Tool for Constructing a Large-Scale Corpus of Code Comments and Other Source Code Annotations

Luna Peck, Susan Windisch Brown
University of Colorado Boulder
{luna.peck, susan.brown}@colorado.edu

Abstract

The sublanguage of source code annotations—explanatory natural language writing that accompanies programming source code—is little-studied in linguistics. To facilitate research into this domain, we have developed a program prototype that can extract code comments and changelogs (i.e. commit messages) from public, open-source code repositories, with automatic tokenization and part-of-speech tagging on the extracted text. The program can also automatically detect and discard "commented-out" source code in data from Python repositories, to prevent it from polluting the corpus, demonstrating that such sanitization is likely feasible for other programming languages as well. With the current tool, we have produced a 6-million word corpus of English-language comments extracted from three different programming languages: Python, C, and C++.

Keywords: text extraction, corpora, code comments

1. Introduction

Within the already-dizzying number of human languages spoken around the world, one can find an even greater number of sublanguages: niche subsets of a language that emerge for specialized purposes. While some sublanguages, such as cooking recipes (e.g. Brdar-Szabó and Brdar, 2009; Gerhardt et al., 2013; DiMeo and Pennell, 2018), have received widespread linguistic attention, others, such as source code annotations, remain woefully understudied.

To the end of eventually developing a large, well-curated corpus of source code annotations, we have developed a tool[1] for automatically extracting such data from source code repositories. By "source code annotation," we are referring to explanatory natural language writing that accompanies formal language programming source code. Examples include comments, changelogs (commit messages)[2], and documentation.

The current version of the program builds a corpus of approximately 6-million words, across 90,461 changelogs and 76,445 comments, taken from three English-language repositories written in the program's supported programming languages: Python (django[3]), C (libvirt[4]), and C++ (dlib[5]). These repositories were chosen as they host code for well-established, widely-used software implemented in the languages currently supported by the program. The program automatically performs tokenization and part-of-speech tagging on the extracted data. The program does not currently specially extract documentation, and documentation that is encoded as comments (e.g., Doxygen) is not differentiated from other comments.

We have discovered that code comments in particular are a very messy domain, requiring much sanitization, such as:

1. Resolving inconsistent encoding and stripping invalid characters (e.g., a changelog contained a bell character, unicode 0x07).

2. Identifying and discarding source code that has been disabled by "commenting it out."

3. Identifying snippets of code embedded in natural language comments and preprocessing them before annotation (e.g., part-of-speech tagging).

The program currently performs (1)[6] and (2) automatically[7]. (3) will be handled in a future version of the program. Due to space constraints, we have only devoted significant discussion to how we have

[1] https://github.com/lunaria-bee/ccc-tools/tree/cawl2024-docs

[2] "Changelog" is a generic term we have chosen to refer to any description of changes made to the source code. In the current version of the tool and its resulting corpus, "changelog" is synonymous with "commit message." However, some codebases, such as those that are not under version control, may log changes in other ways, e.g. simply with comments. The use of "changelog" anticipates a possible future version of this tool that extracts data from such repositories.

[3] https://github.com/django/django

[4] https://github.com/libvirt/libvirt

[5] https://github.com/davisking/dlib

[6] Although we are handling this somewhat reactively, with special handling added for each new inconsistency as it is encountered.

[7] Although (2) currently only functions for comments in Python code.

approached (3), as we consider it the most interesting of the challenges to which we have some solutions.

2. Motivations and Prior Work

This project was inspired by previous (unpublished) work in which we made basic analyses of source code annotations from a small, largely hand-collected dataset. While the smaller dataset was adequate for primarily qualitative analysis, substantive statistical linguistic analysis demands larger datasets.

