Survey of Methods to Generate Natural Language from Source Code

Graham Neubig
Graduate School of Information Science, Nara Institute of Science and Technology

Abstract

This paper is an informal survey of methods to generate natural language from source code.

1 Survey Papers

Nazara et al. (2015) gives a nice survey on code summarization.

2 Generation Methods

2.1 Manual Rules/Templates

Perhaps the most common way of creating comment generation systems is through the creation of human rules.

The Software Word Usage Model (SWUM) is one of the first models of this type, and can be used for converting Java method calls into natural language statements (Hill et al., 2009). Sridhara et al. (2010) use the SWUM to create a rule-based model that converts Java to natural-language descriptions.

Rule-based approaches have been expanded to cover segments over multiple lines (Sridhara et al., 2011a), full methods (Abid et al., 2015), or full classes (Moreno et al., 2013). There have also been used to cover special types of code such as test cases (Zhang et al., 2011; Kamimura and Murphy, 2013), code changes (Buse and Weimer, 2010; Cortés-Coy et al., 2014), or exceptions (Buse and Weimer, 2008).

These methods use a variety of information other than the code itself, including context through the codebase (McBurney and McMillan, 2014), execution paths (Buse and Weimer, 2008; Zhang et al., 2011), or stereotypical method types (Moreno et al., 2013; Abid et al., 2015).

2.2 Keyword Lists

Haiduc et al. (2010a) convert all comments and identifiers in the source code, and rank them by relevance, converting it into a keyword list. This approach has been examined in detail in following studies (Haiduc et al., 2010b; De Lucia et al., 2012; McBurney et al., 2014).

It is also possible to automatically guess keywords based on feature vectors calculated from the corresponding code (Wang et al., 2015).

2.3 Comment Retrieval

Wong et al. (2013) propose a method to retrieve comments that mines code/text pairs from Stack Overflow, and then tries to match the code in question with an example in stack overflow.

Allamanis et al. (2015b) create a probabilistic model over code, but use it in the opposite direction to also retrieve full natural language snippets.

2.4 Word-by-Word Predictive Models

Movshovitz-Attias and Cohen (2013) predict the next word of comments using n-gram or topic models.

Allamanis et al. (2015a) use a similar method based on bi-linear language models for suggesting variable names, and find it works better than n-grams. They also suggest a method of breaking down variable names that makes it possible to suggest neologisms.

2.5 Statistical Machine Translation

Oda et al. (2015) use statistical machine translation to learn a system that converts from syntax trees of the source code to natural language descriptions.

3 Content Selection Methods

In addition to methods to generate text itself, it is important to choose which text is salient when creating a code summary. There are several methods to do so:

Keyword Scoring: Haiduc et al. (2010a) examine a number of methods for selecting which keywords to use in a summary, including the lead method, TF-IDF, or LSI. There has also been use of topic models, or hierarchical topic models in a similar way (McBurney et al., 2014).

Classes Affected: Cortés-Coy et al. (2014) measure an impact value based on the number of classes affected by the code at question.

Invocation Frequency: Kamimura and Murphy (2013) simply choose the least frequent invocations up to a certain length threshold.

Node Centrality: Similarly to invocation frequency, Rastkar et al. (2011) build an ontology from the code base, and choose the nodes that are most central to the ontology graph.
Eye Tracking: Rodeghero et al. (2014) perform an eye-tracking study, and find that users focus on method signatures more than other parts of the method, and use this method to improve selection heuristics.

4 Targeted Software Units

It is possible to generate natural language descriptions of code on many levels.

Variables: It is possible to generate comments (Sridhara et al., 2011b) and descriptive variable names (Allamanis et al., 2015a) based on the content and context of the method/variable.

Lines of Code: It is also possible to step through and describe code line by line (Oda et al., 2015).

Multi-line Blocks: Descriptions can be generated for multi-line blocks (Sridhara et al., 2011a) or snippets on QA sites (Wong et al., 2013; Allamanis et al., 2015b).

