# Polyglot: Feature Location Using Doc2Vec

Chuong Ngo
The College of William and Mary
chngo@email.wm.edu

## Abstract

*The ABSTRACT is to be in fully-justified italicized text, at the top of the left-hand column, below the author and affiliation information. Use the word "Abstract" as the title, in 12-point Times, boldface type, centered relative to the column, initially capitalized. The abstract is to be in 10-point, single-spaced type. The abstract may be up to 3 inches (7.62 cm) long. Leave two blank lines after the Abstract, then begin the main text.*

## 1. Introduction

**TODO**

## 2. Background

**TODO**

## 3. Related Works

### 3.1. Related Works in Feature Location

Wilde et al. [29] and Biggerstaff et al. [5] provide a starting point for this work. Wilde et al. tackled the problem of feature location with carefully crafted test cases. However, their approach has a number of limitations that reduces the general practicality of the method. DESIRE, put forward by Biggerstaff et al. utilized a knowledge-based pattern recognizer for feature location by looking at suggestive data names, comments, and the program's call graph to decipher the structure and function of various components of the program.

Antoniol et al. [4] presented a technique using static analysis of source code and dynamic analysis of the program execution to associate features with the relevant subsets of the program source code of object-oriented languages. The subsets of the source code, termed *microarchitectures* by the authors, were identified using static analysis. Execution traces covering a multitude of scenarios related to the features of interest are collected and then correlated with the *microarchitectures*, allowing for the comparison of different features or the same feature across different scenarios. Antoniol et al. based their work on that of Eisenbarth et al. [10], Wilde et al. [30], Salah et. al [28], and their own previous work [3].

Chen et al. [7] extended the *system dependence graph* [13] [12], which is an extension of the *procedure dependence graph* [24], and proposed the *abstract system dependence graph*. Once their system has created the *abstract system dependence graph*, a person must then traverse the graph and decide whether each feature is part of the graph using one of four techniques that the authors outlined.

Poshyvanyk et al. [26] demonstrated a combination of Latent Semantic Indexing (LSI) and Scenerio Based Probablistic (SBP) ranking of events could be used to connect bug reports to relevant sections of source code. In fact, compared to previous studies of using LSI [18] and SBP rankings separately, the combination of the two resulted in better precision and recall while eliminating the need for the knowledge-based filtering that is usually needed with SBP.

McMillian et al. [19] presented their system, Portfolio, as an improved code search engine. Using PageRank and Spreading Activation with textual similiarity at its heart, McMillian et al. showed that Portfolio was more capable than the current tools of the time, Google Code Search and Koder, at returning relevant results earlier.

Dit et al. [9] explores different combinations of LSI, dynamic analysis of the binary, and the Hyperlinked-Induced Topic Search (HITS) and PageRank algorithms for feature location. They concluded that the use of LSI, dynamic analysis, and the HITS algorithm together proved to be the most effective way to locate features and its component parts.

### 3.2. Related Works for Log-bilinear Language Models in Software Engineering

Since log-bilinear language (LBL) models have had a number of successes in regards to modeling natural language [14] [20] [21] [22], their application to source code

seems like a natural extension. This is possible because of the naturalness of software source code, as demonstrated by Hindle et al. [11] previously. In terms of the direct application of log-bilinear models to program source code, Maddison et al. [16] extended the work of Mnih et. al [23] to capture the structure in source code and showed that the use of LBL models yielded large improvements of n-gram and probabilistic context free grammar models in predictive capabilities.

Allamanis et al. [1] also used a LBL model and subtoken analysis to suggest method and class names that conform to the naming conventions seen in the LBL model's training corpus. Allamanis et al. [2] also adapted the work of Kiros et al. [14] to natural text and source code in order to create a mapping between questions posted on StackOverflow.com and source code included in the answers for those questions. That establishes a link between the natural text question and the source code given as the answer, allowing for searches with natural language queries yielding relevant source code and source code queries yielding the natural language context to which it applies. Because that work creates a link between natural language text descriptions and source code, it is most related to the work presented in this paper.

# 4. Methodology

Our tool, *Polyglot*, gives developers insights into the function and features of the methods in their program by issuing natural language queries to it. Not only does this provide a way for new developers to learn about a project, but also helps identify relevant methods for bug reports. Both versions of our tool, *Polyglot-A* and *Polyglot-B*, uses LBL models trained on the program source code and commit messages. Going forward, *Polyglot* will refer to both versions and be used when discussing things that are common in both versions or affect both versions.

*Polyglot* consists of a preprocessing pipeline that feeds into the trained LBL model. The following sections will go into further details on both the preprocessing pipeline and the LBL model.

## 4.1. Text Preprocessing

In order to interact with *Polyglot*, users must format their query as a sequence of text tokens. This must also be done to the training corpora prior to training the LBL models. The preprocessing pipeline consists of a tokenizer, a stopword filter, and a lemmatizer. For the purposes of this tool, two pipelines will be explored.

