turbocommit

A Case Study in Applying Al to Streamline the Software Development Process

Conventional commit messages play a vital role in maintaining a clear and well-structured version control history in software development. However, writing informative and consistent commit messages can be challenging and time-consuming for developers. This paper presents `turbocommit`, a novel tool that leverages the power of AI to generate conventional commit messages from git diffs. Utilizing the GPT-3.5 Turbo language model, `turbocommit` aims to streamline the software development process by automating commit message creation and ensuring compliance with the Conventional Commits specification. The paper delves into the requirements, implementation, and evaluation of the tool. The performance of AI-generated commit messages with manually written ones in terms of accuracy, readability, and adherence to the specification.

Raik Rohde 4380676

Table of Contents

1 Introduction 3
2 Theory 3
    2.1 Conventional Commits 3
    2.2 Git and Git Diffs 3
    2.3 GPT-3.5 Turbo Language Model 3
    2.4 Requirements for turbocommit 4
    2.5 Evaluation Metrics 4
        2.5.1 Adherence to Conventional Commits Specification 5
        2.5.2 Readability 5
        2.5.3 Accuracy 5
3 Methods 5
    3.1 Gathering Data 5
    3.2 Implementation 6
    3.3 Evaluation 6
4 Results 6
5 Discussion 6
6 References 7

1 Introduction

Commit messages play a crucial role in software development by maintaining a clear and well-structured version control history. However, developers often face difficulties in creating consistent and informative commit messages, which can hinder the efficiency of the development process. In this paper, we introduce `turbocommit`, a tool that harnesses the power of AI, specifically the GPT-3.5 Turbo language model, to generate conventional commit messages from git diffs. The primary aim of `turbocommit` is to streamline the software development process by automating commit message creation and ensuring adherence to the Conventional Commits specification.

The objective of this term paper is to show the development process, requirements, and evaluation of `turbocommit`. We will compare the performance of AI-generated commit messages with manually written ones using various metrics, such as accuracy, readability, and compliance with the Conventional Commits specification.

2 Theory

2.1 Conventional Commits

Conventional Commits is a specification for structuring commit messages in a consistent and easily understood manner. The specification aims to improve the communication between team members, simplify the process of generating release notes, and streamline version management.

A conventional commit message has a rigid structure, which includes a type, an optional scope, a subject, and optional body and footer elements. The type is a short descriptor of the change, such as `feat` for a new feature or `fix` for a bug fix. Other common types include `refactor`, `docs`, `test`, and `chore`. The scope is an optional noun that specifies the part of the codebase affected by the change, enclosed in parentheses.

The subject is a concise description of the change, written in the present tense. The body and footer elements are optional and provide additional context or information about the commit, such as further explanations or issue references. If the commit represents a breaking change, it must be marked with an exclamation mark at the end of the type, or include a footer element with the text `BREAKING CHANGE: <explanation>`. [1]

By adhering to the Conventional Commits specification, development teams can maintain a clear and informative version control history. This is not only beneficial to the collaboration and communication but also allows for the automation of tasks such as generating release notes or determining the next semantic version number. In the context of `turbocommit`, the goal is to generate commit messages that meet these criteria while accurately reflecting the changes made in the git diff.

An example of a conventional commit message is shown below:

```
feat(backend): add new endpoint for registration

The endpoint accepts a POST request with a JSON
body containing the users email and password.
```

2.2 Git and Git Diffs

Git is a distributed version control system, widely used in software development for tracking changes in codebases and facilitating collaboration among developers. It enables developers to create branches, manage versions, and merge changes efficiently while maintaining the integrity of the codebase.

Git diffs are an integral part of the git workflow, as they provide a human and machine readble representation of the differences between two sets of code, most often between the working directory and the staged or committed changes. Diffs display the lines of code that have been added, modified, or deleted and can be generated using the `git diff` command.

In the context of `turbocommit`, git diffs serve as the primary context for generating conventional commit messages. By analyzing the differences in code, `turbocommit` can determine the nature of the changes and use this information to create a commit message that accurately reflects the modifications made to the repository.

2.3 GPT-3.5 Turbo Language Model

Large language models, such as the GPT-3 family developed by leading AI research organization OpenAI, are a type of artificial intelligence (AI) model falling under the broader category of neural networks. These networks mimic the human brain's information processing, learning

from extensive data to recognize patterns and make predictions. Specifically designed to understand and generate human-like text, large language models are trained on vast amounts of text from diverse sources, enabling them to learn language patterns, grammar, and context, which allows for a more accurate understanding and generation of text. [2]

The core architecture behind large language models like `gpt-3.5-turbo` is the Transformer. [3] The Transformer architecture has become the foundation for most state-of-the-art natural language processing (NLP) models. Its key innovation is the self-attention mechanism, which allows the model to weigh the importance of different tokens in a given context. This mechanism, combined with the model's ability to process input in parallel, has led to significant advancements in NLP tasks.

