Автономная некоммерческая организация высшего образования
«Университет Иннополис»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
(БАКАЛАВРСКАЯ РАБОТА)
по направлению подготовки
09.03.01 - «Информатика и вычислительная техника»

GRADUATION THESIS
(BACHELOR'S GRADUATION THESIS)
Field of Study
09.03.01 – «Computer Science»

Направленность (профиль) образовательной программы
«Информатика и вычислительная техника»
Area of Specialization / Academic Program Title:
«Computer Science»

Тема / Topic	**Генерация сообщений к коммитам с помощью методов глубокого обучения/ Generation of commit messages with deep learning methods**

Работу выполнил / Thesis is executed by	**Сергеев Никита Олегович/ Sergeev Nikita Olegovich**	подпись / signature
Руководитель выпускной квалификационной работы / Supervisor of Graduation Thesis	**Иванов Владимир Владимирович / Ivanov Vladimir Vladimirovich**	подпись / signature
Консультанты / Consultants	**Арапов Даниил Дмитриевич / Arapov Daniil Dmitrievich**	подпись / signature

Иннополис, Innopolis, 2024

Contents

List of Tables

List of Figures

Abstract

My thesis explores the domain of automatic commit message generation with deep learning methods. The results achieved in deep learning have increased significantly in 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 models training, evaluation, and discussion of results. Regarding model training, I experimented with models of different sizes, an additional module to solve the problem with message style adaption, and an additional module to solve the problem of limited context. During the literature review, metrics analysis is conducted to identify the most suitable metrics to generate commit messages. I also carried out the experiment to prove that my metrics make sense and correctly identify how good the generated message is.

In summary, my thesis includes an analysis of the problem from a data, metrics, and model perspective. I also carried out experiments to reproduce the results from other papers. In addition, I proposed an idea to mitigate the problem of handling large commits. At the end of my work, I evaluated all trained models and came to a conclusion about the current state of 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. Commits creates a snapshot of code modifications in the project, allowing developers to record their progress. They 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 to 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 writing of concise commit messages is a fundamental 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 open source commit messages are under-informative. State-of-the-art approaches to commit message generation (CMG) show promising results. However, most current research proposes methods that involve training encoder-

decoder transformer-based models for this task. Limitations of current CMG methods are the inability to handle the full context of code changes for the big commits, due to the limited context window of the model, and problems with predicting on the noisy data.

1.2 Methods

Analyzing the literature on the topic of automatic commit message generation, I derived 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 seq2seq neural machine translation task, take code changes as unstructured text, and train the model with a causal language modeling objective.

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 modeling. 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 adaption. 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 the generated messages to the style of the 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 ways to deal with this problem is to consider each commit file separately. In my method, I implemented an additional module between the 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. These scores for averaging are trainable parameters and represent the importance of the file in the commit.

Another part of my work, separate from the deep learning model training, includes data collection and preparation, and metrics analysis. From the data perspective, my work includes collecting my own dataset of commits from GitHub and the analysis of data from other works. My data selection process 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 analyzed most widely used metrics from the literature and the idea behind them. To have a complete analysis of the model performance, I used an additional metric for the semantic similarity measurement. Despite the

infrequent use of this kind of metrics in current research, it is proved that this metric outperforms the existing statistical metrics in the relations with the human judgement.

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 provides 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 used for CMG research. Section 5 offers a detailed examination of the prevailing evaluation metrics used in the evaluation of CMG models. Section 6 identifies and elaborates on the significant gaps and deficiencies present within current research. Section 7 brings this chapter to a conclusion, summarizing the key insights and findings presented in 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 architecture [1], have revolutionized the field since 2018. Models like **BERT** (Bidirectional Encoder Representations from Transformers) [2], **GPT** (Generative Pretrained Transformer) [3], and their derivatives have set new standards for NLP tasks, achieving state-of-the-art results for most NLP tasks. Transfer learning is a dominant paradigm, where pretrained 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. However, 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 obtain 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 a certain commit. The output tokens are the generated commit message $Y = (y_1, y_2, \ldots, y_m)$. The language

modeling model is training to learn a distribution of conditional probabilities of Commit Message tokens, given the information about commit $P(y_1, \ldots y_m \mid x_1, \ldots x_n)$. Modern CMG solutions mostly use the encoder-decoder transformer neural network architecture [4]–[7]. In this approach, the encoder part transforms the input X into a hidden representation $h \in \mathbb{R}^d$, and then the decoder, using this h, generates an output Y. At this point, the structure of the model may vary. The approach might be based on the pure transformer model [5], [6] or **Graph Neural Network** (GNN) combined with the transformer model [8], [9]. Some approaches combine the transformer model with the retrieval mechanism [10], which increases the performance of the method.

