SHOULD BE REPLACED ON REQUIRED TITLE PAGE

Instruction

1. Open needed docx template (folder "title"/<your department or bach if bachelor student>.docx).

2. Put Thesis topic, supervisor's and your name in appropriate places on both English and Russian languages.

3. Put current year (last row).

4. Convert it to "title.pdf," replace the existing one in the root folder.

Contents

1 Introduction **7**

1.1 Problem statement 7

1.2 Methods 8

1.3 Work objective 9

2 Literature Review **11**

2.1 Introduction 11

2.2 Background of the Deep Learning in the NLP 12

2.3 Approaches to Commit Message Generation 12

2.3.1 Transformer based models 13

2.3.2 Graph Neural Network approach 13

2.3.3 Transformer-based architecture with the retrieval 14

2.4 Datasets for CMG 15

2.5 Evaluation metrics for CMG 17

2.6 Research gaps 18

2.6.1 Limited context window 18

2.6.2 Stalling performance on the out-of-filter data 19

2.6.3 Lack of the evaluation metrics 19

2.6.4 Lack of the research on CMG using the LLM 19

2.7 Conclusion 20

3 Methodology 21

3.1 Experiment design 21

3.2 Data retrieval 22

3.2.1 Retrieval process 22

3.2.2 Retrieved components 23

3.2.3 Structure of the data 23

3.3 Collected data analysis and filtering 24

3.4 Analysis of datasets from other works 28

3.5 Conclusion about the data 31

3.6 CodeT5+ training 32

3.7 Larger model experiments 33

3.8 CodeT5+ with retrieval components training 34

3.8.1 Retrieval approaches 34

3.8.2 My experiment with retrieval 35

3.9 Possible way to resolve limited context window problem 37

3.9.1 Efficient handling of big commits 38

3.9.2 Model architecture 39

3.10 Metrics to evaluate CMG results 39

3.10.1 BLEU score description 39

3.10.2 BERTScore description 41

4 Implementation 45

4.1 Vanilla CodeT5+ training details 45

4.1.1 Input sequences representation 45

4.1.2 Training hyper-parameters 47

4.1.3 Generation hyper-parameters 49
4.1.4 Training process 51
4.2 CodeT5+ with retrieval training results 51
4.3 CodeT5+ with file attention training results 51
4.4 Analysis of BERTScore metric 51

5 Evaluation and Discussion **52**
5.1 Models performance analysis 52
5.2 Models speed analysis 52

6 Conclusion **53**
6.1 Future work 53

Bibliography cited **54**

List of Tables

3.1 Commit Data Attributes 23
3.2 Distribution of commit messages 27
3.3 Distribution of commit messages 30
3.4 Comparison of CodeT5+ with 220M and 770M parameters . . . 34

4.1 Fine-tuning hyper-parameters choice 47
4.2 Generation hyper-parameters choice 49

List of Figures

3.1 Distribution of the number of commits in the repositories. . . . 25
3.2 Distribution length of the commits in the dataset. 25
3.3 Distribution of the number of files in commits. 26
3.4 Distribution of the number of files in commits in CommitChronicle. 29
3.5 Distribution of length of commits in CommitChronicle. 30
3.6 Number of commits in the repositories in CommitChronicle. . . 31

Abstract

My thesis aims to develop a system that automates the process of generating commit messages. The results achieved in deep learning have increased significantly over the last six years. This progress is especially noticeable in natural language processing (**NLP**). The generation of commit messages can be viewed as a neural machine translation task in NLP. It involves translating code into natural language to describe the changes made in the commit. I claim that considering the current progress in NLP, it is possible to generate descriptive commit messages from code modifications.

My work consists of a literature review, data collection and analysis, deep learning model training, evaluation, and results discussion. Regarding the model training, I experimented with models of different sizes, an additional module to solve the problem with message style adoption, and an additional module to resolve the problem of limited context. During the literature review, metrics analysis is conducted to identify the most suitable metrics for generating commit messages. I also conducted the experiment to prove that my metrics make sense and correctly identify how good the generated message is.

In summary, my thesis includes the analysis of the problem from the data, metrics and model perspective. I also conducted experiments to reproduce the results from other papers. In addition, I proposed an idea to mitigate the problem of handling bit commits. At the end of my work, I evaluated all the models trained and made a conclusion about the current state of the commit messages generation.

Chapter 1

Introduction

1.1 Problem statement

In software development, commit is an operation that captures the current state of a project's files. It creates a snapshot of code modifications in the project, allowing developers to record their progress. Commits enable the tracking of specific changes, rolling back to previous states, and collaborating within a team of developers. A commit message is a brief description attached to each commit explaining the changes made. This message is critical for understanding the purpose of the changes. Most of the time, manual writing of commit messages is a tedious task and may take up to several minutes of developer time.

The task of automatically generating commit messages is the perspective research topic. The writing of concise commit messages is a crucial task in the software development process. They serve as a valuable tool for developers to track changes in the project and to collaborate effectively. Despite this, a significant proportion of commit messages in open source are under-informative. State-of-the-art approaches to commit message generation (CMG) show promis-

ing results. However, most current research proposes methods that involve training encoder-decoder transformer-based models for this task. And one of the limitations of current CMG methods is the inability to handle the full context of code changes for the big commits, due to the limited context window of the model.

1.2 Methods

Analyzing the literature on the topic of automatic commit message generation, I concluded that most modern deep learning methods can be divided into two main groups: (I) Those that utilize the structured format of code dependencies and use graph neural networks to obtain an embedded representation of code changes. (II) Approaches that consider the generation of commit messages as a neural machine translation seq2seq task, take code changes as unstructured text, and solve the task with a causal language modeling objective.

Another possible but less explored method for this task is to use a large language model, such as ChatGPT4 [1] or LLaMa [2], that is tuned for general-purpose instructions. These models have gained popularity fairly recently. Therefore, research in the field of generating commit messages using this method has not yet been sufficiently studied. The disadvantage of this approach is the excessive number of parameters in the models, leading to slow inference and training times that require more computational resources. Nevertheless, even without specific fine-tuning general purpose large language models show promising results in the commit messages generation. I want to clarify that I will not use large language models in my work. I assume that automating the process of writing commit messages should save the time of developers. Therefore, it should be done quickly and without requiring a huge amount of resources. Large language models mostly

achieve their performance due to the scaling in terms of parameters and therefore do not fit into these conditions.

1.3 Work objective

The primary focus of my work is on experiments with transformer-based encoder-decoder models training for the task of generating commit messages. It includes four different approaches to solving this task with causal language modelling. The first experiment is a straightforward training of the base model to obtain the baseline. Then, experiments are provided with the model scaling in terms of parameters. With this, I investigated how it will affect the task in terms of the quality of the generated commit message, the time efficiency, and the required memory.

In the next experiment, I attempted to mitigate the common problem with the generation of commit messages, style adoption. In the literature, this issue is typically resolved with the help of retrieval methods. Therefore, I modified the baseline to insert the retrieval mechanism in it. This modification enables the model to adjust generated messages to the style of commit messages in the repository. My retrieval method also leverages the model's ability to generate descriptive messages by getting the human written message from the commit with a similar semantic obtained from the database.

In the further experiment, I tried to mitigate the well-known problem of long context processing. Commits can impact multiple files, each of which may contain a significant amount of code. Due to this limited context window of the transformer model, it cannot always process the entire commit. One of the possible way to deal with this problem is to consider each file from the commit separately.

In my method I implemented additional module between encoder and decoder parts of the baseline transformer model. First, each file goes through the encoder separately. After processing all the files for the commit, a weighted average between them is taken to get the final hidden representation for the decoder, as in the standard encoder-decoder model. This scores for the averaging are trainable parameters and represents the importance of the file in the commit.

