Code Fragment Summarization

Annie T. T. Ying and Martin P. Robillard
School of Computer Science
McGill University
{annie.ying,martin}@cs.mcgill.ca

ABSTRACT

Current research in software engineering has mostly focused on the retrieval accuracy aspect but little on the presentation aspect of code examples, e.g., how code examples are presented in a result page. We investigate the feasibility of *summarizing* code examples for better presenting a code example. Our algorithm based on machine learning could approximate summaries in an oracle manually generated by humans with a precision of 0.71. This result is promising as summaries with this level of precision achieved the same level of agreement as human annotators with each other.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Experimentation

Keywords

Machine Learning, summarization, source code analysis

1. INTRODUCTION

The Web is an important resource for a programmer: as much as 20% of a programmer's time could be spent on the Web [2]. When a programmer searches for information related to an Application Programming Interfaces (API), of the various types of documentation the programmer finds on the Web, code examples are one of the most effective [14], important [16], and frequently sought-after [17]. Because searching for code examples is often an indispensable part of programming, researchers have built search engines and recommendation systems for retrieving relevant code examples [21, 4, 1]. Most of these systems focus on improving the accuracy of code example retrieval.

However, even when a relevant example is returned by a general or code-specific search engine, the presentation of

Apache Portable Runtime: File I/O Handling Functions
apr.apache.org/docs/apr/1.2/group__apr__file__io.html
26 Nov 2007 – apr_status_t · **apr_file_read** (apr_file_t *thefile, void *buf, apr_size_t *nbytes). apr_status_t · apr_file_write (apr_file_t *thefile, const void *buf, ...

Figure 1: A result for "apr_file_read" from Google

code examples can hinder the use of the example. Even a state-of-the-art search engine for finding code examples does not provide adequate cues for a programmer to effectively evaluate whether the link is worth-while to follow. Often, the textual snippet accompanying a returned link contains no code, only a textual summary of the Web page being linked. When a summary does contain code, the summary is extracted as if the code were text (Figure 1). These limitations often foil a programmer's attempt to evaluate whether a search hit is worth pursuing, making it necessary for a programmer to open and scan many of the result pages [20]. Yet, in the context of general search engines, textual snippets form a significant part (40% of the time on a search result page [7]) of evaluating whether a particular returned link is worth navigating to.

We propose to summarize code examples, more formally, *code fragments* which we define as partial programs that serve the purpose of demonstrating the usage of an API. A *code fragment summary* is a shorter code fragment than the original one, where any line in the summary is more informative (in the context of a specific query) than any other line not in the summary.

In this paper, we present a feasibility study on one way of generating code fragment summaries: a supervised machine learning approach that classifies whether a line in a code fragment should be in a summary. As an initial investigation, we exploited two types of features: *syntactic* features of the source code, and whether a line is related to the given *query*. For training and evaluating our classifier, we collected a corpus of code fragments and constructed a corresponding summary oracle using four human annotators. The corpus consists of 70 code fragments directly extracted from code examples illustrating the answers to the Eclipse official FAQ.[1] We defined the query, which is the focus of the summary, as the FAQ question. The summary oracle consists of summaries manually created by annotators through selecting lines deemed important for a summary, totaling to 3560 judgments. Figure 2 illustrates one code fragment from our corpus, for the FAQ "How do I distinguish between internal and external JARs on the build path?" The summary we generated is marked as bold.

[1] http://wiki.eclipse.org/index.php/Eclipse_FAQs, accessed on May 29, 2013

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ESEC/FSE'13, August 18–26, 2013, Saint Petersburg, Russia
Copyright 2013 ACM 978-1-4503-2237-9/13/08...$15.00
http://dx.doi.org/10.1145/2491411.2494587

```
1: IClasspathEntry entry = ...
2: IPath path = entry.getPath();
3: IWorkspace workspace=ResourcesPlugin.getWorkspace();
4: IResource jarFile=workspace.getRoot().findMember(path);
5: if (jarFile != null) {
6:   return jarFile.getLocation();
7: } else {
8:   // must be an external JAR (or invalid classpath entry)
9: }
```

Figure 2: A code fragment summary (in bold)

We found promising evidence in the feasibility of generating code fragment summaries using a machine learning approach with light-weight features, with a precision of 0.71 when we allowed summaries to be of the same length as the oracle. This level of precision were as similar to human-generated summaries as summaries generated by different humans to each other. Our syntactic and query-related features are fast to generate (0.09s per code fragment), making it possible to deploy in a real search engine and other software engineering settings. This simple algorithm will serve as a baseline, as we develop more sophisticated code fragment summarization approaches.

2. RELATED WORK

The closest work to ours is a system by Kim et al [11] that augments API documentation with code example summaries. The component of their system responsible for summarization resembles our tool, also using syntactic and query features, though without the use of machine learning. Our summarizer differs in the purpose: their approach is meant for summarizing multiple results from a code search engine, such as Koders, whereas our algorithm works on summarizing any single code fragments.

