Using Natural Language Processing for Documentation Assist

Alexandre Terrasa, Jean Privat, Guy Tremblay
Université du Québec à Montréal
C.P. 8888, Succ. Centre-ville,
Montréal (QC), Canada H3C 3P8

Abstract

Nowadays, most of the code hosting platforms for open-source projects consider the README file as the project cover. As it is the first piece of documentation seen by the project user or maintainer, such a document needs to be crafted with care.

Documentation assist can be a useful tool to help documentation writers produce better documentation like README files. In this paper, we show how an abstract representation of a README file can help documentation assist tools provide better suggestions to writers.

Our approach benefits from natural language processing tools and techniques to analyze the content of a README file. Using this model and the current cursor position within the document, our tool can suggest pieces of documentation, examples, and figures as well as structure improvements and update suggestions to the writer. Suggestions are presented as cards that can be selected to automatically enhance the document under writing.

Introduction

Writing good documentation is difficult. Programs and libraries tend to become larger and more complex, and writing documentation for such software becomes both a time consuming and expert job. Thus, there is a need for tools to help the documentation writer.

Although there is some support for writing documentation through API documentation generators (e.g., JavaDoc style (Friendly 1995)), the support they provide is rather limited—most of them only generate sorted lists of packages, classes, or properties with their documentation extracted from code comments.

When dealing with writing code, one basic service typically provided to help the code writer is code completion. Such a service helps the programmer find the feature(s) he/she might be looking for among numerous packages, classes, or properties. Code completion informs the writer about which entities from the software model are available—and which one(s) might be most appropriate given the current context.

Writing technical software documentation can be seen as similar to writing code. The writer uses a specific structured language—e.g., HTML, Markdown, ReStructuredText, or even \LaTeX—to express the code’s usage and purpose. And the writer also has to build the finished product—the documentation—describing what is found in the software model.

If code completion is useful for writing code, couldn’t it be used and adapted to assist the documentation writer?

In this paper, we present a tool that suggests documentation pieces called suggestion cards to the documentation writer. These suggestions cards are identified by processing both the source code (of the program or library being documented) and the natural language text found in the already available comments and documentation files using natural language processing approaches.

Writing READMEs

READMEs are an important part of software documentation. They are shipped with the software to be read in raw text format by both users and contributors. They are also used as project cover by open-source code hosting platforms such as GitHub, GitLab, or BitBucket. In both cases, they represent the first piece of documentation seen by the software user.

READMEs Content

READMEs are structured pieces of documentation, containing sections and subsections. Here is the typical structure of a README file for an open-source project (Github 2014):

- Brief description of the project and why it is useful
- Project installation or compilation and usage
- List of features and concerns
- Troubleshooting
- Contributing
- Licensing

Current code hosting platforms accept a wide variety of markup languages for the README file. For example, GitHub currently supports nine markup languages, including Markdown, RDoc, ReStructuredText, and AsciiDoc. Markdown is the most used among these, generally presented as the default choice. Discussions and issues are also usually expressed using Markdown.
Markdown was conceived as an easy to write, easy to read text markup language from which HTML can be generated. It uses a light markup that can be read easily in its raw form. Because it was intended as a language to support multimedia documents, it includes features to write both structured and hyper-linked pages.

Documents can be structured using up to six levels of heading, contain links, images, or verbatim pieces of code—either using back-ticks for short code expressions or using indented code for sequences of instructions. One can also (with GitHub as well as GitLab) annotate segments of code with a programming language name, so that the documentation generator can produce formatted and highlighted code.

**Documentation Assist**

The goal of a documentation assist tool is to help the programmer write better documentation. It can be seen as a kind of code assist tool, but instead for formatted documentation production. Thus, it aims at helping the documentation writer by providing suggestions about what should be written.

Documentation quality is difficult to measure and describe. However, researchers have identified some key attributes of good documentation (de Souza, Anquetil, and de Oliveira 2005; Forward and Lethbridge 2002):

- **structured**, i.e., the documentation is divided into sections and subsections;
- **containing examples** that are explicative, minimal, and up-to-date;
- **containing abstractions** such as diagrams (e.g., UML package, class, sequence diagrams, etc.).

