Marcus Roldan
ENGW 3302
Annotated Bibliography
10 October 2023

I. BACKGROUND

As the importance and reliance upon technology continues to increase in the modern world, the importance and reliance upon software systems likewise increases. These software systems consist of different software subsystems which each accomplish different subtasks and are similarly composed of their own subsystems. These pieces of software, often written by many different developers, are woven together, relying upon their stated inputs, outputs, and use cases. The essential attributes of these systems are communicated through a variety of **Natural Language (NL)** components which accompany the **Programming Language (PL)** components. The production and upkeep of these NL components, typically referred to as documentation, carries with it an importance that rivals that of the source code itself. A prototypical developer's view of documentation is that while it is essential to effective and efficient development, it is a drain on a programmer's time and attention.

The task of automatic generation of documentation for any arbitrary source code is one that has garnered attention in recent years, especially as the scale and capabilities of **Generative Neural Networks (GNNs)** and **Large Language Models (LLMs)** has increased dramatically. Specifically, the exploration of different LLMs, **prompt engineering**, and more sophisticated AI models, along with the investigation of the evaluation models used in such experiments has occurred.

This bibliography aims to provide an overview of the recent experimentation with various AI techniques to provide an effective strategy for the automatic generation of arbitrary source code documentation as well as the scrutinization of the standard evaluation models used in these experiments.

II. ANNOTATED BIBLIOGRAPHY

[1] C. D. Newman *et al*., "Automatically generating natural language documentation for methods," *2018 IEEE Third Int. Workshop on Dynamic Software Documentation* (DySDoc3), Sep. 2018. doi:10.1109/dysdoc3.2018.00007

The tool *MethodManual* (MethodMan) was developed to generate method-level summaries of source code without reliance on **NLP** techniques. MethodMan utilizes stereotype summarization, where a method is categorized information from the source code is extracted into an associated summary template based on its categorization, through the use of **srcML** infrastructure. The guidelines followed for the generated summaries are as follows: the summary should at least include a verb that accurately describes the purpose of a method, some object that the method acts upon, and any subsequent calls made during the method. Additionally, the name of the method is considered to contain the most important information about the method. MethodMan is able to generate summaries for entire source codes in seconds, including stereotype and srcML processing.

MethodMan shows overly sophisticated techniques are not always necessary in certain situations. In terms of automatic documentation generation, the tool accomplishes this task, but has a narrow range of efficacy. MethodMan has only been shown to operate effectively on Java code at the method level and does not address any higher level documentation needs, but its simplicity and speed are notable. Because of these aspects, MethodMan serves as a proof of concept that the stereotype technique is effective in certain contexts, but highlights the need for more sophisticated strategies to address its drawbacks, mainly flexibility in language and lack of global insight.

[2] X. Hu *et al*., "Correlating automated and human evaluation of code documentation generation quality," *ACM Trans. on Software Eng. and Methodology*, vol. 31, no. 4, pp. 1–28, 2022. doi:10.1145/3502853

NLP metrics are often used to automatically measure the quality of automatic documentation generation strategies alongside human evaluation; a less rigorous, time-consuming, and expensive strategy with reduced sample size. This study compares the NLP metrics **BLEU**, **METEOR**, **ROUGE-L**, **CIDEr**, and **SPICE** with human evaluations to investigate the effectiveness of these automatic metrics by examining Pearson correlations between them and the human evaluations. Automatic metrics showed strong correlation in extreme cases (less effective for comments rated as average by humans) and METEOR was the most correlated while BLEU was the least (Tab. 6, pp. 20). The authors recommend that METEOR be included in more studies of automatic code documentation generation, while noting there is still a gap between it and human evaluations. Finally, for NL to NL translation, high BLEU scores are strongly correlated to human evaluation while this study found BLEU to be the lowest correlating metric. The authors recommend the inclusion of *Adequacy* (usefulness and understandability) into models by focussing on high-information tokens: API names, identifiers, etc. The authors believe combining *Adequacy* with existing metrics will improve correlations between human and automatic evaluations.

The task of automatic documentation generation has a large amount of variability in both input and output, meaning effective evaluation of proposed models requires that a complete representation of these inputs and outputs. At such a scale, human evaluation quickly becomes unsustainable, demonstrating the importance of the development of effective automatic evaluation metrics. This study found the most popular automatic evaluation metrics are often borrowed from computational linguistics (BLEU, METEOR, etc.) which lack the incorporation of key aspects of documentation. The identification of *Adequacy* as a missing component is a key step in the development of a more effective automatic evaluation metric which will facilitate the efficiency of future studies.