Another part of my work, separated from the deep learning models training includes data collection and preparation and metrics analysis. From the data perspective my work includes the collecting my own dataset of commits from the GitHub and the analysis of data from other works. My process of data selection also includes comparison of the dataset and collection of statistics to came to the final decision about the data that will be used during the training process.
And regarding the metrics, In my work I analysed most widely used metrics from the literature and the idea behind them. Additionally I provided some analysis to statistically prove that the metrics I choose correlate with the quality of the generated commit message.

Chapter 2

Literature Review

2.1 Introduction

The main focus of this chapter is a comprehensive literature review of the **Commit Message Generation** (CMG) task. The chapter is meticulously structured as follows: Section 2 furnishes an in-depth exploration of the foundation of Deep Learning in the realm of **Natural Language Processing** (NLP). Section 3 delineates the list of approaches for the CMG task. Section 4 provides a comprehensive overview of the various datasets created and employed for CMG research. Section 5 offers a thorough examination of the prevailing evaluation metrics used in the assessment of CMG models. Section 6 identifies and elaborates on the significant gaps and deficiencies present within the current research. Section 7 brings this chapter to a conclusion, summarizing the key insights and findings presented within the chapter.

2.2 Background of the Deep Learning in the NLP

Deep learning techniques in NLP have brought remarkable progress and widespread adoption. Deep learning models, particularly those based on transformer architectures [3], have revolutionized the field since 2018. Models like **BERT** (Bidirectional Encoder Representations from Transformers) [4], **GPT** (Generative Pre-trained Transformer) [5], and their derivatives have set new standards for NLP tasks, achieving state-of-the-art results for most of the NLP tasks. Transfer learning is a dominant paradigm, where pre-trained models are fine-tuned for specific tasks, reducing the need for large labeled datasets. These models have made NLP more accessible, enabling researchers and practitioners to build language understanding systems with relatively modest computational resources. Despite these advancements, several challenges remain: mitigating biases, improving the model's understanding of context and semantics, and hallucinations in the model's answers. Nevertheless, deep learning in NLP has revolutionized this research area, offering unprecedented capabilities for processing and generating natural language.

2.3 Approaches to Commit Message Generation

Almost all modern commit message generation solutions use **Language modeling** to get the result.

In this approach, CMG is represented as a code2text **Neural Machine Translation** (NMT) task. In this setting, the input and output of the model are considered as a sequence of tokens. Input tokens are represented as $X = (x_1, x_2, \ldots x_n)$, which is information about the certain commit. Output tokens are the generated commit

message $Y = (y_1, y_2, \ldots, y_m)$. Language Modeling model is training to learn a distribution of conditional probabilities of the Commit Message tokens, given the information about the commit $P(y_1, \ldots y_m \mid x_1, \ldots x_n)$. Modern CMG solutions mostly use encoder-decoder neural network architecture [6]–[9]. In this approach, the encoder part transforms input X into a hidden representation $h \in \mathbb{R}^d$, and the decoder then, using this h generates an output Y. At this point, the structure of the model may vary. The approach might based on the pure transformer model [8], [9] or **Graph Neural Network** (GNN) combined with the transformer model [10]. Some approaches combine the transformer model with the retrieval mechanism [11], and this helps to increase the performance of the method.

2.3.1 Transformer based models

In this approach, code differences from the commit are treated as a sequence of tokens, and the models excel at capturing the nuances within token sequences. This information is crucial for understanding the specific changes made in the code and their impact on the overall software project. The self-attention mechanism in transformers allows the model to weigh the importance of each token in the context of the entire code snippet, enabling it to identify added or deleted lines, modified functions, and other code alterations. Additionally, transformers can efficiently handle the structure of code changes. This model is capable of ensuring that the generated commit messages are not only accurate but also contextually relevant.

2.3.2 Graph Neural Network approach

The GNN approach in CMG represents an innovative and effective way to tackle the task. Unlike the traditional sequence-based methods, GNNs work with

the **Abstract Syntax Tree** (AST) of the code changes. This representation of the relationships helps the model to capture the code's structural and semantic relationships, often defined as a graph. GNNs excel in learning from these graph-structured representations, allowing them to understand how code changes affect each other in a complex codebase. The GNN model can propagate information through the graph, aggregating context from neighboring code snippets, and use this rich context to generate more contextually aware commit messages. In summary, the GNN approach in CMG leverages graph-based representations to enhance the quality and relevance of commit messages, providing developers with a deeper understanding of code changes within the broader context of a software project.

2.3.3 Transformer-based architecture with the retrieval

The transformer-based architecture with retrieval mechanisms is a sophisticated approach that enhances CMG by combining the strengths of the transformer model with retrieval techniques. In this context, the retrieval mechanism allows the model to access and incorporate relevant information from a database of previous commits and corresponding commit messages. This approach is particularly valuable because it addresses the challenge of generating commit messages that are not only informative but also consistent with past practices and project-specific terminology. The retrieval mechanism can be used to identify and incorporate snippets of text or phrases from historical commit messages that are relevant to the current code changes. This helps in producing commit messages that maintain consistency in terminology and style, which is essential for codebase documentation and understanding. The transformer-based architecture,

with its ability to model code-related information effectively, is well-suited for this task. It can combine the retrieved information with its understanding of the code changes to generate commit messages that are both contextually rich and in line with the development team's conventions. In summary, the combination of a transformer-based architecture with retrieval mechanisms in CMG results in commit messages that are not only accurate and informative but also consistent with the project's history and established practices, contributing to more effective code documentation and collaboration.

2.4 Datasets for CMG

Multiple datasets are available for the generation of commit messages. These datasets can be categorized into two distinct groups: the first category comprises datasets that include pairs of code differences created in the commit and their corresponding commit messages, and the second category encompasses datasets that contain supplementary information related to the commit, such as repository name, commit timestamp, SHA (Secure Hash Algorithm) for unique identification of the specific commit, and other relevant metadata.

- The **CommitBERT$_{\textbf{data}}$** dataset, presented in [9], encompasses a collection of 345,000 code modification instances paired with corresponding commit messages. These instances come from 52,000 repositories representing six distinct programming languages (Python, PHP, Go, Java, JavaScript, and Ruby), parsed from Github. It is worth noticing that this dataset exclusively contains information about the altered code segments and does not include any supplementary details regarding the associated commit actions. Unlike other datasets, CommitBERT$_{\text{data}}$ is specifically tailored

to focus only on modified lines within a commit, potentially resulting in the omission of crucial contextual information that may be present in the unaltered sections of the code. Other strict filterings on the data: limitation on the number of files in the code changes; usage of only the first line of the original commit message as a prediction target; collecting the only commit messages, beginning from the verb.

- **$\text{ATOM}_{\text{data}}$** - Liu *et al.* in [7] meticulously curated a dataset obtained from a selection of 56 Java projects, with project inclusion determined based on the number of stars in the corresponding GitHub repositories. Following the exclusion of commits with ambiguous or irrelevant message content and those devoid of substantive source code modifications, the resulting $\text{ATOM}_{\text{data}}$ corpus encompasses 197,968 commits. This dataset comprises the raw commit records and includes information about the extracted functions influenced by each commit. Beyond the code-related data and the corresponding target commit messages, the dataset incorporates essential metadata such as repository names, commit SHA, and commit timestamps.

- Multi-programming-language Commit Message Dataset (**MCMD**) presented at [12] tries to mitigate the issues from the previous datasets, including (a) only the Java language; (b) small scale of 20,000–100,000 commits; (c) limited information about each commit (hence, no way to trace back to the original commit on GitHub). MCMD collects the data from the five programming languages (Java, C#, C++, Python, JavaScript), and for each language, the dataset contains commits before 2021 from the most starred projects on GitHub. The total size of the dataset after the collection, filtering, and balancing of the data is 450,000 commits for each language.

This dataset also contains additional info about the commit.

