turbocommit

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

Raik Rohde (4380676)

Abstract

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 and evaluates `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 performance of AI-generated commit messages is compared with manually written ones in terms of accuracy, readability, and adherence to the specification. The evaluation methodology encompasses a mix of manual assessments and automated scoring systems, providing a comprehensive analysis of the effectiveness of AI-generated commit messages in a software development context.

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.

To demonstrate the effectiveness of `turbocommit`, this term paper presents the requirements and a comprehensive evaluation methodology. The evaluation includes a combination of manual assessments and automated scoring systems, comparing the performance of AI-generated commit messages with manually written ones using various metrics, such as accuracy, readability, and compliance with the Conventional Commit 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 communication between team members, simplify the process of generating release notes, and streamline version management.

As described in the official Conventional Commit documentation [1], 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>`.

By adhering to the Conventional Commits specification, development teams can maintain a clear and informative version control history. This is not only beneficial to 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 introduced by OpenAI in 2020 [2], 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.

The core architecture behind large language models like `gpt-3.5-turbo` is the Transformer, introduced in 2017 [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, instead 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, 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, featured in the SpaCy package [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.

Measuring the readability of the text in the body of the commit message involves using the Flesch Reading Ease Score (FRES), which originates from Kincaid's 1975 work on deriving new readability formulas [5]. 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 the following rubric.

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], resume [7], and kotlin-faker [8]. While kotlin-faker does not adhere to Conventional Commits, its commit messages maintain a consistent structure, this will be taken into account later in the paper. 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. Commits that were automatically generated (e.g. merge commits, or commits authored by bots) were filtered out and placed into a separate table.

From the three repositories using Conventional Commits, on average 101.5 non-filtered commits were collected from taxonomy and resume (99 from taxonomy, and 104 from resume). To maintain a balanced evaluation, only 0 random commits from kotlin-faker were included. This approach ensures that the overall evaluation is not disproportionately influenced by the kotlin-faker repository.

This totals to 203 commits that are evaluated in this paper.

3.2 Implementation

`turbocommit` works by analyzing the changes in a code repository, sending the relevant information along with a well-crafted system message to the `gpt-3.5-turbo` API, which then returns a defined um number of suggested commit messages. It is important to note that the `gpt-3.5-turbo` language model has a token limit of 4096 tokens; therefore, if a diff is too large, it must be split so that not all staged files are included in the diff.

In order to adapt `turbocommit` for this paper, several modifications were made:

1. Instead of using libgit2 to obtain the diff, the adapted version reads the diff from a diff.txt file.
2. Rather than prompting the user to choose what to do with the generated message (e.g., commit, edit and commit), the adapted version writes the message to an output.txt file for further analysis.
3. For the sake of time efficiency, only one message is generated for each commit, as opposed to the typical process where users can generate multiple messages and choose the best one.

These adaptations allow `turbocommit` to function effectively within the scope of this paper while still maintaining its core capabilities as a commit message generation tool.

3.3 Evaluation

Two Python scripts were developed to facilitate the evaluation process, with significant assistance from generative AI tools, specifically GPT-4. These tools played a crucial role in authoring the scripts, under careful guidance and supervision.

The first script generates a new table within the SQLite database and iterates through all human-authored commits. It then calculates the scores for

each commit based on the metrics outlined in the theory section of this paper.

The second script is responsible for the manual accuracy scoring process. It iterates through all partially scored commits in random order. For each commit, the script displays the git diff, followed by either the human-written or AI-generated commit message. The evaluator manually inputs a score for the commit message, and then the script reveals the other message, which is also scored.

As stated in a previous section,

In the interest of brevity, the complete details and workings of the scripts have not been included in this paper.

4 Results

The histograms in Figure 1 and Figure 2 reveal a notable difference in the distribution of overall scores between human-written and AI-generated commit messages. AI-generated messages exhibit a higher spike at the 0.9 bin, with significantly lower counts in other bins. In contrast, human-written messages display the highest spike just below the 0.9 bin, with a more equalized distribution across bins and lower values in general compared to AI-generated messages.