Tokens are the basic units of text processed by language models. They can represent single characters, words, or subwords, depending on the language and tokenization method used. Tokenization is the process of breaking down text into these smaller units, which allows language models to analyze and manipulate text more effectively. In essence, tokens enable the models to learn and understand the underlying structure and relationships between various textual elements, insteaf of only learning existing words and how they are used in a given context.

Training a large language model involves a two-step process: pre-training and fine-tuning. In the pre-training phase, models are trained on a massive dataset, typically containing billions of tokens from diverse sources, such as books, articles, and websites. This unsupervised learning allows the model to acquire a general understanding of language, grammar, and context. The fine-tuning phase involves training the model on a smaller, more specific dataset to adapt it to a particular task or domain.

Despite their impressive capabilities, large language models are not without limitations. They require substantial computational resources for training and deployment, raising concerns about energy consumption and accessibility. Additionally, these models can be prone to bias and what is called 'hallucinations', where it makes reasoning errors or creates facts. Researchers and practitioners must address these challenges to ensure the responsible and ethical use of large language models in various applications.

2.4 Requirements for turbocommit

One of the primary requirements for `turbocommit` is its ability to generate commit messages that adhere to the Conventional Commits specification. Ensuring consistency in the commit history is essential, as it makes the codebase more informative and accessible for developers while enabling automated tasks such as generating release notes and determining semantic versioning.

Integration with the git workflow is also crucial for `turbocommit`. Allowing the tool to work seamlessly alongside git commands and be easily accessible from the command line is important, as it caters to the lowest common denominator of developers' working environments.

Accurate reflection of the changes made in the git diff is another key requirement. Ensuring that `turbocommit` generates commit messages that not only comply with the Conventional Commits specification but also accurately represent the modifications made to the codebase is vital, as it provides developers with a clear and informative commit history.

Efficiency and performance, along with cost-effectiveness, are important factors for `turbocommit`. Enabling the tool to utilize the GPT-3.5 Turbo model effectively is necessary, as it delivers fast response times and cost-effective usage, ensuring that developers can generate commit messages quickly without incurring excessive operational costs.

Finally, a user-friendly interface is essential for `turbocommit`. Making the command-line interface intuitive and providing users with clear instructions and options to customize the commit message generation process is important, as it makes it easy for developers to adopt and integrate into their existing workflows.

2.5 Evaluation Metrics

In order to evaluate a commit message, a set of evaluation metrics will be employed, each providing a score between 0 and 1. A higher score indicates better performance in that particular metric. These metrics assess various aspects of the message, such as adherence to the specification, readability, and accuracy of the described change. To obtain an overall evaluation score, the individual scores for each metric will

be combined with equal weighting. By combining these scores, the final evaluation score will also range between 0 and 1, with a higher score signifying a better commit. With such a scoring system, it is possible to compare commits written by a person and those generated by `turbocommit`.

2.5.1 Adherence to Conventional Commits Specification

To assess this, a Conventional Commit parser is used, which validates the structure and syntax of the generated commit messages. This parser examines elements such as the type, scope, and subject of the commit message to ensure that they conform to the established guidelines. In this evaluation metric, the adherence to the spec is measured as a binary score (0 or 1), where a score of 1 indicates full compliance with the specification, and a score of 0 represents non-compliance.

2.5.2 Readability

To assess the readability of commit messages, two aspects will be evaluated: the title and the body.

For evaluating the title, we will use Natural Language Processing (NLP) techniques, specifically Part-of-Speech (POS) tagging [4] . POS tagging helps us identify the grammatical structure of the title and determine if it is a proper sentence written in the present tense. If the title is detected to be a correct sentence and written in the present tense, it will be assigned a score of 1, otherwise it will be assigned a score of 0.

For the body of the commit message, the Flesch Reading Ease Score (FRES) [5] is used, to measure the readability of the text. FRES is a widely used readability metric that assesses the complexity of the text based on the number of words, sentences, and syllables. The score ranges from 0 to 100, with higher scores indicating greater readability.