- **CommitChronnicle** is the dataset presented in the [6]. The authors of this research paper posit that datasets for CMG face not only the issues outlined [12], but also significant alterations to the original commit history and stringent data filtering. In the CommitChronnicle dataset, the majority of the filters previously applied in prior datasets are eliminated, resulting in increased data diversity and enhanced dataset generality. Notably, CommitChronnicle encompasses 10.7 million commits after the application of various filtration steps. Additionally, this dataset incorporates supplementary information about commits, facilitating the tracking of commit history for specific users and projects during the training process. Leveraging this supplementary information, it becomes possible to generate commit messages that exhibit greater alignment with the project's historical context. In conclusion, CommitChronnicle stands out as the most comprehensive and diverse dataset available for the CMG task, successfully addressing the issues present in previous datasets while offering a significantly larger volume of data.

2.5 Evaluation metrics for CMG

Numerous metrics are commonly used to assess the performance of CMG methods. Foremost among these is BLEU [13], a metric widely adopted in the evaluation of machine translation models. Given that CMG can be conceptualized as a code-to-text task analogous to a neural machine translation, BLEU, and related metrics find relevance in evaluating CMG methods. B-Norm, a variant of BLEU,

has been demonstrated to align most closely with human judgments regarding the quality of commit messages, as observed by Tao *et al.* [12]. BLEU, reliant on the precision concerning shared n-grams between generated and reference sequences, has been a staple in prior commit message generation studies.
Nevertheless, BLEU presents limitations in assessing the quality of the generated text. This metric lacks sensitivity to the semantic nuances embedded within generated text and is incapable of capturing the semantic similarity between the generated output and reference text. To address these limitations, more advanced metrics rooted in neural networks have been developed. Notably, BERTScore [14], which leverages the BERT model, exhibits heightened sensitivity to the semantic nuances of generated text. Despite these advantages, BERTScore has seen relatively limited adoption in the evaluation of CMG methods.

2.6 Research gaps

Despite the remarkable progress in CMG in recent years, several challenges remain.

2.6.1 Limited context window

Modern CMG methods are limited in capturing the broader context of code changes. They can handle only small pieces of code changes, typically limited to a single file. It's a significant limitation, as code changes often span multiple files, and the context of these changes is crucial for generating accurate commit messages.

2.6.2 Stalling performance on the out-of-filter data

Even in the CommitChronnicle dataset, which is the most comprehensive and diverse dataset available for the CMG task, the performance of modern methods is stalling on the out-of-filter data. It is a significant limitation, as the out-of-filter data is the most relevant for real-world applications.

2.6.3 Lack of the evaluation metrics

Metrics used for NMT can not evaluate the generated commit messages in terms of semantics. Most of the metrics are based on the BLEU, which is not sensitive to the semantic nuances of the generated text. It is a significant limitation, as the semantic nuances are crucial for the commit messages.

2.6.4 Lack of the research on CMG using the LLM

Current SOTA for most of the NLP tasks uses the encoder-based transformer models. For example - CodeLLaMa [2] - is the current SOTA for the code generation task. However, LLM usage for the CMG task does not have enough exploration. Authors in [6] attempt to approach the CMG using the ChatGPT with a well-chosen prompt. However, this approach performs worse than the fine-tined encoder-decoder models. The results are the following since ChatGPT is not trained for this task. It is a significant limitation, as the LLMs are the current SOTA for most NLP tasks.

2.7 Conclusion

This chapter offers a comprehensive review of prior research related to the CMG task. The sections above describe the backdrop of Deep Learning within the realm of NLP, explore various approaches employed for CMG, examine the available datasets tailored for this task, scrutinize the evaluation metrics applied to assess CMG models, identify areas of deficiency in the current body of research, and provided a list of pertinent references. In the subsequent chapter, we will expound upon our proposed solution for the CMG task.

Chapter 3

Methodology

Literature Review chapter (2) defines current approaches to solve the task of Commit Messages Generation(**CMG**). It includes deep learning models used to generate messages, data sets constructed for this purpose, and metrics typically used to evaluate model performance. The objective of this chapter is to describe the steps I took to achieve the goal from the Introduction chapter (1). Section 3.1 describes the overall structure of the experiments conducted and precisely defines the goal of the work. In Section 3.2 I will describe the process of constructing a dataset of commits, including data collecting and scrubbing of data outliers.

3.1 Experiment design

The main objective of this study is to develop a system for automatically generating commit messages that rival state-of-the-art (SOTA) methods. To achieve this, I first need to construct my dataset. The next step involves analyzing these data and those from other studies, crucial for understanding the data structure

and extracting relevant statistics. This analysis also involves filtering to ensure data quality. Subsequently, my research will examine the metrics used in Neural Machine Translation (NMT) tasks, requiring an understanding of their calculation methods and underlying intuition. Finally, I will train various models, providing a detailed description of each model's architecture and the rationale for its expected success in the commit message generation (CMG) task.

3.2 Data retrieval

The first step I took in my work was to obtain the data from open sources. In the task of the automatic commit messages generation, the data to train or evaluate the models should be in the format of labelled pairs code changes and the corresponding commit message.

3.2.1 Retrieval process

To get the data, I decided to parse the most starred GitHub[1] repositories which were mostly written in Python programming language. For this purpose, I first get the names of the repositories and links for them via GitHub API. One way of getting the commits content from the repository is directly using the GitHub API, but due to the API calls limit I decided to do this in another way. With the use of the GitPython[2], which is a python library used to interact with git repositories I firstly cloned all the repositories from the retrieved list into my local machine and then fetch the information about all the commits from the .git directory.

[1] GitHub: Cloud storage for code projects.
[2] Description of the library

3.2.2 Retrieved components

The main components of the data sample are the code changes and corresponding commit messages. However, for further research, I included some additional fields in my dataset. At the end of the data parsing process, I came up with the features mentioned in the table 3.1

TABLE 3.1
Commit Data Attributes

Attribute	**Description**
Name of repository	Name of the project in which the current commit was added. Useful to associate the commit with its project.
Commit message	Label for the prediction that will be used by the model in the training phase.
Commit changes	Input data for the model.
Number of changed files in the commit	Statistics about the retrieved data.
Length of code changes in chars	Statistics about the retrieved data.
Hash of the commit	Unique identifier for the current commit to avoid duplicates.

3.2.3 Structure of the data

Before passing data to the deep learning model, we first need to preprocess it. For my task, I decided to add some special tokens to the code changes and commit messages. I used the following special tokens:

- <file_name> for the name of the file before and after the commit
- <code_del> for code lines deleted in the commit
- <code_add> for code lines added in the commit
- <commit_msg> for the commit message

These special tokens are used to separate the most important parts of the input. Tokens described above are used at the beginning of the line, and the opposite with a backslash is used to signify the end of this part. The final format of the input data from the model is the following.

<file_name> old file name </file_name>
<file_name> new file name </file_name>
<code_del> deleted code </code_del>
<code_add> added code </code_add>
<commit_msg> commit message </commit_msg>

3.3 Collected data analysis and filtering

For my dataset, I collected the data from 400 most-starred GitHub repositories. But due to the repetitions at the end, I came up with ~300 thousand commit samples from 311 different repositories with in total ~1.2 million code files changed. Fig 3.1 shows the distribution of the number of commits among the repositories I took from GitHub. It does not include some data outliers, where the number of commits is too big. From this histogram, we can conclude that in an average repository has ~3000 commits.