2.3.1 Transformer based models

In this approach, code differences from commit are treated as a sequence of tokens, and models excel at capturing the nuances within token sequences. This information is crucial to 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. In addition, 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 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 uses 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 code base 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$_{\text{data}}$** dataset, presented in [6], 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 should be noted 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 filters 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 [9] 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 [11] 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 [4]. The authors of this research paper posit that datasets for CMG face not only the issues outlined [11], 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. The first among these is BLEU [12], 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 shown to align more closely with human judgments regarding the quality of commit messages, as observed by Tao *et al.* [11]. 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 the generated text and is incapable of capturing the semantic similarity between the generated output and the reference text. To address these limitations, more advanced metrics rooted in neural networks have been developed. In particular, BERTScore [13], which utilizes the BERT model, exhibits increased sensitivity to the semantic nuances of the 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. In this section I will describe the current drawback from the models, data, and metrics sides.

2.6.1 Limited context window

Modern CMG methods are limited in capturing the broader context of code changes. They can handle only small changes in code, typically limited to a single file. This is 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. This 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 cannot 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. This 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 NLP tasks uses decoder-based transformer models. For example - CodeLLaMa [14] - is one of the powerful model for the code generation task. However, the LLM usage for the CMG task is not explored enough. The authors in [4] attempt to approach the CMG using ChatGPT with a well-chosen prompt. However, this approach performs worse than the fine-tuned encoder-decoder models. The results are as follows, as ChatGPT is not trained for this task. This is a significant limitation, as 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 background of Deep Learning within the realm of NLP, explore various approaches for CMG, examine the available datasets for this task, scrutinize the evaluation metrics applied to assess CMG models, and identify the current research gap. In the subsequent chapter, I will expound upon the proposed solution for the CMG task.

Chapter 3

Methodology

Literature Review chapter (2) defines modern 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 that 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 Sections 3.2-3.5 I will describe the process of constructing a commits dataset, including data collection, analysis, and the structure of the model input. Sections 3.6-3.9.1 describe the theory behind the models and training process. In concluding this chapter, Section 3.10 provides an overview of the choice of evaluation metrics for the CMG task.

3.1 Experiment design

The main objective of this study is to develop a system to automatically generate commit messages that rival state-of-the-art (SOTA) methods. The initial step for it is to construct my dataset. The next step involves analyzing these data and those from other studies, important to understand the data structure and extracting relevant statistics. This analysis also involves filtering to ensure data quality. Subsequently, my research examined the metrics used in Neural Machine Translation (NMT) tasks, requiring an understanding of their calculation methods and the underlying intuition. Finally, I trained 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 automatically generating commit messages, the data to train or evaluate the models should be in the format of labeled pairs of 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 that were mainly written in Python programming language. For this purpose, I first obtained the names of the repositories and the links to them through the GitHub API@. One way of getting the commit content from the repository is directly

[1] GitHub: Cloud storage for code projects

using the GitHub API, but due to the API calls limit, I opted for an alternative methodology. Utilizing 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.

3.2.2 Retrieved components

The main components of the data sample are 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 Table I

3.2.3 Structure of the data

Before feeding data into the deep learning model, it is essential 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. The tokens described above are used at the beginning of the line, and the opposite

[2]Description of the library

TABLE I
Commit Data Attributes

Attribute	**Description**
Name of repository	Name of the project in which the current commit was added. Useful for identifying the commit to its project.
Commit message	Label for the prediction that is 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.
The hash of the commit	Unique identifier for the current commit to avoid duplicates.

with a backslash is used to signify the end of this part. The final format of the input data from the model is as follows.

<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 large. This histogram indicates that an average repository contains approximately 3000 commits.

