_4 and RQ_5, we found evidence that some terms really do help programmers more than other terms: the top predicted and actual terms led to less difficulty than the bottom quartile predicted and actual terms.

This finding may seem in contradiction to the findings in the synthetic study. In the synthetic study, we found that approximately 50% of the actual top terms occurred in the predicted top terms (see

Section 6.1). One possible explanation is that these terms still contained enough information to convey the meaning of the source code to the programmers. The lowest quartiles did not contain this information, leading to greater difficulty of the comprehension task.

9. Conclusion

This paper contributes to Software Engineering research in two ways. First, we presented a unified prediction model for identifying the important terms in source code. We trained a machine learning algorithm using a dataset of actual programmer eye gaze times on terms. Longer gaze times are strongly correlated with importance (see Section 3.3), suggesting that our predictions will detect gaze times. Furthermore, a second contribution is that we evaluated our technique with professional programmers in the field, from a variety of industries. Our predicted important terms did as well as the eye-tracking gold set of important terms, when used for program comprehension tasks.

We recommend our unified model over tf/idf and other importance metrics that are currently in use in SE research. While a specific problem may benefit from using one metric or another, it is our belief that this unified model will lead to superior performance for tools in a general case. However, we acknowledge that future work is necessary to test our model as a replacement for alternatives in the context of different SE problems apart from program comprehension.

References

[1] A. Marcus, J. Maletic, *et al.*, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Software Engineering, 2003. Proceedings. 25th International Conference on*, pp. 125–135, IEEE, 2003.

[2] A. D. Lucia, M. D. 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*, pp. 193–202, June 2012.

[3] 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.

[4] 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*, ICPC 2014, (New York, NY, USA), pp. 279–290, ACM, 2014.

[5] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: A comparative study of generic and composite text models," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, (New York, NY, USA), pp. 43–52, 2011.

[6] 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*, pp. 31–35.

[7] 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, pp. 470–495, May 2009.

[8] G. Díaz and J. R. Bermejo, "Static analysis of source code security," *Inf. Softw. Technol.*, pp. 1462–1476, 2013.

[9] D. Binkley, M. Davis, D. Lawrie, J. I. Maletic, C. Morrell, and B. Sharif, "The impact of identifier style on effort and comprehension," *Empirical Software Engineering*, vol. 18, no. 2, pp. 219–276, 2013.

[10] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster, "Program understanding and the concept assignment problem," *Communications of the ACM*, vol. 37, no. 5, pp. 72–82, 1994.

[11] J. Starke, C. Luce, and J. Sillito, "Searching and skimming: An exploratory study," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pp. 157–166, IEEE, 2009.

[12] 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*, pp. 390–401, ACM, 2014.

[13] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *Reverse Engineering, 2010 17th Working Conference on*, pp. 35–44.

[14] S. Haiduc and A. Marcus, "On the use of domain terms in source code," in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pp. 113–122, IEEE, 2008.

[15] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie, "Exemplar: A source code search engine for finding highly relevant applications," *Software Engineering, IEEE Transactions on*, pp. 1069–1087, 2012.

[16] M.-A. Storey, "Theories, tools and research methods in program comprehension: past, present and future," *Software Quality Journal*, vol. 14, no. 3, pp. 187–208, 2006.

[17] A. J. Ko and B. A. Myers, "A framework and methodology for studying the causes of software errors in programming systems," *Journal of Visual Languages and Computing*, vol. 16, no. 12, pp. 41 – 84, 2005.

[18] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway, "Mental models and software maintenance," *Journal of Systems and Software*, vol. 7, no. 4, pp. 341 – 355, 1987.

[19] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer, "Example-centric programming: integrating web search into the development environment," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, (New York, NY, USA), pp. 513–522, ACM, 2010.

[20] G. Kotonya, S. Lock, and J. Mariani, "Opportunistic reuse: Lessons from scrapheap software development," in *11th International Symposium on Component-Based Software Engineering*, (Berlin, Heidelberg), pp. 302–309, Springer-Verlag, 2008.

[21] 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*, ICSE 2012, (Piscataway, NJ, USA), pp. 255–265, IEEE Press, 2012.

[22] R. Schmidt, K. Lyytinen, and P. C. Mark Keil, "Identifying software project risks: An international delphi study," *Journal of management information systems*, vol. 17, no. 4, pp. 5–36, 2001.

[23] 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*, ICGSE '08, (Washington, DC, USA), pp. 182–191, IEEE Computer Society, 2008.