To produce better documentation, a documentation assist tool should thus provide a way to help the writer produce a documentation having these attributes.

Structuring a document becomes more difficult as the document size grows. During its lifespan, the document will go through multiple revisions. Helping the writer provide a good structure means suggesting a structure as well as checking the structure the writer may have already provided.

When writing examples, the documentation writer seeks to provide examples, generally short, of how to use parts of the API or how to handle certain use cases. For projects with unit tests, many if not most of the use cases are already checked. Providing pieces of code matching certain use cases from the test suites can then be a way to help the writer provide better examples.

Writing diagrams can be a tedious task even for small group of classes. It commonly implies drawing figures in an external tool, then importing those figures as images in the documentation. When the code changes, the figures must then manually updated. A documentation assist tool should provide a way to easily generate figures and keep them up-to-date.

**Suggestion Cards**

Code completion proposes a list of model entities that match the current context (i.e., receiver type, method name, etc.). The entities appear in a drop-down list along with their signature and comments. For our documentation assist tool, the suggestions to be presented—which consist of code examples, UML diagrams, or structure modifications—require more than a simple drop-down list.

One possible way to present complex suggestions is illustrated by the Google Spreadsheet Explore feature,¹ which presents a set of suggestions based on the selected data in the spreadsheet. Suggestions are presented as a list of cards in the right panel of the page, each card providing an action link. When selected, the link will edit the data, format it, include new formulas, etc., depending on the card’s kind.

We reuse this concept to provide documentation suggestions formatted as cards that can be selected on to enhance the documentation under writing. We group these cards into two categories, *insert suggestions* and *change suggestions*, which we describe below.

**Insert Suggestion Cards**

Insert suggestions are cards that can be selected to insert new content into the documentation at the current cursor position.

Some insert cards pertain to the top-level of the document to scaffold the overall structure of the README. These cards are related to the project as a whole and are inferred from README templates:

- **Features list**: Concerns extracted from code and documentation;
- **Usage**: Usage example, e.g., code example for a method or command line command example for a program;
- **License**: Authors and copyright;
- **Collaborating/testing**: Link to the code repository, how to run tests;
- **Troubleshooting**: Link to the repository issues.

Other insert cards are related to free text documentation, where the writer describes technical aspects of the source code. These cards are related to the model entity (i.e., module, class, method, or property) which is being documented by the writer.

- **Links and comments**: Includes a link to the API documentation page of the entity or include the entity comment;
- **Usage**: Include an example of how to import, extend, initialize, or call the entity (example generated or extracted from the unit tests);
- **UML figures**: Generated package and class diagrams about the entity (inheritance and usage);
- **Lists**: Related entities like a *see also* list suggestion for related methods, sub-classes, super interfaces, etc.;
- **Next section**: What the writer should document next.

When selecting an insert card, the editor inserts a custom instruction at the current cursor position:

```
[[card name: argument1, argument2...]]
```

¹https://support.google.com/docs/answer/6280499
These instructions are then parsed by a custom Markdown to HTML processor which inserts the corresponding HTML for each card (i.e., inserts the list of entities from a list card or the image generated from a UML diagram card).

**Change Suggestions Cards**

Change suggestions cards are based on the analysis of what was already written by the documentation writer. Their goal is to suggest improvements to what already exists:

- Fixing spelling and grammar errors;
- Updating references, code, and figures;
- Inserting missing links or examples;
- Reordering section or inserting subsections;
- Removing unrelated content.

When clicked, these cards reorder or modify parts of the document content.

**Analyzing Markdown READMEs**

To provide good suggestions based on the current content of the README, we need to understand and represent this content.

We analyze the structure of the Markdown document to extract the document tree composed of sections and subsections. For each section, we extract the section paragraphs, code blocks, and the custom Markdown instructions of already inserted cards.