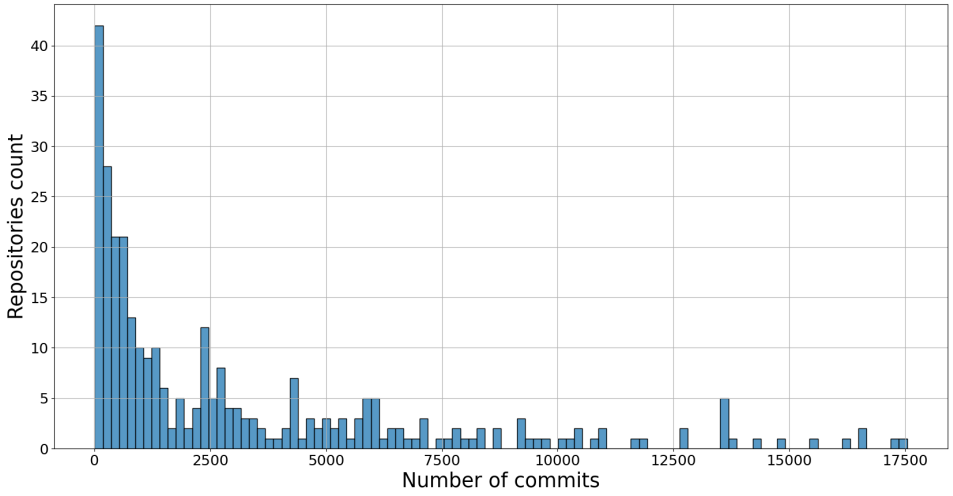

Fig. 3.1. Distribution of the number of commits in the repositories.

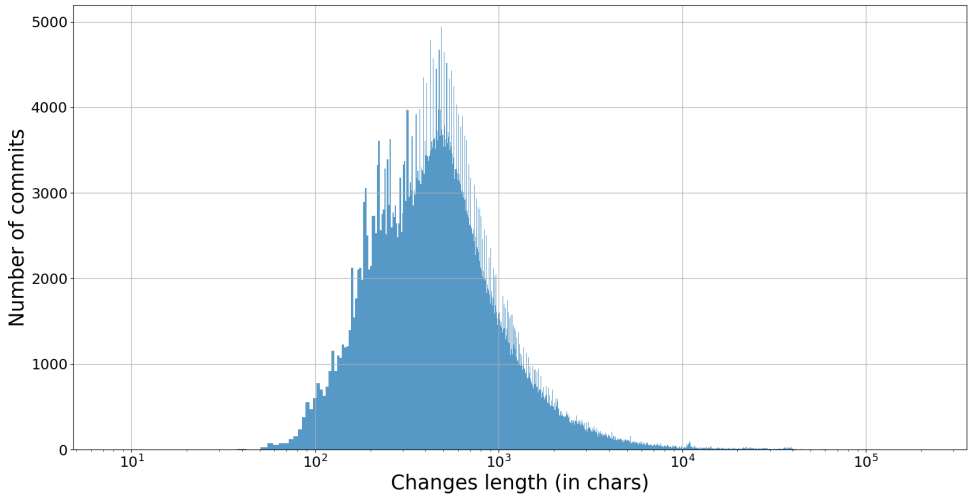

Fig. 3.2. Distribution length of the commits in the dataset.

Fig 3.2 represents the distribution of the code changes in commits. From this distribution we can see that the average length of code changes is ~5000 characters. From this we can conclude that the code changes on average in a very

long sequence. Most of the modern Deep Learning models are transformer-based and therefore experience significant degradation of the performance due to the quadratic complexity of the self-attention mechanism as mentioned in [15]. In this step, I decided to trim my dataset to have only samples with less than 6000 characters. With this truncation, we lose only 15% of the data, but we have much shorter sequences on average.

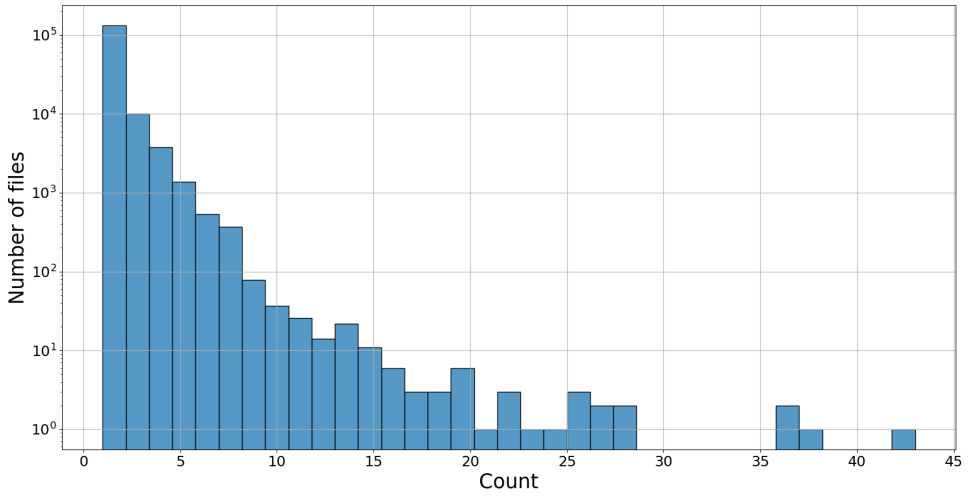

Fig. 3.3. Distribution of the number of files in commits.

Figure 3.3 shows how commits in my dataset are distributed based on the number of files they modify. From this histogram, we can see what significant number of commits change more than one file. To be precise, single file commits are two-thirds of all data samples.

One more insight that we can get from the collected data is the most popular beginning word of commit messages. This part of the data analysis is similar to the one from [9]. The authors of this work state that commit typically states with the common verbs. From the collected dataset I extracted the 10 most

used starting words for commits and got the results, presented in the table 3.2. From this table, it is visible that there exist some common words for starting a commit message. Even among these 10 words, there are several repetitive ones with modified writing or form. These words cover ~35% of the total number of samples in the dataset.

Commit Message	Count
Merge	25313
Fix	17891
Add	15476
Update	10473
Issue	7979
Added	7514
fix	6496
add	5600
Remove	5328
Fixed	4820

TABLE 3.2
Distribution of commit messages

In the data filtering step my goal was to get rid of too long samples and samples with bad quality commit messages, which may lead to the model degradation. After all, I came up with the following filtering criteria:

- Samples with too long code changes (more than 6000 characters, 85 percentile).
- Samples from repositories with too many commits
- Samples with too long commit messages (more than 950 characters, 98 percentile).
- Samples with non-ASCII symbols in the commit message.

- Samples without Python code changes.

In the end, the filtered dataset I got is made up of ~ 300 thousand commits from 170 repositories.

3.4 Analysis of datasets from other works

In the literature review chapter (2) I described most of the existing datasets for the commit messages generation task. But in this section, I would like to focus only on the CommitChronicle dataset, presented in [6]. It consists of 10.7M commits in 20 programming languages from 11.9k GitHub repositories. It also includes not only code changes with the corresponding commit message, but a lot of metadata about the commit, including the hash of the commit, commit date and time, and language which was used in the.

Regarding the filtration stage, CommitChronicle utilizes the following strategy: The authors of this dataset dropped samples out of the $[5\%, 95\%]$ percentile range of the length of code difference and samples with more than 16 files changed. They also remove commits with non-ASCII symbols in the message, merge and revert commits, and samples with trivial messages [16], which do not provide any useful information about code updates.

For this dataset, I performed the same analysis to compare it with my data. Figure 3.4 displays the distribution of files changed within a single commit. Comparing CommitChronicle with my dataset in this regard, we can observe that the dataset presented in [6] has more balanced data. The number of samples decreases uniformly with an increasing number of files, at the same time my dataset has some outliers. Fig. 3.5 represents the distribution of code changes length in characters. On this side, the statistics are similar, so we can say that

the datasets are the same from this perspective. The distribution of the number of commits among the CommitChronicle repositories is shown in Fig. 3.6. In this figure, I displayed only information for the repositories with less than 5000 commits to make the plot more representable. From this side, CommitChronicle has more balanced data, as the distribution is smoother, but it is mostly connected with the much fewer repositories in my dataset. The last thing that I would like to mention about the statistics of CommitChronicle is the most popular first words of commit messages. They are presented in Table 3.3. As was mentioned above, merge commits were filtered out by the authors, and in all other respects, the results are the same as for my dataset. Therefore, datasets are identical from this side.