Source code annotations are particularly interesting in that they exist almost purely in written form. While they interact with the often-spoken sublanguage of software development jargon, comments themselves are both constructed and interpreted overwhelmingly as written language. Further, source code annotations offer an unusual opportunity to study the ways natural language interacts with formal languages: How does writing about formal language influence our production choices in natural language? Are such influences consistent or varied across different formal + natural language pairings?

There is little existing purely linguistic research into this domain. Much existing work has focused on instrumental tasks, like automatically generating comments from source code (Song et al., 2019), automatically generating changelog messages from source code (e.g. Cortés-Coy et al., 2014), and automatically generating source code from natural language prompts (e.g. Barone and Sennrich, 2017; Wei et al., 2019). Even for such instrumental purposes, having a linguistic corpus of source code annotations should prove valuable; the better we understand how programmers write about code, the better we can create tools to translate between natural language and source code.

The small amount of purely linguistic research on the topic has primarily involved Letha Etzkorn (Etzkorn et al., 2001; Vinz and Etzkorn, 2008). This research studied the linguistics of code comments in 11 C++ packages, as opposed to this project, which aims to construct a corpus of multiple annotation types extracted from multiple different programming languages. To our knowledge, the corpus used in Etzkorn et al.'s research has not been published.

A similar project to automatically extract source code annotations from source repositories has been conducted (Barone and Sennrich, 2017), although that project was much more limited in scope than this current project. The Barone & Sennrich project focused only on extracting docstrings[8] from Python programs. This current project, on the other hand, aims to extract a broad range of annotations from source code, including comments, changelogs, and eventually documentation.

Automatically filtering comments for linguistic use has been previously explored (Matskevich and Gordon, 2022). Similarly to Barone and Sennrich (2017), this research focused only on comments, excluding other forms of source code annotations. Nonetheless, the preprocessing techniques described by Matskevich and Gordon (2022) will almost certainly be useful for data sanitization in future versions of this tool.

3. Data Structure

Corpus data is stored as XML. Each source code annotation, be it a changelog or comment, is generically referred to as a "note." The XML structure for each corpus file contains a root node, named `<notes>`, that in turn contains a series of `<note>` elements, each representing a single source code annotation.

Each <note> element contains the raw text of the annotation, its tokenization and part-of-speech tags, and assorted metadata, represented by the following subelements:

- `<author>`: First 8 bytes of the SHA-256 hash of the author's version control username. When an annotation has multiple authors, there is one `author` subelement per author.
- `<repo>`: Name of the repository the data came from.
- `<revision>`: First 7 nibbles[9] of the revision hash, as provided by Git. The first 7 nibbles of the hash is the same ID scheme used by GitHub. When an annotation was edited across multiple revisions, there is one `revision` subelement per revision.
- `<note-type>`: One of 'changelog' or 'comment'.
- `<language>`: Programming language the data was extracted from. Only applicable for comments, not changelogs.
- `<file>`, `<first-line>`, & `<last-line>`: Path to the file a note was extracted from, and the span of lines within that file on which it appears. Only applicable for comments, not changelogs. Facilitates finding the source code associated with a comment.
- `<raw>`: Raw text of the annotation, in which comment delimiters (if applicable) and newlines & other whitespace are preserved.

[8]In-source documentation, see https://peps.python.org/pep-0257/.

[9]A half-byte, i.e. four bits.'

```
<note>
  <repo>dlib </repo>
  <author>0e8171ad9c374e5d</author>
  <revision>754da0e</revision>
  <note-type>comment</note-type>
  <file>repos/dlib/dlib/algs.h</file>
  <first-line>205</first-line><last-line>206</last-line>
  <language>c</language><
  raw>// set the initial guess for what the root is depending on
// how big value is
</raw><
  tokens>set the initial guess for what the root is depending on how big value is
  </tokens>
  <pos>VB DT JJ NN IN WP DT NN VBZ VBG IN WRB JJ NN VBZ</pos>
</note>
```

Figure 1: An example `<note>` element.