With this information, we create a set of multidimensional vectors describing the defining attributes of the section:

- The natural language vector is created by analyzing the paragraphs content with the Stanford NLP parser (Manning et al. 2014);
- The code vector is created by analyzing the code blocks content with a custom parser;
- The references vector is created using the references to model entities found directly in the text, if any—looking for qualified names such as `java::lang::Object` or simple names like `Object` or `toString();`
- The intent vector is created by looking for certain keywords, such as example, list, figure, etc. These keywords are selected by calculating the statistical probability of the apparition of a card after a specific word;
- The table of contents vector is created by looking at the parent section, children subsections, previous and following sections.

Combined together, these vectors provide an abstract representation of the content of each section, and thus of the whole document.

**Selection of Suggestion Cards**

Fig. 1 provides an overall view of the suggestion process. We first index both source code and comments from the target program by creating natural language and code vector for each model entity. The resulting vectors are used to create a reversed index (Ramos and others 2003).

**Insert Suggestion Cards**

To provide the writer with relevant insert suggestion cards, we look at the current cursor position within the document to determine the section being worked on, along with the position within the section.

We compute the cosine similarity between the section combined vectors (natural language, code, references, and table of contents) and the vectors from the index to match the most relevant entities. From the list of matched entities, we generate the list of suggestions depending on the intent vector of the section and the kind of entity matched from the index.

- Packages: Features, bin usage, troubleshooting, contributing, license;
- Classes: Initialization example, interesting methods, inheritance diagram;
- Methods: Call examples, list of related methods.

Suggestions are then filtered—to exclude cards that were already used—and ranked. For the ranking, we use a combination of both the cosine similarity of the entity with the document vector and the distance of the entity from the cursor position. Closer entities are ranked before farther entities.

**Change Suggestion Cards**

Change suggestions are based on the content of the document itself. We use the position of the cursor to provide changes relevant to the current section before changes to be applied to the whole document.

**Cross-Reference Change Suggestions**

When parsing the document, we look for links and references to entities. We also check already inserted cards instructions. When a referenced entity cannot be found in the model of the program, we display a warning which includes a list of change suggestions to the entities matched from the index.
Structure Change Suggestions  We use FCA (Formal Concepts Analysis (Poshyvanyk, Gethers, and Marcus 2012)) to suggest structure changes. We reuse the document vectors as objects represented by their attributes that are the vectors content. With these objects, we create a concept lattice representing the ideal document structure. We then match this concept lattice with the actual document structure to suggest new sections, new subsections, sections reordering, or new sections nesting.

Prototype
We developed a prototype for the Nit language (Privat 2006), based on its document generator Nitdoc. Nit is an object-oriented language with a script-like syntax but with adaptive static type-checking. It supports multiple inheritance with a strong module system orthogonal to classes and methods.

Nitdoc provides an online documentation server with dynamic rendering, along with a REST API to access the program model and documentation features. Our documentation assist tool interacts with this REST API using Javascript/Ajax.

Nit and Nitdoc already use the Markdown format as default documentation markup. Nit also provides a way to document packages through a README.md file located in the package directory.

Content Assist Presentation
We present the writer with a text field supporting Markdown syntactic highlight and a right panel to display the suggestions—see Fig. 2. When the content of the text field is updated, its content is sent to the assist server along with the current cursor position. A list of suggestions is returned as response and the right panel is updated accordingly.

For each card, we specify an action that will be triggered. When selecting the card, the whole content of the text field is sent again along with the selected card. The server then responds with the text updated through the applied action, and the text field is updated accordingly. This way, the front-end can handle both insertion and structure changes in a generic way.

Once satisfied with its document, the writer must save it. Three output format are provided: i) README.nitdoc, a markdown format with support of cards commands; ii) README.md, where all cards commands have been replaced by either markdown formatted text or image inclusion; iii) README.html, a standalone HTML page.

Conclusion and Future Work
With our prototype, we showed that an abstract representation of a README file combined with natural language processing can support the implementation of a documentation assist tool.

We are now focusing our work on the validation of the quality of the produced documents, to evaluate whether documentation assist features are helpful for writing READMEs and enhance their quality, and also try to measure the approach’s usefulness for documentation writers.

References