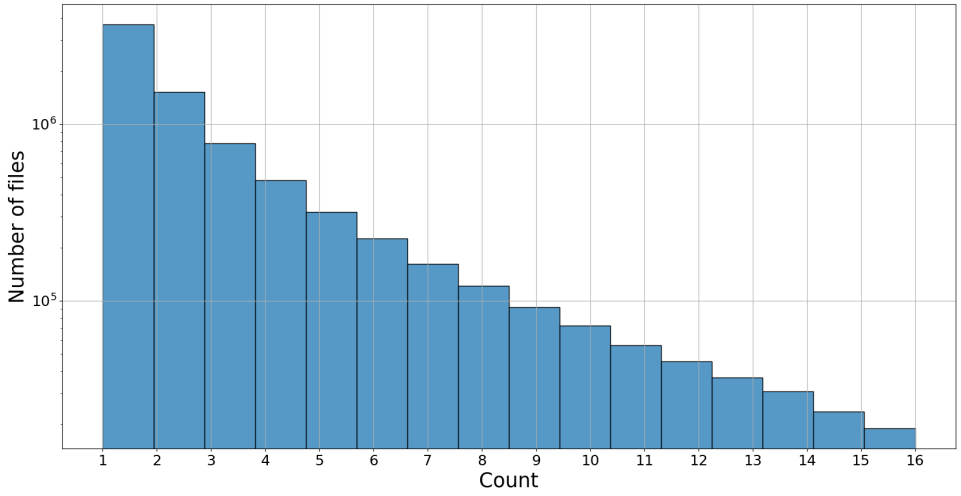

Fig. 3.4. Distribution of the number of files in commits in CommitChronicle.

TABLE 3.3
Distribution of commit messages

Commit Message	Count
Add	541205
Fix	427753
Update	251594
add	198032
fix	177001
Added	158065
Remove	157336
fix:	118704
Use	93908
Fixed	89186

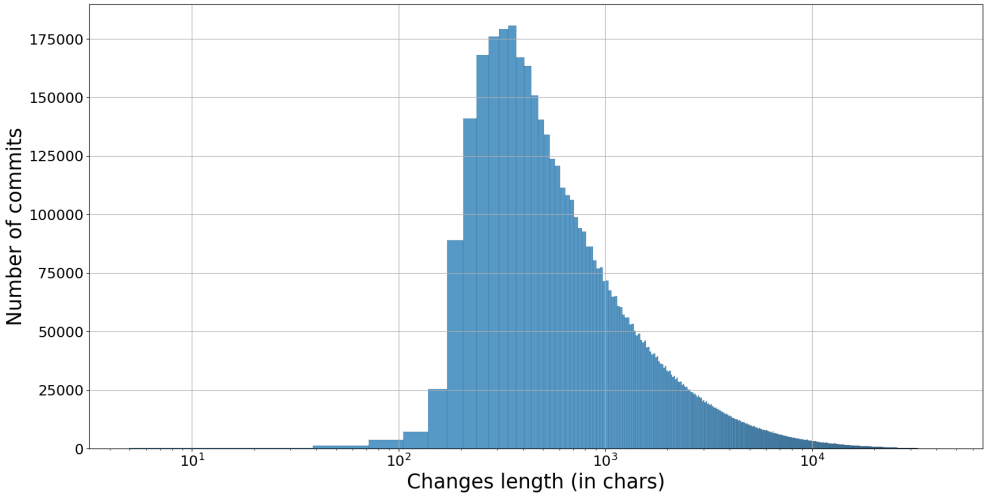

Fig. 3.5. Distribution of length of commits in CommitChronicle.

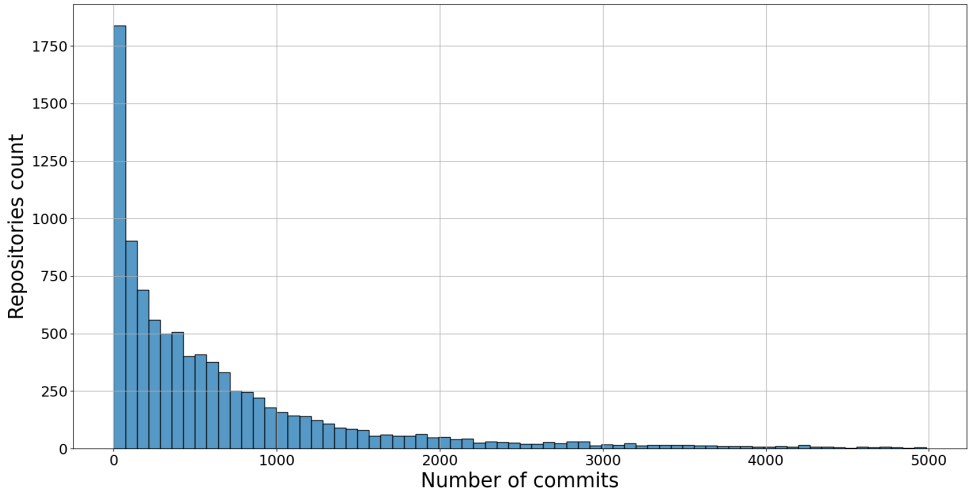

Fig. 3.6. Number of commits in the repositories in CommitChronicle.

3.5 Conclusion about the data

In this section, I would like to make a choice of the data used for the model training, evaluation, and testing. From the data analysis, I got that CommitChronicle has a better representation of the data. It also has much more samples. So, it will be logical to use CommitChronicle as a train set. As a validation set, I will use a validation split of the CommitChronicle to be sure that I have no repetitive samples with the train data. For the test of my models, I will utilize both my custom dataset and the CommitChronicle test split. The results of the models in the custom dataset may be represented as the results of out-of-domain data, as the process of getting the data differs, and some samples filtered out from CommitChronicle may be included. So, this test will examine the fair performance of the model in the ‘real world data‘.

3.6 CodeT5+ training

My first attempt at approaching the task of commit message generation is straightforward. I will consider this task as a simple NMT (Neural Machine Translation) problem. In this setting, the model aims to translate the input code into a natural language description of the code changes. Current SOTA solutions in the NMT task utilize the transformer architecture approach presented in [3]. According to the specific domain of the input data for my task, the best choice is the model pre-trained on the massive amount of code. The model used as the backbone in the current SOTA solution of the CMG task presented at [6] is codeT5. The model was pre-trained on a variety of coding tasks and demonstrated proficient performance in code comprehension. However, since that time, the authors of CodeT5 have introduced an updated version of the model, called CodeT5+ [17]. This new model outperforms CodeT5 in performance in most code-related benchmarks. So, I decided to train CodeT5+ on the CMG task as the first experiment in my work. In this setting, I will use the code difference from the commit as plain text.

As mentioned in Section 2.3, the most common method for generating commit messages involves representing both input and output as a sequence of numerical tokens. In Sect. 3.2.3, I described the specific format of the input and output data for my task. Therefore, before training the model, it is necessary to add special tokens to the tokenizer vocabulary. The model will learn the embedded representation of these special tokens during training. This step will aid the model

in better understanding the structure of code changes.

$$L(y, \hat{y}) = -\sum_{i=1}^{C} \log \left(\frac{e^{\hat{y}_i}}{\sum_{j=1}^{C} e^{\hat{y}_j}} \right) \times y_i \tag{3.1}$$

The loss function that should be optimized by the neural network is cross entropy loss represented at 3.1. y here represents the true probability distribution of the next token choice from the tokenizer vocabulary in the target sequence. This is a vector with 1 in one position and 0 in all the others, as we exactly know what token should come next. And $\hat{y}$ is the predicted vector of probability distribution to choose the next token. This function tends to make the predicted vector as similar to the true vector as possible, *i.e.* predict the next token correctly. Predicting the next token in an autoregressive manner model will generate a commit message for the given code edits.