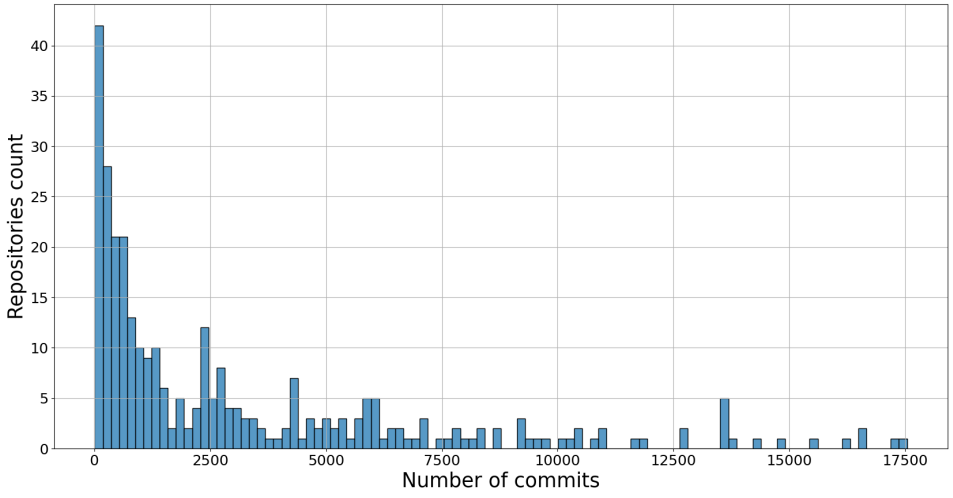

Fig. 3.1. Distribution of the number of commits in the repositories.

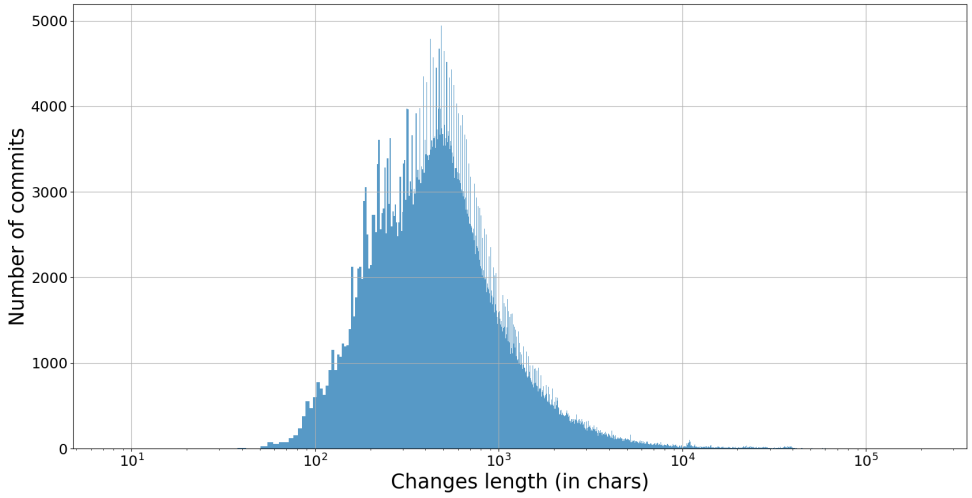

Fig. 3.2. Distribution length of the commits in the dataset.

Fig 3.2 illustrates the distribution of code modifications length within commits. This distribution indicates that the typical length of code modifications is approximately 5000 characters. This suggests that on average, the sequences of code changes are quite extensive. Most modern Deep Learning models are transformer-based and therefore experience significant performance degradation 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. By implementing this truncation, only 15% of the data are lost, yet the average sequence length was significantly reduced.

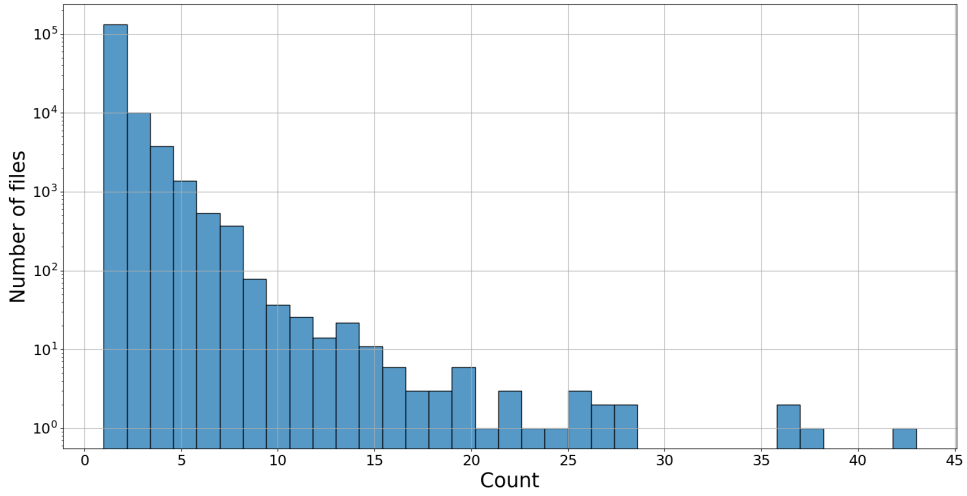

Fig. 3.3. Distribution of the number of files in commits.

