2016 IEEE/ACM 38th IEEE International Conference on Software Engineering Companion

Discovering Important Source Code Terms

Paige Rodeghero
Advisor: Collin McMillan
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN, USA
prodeghe@nd.edu

ABSTRACT

Terms in source code have become extremely important in Software Engineering research. These "important" terms are typically used as input to research tools. Therefore, the quality of the output of these tools will depend on the quality of the term extraction technique. Currently, there is no definitive best technique for predicting the importance of terms during program comprehension. In my work, I perform a literature review of several techniques. I then propose a unified importance prediction model based on a machine learning algorithm. I evaluate my model in a field study involving professional programmers, as well as a standard 10-fold synthetic study. I found that my model predicts the top quartile of most-important source code terms with approximately 50% precision and recall, outperforming tf/idf and other popular techniques. Furthermore, I found that, during actual program comprehension tasks, the predictions from my model help programmers equivalent to a real set of most-important terms.

1. PROBLEM AND MOTIVATION

"Terms" are alphanumeric strings found in source code such a variable name, a function name, or words found in comments. Software engineering research relies heavily on the idea that terms in source code are useful as inputs to various research tools. The main problem is that researchers are unsure which terms are more valuable as inputs because there is a lack of evidence to support deciding which terms to choose. Without this evidence to guide them, researchers may unknowingly extract terms that have little to no value for their current task. If a solution is found to this problem, the beneficial impact on software engineering research could be significant. As the quality of the output of tools that require specific terms as input increases, the quality of results of several research projects are likely to increase, as well. [1]

[1]These studies are part of a paper that will soon be under review at TSE.

ICSE '16 Companion, May 14 - 22, 2016, Austin, TX,USA

ACM ISBN 978-1-4503-4205-6/16/05.
DOI: http://dx.doi.org/10.1145/2889160.2891037

2. BACKGROUND AND RELATED WORK

2.1 Important Terms

To say that a term is "important" means that that term is more beneficial to programmers during program comprehension than other terms. A substantial amount of research exists on studying how and why programmers have tendencies towards certain terms [4, 5, 7, 9, 10, 11, 12, 13, 14, 17, 18, 20]. The main consensus seen within this literature is that with the ever increasing size and scope of software engineering projects, professional programmers rely more heavily on the ability to skim source code for needed information than they have in the past. This means that programmers no longer have the luxury of reading all source code terms to glean the meaning of a method or class, but must now be able to quickly find the most important terms for their current comprehension needs.

2.2 Modeling Importance

Software engineering researchers have previously attempted to create useful models of term importance, such as tf/idf [8, 15]. I have performed a literature review to create a list of 20 individually widely used metrics to combine and use in my model. These metrics have been broken down into two main categories: structural and textual. Structural metrics represent the placement and use of a term with respect to the method, class, and package. These metrics help determine a term's inherent importance to the overall project. The structural metrics are assignment, complexity, conditional, input, invocation, line, LSA, nesting, location, output, scope, signature, tf/idf, and variable type. Textual metrics represent the spelling and design of a term. These metrics help determine the importance given to the term by the programmer who wrote that section of source code. The textual metrics are abbreviation, capitalization, compound, length, line size, and part of speech. [2]

2.3 Eye-tracking Data

Eye-tracking technology is starting to be more widely used in software engineering studies [1, 2, 3, 6, 19]. These studies, among others, show that eye-tracking can be useful for helping improve software engineering research. In this study, I used my previous eye-tracking data as a basis for determining what gaze times should be considered important. In the previous study, I examined the source code terms and ar-

[2]This material is based upon work supported by the National Science Foundation Career Award under Grant No. CCF-1452959.

eas of code programmers focused on when reading a method for the purpose of comprehension [16]. I recruited 10 professional programmers to scan unfamiliar source code and write a summary about it. The source code was randomly gathered from 6 common open source projects : Jajuk, JEdit, JHotDraw, JTopas, NanoXML, and Siena. As the programmers scanned the source code, I had an eye-tracking system keeping record of their eye movements, including both position and timing. The study determined that there were both areas of code and certain terms in each method that programmers tended to focus on.

3. APPROACH AND UNIQUENESS

The uniqueness behind my approach is discovering importance through a combination of metric calculations for every term in a source code project. First, I parse the source code for all possible terms. I log 15 characteristics of each term as I go, such as class name, method name, and line number. Second, I extract all the metrics associated to each term. Some metrics only require the simple information gathered during parsing, while others require an examination of the entire project that contains the term. Third, I create test data that can be used for classification using a combination of the various metrics. Fourth, I use a Naive Bayes classifier to place each term into specific categories that fit its metrics. Fifth, I calculate a predicted gaze time for each term using those specific categories. This calculation requires some tuning to ensure that the correct metrics are used. Finally, I then determine which terms are important on a per method basis. Importance is achieved by a term if its predicted gaze time is above a specified "importance threshold". This list of important terms may help improve a future auto-generated comment for that method.