Mica [20] and Assieme [9] are examples of the line of work aiming at augmenting search engines. Mica aims at helping a programmer evaluate if a search hit from a general search engine contains relevant API elements, and for Assieme, relevant code examples. Mica works by augmenting the search result page with API methods, classes and field names contained in Web pages linked by the search hits. Assieme extracts the code examples from the returned Web pages as the result and displays various statistics and links to the API elements. Assieme attempts to better adapt a general search engine to code search by expanding a programmer's query on a general search engine with programming language keywords. Our system differs in that instead of simply displaying all the API elements (in Mica's case) or the code fragments (in Assieme's case) contained in a Web page linked by the search hits, we *summarize* code fragments contained in the Web page to improve on the often inadequate textual snippets provided by general search engines.

Researchers have proposed code example search engines (e.g., Strathcona [10], SNIFF [5], Sourcerer [1], and a recent system by Buse and Weimer [4]) and code completion tools (e.g., the Intelligent Code Completion system [3]). Most of these systems attempt to synthesize code examples or find the relevant APIs from code repositories that match a programmer's query, whether the query is explicitly formed or implicitly inferred from the context. However, these systems do not help a programmer searching on the Web for code examples and wishing to use the context accompanying the desired code fragment in a Web page.

Several efforts have investigated reduced representations of software artifacts, including summarizing bug reports [15, 12] and producing a succinct set of textual keywords [8] or textual summary [18, 19] given the source code of a method.

3. CODE FRAGMENT SUMMARIZER

Our classifier uses two types of features: whether a line contains certain syntactic constructs and whether a line is related to a query. We set aside 17 code fragments and summaries for the the development of features (development set) and the rest, 53, for evaluation.

3.1 Syntactic Features

We observed from the summaries in the development set that when a line contains a certain type of syntactic constructs, the line is more likely to be in a summary. For example, a line containing an anonymous class declaration and instantiation tended to be in a summary in the oracle, whereas a line containing an *if* conditional tended *not* to be in a summary.

In total, after experimenting with the development set, we employed 49 syntactic features. These features include whether a line in a code fragment contains a certain part of a type signature (e.g., *typeIsPublic*), contains an anonymous declaration and instantiation, contains an exception handling or conditional keyword, is in a block, contains a variable or field declaration, contains a part of a method signature, contains a method invocation, contains a method exist statement, contains a comment, contains an assignment, contains a call declared in the code, and contains a call in the Java SDK.

We extracted these syntactic features of a code fragment from its abstract syntax tree (AST). All syntactic features are discrete variables, with a binary value depending on whether a line contains a feature.

3.2 Query-Related Features

We also observed that annotators are more likely to include in the summary the lines containing the terms from the query, which is the question in the FAQ. Analogous to the syntactic features, query related features are discrete variables, with a binary value depending on whether a line contains a feature. We constructed three features that indicate whether an identifier in a code fragment contains a query term or not. We constructed two additional features that looked beyond just individual lines: *mostTerms* is true when a line contains the most number of matching terms among all the lines in the same code fragment, and *mostDiverseTerms* is analogous except it indicates the most number of *distinct* terms. When computing the features, we split the identifiers according to the common camel case identifier naming convention.

3.3 Classifier, Training and Evaluation Data

We experimented with two separate classifiers using Naive Bayes (NB) and Support Vector Machine (SVM).

Regarding training data, the summaries from the four annotators had a Cohen Kappa [6] agreement of 0.487. A value between 0.4-0.6 is considered a moderate agreement when the task is well-defined [6]. Since the same code fragment line could have contradictory markings by the four annotators. Such noise could confuse machine learning models, more so for SVM than Naive Bayes. Other previous efforts in

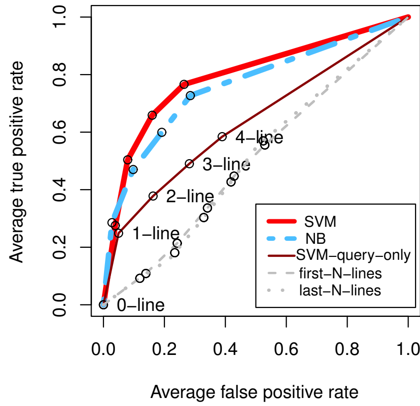

Figure 3: ROC curves

summarization have addressed this problem by training and evaluating a model using a **gold standard summary** [15]. A gold standard summary for a given code fragment consists of lines marked as in-summary by at least two annotators, and lines marked by one annotator or none are not considered to be in a gold standard summary. On average, the size of a summary in the gold standard was 33.5% (sd=14.5%) of the original code fragment.