Methods/Functions: Perhaps the most common variety is description of methods or functions (Abid et al., 2015). These include specialized methods such as test cases (Kamimura and Murphy, 2013).

Classes: We can also go beyond methods and create summaries of whole classes (Moreno et al., 2013).

Multi-class Units: It is also possible to create natural language descriptions of changes crossing multiple classes (Cortés-Coy et al., 2014), or “cross-cutting code concerns” (Rastkar et al., 2011).

5 Training Data Creation

The majority of methods proposed so far are heuristic methods that don’t require explicit training data creation. Sometimes these methods are tested on manually created test data. For methods that do require training data, there are a number of ways to create it.

Manual Creation: It is possible to create data for training manually, such as 18,000 annotated lines of Python code (Oda et al., 2015).

Communications: There is also some work on mining data from communications. The first work focused on developer mailing lists (Panichella et al., 2012), and it is now common to mine Stack Overflow (Wong et al., 2013; Allamanis et al., 2015b).

Comments: Another source that has been mined is comments corresponding to specific methods (Movshovitz-Attias and Cohen, 2013) or blocks of code (Wang et al., 2015; Wong et al., 2015).

6 Evaluation

6.1 Intrinsic

6.1.1 Manual Evaluation

The simplest way to judge appropriateness of comments is through manual evaluation. There are several measures:

Accuracy/Adequacy: Whether the generated comments are correct, and appropriately reflect the content of the corresponding code (Sridhara et al., 2010; Oda et al., 2015).

Conciseness: Whether the comments avoid saying anything that is not necessary (Sridhara et al., 2010).

Preference: Whether one type of summary is preferred over the other (Cortés-Coy et al., 2014), or even whether it is better than the language already existing in the code (Buse and Weimer, 2008). There is quite a bit of variability in how subjects perform this process (Eddy et al., 2013).

Commitability: Whether the comments can be committed (Wong et al., 2015), or whether they are actually committed by developers (Wong et al., 2013; Allamanis et al., 2015a).

6.1.2 Similarity to Manually Created Language

Haiduc et al. (2010a) measure the pyramid method (Nenkova and Passonneau, 2004), which compares keyword lists to gold-standard keyword lists created by multiple human annotators.

Oda et al. (2015) measure the BLEU score (Papineni et al., 2002) with comments created by human annotators.

Allamanis et al. (2015a) measure the sub-token F-measure for generated method names.

6.1.3 Retrieval Accuracy

Allamanis et al. (2015b) measure the accuracy of their retrieval model using mean reciprocal rank of the correct natural language query within a set of distractors.

6.2 Extrinsic

6.2.1 Comprehension

Several studies have found that automatically generated natural language helps developers (McBurney and McMillan, 2014) or programming beginners (Oda et al., 2015) understand code better.

6.2.2 Reading Speed

Oda et al. (2015) find that automatically generated pseudo-code actually decreases reading speed, as it is necessary to parse the potentially incorrect comments.

6.2.3 Reduces Comment Typing Time

Movshovitz-Attias and Cohen (2013) measure the reduction in characters typed by using comment auto-complete.
6.2.4 Task Dependence

Binkley et al. (2013) note that different varieties of natural language description are necessary for different tasks such as code reuse and testing. McBurney and McMillan (2015) also note that there is a disconnect between the summaries written by authors of the code, and summaries written by readers of the code.

7 Acknowledgements

Thank you to Dawn Lawrie and Hideaki Hata for suggestions on papers to include in this survey.

References


Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010b. On the use of automated text summarization techniques for summarizing source code. In Reverse Engineering (WCRE), 2010 17th Working Conference on, pages 35–44. IEEE.


Najam Nazara, Yan Hua, and He Jiang. 2015. Summarizing software artifacts: Classifications, methods, and applications. Submitted to Knowledge-Based Systems.


