Automatic Generation of Git Commit messages from Changes in Code

Nazia Tasnim
nimzia@bu.edu

Naima Abrar
anaima@bu.edu

Abstract

In the realm of software development, the quality and clarity of commit messages play a crucial role in maintaining a well-organized and understandable codebase. Automation of this aspect could significantly enhance the efficiency and reliability of software development processes. Over the years, various approaches utilizing generation or retrieval techniques have been proposed to automatically generate commit messages. In this work, we explore several such models of varying architectural complexities. We perform a benchmarking of simple to modern heavier SOTA models and compare their performances using popular text similarity metrics. The summary of our code and generated output is available on github [1]

1 Introduction

Professional software developers spend a lot of time reading and maintaining legacy codes. Sometimes, such maintenance may takeup to 80% of the total development lifecycle. Source code modifications are often documented in the commit messages. These short snippets can function as key component to tracking software changes, by allowing the developers to quickly locate, validate and triage the changes. If they are well written, it is also possible to synthesize human-readable product releases from the commit messages. Such application would effectively reduce the time spent on writing tedious documentations for each version/patch release. Since commit messages are so important, it is also important that they are well-written and follows certain standard. A good commit message allows developers to visualize the changes in the project at-a-glance. However, developers often write commit messages in a hurry, and they are not always well-written. This is especially true for open source projects, where - if strict PR guidelines are not enforced - the commit messages are often written in substantially different styles - rendering them useless for automatic processing. This trend may also be prevented by employing an automatic commit message generation system, which can generate succinct and informative commit messages following a certain convention adopted by the project.

In this work, we aim to explore this particular subarea of text generation. Using publicly available large datasets, we have explored some of the architectures discussed in our class, i.e: KNN-clustering, Seq2Seq models, transfer learning and current state-of-arts generative model. We also performed some basic analysis on the dataset, as well the commit messages generated by our models. We hope that this work can serve as a summary of the evolution of commit message generation, and provide some insights on the current state-of-the-art models.

2 Related Works

Modeling the relationship between source code and natural language has been studied from different application perspectives (Murali et al., 2018; Iyer et al., 2016; Balog et al., 2017; Allamanis et al., 2015; Rabinovich et al., 2017; Yin and Neubig, 2017) from automatic code summarization to code generation. In this section, we will discuss some of the related works that are relevant to our project.

Many of the earlier works in this field utilize a compilation of expert rules (Cortes-Coy et al., 2014; Buse and Weimer, 2010). However, such hand-crafted rules often cannot capture the key intention of code changes and may be longer than necessary. In later works, (Liu et al., 2018) proposed a simple retrieval-based approach to com-

[1]Project-CS505

mit generation that employs nearest-neighbor algorithm of the bag-of-words representation of the code changes, to obtain the most similar commit message from the training set. Similar syntax and semantics similarity dependant approaches have also been explored by other researchers (Huang et al., 2020). These approaches, while substantially robust, are not very effective in capturing the exact current changes in the code and depends on the quality of the training data.

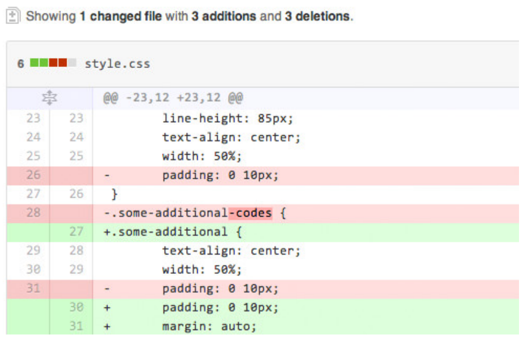
With the advent of deep learning, it became popular to frame the problem as a Neural Machince Translation (NMT) task (Jiang et al., 2017; Loyola et al., 2017; Jiang, 2019), which allowed the researchers to utilize the powerful Seq2Seq models that could process very long source sequences. Even more sphisticated models have been proposed, that attempt to jointly learn the hierarchical AST structure of code as well the semantics of the commit messages (Xu et al., 2019; Liu et al., 2020). Until very recently the state-of-the-art models were based on the Transformer architecture (Vaswani et al., 2023), coupled with transfer learning. Especially, codeBERT (Feng et al., 2020), which is a pre-trained model based on the Transformer architecture, has been shown to have good performance on a variety of code-related tasks. Finally, the most recent work in this field has explored the application of openAI's GPT (OpenAI et al., 2023) models with great success. With the availability of such powerful models, it is possible to generate commit messages that are very similar to the convention-abiding quality human-written messages. And likewise, many integration tools [2] [3]have been developed to automatically generate commit messages from the code changes. placeins

3 Methodology

In this section we describe the resources and techniques we have used for this project. In the first part, we describe the dataset we used and the second part discusses our choice of models.