4. FEASIBILITY STUDY

Our main research question is whether it is feasible to generate code fragments summaries using only syntactic and query-related features. On average, our summarizer could generate features for a code fragment in 0.09 seconds per code fragment. If we can generate reasonable code fragment summaries using these two simple types of features, it would provide us with a promising feasibility argument and baseline for developing more sophisticated techniques.

4.1 Effectiveness Metrics

We used *leave-one-out cross validation*, where "one" (also known as a fold) refers to one code fragment.[2] For each fold, we defined the correctness of the generated summary as the corresponding gold standard summary.

To evaluate how well a generated summary resembles the gold standard summary in the oracle, we compared the two using **R-precision**, an evaluation metric from the field of information retrieval [13]. R-precision is similar to precision-at-k, the precision for summaries of length k. Precision-at-k determines out of the top k lines predicted by our classifier ($predicted_k$), how many are correct ($|oracle \cap predicted_k|$). R-precision differs from precision-at-k by allowing summaries of variable lengths. The R-precision of a code fragment evaluates the top R lines returned by our classifier ($predicted_R$), where R is the length of the summary oracle. More formally, R-precision is given by $\frac{|oracle \cap predicted_R|}{|predicted_R|}$.

4.2 Effectiveness Results

As a preliminary evaluation, we assessed the performance of our summarizer by comparing whether the classifier was better than three baselines: the *first-N-lines* classifier which constructs a summary of length N by picking the first N lines of a code fragment, the *last-N-lines* classifier which picks the last N lines, and the *SVM-query-only* classifier that only uses the two query related features described in Section 3.2. We conducted the performance comparison through a receiver operator characteristic (ROC) curve. An ROC curve depicts the trade-off between the true positive rate and false positive rate as we varied N from one line to four lines. This curve enables us to understand explicitly the performance trade-off among different summary lengths. The coordinate of a point on the curve is given by the average true positive rate (average of each of the true positive rate per code fragment) and average false positive rate (average of each of the false positive rate per code fragment). The *true positive rate* of a code fragment c is given by:

$$\frac{|\{\text{lines in gold st. summary of } c\} \cap \{\text{lines in gen. summary of } c\}|}{|\{\text{lines in gold st. summary of } c\}|}$$

The *false positive rate* of a code fragment c is given by:

$$\frac{|\{\text{lines in } c \textit{ not} \text{ in gold st. summary}\} \cap \{\text{lines in gen. sum. of } c\}|}{|\{\text{lines in } c \textit{ not} \text{ in gold st. summary}\}|}$$

Averaging the rates per code fragment (rather than for all lines) aligns better with the actual task of providing a summary for a code fragment (rather than being just an exercise of predicting summary-membership of lines). The closer the ROC curve is to the upper left corner (with fewer false positives and more true positives), the better the classifier. The area under the curve sums up this intuition: the better the classifier, the closer its area under a ROC curve is to 1.

Figure 3 shows five ROC curves: two versions of our classifier *(SVM* and *NB*, the two thicker lines) and three baselines *(SVM-query-only*, *first-N-lines*, and *last-N-lines*, the three thinner lines). Our two classifiers have area under the curve of 0.806 for *SVM* and 0.772 for *NB*. Both clearly lie above the *first-N-lines* baseline (the area under the curve is 0.493) and *last-N-lines* baseline (the area under the curve is 0.503). In addition, the two classifiers using both syntactic and query features out-perform the baseline using only query-related features, *SVM-query-only*, whose area under the curve is 0.629.

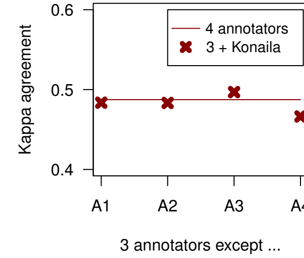

Figure 4: Agreement result

4.3 Generated against Annotators' Summaries

Figure 4 illustrates the Kappa agreement of the four annotators (kappa=0.487) and how the agreement changed when we left out the summaries provided by each one of the annotators and replaced the summaries of the left-out annotator with summaries generated by the classifier. In one case (A3), swapping in the generated summaries even *improved* the agreement, whereas in two cases the agreement decreased slightly and in the third case decreased more. The average of the four kappa statistics of the three-annotators-plus-our-classifier settings was 0.484, almost the same as the agreement of the four annotators.

[2]To maximize the training data but still evaluate on the unseen evaluation data, the cross validation consisted of 53 folds, each fold yielded a prediction for one summary trained on 69 (i.e., 17+53-1) gold standard summaries.

One could argue that the average R-precision was 0.705 and the area under the ROC curve were far from 1, the perfect score. However, given the agreement among the human annotators on the summaries was considered moderately low (kappa=0.487), there is a limit to the performance one could expect from a machine learning approach that relied on this same set of annotations. These initial results show promising feasibility in using light-weight features to generate code fragment summaries.