Fig. 3.3 shows how the commits in my dataset are distributed based on the number of files they modify. This histogram indicates the significant number of commits that change more than one file. To be precise, single-file commits are two-thirds of all data samples.

One more insight 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 [6]. The authors of this work state that commit typically states with the common verbs. From the collected dataset, I extracted ten most used starting words for commits and got the results, presented in Table II. From this table, it is visible that there exist some common words for starting a commit message. Even among these ten words, there are several repetitive ones with modified writing or form. These words cover ~35% of the total number of samples in the dataset.

In the data filtering step, my goal was to eliminate long samples and samples with bad-quality commit messages, which may lead to model degradation. After

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

TABLE II
Distribution of commit messages

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 that 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 message generation task. But in this section, I would like to focus only on the CommitChronicle dataset, presented in [4]. 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 the 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 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. When comparing CommitChronicle with my dataset in this regard, it was observed that the dataset presented in [4] 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 the length of code changes in characters. On this side, the statistics are similar, so 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, only data for repositories with fewer than 5000 commits are shown to enhance the clarity of the plot. 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 I would like to mention about the CommitChronicle statistics is the most popular first words of commit messages. They are presented in Table III. 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, the data sets are identical on this side.

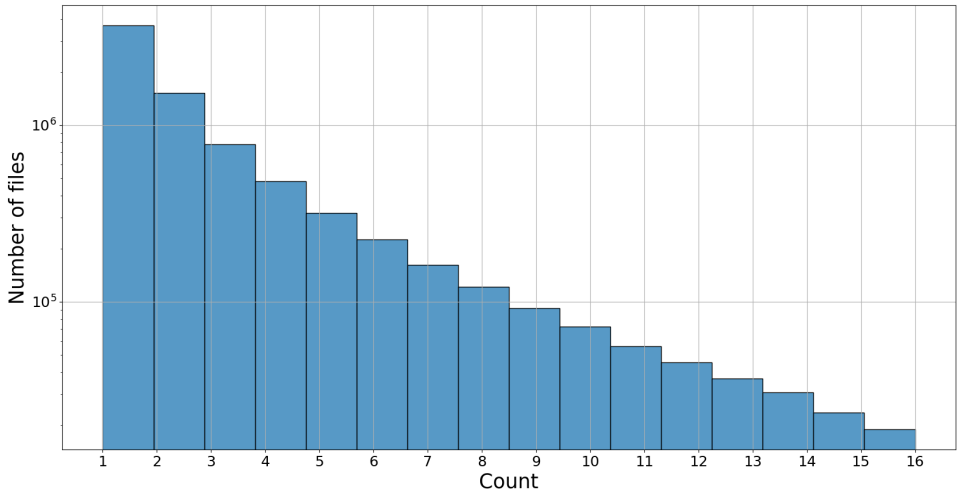

Fig. 3.4. Distribution of the number of files in commits in CommitChronicle.

TABLE III
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

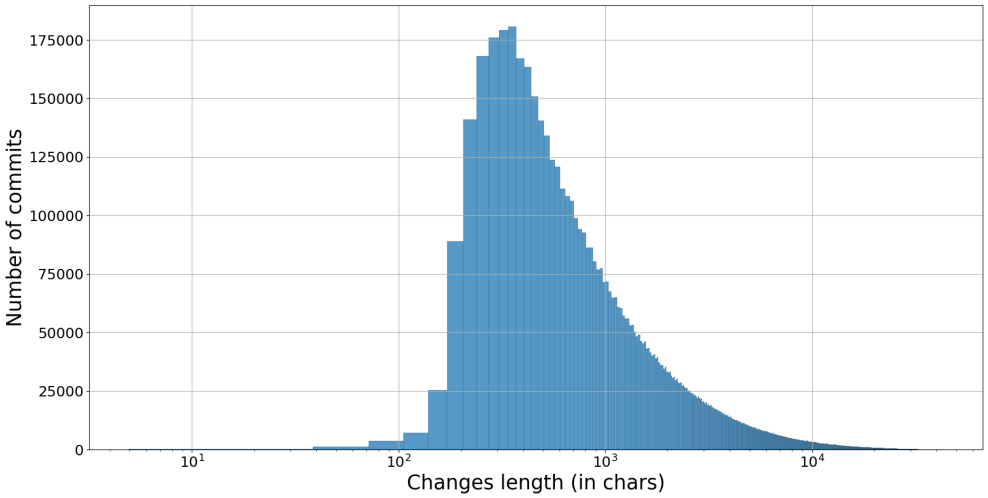

Fig. 3.5. Distribution of length of commits in CommitChronicle.