3.7 Experiments with larger CodeT5+

To assess the impact of scaling the model in terms of parameters on the quality of generated commits, I trained the CodeT5+ model with 770 million parameters. The differences between the base version and this model are presented in Table 3.4. From this table, it is clear that the distinctions between these models are primarily in the hidden representations and the number of blocks in both the encoder and decoder parts. A larger model version does not process longer sequences but has better internal representation, potentially enabling the generation of better commit messages. The training objective I utilized and the data format are identical to those used for training the base CodeT5+. This experiment demonstrated how scaling the depth of the neural network and improving representation can affect

model performance. Additionally, this phase allowed me to determine whether 220 million parameters are sufficient for constructing commit messages that are relatively simple sequences. The final aspect requiring analysis in this section is how this scaling influences the model's inference time, since efficiency in terms of time is as crucial as message quality when considering real-world applications.

TABLE 3.4
Comparison of CodeT5+ with 220M and 770M parameters

Feature	**220M model**	**770M model**
Context window tokens	512	512
Hidden state dimension	768	1024
Encoder transformer blocks	12	24
Decoder transformer blocks	12	24
Embedding dimension	32100	32100

3.8 CodeT5+ with retrieval components training

One of the problems with the traditional language modeling approach to the task of commit messages generation is the shared style of messages within the same repository. For example, repository of Linux kernel source code have a strictly specified format of commit messages. Using only code modifications it is impossible to adapt generated commit message to have the same specified format.

3.8.1 Retrieval approaches

To mitigate issue with a message style, works like [11], [7], and [18] adopt additinal retrieval module to the standard endcoder-decoder architecture. These

methods are mostly uses the database of commits and corresponding commit message. And then new commit is given as an input to the model, this method firstly use encoder to get a embedded vector representation of code modifications. This representation is further used to find the most similar code changes from the dataset, and get both modifications and commit message. This additional information is then passed to the model to get the result. Analysis of these method shows, that retrieval mechanism leverages the results of commit generation.

One more way to use the historical data in generation of commit messages is to retrieve previous commits from the same repository, or previous commits of the same person. This method way used in the [6], and showed improvement of model performance. If previous methods are mostly used to improve the understanding of code changes, then this historical data retrieval method is used only for message style adaptation.

3.8.2 My experiment with retrieval

For my experiment, I decided on the method combining retrieving the most similar code modifications and getting historical data from the previous commits. My experiment is close to the one provided in [6] with a combination of the RACE method from [11] and the history of messages added to the model input. The main difference between my approach and the RACE is that the RACE leverage the commit generation by passing the combination of the input code changes with the most similar one from the database. At the same time, my approach is utilizing only the message from the most similar commit to adapt to the style of the commit message of the same code theme. Additionally, I pass to the model input history samples of the messages from the same repository. In the end, my input to the

model have the following format:

<commit_msg> most simmilar message </commit_msg>
<commit_msg> previous message </commit_msg>
code modifications

Generally, I construct my pipeline in the following manner. Firstly, I need to get the database from which I will retrieve the most similar commit. As a database, I used my training data. The similarity metric for my retrieval process is the cosine distance between the target embedding and embeddings from the dataset. For getting the embeddings of my data, I need the model pre-trained to understand code modifications and commit messages. For this purpose, I used a separated encoder part of the CodeT5+ fine-tuned for commit generation from my initial experiment. After the training on the large dataset of code changes, it should be able to construct meaningful embeddings. With the usage of the hidden states from the encoder, I extended my training data with the corresponding embeddings of code changes that I will further refer to as the database. One additional extension to the dataset is the previous commit message from the same repository. To obtain the message history, the training data was grouped by repositories and sorted in historical order based on commit timestamps. The model's forward pass involves running input code modifications through the encoder of the codeT5+ model to obtain embedded representations of input code changes. After calculating the embedding, I searched for the most similar one in the database and retrieved its commit message.

An important detail in the search process is to track the timestamp of a similar commit and the similarity of embeddings. During the training process, the model takes a sample from the search database. Therefore, the most similar commit will

always be its exact copy with the real commit message, as we have its copy in the search space. This behaviour will break the logic behind the method and lead our model to overfit, thus we should handle this situation and consider a threshold on the similarity for it not to be too big. One more restriction on the similar commit search is the timestamp of the retrieved sample. In the training phase model must not have access to the commits from the future. This may cause a situation of retrieving the future commit from the same repository, changing the same piece of code, and leading the model to overfitting. Therefore I restricted my search space to the most similar sample with the similarity threshold and commits made only before the input one.

With the help of the retrieved commit message and the history of the previous commits from the repository, I'm constructing the input in the format described above. These modifications will enhance the generated commit message adaptation to the repository's overall style and code change themes, thereby improving the text-matching metrics.

3.9 Possible way to resolve limited context window problem

Another problem with generating commit messages is the limited context window of the transformer-based models. As stated in [3], the self-attention mechanism - the core module of all the transformers - has quadratic time complexity. For example, a model that I used in the previous experiments - CodeT5+ can handle only sequences with up to 512 tokens in the input. This amount is insufficient to handle a large commit that includes changes of multiple files. One

possible way to mitigate this limitation is to use a larger model. For example, the mistral model presented in [19] has a maximum context length of 8192 tokens. However, the problem with this solution is that this model has 7 billion parameters, compared with 220 million in CodeT5+. CMG is a relatively simple task since generated messages are not that varied. According to my analysis of the data from 3.3 in general, most of the commit messages have almost the same structure. Thus, the inference speed of the used model should be high. And with the larger model, I will lose the performance in terms of the speed too much. That is why I decided to handle big commits more efficiently in another way.

3.9.1 Efficient handling of big commits

My idea of handling big commits consists of the following. In most cases, big commits involve changes in multiple files. Each file in the commit is relatively simple and can be processed by the codeT5+ model. Therefore, if I get the embedding of each file separately, I will have the full context of the commits after the encoder. Suppose we have a commit with modifications in 5 code files. Firstly, I separately pass each file to the encoder part of my model. Even if some of the files are too big and can not fit into the context window, I will extract the information from the beginning of the changes and get the general goal of the commit. At the same time, passing the whole commit as a text will lead to the truncation of the data. It is possible to exclude some files from the model input, which may lead to the loss of information about the commit goal. Before passing the separate embeddings of modification files to the decoder, I need to aggregate them to match the shape of the hidden encoder representation with the standard one. For this purpose, I use the weighted average of the embeddings. These

weights for each embedding represent the importance of certain file changes for the general changes made in the commit. The importance score for each file is trainable parameters. I get them through the additional module between the encoder and decoder part of the model.

3.9.2 Model architecture

TODO

3.10 Metrics to evaluate CMG results

In the literature review section, I have described the most popular metrics to measure the performance of the models in terms of generated commit message quality 2.5. For my experiments, I used bath metrics described in the literature review. BLEU for the syntax similarity and BERTScore to access the semantic similarity with the original commit message. In this chapter, I aim to give an extensive definition of these metrics and the intuition behind them.

3.10.1 BLEU score description

Bilingual evaluation understudity (**Bleu**) is the metric invented in 2002 and presented at [13]. The original goal of this metric was to evaluate the quality of machine translation. The task of generating commit messages from code modifications is considered a translation from code to natural language. The original commit message in this setup is the reference translation. A comparison of the generated commit message with the original one matches the idea of the

BLEU. Therefore, the results of this metric properly reflect the quality of the generated commit message.