3.1 Dataset

There are multiple different datasets available for the code to natural language generation task. For our work, we are particularly interested in git commits. Whenever a developer makes a change in a git repository, Git generates a *"commit"* for that change and enables the developer to provide a descriptive *"commit message"*. A Git commit consists of both the actual change and its associated commit message. The change itself can be portrayed by a "diff," illustrating the distinctions between two versions of a program. It is structured in a specific format. Lines that begin with $-$ indicate lines that were present in the original file but are not present in the modified file. Lines that begin with $+$ indicate lines that are present in the modified file but were not in the original file. An illustrated example of git diff can be seen on figure 3. we have used the dataset proposed by (Liu et



Showing **1 changed file** with **3 additions** and **3 deletions**.

```
6 style.css
@@ -23,12 +23,12 @@
          line-height: 85px;
          text-align: center;
          width: 50%;
-         padding: 0 10px;
 }
-.some-additional-codes {
+.some-additional {
          text-align: center;
          width: 50%;
-         padding: 0 10px;
+         padding: 0 10px;
+         margin: auto;
```

Figure 1: Sample Git Diff

al., 2018) which is generated by cleaning the noise and introducing quality assurance protocols into another large dataset named CmtGen (Jiang and McMillan, 2017). The nngen dataset, hosted on GitHub [4], is a specialized dataset for training neural network models to generate commit messages from code diffs. The dataset consists of paired files containing code diffs and their corresponding human-written commit messages.

The collection and filtration process of this dataset can be summarized as follows:

- The dataset is collected from the top 1K Java repositories from github
- Branch merging, rollback commits and very large diffs were removed from the raw commit dataset
- To remove noisy and low-information commits, commit ids, issue ids, messages longer than 30 characters were removed.
- A Verb - Direct message filtration scheme was used to commits with conventional pat-

[2]CommitGit,
[3]auto-commit

[4]`https://github.com/Tbabm/nngen/tree/master`

terns, e.g: "[delete] : redundant text normalization function"

- The final dataset contains $26,208$, $3,000$, and $3,000$ commits for train-validation-test splits, respectively.

While larger and more updated dataset contituting instances from multiple programming languages and other metadata are available, given the time and resource restriction we decided that using the NNGen dataset is sufficiently standard for our project.

3.2 Exploratory Data Analysis (EDA)

As part of our EDA, we performed a thorough examination of the commit messages within the nngen dataset. A summary of the data distribution across the splits is available in table 1 The following visualizations (fig. 2, 3, 4 and 5 were generated to better understand the characteristics of the data. A more comprehensive analysis is provided in the 'EDA.ipynb' notebook available the git repo.

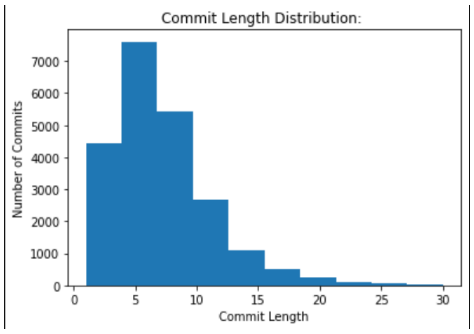


Figure 2: Histogram of Commit Message Lengths

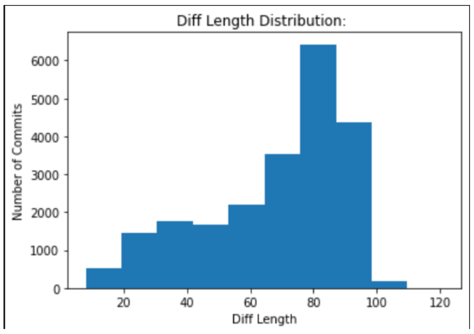


Figure 3: Histogram of Git Diff Lengths

In Figure 5, a word cloud displays the frequency of words appearing in commit messages. Larger fonts indicate higher frequency. This visualization highlights the most prominent terms such as 'fix',