[3] J. Y. Khan and G. Uddin, "Automatic code documentation generation using GPT-3," *Proc. of the 37th IEEE/ACM Int. Conf. on Automated Software Eng.*, 2022. doi:10.1145/3551349.3559548

The goal of this study is to evaluate the effectiveness of **GPT**-3 model Codex for use in code documentation generation utilizing data from CodeSearchNet: a dataset containing code and some associated documentation across several common programming languages, preprocessed to remove non-English code, code outside the selected token amount (3-256, inclusive) or containing special tokens. The experiment used Prompt Engineering, where a task is supplied for Codex to perform. Two methods were used: zero-shot learning, where the model is told to generate documentation from a supplied piece of code, and one-shot learning, where the model is given an example piece of code, its associated documentation, and another separate piece of code for which it should generate documentation. Codex was evaluated using both quantitative metrics (BLEU) and qualitatively using the length and **Flesch-Kincaid Grade Level (FKGL)**, which showed satisfactory informativeness (Sec. 3.3.2), but in some cases the Codex generated documentation was more comprehensible (using FKGL) and included additional useful information. Codex provided satisfactory results but often failed to gauge the expected documentation format with zero-shot learning, but with one-shot learning, Codex outperformed other models in 4 of the 6 studied languages, and was second-best in the remaining two (Tab. 2). In the future, the use of more advanced parameter tuning, few-shot learning, and fine-tuning will be studied to attempt to improve the performance further.

This study suggests the vast amounts of training on not only NL, but specifically on Internet content have created a generative AI model which can serve as an excellent foundation for descendants more tailored towards code documentation generation. However, the study does have a limited number of programming languages where all are higher-level, and none are explicitly functional. Additionally, following from [2], BLEU evaluation is not the best option, but the inclusion of FKGL attempts to remedy the shortcomings of BLEU and aids in the evaluation of Codex.

[4] Y. Su, C. Wan, U. Sethi, S. Lu, M. Musuvathi, and S. K. Nath, "HOTGPT: How to make software documentation more useful with a large language model?," *Proc. of the 19th Workshop on Hot Topics in Operating Syst.*, 2023. doi:10.1145/3593856.3595910

This study experiments with Codex for use interacting with NL components of software systems, specifically **JavaDoc** comments. The goal was to convert exception conditions from JavaDoc comments (denoted by **@throws**) into Java exception predicates. The researchers developed a prompt consisting of three sections: some instruction text, an example to learn from, and the task itself which were given to Codex. The researchers noted that the output was not simply the desired exception predicate, but included some extraneous information, that once filtered, yielded the desired predicate. This methodology was tested against 6 Java libraries where sixty percent of @throws tags had associated predicates, allowing for the effectiveness of Codex to be assessed on ground truth examples on the basis of both precision (accuracy of produced predicates) and recall (rate of predicate production). Codex was found to be effective at this translation task (Tab. 2). The use of LLMs to support SE tasks such as developing exception predicates from signatures and comments shows promise, but there is no guarantee of correctness, and they depend heavily on prompt engineering.

Similarly to [1], this study aims to propose a model effective at a very distilled task, generating exception predicates based upon JavaDoc @throws. This study and [3] utilize Codex and note that prompt engineering is important for quality output, highlighting the need for further investigation into prompt engineering techniques, despite the translation directions being opposite (PL to NL in [3], NL to PL in [4]). This study's data source was only 6 Java libraries whose development strategies and intended uses are unknown. Depending on the complexity of the methods contained within the library, the outputs of this study may vary according to their complexity, length, quality of Javadoc comment, etc. How these qualities affect the ability of Codex to generate predicates is left unanswered, but it would be reasonable to speculate based on the increased effectiveness of Codex with one-shot learning in [3] that the qualities of the input greatly influence the received output.

[5] F. Mu, X. Chen, L. Shi, S. Wang, and Q. Wang, "Automatic comment generation via multi-pass deliberation," *Proc. of the 37th IEEE/ACM Int. Conf. on Automated Software Eng.*, 2022. doi:10.1145/3551349.3556917

Many code documentation generation models utilize the encoder-decoder translation framework to 'translate' source code into NL, a strategy with two major drawbacks: error accumulation and inability to leverage global level information. This article proposes a multi-pass deliberation model DECOM. The most similar code from a predefined corpus is identified, and its associated comment becomes the first draft, which along with the source code and extracted keywords (from identifiers) are given as input to DECOM, which performs optimization upon the draft: evaluating the semantic similarity between each new draft and the original source code, terminating when either the deliberation limit is reached or the generated documentation is of sufficient quality. The DECOM model extracts identifiers by tokenizing the source code, extracting the most semantically useful tokens and sub-tokenizes them according to naming conventions (CamelCase, snake_case, etc.), then utilizes three levels of Transformer Encoders (code, keyword, comment) followed by a Decoder. DECOM was shown to outperform SOTA models across seven metrics and two langauge datasets (Tab. 2). The multi-pass deliberation and evaluation models were shown to contribute positively to the performance of DECOM (Tab. 3). The authors acknowledge that the retrieval of similar source code and comment pair is a weak point, and that DECOM was only tested against two languages (Java, Python), and plan to improve the quality of DECOM by leveraging reinforcement learning techniques into the model.