[24] 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*, (Washington, DC, USA), pp. 35–44.

[25] M. E. Crosby and J. Stelovsky, "How do we read algorithms? a case study," *Computer*, vol. 23, pp. 24–35, Jan.

[26] R. Bednarik and M. Tukiainen, "Temporal eye-tracking data: evolution of debugging strategies with multiple representations," in *Proceedings of the 2008 symposium on Eye tracking research & applications*, ETRA '08, (New York, NY, USA), pp. 99–102, 2008.

[27] N. Ali, Z. Sharafl, Y. Gueheneuc, and G. Antoniol, "An empirical study on requirements traceability using eye-tracking," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pp. 191–200, Sept 2012.

[28] B. Sharif, M. Falcone, and J. I. Maletic, "An eye-tracking study on the role of scan time in finding source code defects," in *Proceedings of the Symposium on Eye Tracking Research and Applications*, ETRA '12, (New York, NY, USA), pp. 381–384, ACM, 2012.

[29] P. Oman and J. Hagemeister, "Metrics for assessing a software system's maintainability," in *Software Maintenance, 1992. Proceerdings., Conference on*, pp. 337–344, Nov 1992.

[30] G. Gill and C. Kemerer, "Cyclomatic complexity density and software maintenance productivity," *Software Engineering, IEEE Transactions on*, vol. 17, pp. 1284–1288, Dec 1991.

[31] T. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, no. 4, pp. 308–320, 1976.

[32] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi, "An empirical investigation into a large-scale java open source code repository," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, (New York, NY, USA), pp. 11:1–11:10, ACM, 2010.

[33] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman, "Indexing by latent semantic analysis," *JAsIs*, vol. 41, no. 6, pp. 391–407, 1990.

[34] S. T. Dumais, "Latent semantic analysis," *Annual review of information science and technology*, pp. 188–230, 2004.

[35] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Inf. Process. Manage.*, vol. 24, pp. 513–523, Aug. 1988.

[36] J. Ramos, "Using tf-idf to determine word relevance in document queries," in *Proceedings of the first instructional conference on machine learning*, 2003.

[37] M. J. Pazzani and D. Billsus, "Content-based recommendation systems," in *The adaptive web*, pp. 325–341, Springer, 2007.

[38] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker, "Mining source code to automatically split identifiers for software analysis," in *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, pp. 71–80, IEEE, 2009.

[39] D. Binkley, M. Davis, D. Lawrie, and C. Morrell, "To camelcase or under_score," in *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, pp. 158–167, IEEE, 2009.

[40] J. Chen, C.-H. Yeh, and R. Chau, "Identifying multi-word terms by text-segments," in *Web-Age Information Management Workshops, 2006. WAIM'06. Seventh International Conference on*.

[41] I. Fahmi, "C-value method for multi-word term extraction," in *seminar in Statistics and Methodology*, 2005.

[42] K. Kawamoto and O. Mizuno, "Predicting fault-prone modules using the length of identifiers," in *Empirical Software Engineering in Practice (IWESEP), 2012 Fourth International Workshop on*, pp. 30–34, 2012.

[43] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Software Maintenance, 1999. Proceedings. IEEE International Conference on*, pp. 109–118.

[44] E. Hill, L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of nl-queries for software maintenance and reuse," in *Proceedings of the 31st International Conference on Software Engineering*, pp. 232–242, IEEE Computer Society, 2009.

[45] N. Madani, L. Guerrouj, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "Recognizing words from source code identifiers using speech recognition techniques," in *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pp. 68–77, IEEE, 2010.

[46] S. L. Abebe, S. Haiduc, P. Tonella, and A. Marcus, "Lexicon bad smells in software," in *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pp. 95–99, IEEE, 2009.

[47] J.-R. Falleri, M. Huchard, M. Lafourcade, C. Nebut, V. Prince, and M. Dao, "Automatic extraction of a wordnet-like identifier network from software," in *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pp. 4–13, IEEE, 2010.

[48] P. Rodeghero, C. Liu, P. McBurney, and C. McMillan, "An eye-tracking study of java programmers and application to source code summarization," *Software Engineering, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2015.

[49] 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.

[50] M. Grechanik, C. McMillan, T. Dasgupta, D. Poshyvanyk, and M. Gethers, "Redacting sensitive information in software artifacts," in *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, (New York, NY, USA), pp. 314–325, ACM, 2014.

[51] D. A. Wolfe and M. Hollander, "Nonparametric statistical methods," *Nonparametric methods*, 1973.