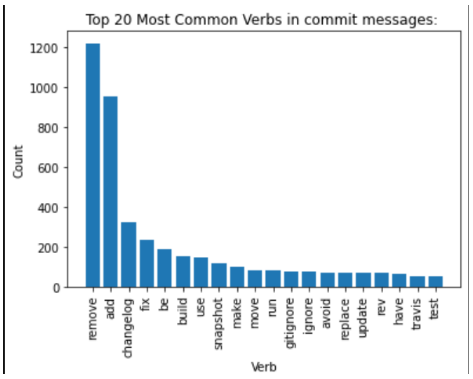


Figure 4: Frequency Plot of Most common verbs

'update', 'add', and 'remove', which are common in software development vernacular.

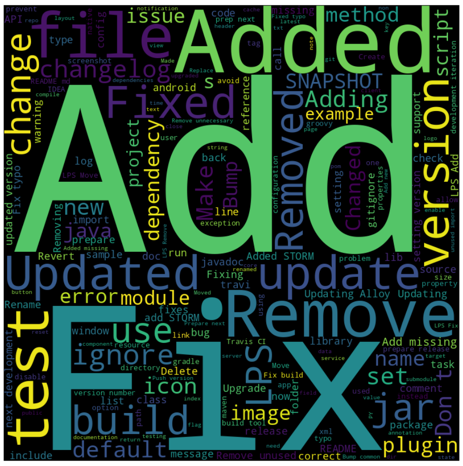


Figure 5: Word Cloud of Commit Messages

3.3 Models

We have primarily used **four** types of models for our analysis. In this section, we will briefly describe the underlying architecture of each of these models.

1. KNN-Clustering For our baseline we have used a simple clustering-based algorithm, adapted from NNGen (Liu et al., 2018). The algorithm is very simple and can be summarized in the following steps:

- Preprocess the dataset to obtain the bag-of-words representation of the code changes and the commit messages
- The similarity between the representations of training and test diffs are computed using cosine similarity
- The top-k most similar commit messages are retrieved from the training set

Table 1: Data Distribution across NNGen split

	Mean Length	**Max Length**	**Min. Length**
Commit Message	Train/Valid/Test: 6.94/6.94/6.92	Train/Valid/Test: 30/29/29	Train/Valid/Test: 1/2/1
Git Diff	Train/Valid/Test: 68.22/68.25/67.22	Train/Valid/Test: 121/112/108	Train/Valid/Test: 8/8/8

- BLEU-4 score between the new diff and each of the top-k training diffs are computed. Training diff with the highest BLEU-4 score is regarded as the nearest neighbor of the new diff
- Finally, the approach simply outputs the reference message of the nearest neighbor as the final result

2. Seq2Seq The Seq2Seq model is a very popular architecture for NMT tasks. It consists of two parts: an encoder and a decoder. The encoder encodes the input sequence into a fixed-length vector, which is then fed to the decoder to generate the output sequence. The encoder and decoder are typically implemented using RNNs or LSTMs. In our project, we have used a simple Seq2Seq model (See figure 6) with LSTM cells and attention layers.

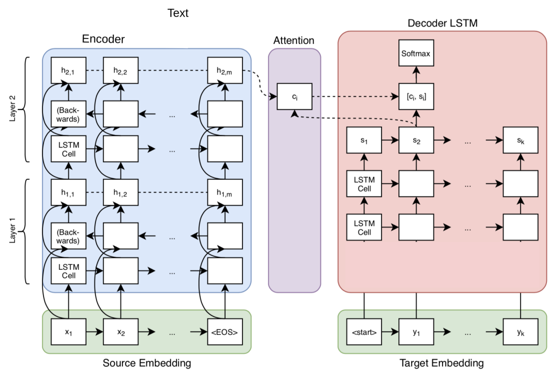


Figure 6: Standard Seq2Seq Model

3. CodeBERT This approach is a demonstration of the power of transfer learning. The CodeBERT model is a pre-trained model based on the Transformer architecture. It is trained on a large corpus of code across different programming languages. We use this model as the initial weight for our encoder. The decoder is a simple LSTM layer with multiple attention heads. This architecture helps to reduce the gap between the contextual representation of the code and the commit message.