Considering details of this metric, its mathematical formulation is presented in formulas 3.2-3.4. The final formula for the BLEU score shown in 3.4, is calculated as a product of brevity penalty (**BP**) and the weighted sum of n-gram precision. Bravity penalty is calculated according to the formula 3.3 and penalizes metric for too long generated text. In this formula r stands for the length of generated text and c is the length of the reference. The precision of the n-grams formula presented at 3.2 calculates the ratio matched n-grams to the total number of n-grams in the reference text for each sentence. Default parameters for BLEU score is $N = 4$ and uniformly distributed $w_n = \frac{1}{4}$

$$p_n = \frac{\sum_{\mathcal{C} \in \{\text{Candidates}\}} \sum_{\text{n-gram} \in \mathcal{C}} Count_{clip}(\text{n-gram})}{\sum_{\mathcal{C}' \in \{\text{Candidates}\}} \sum_{\text{n-gram}' \in \mathcal{C}'} Count(\text{n-gram}')} \tag{3.2}$$

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{1-r/c} & \text{if } c \leq r \end{cases} \tag{3.3}$$

$$BLEU_N = BP \times \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{3.4}$$

From the explanation above, it is clear that the formula for BLEU strongly depends on configured parameters, n-grams N, and importance weights w_n. That makes this metric variative, and therefore, it is hard to compare the results among

different works. To overcome this problem, the authors of [20] presented the package SacreBLEU. This Python script helps calculate the BLEU score in a unified way to have reproducible results.

Analyzing the BLEU score specifically for the commit messages generation task, results from [12] show that in this specific task related to code, it is more representative to use a normalized version of BLEU called B-Norm. The difference from the original method is to first convert both prediction and reference to lowercase, as it is not as important for this task as for translation. The second difference is to add one to both the numerator and denominator of 3.2 to add smoothness to it.

In my experiments, I decided to use SacreBLEU and B-Norm. This will make the evaluation results more extensive and fair. Counting SacreBLEU will help me compare my results with other approaches to solving CMG tasks. B-Norm is mostly used in my work to compare the results with the work of JetBrains Research [6], as I used their method as a baseline for my work. To make a fair comparison in terms of B-Norm, I used the script to calculate the metrics from the source code of [6], as it is open and free to use.

3.10.2 BERTScore description

BERTScore is the metric for automatic evaluation of text generation presented at [14]. This metric is based on the BERT [4] - encoder-only transformer-based model. It was shown that the BERT model achieves outstanding results in encoding text and getting meaningful hidden representation. Utilizing this fact, BERTScore gets embeddings for each token of both generated text and reference. Due to the transformer-based architecture, the representation of the tokens may differ

depending on the surrounding context. This feature makes this metric more representative than the BLEU score, which considers only matching n-grams, *i.e.*, based only on the syntax similarity between generated text and reference. At the same time, BERTScore depends on the context and meaning of the word, therefore can catch the situation, then we have generated text with the same meaning, but written in other words.

Suppose that we have reference text $x = \langle x_1, x_2 \dots x_k \rangle$ and generated candidate text $\hat{x} = \langle \hat{x_1}, \hat{x_2} \dots \hat{x_k} \rangle$ represented in tokens. The first step to calculate the BERTScore is to get embeddings of size h of each token for both texts, resulting in two embeddings matrices $H_x \in \mathbb{R}^{k \times h}$ for the reference text and $H_{\hat{x}} \in \mathbb{R}^{l \times h}$ for the generated text. Further is to calculate cosine similarity for each hidden vector from matrices. Cosine similarity between reference token x_i and candidate token $\hat{x_j}$ calculated as shown in 3.5. The result of cosine similarity lies in the interval $[-1, 1]$ and represents the cosine of the angle between these two vectors. The closer these vectors are, the closer the cosine similarity to one.

$$s = \frac{x_i^T x_j}{\|x_i\| \|\hat{x_j}\|} \tag{3.5}$$

Constructing a similarity matrix from pairwise cosine similarity of tokens, I get matrix $S \in \mathbb{R}^{k \times l}$. One more detail about this metric is the importance of weighting for each token in the reference text. For this, the authors of this method used inverse document frequency (idf) scoring. Given M reference sentences $\{x^{(i)}\}_{i=1}^{M}$ idf score for the token w calcualted according to the 3.6.

$$\text{idf}(w) = -\log \frac{1}{M} \sum_{i=1}^{M} \mathbb{I}\left[w \in x^{(i)}\right] \tag{3.6}$$

Complete BERTScore matches each token in x to a token in $\hat{x}$ to compute recall 3.7, and each token of $\hat{x}$ to a token in x to calculate precision 3.8. The authors used a greedy strategy for token matching. Each token is matched to the most similar token in the other text.

$$R_{BERT} = \frac{\sum_{x_i \in x} \text{idf}(x_i) \max_{x_j \in \hat{x}} x_i^T x_j}{\sum_{x_i \in x} \text{idf}(x_i)} \tag{3.7}$$

$$P_{BERT} = \frac{\sum_{x_j \in \hat{x}} \text{idf}(x_j) \max_{x_i \in x} x_i^T x_j}{\sum_{x_j \in \hat{x}} \text{idf}(x_j)} \tag{3.8}$$

For my experiment, I used the F1 score 3.9. It is calculated as a doubled geometric mean of precision and recall, thus including information from both these metrics.

$$F_{BERT} = 2\frac{P_{BERT} \times R_{BERT}}{P_{BERT} + R_{BERT}} \tag{3.9}$$

The last important point about the BERTScore metric is the sensitivity to the base BERT model. As was written before, the most important step in calculating this metric is to get representative embedding for each of the tokens with BERT. And

the choice for this is crucial to get an informative metric. For my experiments, I got deberta-xlarge-mnli from [21]. The reason I choose this exact model is that authors of the evaluate[1] library, have analyzed[2] the performance of most popular encoder models and states, that this version of DeBERTa shows the best performance for calculating the BERTScore.

[1] Evaluate - library with the implementation of the most modern ML metrics.
[2] Documentation for BERTScore from evaluate

Chapter 4

Implementation

4.1 Vanilla CodeT5+ training details

The theoretical aspects of training the transformer-based encoder-decoder model for the commit message generation task were described in the corresponding methodology section 3.6, and in this part of the work, I will give details of the implementation of the training process. This section includes details of the model input sequence format and the representation of the label vector to compute the loss function. Further, I will describe the hyper-parameters required for the model training and inference and the logic behind them.

4.1.1 Input sequences representation

The first step in training a neural network involves preparing and converting the data to the input format of the model. CommitChronicle dataset stores code changes in the format of JSON. It is required to convert it into plain text and insert "special" tokens in the data preprocessing stage, according to the desired

format 3.2.3. The next step in data preparation is to tokenize the text. Machine learning models cannot process data in text format because they can only work with numeric values. Initially, I needed to convert text into numbers. The standard method is to use a pre-trained tokenizer that's trained with the model. The tokenizer contains a vocabulary with the mapping of words to numbers. The model takes a sequence of numbers as input, each corresponding to a word in the dictionary.

In neural network training, data is typically passed in batches rather than as a single sequence. Batched training is utilized to accelerate GPU computations and enhance loss convergence. For this reason, the input batch should have a form of the matrix. The issue with constructing this matrix arises due to the varying lengths of input sequences. This condition makes it impossible to unify the shape of the batch matrix without additional data transformation. The common technique to resolve this issue in the sequence processing task is to add padding tokens to each sample to equalize all sequences in length. These padding tokens are excluded from the model's self-attention calculation, so expanding the sequence length does not result in additional computations. For my experience with CodeT5+, I expanded all the input sequences to the maximum size of the model context - 512 tokens. The same padding should be applied to the label vectors. The difference is that the cross-entropy loss interprets the padding as correctly guessed tokens, thus making the loss function less sensitive. To avoid this loss degradation, I excluded the padding tokens while computing it.

All the techniques described in this section are required for the efficient interpretation of the data by the model. Furthermore, this preprocessing enables the batched model training and the efficient loss computation.

4.1.2 Training hyper-parameters

The choice of hyper-parameters is the crucial step in neural network training. A model's performance in a given task is significantly impacted by a set of hyperparameters that affect the convergence of the loss function. In this section, I will describe all the hyperparameters chosen for fine-tuning the CodeT5+ model for the commit message generation task. In addition, I will define each parameter and explain how they affect the fine-tuning process.