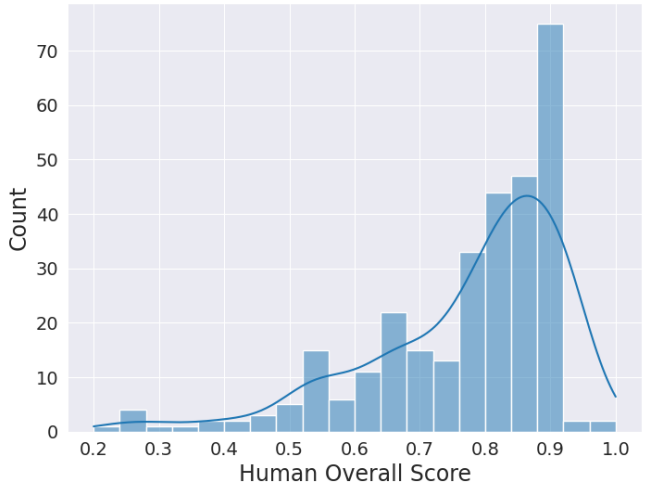

Figure 1: Histogram showing the distribution of Human overall scores

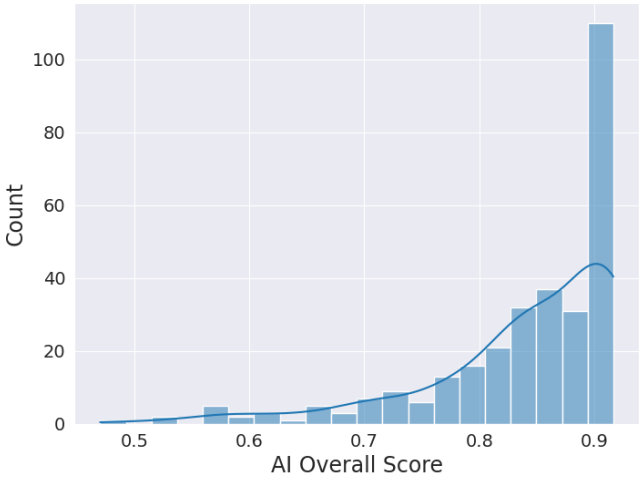

Figure 2: Histogram showing the distribution of AI overall scores

Next, we examine the box plots representing overall scores for both humans and AI. The human median score is 0.833, while the AI median score is slightly higher at 0.858. The box representing AI scores is notably smaller than the human one, indicating a narrower range of scores for AI-generated messages. The upper and lower quartiles for human scores are more spread out, showcasing greater variability in human commit messages.

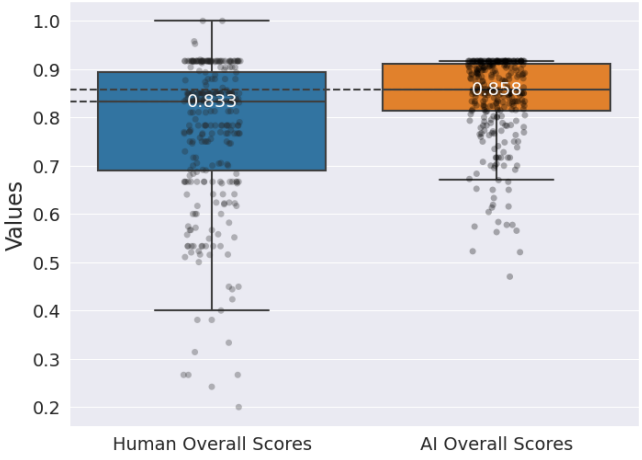

Figure 3: Box Plot showing the quartiles and median of the overall scores, excluding outliers

These results suggest that AI-generated commit messages exhibit a more consistent performance in terms of overall scores, while human-written messages show a wider range of outcomes. This finding supports the notion that AI can streamline the software development process by providing more consistent and high-quality commit messages. Additionally, time savings and ease of use benefits offered by AI-powered tools further

contribute to enhancing the efficiency of the software development process, emphasizing the potential of AI in this domain.

To further understand these results, we analyze the individual evaluation metrics. Figure 4 shows that AI-generated messages adhere to the specification 100% of the time, whereas human-written messages fail to adhere to it in 17.43% of cases. This highlights the AI's ability to strictly follow the conventional commit specification, contributing to its higher overall scores.

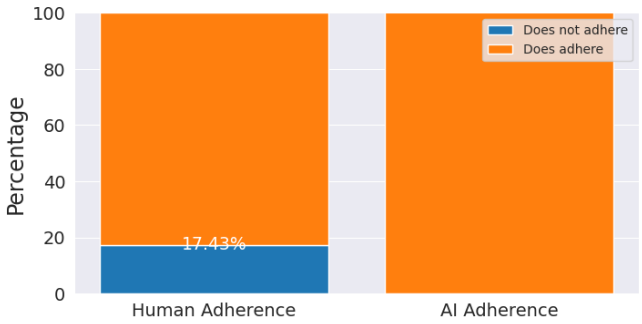

Figure 4: Stacked Bar Graph showing the adherence to the specification

The histogram for accuracy scores, in Figure 5, reveals interesting patterns. While both human and AI-generated messages have the highest counts at a score of 1.0, the difference between the AI 1.0 bin and the AI 0.8 bin is much larger than the difference between the human 1.0 and human 0.8 bins. This indicates that AI-generated messages more frequently achieve perfect accuracy scores compared to human-written ones. However, AI-generated messages also display a higher concentration of lower scores (0, 0.2, and 0.4) than human-written messages. No human-generated messages received a 0.0 score. A possible explanation for lower accuracy scores in AI-generated messages is that they often misinterpret changes in README or changelog files as actual code changes. This issue could potentially be mitigated by providing the AI with more explicit information about the changes, such as specifying that a change only involves documentation.