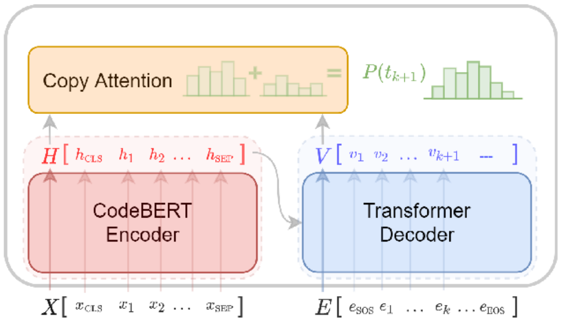

Copy Attention
$P(t_{k+1})$
$H[\ h_{CLS}\ h_1\ h_2\ \dots\ h_{SEP}\ ]$
$V[\ v_1\ v_2\ \dots\ v_{k+1}\ -\ ]$
CodeBERT Encoder
Transformer Decoder
$X[\ x_{CLS}\ x_1\ x_2\ \dots\ x_{SEP}\ ]$
$E[\ e_{SOS}\ e_1\ \dots\ e_k\ \dots e_{EOS}\ ]$

Figure 7: Transfer Learning with CodeBERT

4. Zero-shot Inference Zero-shot inference is a very popular technique used in transfer learning. It allows us to use a pre-trained model for a task that it was not trained for. In our case, we use an off-the-shelf t5 [5] model, specifically trained to generate commit messages from code changes. We limit the maximum generated tokens to 50 and use the default parameters for the rest of the model.

3.4 Evaluation Metrics

We adopted two popular metrics used for abstractive text summarization, and neural machine translation tasks to evaluate the generated outputs of our models.

1. BLEU-4: BLEU is a popular metric used to evaluate the quality of machine-generated text (Papineni et al., 2001). It is based on the n-gram precision of the generated text with respect to the reference text. BLEU-4 is the BLEU score computed using 4-grams. The BLEU score is computed as follows: 1. Compute the n-gram precision of the generated text with respect to the reference text. 2. Compute the brevity penalty, which is the ratio of the length of the generated text to the length of the reference text. 3. Compute the final BLEU score as the product of the n-gram precision and the brevity penalty.

[5]Parrot-t5

$$BLEU = BP \times exp(\sum_{n=1}^{N} w_n \log p_n) \quad (1)$$

2. ROUGE-L: ROUGE-L is another popular metric used to evaluate the quality of machine-generated text (Lin, 2004). It is based on the longest common subsequence (LCS) between the generated text and the reference text. The ROUGE-L score is computed as follows: 1. Compute the LCS between the generated text and the reference text. 2. Compute the precision and recall of the LCS with respect to the generated text and the reference text. 3. Compute the final ROUGE-L score as the F1 score of the precision and recall.

$$ROUGE - L = \frac{2 \times precision \times recall}{precision + recall} \quad (2)$$

3.5 Experimental Setup

We used the same training and testing splits to train and evaluate all of our models. We used 80% of the dataset for training, 10% for validation and the remaining 10% for testing. We used the same optimizer and criterion for all of our models: Adam optimizer with a learning rate of 0.001, loss function as categorical cross-entropy. All the experiments were performed on a single GPU on the SCC cluster.

4 Results and Discussion

Table 2: Performance of the experiment models

	BLEU-4	**Rouge-L**
Baseline	16.41	26.93
Seq2Seq	15	15
Transfer Learning	15.08	18.23
Zero-shot Infer.	7.5	13.67

The results of our experiment, as summarized in Table 2, indicate varied performance across different models with respect to BLEU-4 and Rouge-L metrics. The baseline model shows a strong performance in Rouge-L, suggesting a good overlap of the longest common subsequence with the reference messages. However, its BLEU-4 score indicates there is room for improvement in matching the finer-grained n-gram sequences. As the baseline operates on retrieval, we hypothesized that it can easily attain high performance on datasets containing replicated information. And indeed, our examination of the NNGen dataset and its corresponding paper revealed that the NNGen test set contains 16.02% duplicated commit messages, along with 5.16% duplicated pairs of $\langle$Diff, Message$\rangle$ from the training set.

Transfer Learning, utilizing the CodeBERT model, shows comparable performance to Seq2Seq in BLEU-4 but a significant drop in Rouge-L. Initially we assumed that the gap in contextual representation between code and natural language would be reduced through this approach. But as the roBERTa-weight we used has limited token consumption capacity, it might have drastically impacted the trained model's performance as well.