TABLE 4.1
Fine-tuning hyper-parameters choice

Hyper-parameter	**value**
batch size	32
optimizer	AdamW
learning rate	2×10^{-5}
learning rate scheduler	linear
weight decay	10^{-2}
warm-up steps	100
warm-up strategy	linear
bf16	True

Table 4.1 represents a list of all the hyper-parameters configuration for CodeT5+ fine-tuning. The **batch size** for training and evaluation was set to 32. According to empirical findings, I get that this batch size is optimal in training time and computation resource usage. As an **optimizer** I chose an improved version of Adam Optimizer [22] that decouples weight decay from the optimization steps, applying it directly to the weights - AdamW [23]. In recent research, it has been shown that this optimizer leads to better convergence in the task of

causal language modeling. For the **learning rate** I set the maximum value to be 2×10^{-5}. The learning rate can be interpreted as a "step size" for the gradient descent method. In the general form, it corresponds to γ in model parameters update formula $w_{\text{new}} = w_{\text{old}} - \gamma \nabla f(w_{\text{old}})$. During the training, the learning rate is configured using the **linear scheduler**, which monotonically decreases the learning rate to 0 till the end of the training. One more parameter to configure the learning rate is the **linear warm-up**, which monotonically increases it to the maximum specified by the current from 0 during the **warm-up steps**. Weight decay is the hyperparameter that is responsible for neural network parameters regularization. The optimization objective then takes the form of $L_{\text{new}(w)} = L_{\text{original}}(w) + \lambda\, w^T w$, where λ is the specified strength of the weight decay penalty. Hyperparameter **bf16** indicates that instead of full precision with 32-bit float model parameters and optimizer states, I used brain float with 16 bits. Bfloat differs from the usual half-precision float in that it has 8 bits in the exponent part, while fp16 has only 5 bits. With this modification, bfloat16 can handle the same range of numbers as a full-precision float with 32 bits but with less precision than fp16, as it has fewer bits in mantissa. Recent research [24] has shown that training neural networks with bf16 achieves the same performance as full precision, but with fewer computational resources required.

All the hyper-parameters described above are used to configure the training process. The set of parameters from this section, except for batch size, is shared across all models trained in my work. This set of parameters was taken from [6], as authors of this work achieved SOTA on the commit message generation task using the same dataset.

4.1.3 Generation hyper-parameters

The output of a trained language model represents the probability distribution of the next text token. To construct an output, the model predicts all tokens in an autoregressive manner. This means that when predicting the output $\hat{Y}$, $\hat{y}_n$ depends on the previously generated context $(\hat{y}_0 \dots \hat{y}_{n-1})$. One of the possible ways to construct an output sequence is to choose the most probable token every time. But this greedy strategy may lead to the generated text degradation. There exist methods to make the generated text more human-readable and increase the cumulative probabilities of the choices. In this section, I will describe the generation hyperparameters I used for my model and explain the intuition behind them to justify my choice.

TABLE 4.2
Generation hyper-parameters choice

Hyper-parameter	**value**
max new tokens	128
top k	100
number of beams	5
early stopping	True
no repeat n-gram size	2
do sample	True
top p	0.95

Table 4.2 lists the set of hyperparameters that I use to generate commit messages. **Max new tokens** parameter configures the maximum length of the generated message. According to the training dataset statistics, the commit messages are $\sim$ 60 characters on average. To make the results similar to the actual

message I limited the maximum number of generated tokens. **Top k** parameter is responsible for the number of tokens that the model considers in the prediction. It makes the model choose only among the k most probable words. This technique prevents the model from hallucinating by randomly selecting the inappropriate token, making the generation more stable. The parameter **number of beams** refers to the beam search decoding strategy. It implies the use of heuristics to make the output more diverse. Each beam starts from the initial position and predicts the next token according to the probabilities of the model. At the end of the generation, beams are reranked according to the cumulative probability of the predicted sequences. **Early stopping** flag also corresponds to beam search. It controls that the generation process should stop after the first **num_beams** beams are done. I also prevented the model from generation of the same bi-gram several times. It is a known issue of the large language models that they may hallucinate and start repeating some phrase till the end of the response. **Do sample** parameter just states what tokens are selected according to the probabilities, and not just with a greedy strategy. **Top p** parameter corresponds to the nucleus sampling method proposed in [25]. According to this method only the smallest set of most probable tokens with probabilities that add up to **top_p** or higher are kept for generation. This technique is similar to the **top_k** and prevent the model from hallucinating and predicting irrelevant tokens.

All of the generation hyperparameters described in this section helps the model to generate more relevant text. Moreover, some of the parameters, like sampling, make the generated commit message more human readable.

4.1.4 Training process

4.2 CodeT5+ with retrieval training results

4.3 CodeT5+ with file attention training results

4.4 Analysis of BERTScore metric

Chapter 5

Evaluation and Discussion

5.1 Models performance analysis

5.2 Models speed analysis

Chapter 6

Conclusion

6.1 Future work

Bibliography cited

[1] J. Achiam, S. Adler, S. Agarwal, *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[2] B. Roziere, J. Gehring, F. Gloeckle, *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[3] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[4] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[5] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, *et al.*, "Improving language understanding by generative pre-training," 2018.

[6] A. Eliseeva, Y. Sokolov, E. Bogomolov, Y. Golubev, D. Dig, and T. Bryksin, "From commit message generation to history-aware commit message completion," *arXiv preprint arXiv:2308.07655*, 2023.

[7] S. Liu, C. Gao, S. Chen, L. Y. Nie, and Y. Liu, "Atom: Commit message generation based on abstract syntax tree and hybrid ranking," *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1800–1817, 2020.

[8] S. Liu, Y. Li, X. Xie, and Y. Liu, "Commitbart: A large pre-trained model for github commits," *arXiv preprint arXiv:2208.08100*, 2022.

[9] T.-H. Jung, "Commitbert: Commit message generation using pre-trained programming language model," *arXiv preprint arXiv:2105.14242*, 2021.

[10] J. Dong, Y. Lou, Q. Zhu, *et al.*, "Fira: Fine-grained graph-based code change representation for automated commit message generation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 970–981.

[11] E. Shi, Y. Wang, W. Tao, *et al.*, "Race: Retrieval-augmented commit message generation," *arXiv preprint arXiv:2203.02700*, 2022.

[12] W. Tao, Y. Wang, E. Shi, *et al.*, "On the evaluation of commit message generation models: An experimental study," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2021, pp. 126–136.

[13] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: A method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[14] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi, "Bertscore: Evaluating text generation with bert," *arXiv preprint arXiv:1904.09675*, 2019.

[15] F. D. Keles, P. M. Wijewardena, and C. Hegde, "On the computational complexity of self-attention," in *International Conference on Algorithmic Learning Theory*, PMLR, 2023, pp. 597–619.

[16] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, "Neural-machine-translation-based commit message generation: How far are we?" In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 373–384.

[17] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. Hoi, "Codet5+: Open code large language models for code understanding and generation," *arXiv preprint arXiv:2305.07922*, 2023.

[18] H. Wang, X. Xia, D. Lo, Q. He, X. Wang, and J. Grundy, "Context-aware retrieval-based deep commit message generation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 4, pp. 1–30, 2021.

[19] A. Q. Jiang, A. Sablayrolles, A. Mensch, *et al.*, "Mistral 7b," *arXiv preprint arXiv:2310.06825*, 2023.

[20] M. Post, "A call for clarity in reporting bleu scores," *arXiv preprint arXiv:1804.08771*, 2018.

[21] P. He, X. Liu, J. Gao, and W. Chen, "Deberta: Decoding-enhanced bert with disentangled attention," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=XPZIaotutsD.

[22] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[23] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.

[24] D. Kalamkar, D. Mudigere, N. Mellempudi, *et al.*, "A study of bfloat16 for deep learning training," *arXiv preprint arXiv:1905.12322*, 2019.

[25] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," *arXiv preprint arXiv:1904.09751*, 2019.