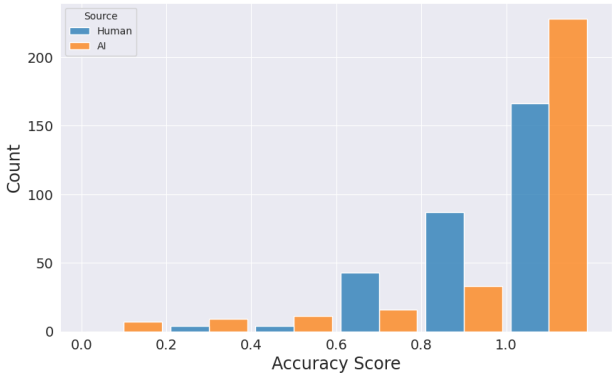

Figure 5: Histogram if Accuracy Scores

Lastly, we examine the histograms for readability scores. Both human and AI-generated messages have peaks in the distribution at 0.7. However, the human score distribution is flatter, with a wider range of scores from 0.0 to 1.0. The AI-generated messages show a narrower distribution, with no extreme values. This suggests that while AI-generated messages have consistent readability scores, they do not significantly impact the overall evaluation scores compared to human-written messages.

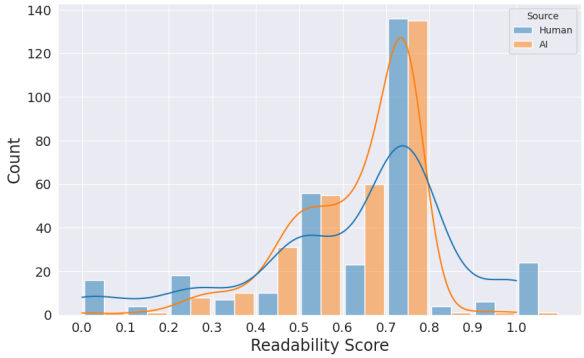

Figure 6: Histogram of Readability Scores

5 Discussion

Limitations

One limitation is the inability of the AI to automatically reference issues or pull requests, with the current context provided. This shortcoming can be addressed by using `turbocommit`'s feature that allows users to edit the generated message before committing, ensuring that the necessary references are included. Another limitation is that changes in binary data are not detectable by the AI, as only the filename is provided for those changes. Consequently, the AI might struggle to accurately describe changes related to binary files, which could impact the accuracy of commit messages in certain situations.

Potential Bias

It is important to acknowledge that personal bias may have influenced the manual accuracy scoring process, despite efforts to maintain objectivity. As the evaluator of the generated commit messages, there is a possibility that familiarity with the AI-generated messages or preconceived expectations about their quality could have affected the assessment. To minimize the impact of such bias, future studies could involve multiple evaluators, and the results could be averaged to obtain a more impartial assessment of the commit messages' accuracy.

Sample Size and Diversity

The evaluation in this study was based on approximately 300 commits from three repositories using two different programming languages. While these repositories provided a diverse set of commits for analysis, the sample size may not be large enough to generalize the findings to all types of repositories or programming languages. The limited sample size and diversity in this study are primarily due to time constraints. Future research could expand the scope of the analysis by including a larger number of commits from repositories with varied programming languages and project types, to better understand the performance of AI-generated commit messages in different contexts.

AI Model

The current study employed the `gpt-3.5-turbo` language model for generating commit messages. It is worth noting that GPT-4, a more advanced and powerful language model, is available and could potentially yield better results. However, due to the higher costs associated with using GPT-4, it was not utilized in this research. As the field of AI continues to advance, it will be important to assess the benefits and limitations of newer models, like GPT-4, in applications such as generating commit messages, to determine whether the additional costs are justified by improvements in performance.

Bibliography

[1] "Conventional Commits." (https://www.conventionalcommits.org/en/v1.0.0/)

[2] T. B. Brown, B. Mann, et al., "Language Models are Few-Shot Learners," arXiv, 2020.

[3] A. Vaswani, N. Shazeer, et al., "Attention Is All You Need," arXiv, 2017.

[4] "Linguistic Features - spaCy Usage Documentation." (https://spacy.io/usage/linguistic-features#pos-tagging)

[5] J. P. Kincaid, R. P. Fishburne, R. L. Roger, and B. S. Chissom, "Technical reports," in *Derivation New Readability Formulas (Automated Readability Index, Fog Count Flesch Reading Ease Formula) Navy Enlisted Personnel*, Feb. 1975.

[6] shadcn, marmorse, et al., "Shadcn/taxonomy," 2023.

[7] dcyou, and chapeupreto, "Dcyou/resume," 2022.

[8] serpro69, Grisu118, et al., "Serpro69/kotlin-faker," 2023.