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 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.

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 whether a search hit from a general search engine contains the relevant API elements, and for Assieme, relevant code examples. Mica works by augmenting the search result page with API methods, classes and field names that are 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 many code example search engines (e.g., Strachona [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 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 a 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 mostDistinctTerms 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,
better than three baselines: the classifier which first-N-lines of our summarizer by comparing whether the classifier was As a preliminary evaluation, we assessed the performance Effectiveness results on 69 (i.e., 17 + 53 − 1) gold standard summaries. unseenevaluationdata, thecrossvalidationconsistedof53 known as a fold) refers to one code fragment. We used , where “one” (also leave-one-out cross validation Effectiveness Metrics provide us with a promising feasibility argument and base-

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.

Effectiveness Metrics
We used leave-one-out cross validation, where “one” (also known as a fold) refers to one code fragment. 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 (predictedk), how many are correct (oracle ∩ predictedk]). 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), where R is the length of the summary oracle. More formally, R-precision is given by 

\[
R \text{-precision} = \frac{|\text{oracle} \cap \text{predicted}|}{|\text{predicted}|}.
\]

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

2To 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