The first text preprocessing pipeline is implemented using the Natural Language Toolkit (NLTK) [6] and consists of the toolkit's Regular Expression (RegEx) Tokenizer, lemmatizer, and built in English stopwords filter provided by Porter et al. The RegEx Tokenizer uses white space and punctuation as its delimiters. The second pipeline is implemented using the Stanford CoreNLP software suite (CoreNLP) [17] and consists of the suite's PTBTokenizer, lemmatizer, and a stopwords filter provided by Jon Conwell [8].

The above two pipelines are the basic pipelines used. When processing the commit logs, additional processing is done before the commit logs are sent through either of the preprocessing pipelines. The commit logs are broken up into individual commits and only the file names and method names are retained. No additional processing was done on the bug reports. The source code was processed by removing programming language keywords and operators. The source code identifiers and other compound words are split, and everything is stemmed using a Porter Stemmer [25]. This is the same procedure used by Dit et al [9].

## 4.2. Language Model

The Paragraph2Vec [15] implementation provided in the gensim library [27] was used to construct the LBL models for *Polyglot*. Going forward, that gensim's Paragraph2Vec implementation will be referred to as Doc2Vec. Doc2Vec was chosen because it allowed for the grouping of multiple tokens and labeling them as a distinct entity, which works well for our goal of representing methods as whole, distinct units. The feature vectors were fixed at a size of 1 x 500 each and the context window for learning the representations were left at the default 8. A set of models were created using the distributed memory learning algorithm (PV-DM) and another set used the distributed bag of words algorithm (PV-DBOW) [15]. A minimum token appearance count of 5 is be used, meaning that any labels appearing less than 5 times in the corpus is not be learned. Negative samples numbering 5, 10, and 20 are also used.

The corpus used to train the LBL models is the same one used by Dit et al [9]. *Polyglot-A* uses two LBL models, one trained on the source code, labeled as one unit, and one trained on the commit logs cross-referenced with the labeled source code unit. *Polyglot-B* is trained on the source code and commit logs directly, without an intermediary.

# References


[1] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Suggesting accurate method and class names. In *Proceedings of the the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE)*, 2015.

[2] M. Allamanis, D. Tarlow, A. Gordon, and Y. Wei. Bimodal modelling of source code and natural language. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 2123–2132, 2015.

[3] G. Antoniol and Y. Gueheneuc. Feature identification: a novel approach and a case study. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 357–366. IEEE, 2005.

[4] G. Antoniol and Y.-G. Gueheneuc. Feature identification: An epidemiological metaphor. *Software Engineering, IEEE Transactions on*, 32(9):627–641, 2006.

[5] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–82, 1994.

[6] S. Bird, E. Klein, and E. Loper. *Natural language processing with Python*. " O'Reilly Media, Inc.", 2009.

[7] K. Chen and V. Rajlich. Case study of feature location using dependence graph. In *IWPC*, page 241. Citeseer, 2000.

[8] J. Conwell. Corenlp. `https://github.com/jconwell/coreNlp`, 2015. [Online; accessed 2015-09-26].

[9] B. Dit, M. Revelle, and D. Poshyvanyk. Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. *Empirical Software Engineering*, 18(2):277–309, 2013.

[10] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *Software Engineering, IEEE Transactions on*, 29(3):210–224, 2003.

[11] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 837–847. IEEE, 2012.

[12] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *Proceedings of the 14th international conference on Software engineering*, pages 392–411. ACM, 1992.

[13] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[14] R. Kiros, R. Salakhutdinov, and R. Zemel. Multimodal neural language models. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 595–603, 2014.

[15] Q. V. Le and T. Mikolov. Distributed representations of sentences and documents. *arXiv preprint arXiv:1405.4053*, 2014.

[16] C. J. Maddison and D. Tarlow. Structured generative models of natural source code. *arXiv preprint arXiv:1401.0514*, 2014.

[17] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky. The stanford corenlp natural language processing toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60, 2014.

[18] A. Marcus, A. Sergeyev, V. Rajlich, J. Maletic, et al. An information retrieval approach to concept location in source code. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 214–223. IEEE, 2004.

[19] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: finding relevant functions and their usage. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 111–120. IEEE, 2011.

[20] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[21] A. Mnih and G. Hinton. Three new graphical models for statistical language modelling. In *Proceedings of the 24th international conference on Machine learning*, pages 641–648. ACM, 2007.

[22] A. Mnih and K. Kavukcuoglu. Learning word embeddings efficiently with noise-contrastive estimation. In *Advances in Neural Information Processing Systems*, pages 2265–2273, 2013.

[23] A. Mnih and Y. W. Teh. A fast and simple algorithm for training neural probabilistic language models. *arXiv preprint arXiv:1206.6426*, 2012.

[24] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *ACM Sigplan Notices*, volume 19, pages 177–184. ACM, 1984.

[25] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[26] D. Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich. Combining probabilistic ranking and latent semantic indexing for feature identification. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pages 137–148. IEEE, 2006.

[27] R. Řehůřek and P. Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. `http://is.muni.cz/publication/884893/en`.

[28] M. Salah and S. Mancoridis. A hierarchy of dynamic software views: from object-interactions to feature-interactions. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 72–81. IEEE, 2004.

[29] N. Wilde, J. Gomez, T. Gust, D. Strasburg, et al. Locating user functionality in old code. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 200–205. IEEE, 1992.

[30] N. Wilde and M. C. Scully. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, 1995.