5. CONCLUSION AND FUTURE DIRECTIONS

We introduced a new idea of generating code fragment summaries as a way to provide succinct cues for Web pages containing code fragments. These summaries have a wide range of applications in summarizing any artifacts containing code fragments. The approach of using light-weight syntactic and query features achieved a precision of 0.705 when we allowed summaries to be of the same length as the oracle. With this level of precision, our summaries achieved the same level of agreement as human annotators with each other. The features are fast to generate, 0.09 seconds per code fragment, making it possible to deploy in a real search engine and other software engineering settings.

There are promising future directions to take our current summarizer to. Observing the summaries in the Eclipse FAQ oracle revealed the promise in additional code-level analysis, to take into account the relation between lines when generating summaries. For example, simple intra-method data-flow related features—such as whether a return type of a method call in a code fragment is later used–can be indicative of whether the line is important for a summary. Analyses on code fragments can pose technical challenges related to the fact that code fragments are not complete programs.

In the evaluation of our summaries, defining the correctness of a summary as the summary lines more agreed upon by the annotators assumed that a code fragment has a single universal summary suitable for everyone. The moderately low agreement among the annotators motivated us to investigate a different assumption on the correctness of a summary: there does not exist one correct summary; rather, each annotator's version of the summary is correct. With this assumption, an optimal summarizer would be one that *personalizes* summaries for each individual. Personalized summaries is a promising direction for future research.

6. ACKNOWLEDGMENTS

Thanks to the four annotators, J. Pineau, K. Moffatt, P. Duboue, P. Rigby, Y. Chhetri, F. Ferreira, G. Petrosyan & C. Treude. This work is supported by NSERC & McGill.

7. REFERENCES

[1] S. Bajracharya, J. Ossher, and C. Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proc. of FSE*, pages 157–166, 2010.

[2] J. Brandt, P. Guo, J. Lewenstein, M. Dontcheva, and S. Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proc. of CHI*, pages 1589–1598, 2009.

[3] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proc. of ESEC/FSE*, pages 213–222, 2009.

[4] R. Buse and W. Weimer. Synthesizing API usage examples. In *Proc. of ICSE*, pages 782–792, 2012.

[5] S. Chatterjee, S. Juvekar, and K. Sen. SNIFF: A search engine for Java using free-form queries. In *Proc. of FASE*, pages 385–400, 2009.

[6] J. Cohen et al. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[7] E. Cutrell and Z. Guan. What are you looking for?: an eye-tracking study of information usage in web search. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 407–416, 2007.

[8] S. Haiduc, J. Aponte, and A. Marcus. Supporting program comprehension with source code summarization. In *Proc. of ICSE-v2*, pages 223–226, 2010.

[9] R. Hoffmann, J. Fogarty, and D. Weld. Assieme: finding and leveraging implicit references in a web search interface for programmers. In *Proc. of UIST*, pages 13–22, 2007.

[10] R. Holmes and G. Murphy. Using structural context to recommend source code examples. In *Proc. of ICSE*, pages 117–125, 2005.

[11] J. Kim, S. Lee, S.-W. Hwang, and S. Kim. Enriching documents with examples: A corpus mining approach. *Transactions on Information Systems*, 31, 2013.

[12] S. Mani, R. Catherine, V. Sinha, and A. Dubey. AUSUM: approach for unsupervised bug report summarization. In *Proc. of FSE*, pages 1–11, 2012.

[13] C. Manning, P. Raghavan, and H. Schutze. *Introduction to information retrieval.* Cambridge University Press, 2008.

[14] S. McLellan, A. Roesler, J. Tempest, and C. Spinuzzi. Building more usable APIs. *IEEE Software*, 15(3):78–86, 1998.

[15] S. Rastkar, G. Murphy, and G. Murray. Summarizing software artifacts: a case study of bug reports. In *Proc. of ICSE*, pages 505–514, 2010.

[16] M. Robillard. What makes APIs hard to learn? Answers from developers. *IEEE Software*, 26(6):27–34, 2009.

[17] S. Sim, R. Gallardo-Valencia, K. Philip, M. Umarji, M. Agarwala, C. Lopes, and S. Ratanotayanon. Software reuse through methodical component reuse and amethodical snippet remixing. In *Proc. of CSCW*, pages 1361–1370, 2012.

[18] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for Java methods. In *Proc. of ASE*, pages 43–52, 2010.

[19] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *Proc. of ICSE*, 2011.

[20] J. Stylos and B. Myers. Mica: A web-search tool for finding API components and examples. In *Proc. of VL/HCC*, pages 195–202, 2006.

[21] T. Xie and J. Pei. MAPO: mining API usages from open source repositories. In *Proc. of MSR*, pages 54–57, 2006.