In contrast to [3] and [4], this study takes a different approach towards generation and directly addresses the shortcomings of the strategies used in [3] and [4] (error accumulation and the lack of global level information). Additionally, it shares similarities to [1] in which it uses observations about the structure of the code to extract the presumed most informative tokens. Overall this novel strategy shows promise and the improvements identified by the authors are aimed at the largest soft spots of the strategy. Finally, the higher-level strategy follows the strategy commonly employed by humans which, especially in tasks where AI aims to produce human-quality output, lends more credibility in the long-term viability of the approach.

[6] A. Y. Wang *et al*., "Documentation matters: Human-centered AI system to assist data science code documentation in computational notebooks," *ACM Trans. on Computer-Human Interaction*, vol. 29, no. 2, pp. 1–33, 2022. doi:10.1145/3489465

Themisto is an automated code documentation generation system which can accomplish three different tasks: first, novel source code documentation generation using **deep-learning** approaches; second, third-party library and package documentation retrieval using **APIs**; third, semi-automated documentation generation utilizing prompts to aid in effective documentation. To develop Themisto, 80 "well-documented" (received the most votes in popular data science competitions) **computational notebooks** were studied, identifying 9 categories for content across 4 stages of development, consisting of 13 task types, influencing the design of a **Graph Neural Networks** based Translation Model to 'translate' source code into NL documentation. The prepossessing of the data includes standard cleaning methods (removing special tokens, etc.) and the generation of **ASTs**. This model results in documentation with higher than average BLEU scores (pp. 13), but suggests the generation of documentation of computational notebooks is a tougher task than documentation generation of more standard-use **SE** cases. Following evaluation by 24 data scientists, it was found that Themisto reduced the time needed to create documentation, increased the documentation's consistency, and increased the satisfaction the participants felt towards their code.

There are many similarities between computational notebooks used in data science and software systems used in SE. Themisto's design task is useful to examine the applicability of the translation task paradigm of automatic documentation generation in the computational notebook context. The strategy employed incorporates a similar 'stereotyping' mechanism to [1] where in conjunction with a deep-learning model, the different categories of documentation and context were enumerated to aid in the specificity of the output. However, for its automatic evaluation model, the study relies upon BLEU scores, which lack the ability to evaluate the *adequacy* of generated documentation [2], but this drawback is countered by the human-based evaluation which found in favor of Themisto's utility.

III. GLOSSARY

A. Linguistic Terms

Abstract Syntax Tree (AST): A representation of the abstract syntax structure of some piece of text utilizing the tree format.

BLEU, METEOR, ROUGE-L, CIDEr, SPICE: Common evaluation metrics used in experiments with automatic documentation generation, originally developed for computational linguistic uses to measure semantic similarity.

Flesch-Kincaid Grade Level: Formula used to quantify the difficulty level of some piece of English text.

Natural Language (NL): Languages that arose from natural use such as English, Spanish, French, Chinese, etc.

Natural Language Processing (NLP): Field of Computer Science and Artificial Intelligence which attempts to allow computers to understand Natural Languages and operate upon them.

Programming Language (PL): Notation system used by programmers to write instructions for computers to execute. Not used as a primary communication method between humans.

B. Artificial Intelligence Terms

Deep Learning: Artificial Intelligence technique which leverages layers of artificial neural networks to accomplish some task.

Graph Neural Networks: An Artificial Intelligence technique combining neural networks and mathematical graph theory.

Generative Neural Networks (GNNs): A category of Neural Networks and more broadly Artificial Intelligence models whose primary function is to generate novel output based on both their design and given input.

Generative Pre-trained Transformer (GPT): A Large Language Model for generative Artificial Intelligence, with different iterations and variants where the first was developed by OpenAI.

Large Language Models (LLMs): Type of Artificial Intelligence model which utilizes deep-learning techniques and large data sets to accomplish some language related task (generation, summarization, prediction, etc.)

Prompt Engineering: The process of developing input for a Large Language Model or some other Artificial Intelligence system to achieve some desired output.

C. Software Engineering Terms

Application Programming Interface (API): A strategy for allowing two distinct software systems to interact and request data from one another, generally consisting of some set of pre-defined queries, parameters, and protocols.

Computational Notebooks: A Data Science tool which allows source code to be directly embedded and potentially executed within a document that is used to explain the steps taken in some Data Science task.

JavaDoc: A documentation tool for Java source code which allows for simple conversion of documentation into HTML, allowing for online webpages to be generated from documentation.

@throws: A type of tag in JavaDoc which indicates the conditions in which a method will 'throw' an exception (encounter a problem which it cannot solve) as well as the type of exception.

Software Engineering (SE): Field of developing, upgrading, and maintaining a software system.

srcML: A tool developed to interpret source code into its abstract components for use in analysis or other tasks where the structure of source code needs to be extracted.