- `<tokens>`: Tokenized text. Comment delimiters (if applicable) and original whitespace are stripped from the data. Spaces represent word token separators, and newlines represent sentence token separators.
- `<pos>`: Part-of-speech annotations aligned to the tokenized text. Like in the <tokens> subelement, spaces represent word token separators, and newlines represent sentence token separators. The tags are those assigned by NLTK's `nltk.tag.pos_tag()` function (Bird et al., 2009). A future version of the program may include a part-of-speech tagger designed for the specific domain of source code annotations.

4. Data Extraction

4.1. Comments

The program checks each file in each repository against parsers[10] for each supported programming language. If a file validates as a valid source file in a supported language, the contents of the file will be tokenized by an appropriate parser[11]. If two or more comment tokens appear across consecutive lines, those tokens are grouped into a single comment.

As the purpose of this corpus is to study natural language text, programming code that has been disabled by "commenting it out" should not be included in the dataset. Automated detection of commented-out source code proved somewhat tricky. It is not sufficient to discard any comment that could be interpreted as valid code, as there are many possible natural language comments that look like source code to the right parser. For example, all of the following are valid Python code:

- Single words: e.g., *deprecated*
- A word followed by another word in parentheses: e.g., *Tests (Final)*
- Words separated by mathematical operators like +, -, *, or /: e.g., *Pre-increment/decrement*

Naïvely discarding any comment that parses as programming code risks removing valuable data from the corpus, and so a smarter approach is in order.

After experimenting with several different approaches, for Python we settled on the following policy: A comment is discarded as commented-out code if (1) its contents are valid Python code and (2) it contains parentheses, square brackets, equals signs, periods, or the word "return".

These rules work quite well, discarding:

- Function calls and object construction: e.g., *Color(0, 0.56789, 0, .5)*.

[10]Python's built-in `ast.parse()` method for Python, libclang for C and C++.

[11]Python's built-in `tokenize` library for Python, libclang for C and C++.

		Actual	
		Natural	Code
Predicted	Natural	50	0
	Code	40	10

Figure 2: Confusion matrix showing efficacy of our approach to identifying commented-out Python code on random samples of 50 comments predicted to be natural language and 50 comments predicted to be code. Taking Natural Language (i.e. inclusion in the corpus) as the positive label: Precision=1.0, Recall=0.56.

```
Identified using the following command:

$ git grep -I '\(\<[_a-zA-Z0-9]\+\>\) *= *\1 *[-+/*^%&|<>@]'
```

Figure 3: Bash snippet with git command embedded in a changelog.

- Indexing: e.g., *text[col-1]*.
- Assignments: e.g., *ay += node.y*.
- Subscripting: e.g., *self._trigger_layout*.
- Return statements: e.g., *return None*.

...while keeping many natural language comments that coincidentally resemble source code:

- *Initialize*
- *todo: remove*
- *--Save/Cancel*

This approach does sometimes discard natural language comments we would probably want to keep, such as *return everything in strings*, discarded due to the presence of the keyword "*return*", causing this to resemble an instruction to return whether `everything` is present in some collection `strings`. Although discarding such data is unfortunate, we have decided to prioritize precision over recall in this task, as we deem polluting the dataset with source code more damaging than discarding some potentially useful data. This over-discarding could likely be ameliorated by augmenting these symbolic rules with a learned model that classifies strings into code and non-code.

The program currently does not attempt to identify commented-out C/C++ code. libclang is not designed for parsing fragments of C/C++, expecting to produce a full translation unit from any input. One option for overcoming this limitation would be to attempt to parse the AST output by libclang[12] against a simplified C++ grammar. Another option would be to rely entirely on a learned model for identifying commented-out C/C++ code.

Author and revision information for each comment are retrieved using the `git blame` command, which provides the most-recent commit modifying each line of a file.