Since our goal is to have commit messages that are easy to read, we will normalize the FRE scores to a range between 0 and 1. We will set a target FRES of 65 as the upper limit, considering it to be easy enough to read for our purpose. To normalize the FRE scores, any score of 65 or above will be assigned a value of 1, while a score of 0 will remain 0. Scores in between will be linearly interpolated within the 0 to 1 range.

To calculate the overall readability score, both the title and body scores will be combined with equal weight. If a commit message does not have a body, only the title score will be used for the overall readability score.

2.5.3 Accuracy

Evaluating the accuracy of commit messages is the most challenging aspect of this assessment. Since there is no straightforward automated method to measure the accuracy, it will be done manually using a rubric.

The rubric for accuracy is as follows:

1.0: The commit message accurately and completely describes the changes made in the code.

0.8: The commit message accurately describes most of the changes, but misses some minor details.

0.6: The commit message describes some of the changes accurately, but significant portions are missing or unclear.

0.4: The commit message is only somewhat accurate, with numerous missing or incorrect details.

0.2: The commit message barely describes the changes made, with little to no accurate information.

0.0: The commit message does not accurately describe the changes at all.

To ensure a fair comparison between human-written and generated commit messages, the evaluation process will be carried out right after studying the changes in the code. Both types of commit messages will be scored consecutively, maintaining consistency and minimizing bias in the assessment. This approach will help in providing a reliable measure of the accuracy of the described changes within the commit messages.

3 Methods

3.1 Gathering Data

Several GitHub repositories were identified as suitable sources of commit messages. These repositories include: taxonomy [6] , demo-projects [7] , resume [8] , and kotlin-faker [9] . While kotlin-faker does not adhere to

Conventional Commits, its commit messages maintain a consistent structure. The specific functionality of these repositories is not vital for the evaluation.

All selected repositories have multiple contributors, which increases the diversity of the commit messages. A Python script was utilized to traverse all commits in these repositories, extracting both the commit message and the corresponding git diff. These data points were then stored in an SQLite database for further processing. Commit messages authored by bots or merge commits were filtered out and placed in a separate table within the same SQLite database.

From the three repositories using Conventional Commits, slightly more than 100 non-filtered commits were collected for each (103 from taxonomy, 110 from demo-projects, and 127 from resume). To maintain a balanced evaluation, only 113 random commits from kotlin-faker were included. This approach ensures that the overall evaluation is not disproportionately influenced by the kotlin-faker repository.

3.2 Implementation

Adapted `turbocommit` implementation to get its data from a temp file and write its output to a temp file, to easily be used in a python script.

3.3 Evaluation

Python scripts

4 Results

5 Discussion

6 References

[1] "Conventional Commits," *Conventional Commits*. [Online]. Available: https://www.conventionalcommits.org/en/v1.0.0/. [Accessed: Apr. 16, 2023]

[2] T. B. Brown *et al.*, "Language Models are Few-Shot Learners." arXiv, Jul. 22, 2020 [Online]. Available: http://arxiv.org/abs/2005.14165. [Accessed: Apr. 16, 2023]

[3] A. Vaswani *et al.*, "Attention Is All You Need." arXiv, Dec. 05, 2017 [Online]. Available: http://arxiv.org/abs/1706.03762. [Accessed: Apr. 16, 2023]

[4] M. Honnibal, I. Montani, S. Van Landeghem, and A. Boyd, "spaCy: Industrial-strength Natural Language Processing in Python." 2020 [Online]. Available: https://github.com/explosion/spaCy. [Accessed: Apr. 19, 2023]

[5] R. Flesch, "How to Write Plain English," Jul. 12, 2016. [Online]. Available: https://web.archive.org/web/20160712094308/http://www.mang.canterbury.ac.nz/writing_guide/writing/flesch.shtml. [Accessed: Apr. 19, 2023]

[6] shadcn *et al.*, "shadcn/taxonomy." Apr. 19, 2023 [Online]. Available: https://github.com/shadcn/taxonomy. [Accessed: Apr. 19, 2023]

[7] moT01 *et al.*, "freeCodeCamp/demo-projects." freeCodeCamp.org, Apr. 14, 2023 [Online]. Available: https://github.com/freeCodeCamp/demo-projects. [Accessed: Apr. 19, 2023]

[8] dcyou and chapeupreto, "dcyou/resume." Nov. 08, 2022 [Online]. Available: https://github.com/dcyou/resume

[9] serpro69 *et al.*, "serpro69/kotlin-faker." Apr. 12, 2023 [Online]. Available: https://github.com/serpro69/kotlin-faker. [Accessed: Apr. 19, 2023]