Zero-shot Inference demonstrates the potential of using pre-trained models for new tasks, though its performance is notably lower in BLEU-4. This may reflect the model's reliance on context that was not seen during its original training, impacting its ability to generate n-grams that match the reference messages. Like the transfer-learning model, this one also has a constraint on the number of input tokens. Additionally, the particular model we used was trained used repositories specifically following the specifications of *Conventional Commits* [6] while our test set is generated from raw human-commits not specifically adhering to any styles.

These findings suggest that while transfer learning and zero-shot inference are promising, they require careful tuning and adaptation to the specific task of commit message generation. At the same time, we want our commits to follow specific guidelines for being actually useful. And so it is also necessary to build and develop our datasets more carefully.

5 Limitation and Future Directions

In our work, we mainly focused on utilizing the concepts taught to us during the *CS505 - Natural Language Processing Course*. The project itself doesn't claim any technical novelty, but rather functions as a summarization of the different works done in this field. We have replicated standard architectures used for text summarization and generation across varying architectural complexities. It can be realistically assumed that, using a more standard dataset, the results would be

[6]Conventional Commits

more meaningful. In future, we would like to further experiment with more sophisticated models, develop adapter modules and integrable tools to standardize the output of this work.

Contributions

1. Nazia Tasnim

- Exploratory data analysis
- Baseline Model
- Transferlearning Model
- Zeroshot Model
- Report writing

2. Naima Abrar

- Exploratory data analysis on the dataset
- running LSTM on the dataset
- running Seq2Seq on the dataset
- adding snippets to the report

References

[Allamanis et al.2015] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015. Bimodal modelling of source code and natural language. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 2123–2132, Lille, France, 07–09 Jul. PMLR.

[Balog et al.2017] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. Deepcoder: Learning to write programs.

[Buse and Weimer2010] R. P. Buse and W. Weimer. 2010. Automatically documenting program changes. *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*.

[Cortes-Coy et al.2014] L. F. Cortes-Coy, M. Linares-Vásquez, J. Aponte, and D. Poshyvanyk. 2014. On automatically generating commit messages via summarization of source code changes. *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*.

[Feng et al.2020] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages.

[Huang et al.2020] Y. Huang, N. Jia, H. Zhou, X. Chen, Z. Zheng, and M. Tang. 2020. Learning human-written commit messages to document code changes. *Journal of Computer Science and Technology*, 35:1258–1277.

[Iyer et al.2016] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. 2016. Summarizing source code using a neural attention model. *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

[Jiang and McMillan2017] Siyuan Jiang and Collin McMillan. 2017. Towards automatic generation of short summaries of commits. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 320–323.

[Jiang et al.2017] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 135–146.

[Jiang2019] S. Jiang. 2019. Boosting neural commit message generation with code semantic analysis. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[Lin2004] Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain, July. Association for Computational Linguistics.

[Liu et al.2018] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang. 2018. Neural-machine-translation-based commit message generation: how far are we? *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*.

[Liu et al.2020] Shangqing Liu, Cuiyun Gao, Sen Chen, Lun Yiu Nie, and Yang Liu. 2020. Atom: Commit message generation based on abstract syntax tree and hybrid ranking.

[Loyola et al.2017] P. Loyola, E. Marrese-Taylor, and Y. Matsuo. 2017. A neural architecture for generating natural language descriptions from source code changes. *Proceedings of the 55th Annual Meeting of the Association For Computational Linguistics (Volume 2: Short Papers)*.

[Murali et al.2018] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. 2018. Neural sketch learning for conditional program generation.

[OpenAI et al.2023] OpenAI, :, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, and Shyamal Anadkat.. 2023. Gpt-4 technical report.

[Papineni et al.2001] K. Papineni, S. Roukos, T. J. Ward, and W. Zhu. 2001. Bleu. *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics - ACL '02*.

[Rabinovich et al.2017] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. In Regina Barzilay and Min-Yen Kan, editors, *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1139–1149, Vancouver, Canada, July. Association for Computational Linguistics.

[Vaswani et al.2023] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention is all you need.

[Xu et al.2019] S. Xu, Y. Yao, X. Feng, T. Gu, H. Tong, and J. Lü. 2019. Commit message generation for source code changes. *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*.

[Yin and Neubig2017] Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In Regina Barzilay and Min-Yen Kan, editors, *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450, Vancouver, Canada, July. Association for Computational Linguistics.