4. RESULTS

My evaluation was two-fold: I used a synthetic study to calibrate my approach for a field study.

4.1 Synthetic Study

I first conducted a synthetic study using a verifiable and reproducible procedure aimed at configuring my set of input metrics to have optimal prediction quality. For this study, I created several variations of metrics combinations, including one automatically produced by a feature selection algorithm [3]. Then, I trained a machine learning algorithm to create an importance model using each combination configuration. I used a quartile categorization for every metric group to produce my predictions.

From my synthetic study results, I derive two key interpretations. First, I note that the best performance occurred when using a subset of the 20 metrics. Using this subset, I was able to reproduce the real data to approximately 50% top quartile precision and recall. A second interpretation is that textual metrics had better predictive power than structural metrics. Note that even the metric tf/idf, which is extremely popular in software engineering research, achieved only 37.9% top quartile precision and 39.8% top quartile recall.

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

4.2 Field Study

Second, I conducted a field study to determine to determine the power of my predictions when actual humans are involved. I also wanted to compare my list of predicted important terms with the list of important terms based on the real eye-tracking data. I recruited 11 professional programmers to participate in my study. Four of the programmers came from the Center for Research Computing at Notre Dame, while the other seven participants came from various industry companies, including IBM, Alden Systems, and Uber Technologies. The participants were each required to perform comprehension tasks. In each comprehension task, a programmer read the source code of a randomly chosen, quartile-obfuscated Java method that only showed either the predicted or eye-tracking important terms only. Note that the programmer did not know why certain terms were chosen to be hidden. The programmer then wrote an English description of the functionality of that code and gave a rating of the comprehension difficulty. This process was then repeated, but with no obfuscation of the source code. A programmer could spend as much time on each task as he or she desired, for a total duration of 90 minutes.

My main 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. This means that my predicted terms made comprehension just as easy for the professional programmers as the set of important terms taken from eye-tracking data of other professional programmers. This also means that my prediction model performs just as well as gathering actual data. In addition, I 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.

5. CONTRIBUTIONS

My work contributes to Software Engineering research in two ways. First, I introduced a unified prediction model for discovering the important source code terms. I trained a machine learning algorithm using actual programmer eye gaze data, which strongly correlates to term importance. Furthermore, a second contribution is that I evaluated my model with a variety of professional programmers in the field. My predictions and the original eye-tracking data were shown to both be equally useful for program comprehension tasks. I recommend replacing tf/idf and other importance metrics that are currently in use in Software Engineering research with my unified model. I believe that my unified model will produce higher quality output for tools in the general case, although some specific cases may benefit from a more specific metric. I acknowledge that more work is required to ensure that my model is a beneficial a replacement all Software Engineering problems. However, I note that my unified model includes many of these current metrics as input before deciding on how to make predictions.

6. REPRODUCIBILITY

All my input data, raw and processed results, and processing scripts are available via my online appendix [4]. The field study data collection program is also available open source for use by other researchers.

[4]http://www3.nd.edu/~prodeghe/projects/importance/

7. REFERENCES

[1] 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*, pages 191–200, Sept 2012.

[2] R. Bednarik and M. Tukiainen. An eye-tracking methodology for characterizing program comprehension processes. In *Proceedings of the 2006 symposium on Eye tracking research & applications*, ETRA '06, pages 125–132, New York, NY, USA, 2006.

[3] 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, pages 99–102, New York, NY, USA, 2008.

[4] 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, pages 513–522, New York, NY, USA, 2010. ACM.

[5] 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, pages 182–191, Washington, DC, USA, 2008. IEEE Computer Society.

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

[7] J. W. Davison, D. M. Mancl, and W. F. Opdyke. Understanding and addressing the essential costs of evolving systems. *Bell Labs Technical Journal*, pages 44–54, 2000.

[8] 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*, pages 35–44, Washington, DC, USA.

[9] 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*, 16(1–2):41 – 84, 2005.

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

[11] A. Lakhotia. Understanding someone else's code: analysis of experiences. *J. Syst. Softw.*, pages 269–275, 1993.

[12] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 492–501, New York, NY, USA, 2006. ACM.

[13] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. *Journal of Systems and Software*, 7(4):341 – 355, 1987.

[14] 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*, pages 1069–1087, 2012.

[15] M. J. Pazzani and D. Billsus. Content-based recommendation systems. In *The adaptive web*, pages 325–341. Springer, 2007.

[16] 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*, pages 390–401. ACM, 2014.

[17] 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, pages 255–265, Piscataway, NJ, USA, 2012. IEEE Press.

[18] R. Schmidt, K. Lyytinen, and P. C. Mark Keil. Identifying software project risks: An international delphi study. *Journal of management information systems*, 17(4):5–36, 2001.

[19] 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, pages 381–384, New York, NY, USA, 2012. ACM.

[20] M.-A. Storey. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal*, 14(3):187–208, 2006.