Neural Network Machine Translation of Javadoc Tags to Java Specifications

Qifan Lu (lqf96@uw.edu) Nikita Haduong (qu@uw.edu) Samia Ibtasam (samiai@uw.edu)

Motivation

The JDoctor paper [1] proposes generating program specifications from semi-structured Javadoc comments. Program specifications express intended program behaviors and are useful in determining if the program is implemented correctly or not. While formal specifications are rarely practiced, infrequently updated [2] or delayed to a later point in time [3], semi-structured documentation with source code is supposed to convey the program's intention. JDoctor reads documentation accompanying Java classes and functions and constructs executable program specifications in the form of boolean Java expressions.

Javadoc comments are used to annotate code with informal specification and usually include a description of member function followed by a series of tags, which informally define program specifications. For example, the '*@param*' tag describes method parameters and conditions that callers must conform to (called pre-conditions), the *'@return'* tag describes the return value of the function as well as its characteristics (called normal post-conditions); and the *'@throws'* tag describes what and when an exception will be thrown by the function (called exceptional post-conditions).

JDoctor translates Javadoc tags into program specifications with four steps: text normalization, proposition identification, proposition translation, and specification creation. During the text normalization step, JDoctor preprocesses Javadoc tags into grammatical English sentences to accommodate current NLP parsers. To identify propositions, the normalized text is tagged for part-of-speech to extract subject and predicates, based on the observation that nouns tend to correspond to variables or expressions while predicates tend to correspond to function calls. These subject and predicate pairs are translated into propositions by using pattern, lexical, and semantic matching to map natural language to corresponding Java expression fragments (e.g. "is positive" ➔ `>0`). Many of these mappings are manually-created rules that are expensive to create and not easily adaptable to other programming language documentation. They are also not robust to writing style changes (e.g. "is greater than zero" instead of "is positive"). After proposition translation, JDoctor generates program specifications by assembling these fragments into an executable boolean-typed Java expression.

As JDoctor is designed specifically for analyzing Javadoc, with many handcrafted procedures. While it is able to achieve impressive results on Javadoc translation, its rules can be expensive to create (the time and human labor of creating many rules and exceptions). This inspires us to replace the handcrafted translation procedures with a neural network-based model that automatically learns the mapping from natural language to program specifications. Such a model will not only reduce the burden of humans but could also generalize to other programming languages, thus enabling developers to write sound programs regardless of the language they use.

Goal

The project's goal is to develop a neural network-based translator that converts Javadoc tags (or semi-structured documentation for other programming languages) into program specifications, possibly in an end-to-end fashion, where the network's input is solely raw Javadoc tags, and its output is program specifications. Developing this translator comes with many challenges, such as the choice of training data and the structure of our neural network model. We cover major problems and potential solutions to our approach in the "Questions" section in details.

Proposed Design

The initial design will focus solely on translating Javadoc tags into program specifications, whose format is specified below. If the approach is feasible and time permits, we will try to extend our approach to documentation for other programming languages. In either case, the input will be the program documentation tags that are related to method parameters, return values and exceptions. The output will be program specifications in the form of language-dependent executable boolean-typed expressions. In the case of Javadoc, both the input and the output should be identical to [1].

Our neural network (NN) model will use informal Javadoc tags as input and the associated procedural specifications (generated by the JDoctor) as target output. The NN will then learn the relationships between the tags and specifications. The NN might also inform us about the possible weights (or influence) of a few tags over the program specification. This can be useful in understanding the effects or presence (or absence) of a tag or the weakness or strongness of a tag and its impact on the program specifications.

Evaluations

1. **Accuracy (Quantitative):** We have gold-labeled data mapping Javadoc tags to program specifications, so we will evaluate how well our NN performs by having it make predictions on held-out data and calculating the precision, recall, and F1-score of its predictions.
2. **Output Quality (Qualitative):** For cases where the NN prediction is incorrect, we will examine the prediction for how it differs from the gold-label. Are predicted specifications invalid, failing to compile? Are they incomplete, failing to make a complete translation of the tag into program specification?
3. **Adaptation to other programming languages (time permitting):** Once an NN model is trained for particular language-specific expressions, we can also try to evaluate its portability and adaptability to other programming languages by changing the training data (e.g. using Python docstrings instead of Javadoc comments).

Questions

- How do we design the NN so that it minimizes the possibility of producing illegal program specifications? For strongly typed languages, how can the NN model be aware of type systems?
- An NN model typically requires many training samples. How do we choose our training dataset? Possibilities include Java projects using Java Modeling Language (JML) or Contracts for Java (C4J), assignments and JUnit test suites using parameterized test cases and theorems. In case

the quantity of training data is insufficient, we may also consider creating synthetic data by detecting program invariants with systems like Daikon [4] [5].

- Will the NN help us discover new rules and patterns for translating program specifications, will they be interpretable, and will they make sense to a human and be intuitive enough to think of?
- Are there specific JavaDoc tags that cause our NN to make incorrect predictions but are easily and consistently translated correctly by JDoctor?
- Will the NN be more generalizable to other programming languages than JDoctor's handcrafted rules? Can it achieve higher accuracy than JDoctor on the Javadoc tag translation task?

Contribution

The contributions of this project will be:

- To evaluate the suitability of using Neural Networks to translate language-specific descriptors to generate program specifications and its applicability to other languages.
- To measure the effectiveness (see evaluation section) of our approach and compare it with JDoctor, Toradocu (before JDoctor) [6] and @tComment [7].
- Our approach resolves a number of questions mentioned in the previous section.
- To demonstrate the portability and adaptability of our approach to different programming languages if time permits.

References

[1] A. Blasi et al., "Translating Code Comments to Procedure Specifications," in Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, New York, NY, USA, 2018, pp. 242–253.

[2] T. C. Lethbridge, J. Singer, and A. Forward, "How software engineers use documentation: the state of the practice," IEEE Software, vol. 20, no. 6, pp. 35–39, Nov. 2003.

[3] D. Schreck, V. Dallmeier, and T. Zimmermann, "How documentation evolves over time," in Ninth international workshop on Principles of software evolution in conjunction with the 6th ESEC/FSE joint meeting - IWPSE '07, Dubrovnik, Croatia, 2007, p. 4.

[4] M. D. Ernst et al., "The Daikon System for Dynamic Detection of Likely Invariants," Sci. Comput. Program., vol. 69, no. 1–3, pp. 35–45, Dec. 2007.

[5] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," in Proceedings of the 21st International Conference on Software Engineering, New York, NY, USA, 1999, pp. 213–224.

[6] "Toradocu-Randoop (Manual) Evaluation. Feb-2018," GitLab, Feb-2018. [Online]. Available: https://gitlab.cs.washington.edu/randoop/toradocu-manual-evaluation-feb-2018/tree/commons-math. [Accessed: 12-Apr-2019].

[7] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies," in Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, Washington, DC, USA, 2012, pp. 260–269.