4.2. Changelogs

Currently, all data included in the corpus comes from Git repositories, so the program can trivially extract revision information from each repository's commit history. The raw text of the changelog note is simply the commit message, the author is the author of the commit, and the revision is the revision created by the commit.

[12]libclang's `parse()` method produces an AST even for invalid input.

5. NLTK Reader

The demonstration corpus may be explored using the CccReader() class, an extension of NLTK's CategorizedCorpusReader and XMLCorpusReader classes (Bird et al., 2009). It provides the general functions one expects of an NLTK reader: words(), sents(), etc.

6. Future Work

6.1. Handling Code Snippets

While comments that consist entirely of source code should be removed from the corpus, comments and changelogs that include snippets of code embedded within natural language text (often for illustrative purposes) are quite common and should be included in the corpus. However, these source code snippets tend to confuse automated annotation functions (such as those responsible for part-of-speech tagging), as natural language concepts such as part of speech do not map cleanly to source code. Therefore, such snippets must be identified and given special handling.

Identifying code snippets in source annotations is much more challenging than identifying commented-out code, for two main reasons:

- As commented-out code is discarded, and discarding some potentially-useful data is more acceptable than polluting the corpus with source code, that algorithm can simply err on the side of over-discarding. This algorithm, however, would need to identify source code snippets in data that has already been included in the corpus, demanding much higher accuracy.
- The algorithm for discarding commented-out code examines each comment in its entirety. Identifying code snippets, however, would require identifying all spans of text that should be considered source code. Thus, the algorithm

doesn't have to be run against one string, but many substrings of each source annotation.

- Source code snippets are not guaranteed to be written in the same language as the surrounding source code. Consider the Bash snippet in Figure 3.

6.2. Tracking Repositories Across Time

Currently, the program takes a snapshot of each repository at a particular revision level, to ensure that all users invoking the program receive the same corpus as output. However, tracking repositories across multiple revisions could potentially provide useful data regarding how programmers revise comments as code evolves, and allow diachronic analysis of source code annotations.

7. Conclusions

Based on the work completed here, it seems quite feasible to build a corpus of source code annotations much larger than the 6-million-word corpus produced at this time. Although not all of the necessary work can be automated, in our opinion the task is automatable enough to justify pursuing the project further. We intend to continue developing this project, with the goal of building a larger, more diverse corpus of source code annotations, to further develop our understanding of this little-studied domain of human language.

8. Bibliographical References

Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. ArXiv.

Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.".

Rita Brdar-Szabó and Mario Brdar. 2009. Indirect directives in recipes: a cross-linguistic perspective. *Lodz Papers in Pragmatics*, 5(1):107–131.

Luis Fernando Cortés-Coy, Mario Linares-Vásques, Jairo Aponte, and Denys Poshyvaynk. 2014. On automatically generating commit messages via summarization of source code changes. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, Victoria, BC, Canada.

Michelle DiMeo and Sara Pennell. 2018. *Reading and writing recipe books, 1550–1800:*. Manchester University Press, Manchester, England.

Letha H. Etzkorn, Carl G. Davis, and Lisa L. Bowen. 2001. The language of comments in computer software: A sublanguage of english. *Journal of Pragmatics*, 33(11):1731–1756.

Cornelia Gerhardt, Maximiliane Frobenius, and Susanne Ley, editors. 2013. *Culinary Linguistics*. John Benjamins.

Sergey Matskevich and Colin S. Gordon. 2022. Preprocessing source code comments for linguistic models.

Xiaotao Song, Hailong Sun, Xu Wang, and Jiafei Yan. 2019. A survey of automatic generation of source code comments: Algorithms and techniques. *IEEE Access*, 7:111411–111428.

Bradley Vinz and Letha Etzkorn. 2008. Comments as a sublanguage: A study of comment grammar and purpose. In *Proceedings of the 2008 International Conference on Software Engineering Research and Practice, SERP 2008*, pages 17–23.

Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. 2019. Code generation as a dual task of code summarization.