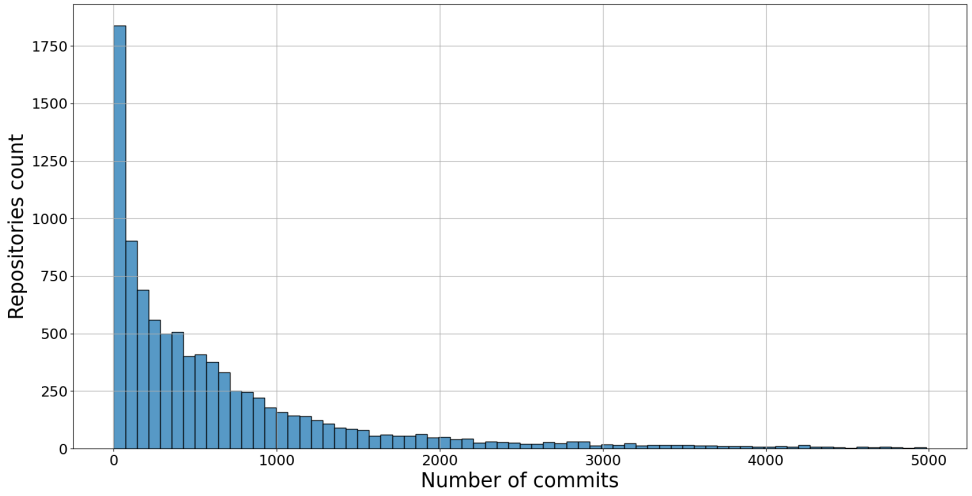

Fig. 3.6. Number of commits in the repositories in CommitChronicle.

3.5 Conclusion about the data

From the data analysis, I found that CommitChronicle has a better representation of the data. It also has many more samples. So, it makes sense to use CommitChronicle as a train set. As a validation set, I used 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 utilized 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 examined the fair performance of the model in the ‘real world data‘.

3.6 CodeT5+ training

My first attempt at approaching the task of generating commit messages is straightforward. I considered 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 [1]. 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 [4] is codeT5. The model was pre-trained on a variety of coding tasks and demonstrated proficient performance in code comprehension. However, since then, CodeT5 authors have introduced an updated version of the model, called CodeT5+ [17]. This new model outperforms CodeT5 in performance in most code-related benchmarks. As the first experiment in my work, I trained CodeT5+ for the CMG task. In this setting, I used 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 Section 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 should learn the embedded representation of these special tokens during the training. This step helps the model better understand 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 the 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 it is known exactly what token to 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 generates 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 IV. 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 version of the model does not process longer sequences, but has a better internal representation, potentially enabling the generation of better commit messages. The training objective I used and the data format are identical to those used to train the base CodeT5+. This experiment demonstrated how scaling the depth of the neural network and improving the representation can affect model performance. Additionally, this phase allowed me to determine whether 220 million parameters are sufficient to construct 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 IV
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 CMG is the shared style of messages within the same repository. For example, a repository of Linux kernel source code has a strictly specified format of commit messages. Using only code modifications, it is impossible to adapt the generated commit message to have the same specified format.

3.8.1 Retrieval approaches

To mitigate the issue with a message style works as [10], [9] and [18] using an additional retrieval module to the standard encoder-decoder architecture. These methods mostly use the commit database and the corresponding commit message. Then a new commit is given as input to the model; this method first uses the encoder to get an embedded vector representation of code modifications. This representation is also used to find the most similar code changes from the dataset

and to get both the modifications and the commit message. This additional information is then passed to the model to get the result. Analysis of these methods showed that the retrieval mechanism uses the results of commit generation.

One more way to use the historical data in the generation of commit messages is to retrieve previous commits from the same repository, or previous commits of the same person. This method was used in [4] and showed an improvement in the performance of the model. If previous methods aimed mainly 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 [4] with a combination of the RACE method from [10] and the history of messages added to the input of the model. The main difference between my approach and the RACE is that the RACE leverages 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 to use only the message from the most similar commit to adapt to the style of the commit message of the same code theme. In addition, I passed input history samples of the messages from the same repository to the model. In the end, my input to the model has the following format:

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

Generally, I construct my pipeline in the following manner. Initially, I acquired the database from which I extracted 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 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 embeddings of code changes that I will refer to as the database. An additional extension to the dataset is the previous commit message from the same repository. To obtain the message history, the training data were grouped by repositories and sorted in historical order based on commit timestamps. The model 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 is always an exact copy of the real commit message since its copy exists in the search space. This behavior breaks the logic behind the method and leads our model to overfit, thus I should handle this situation and consider a threshold on the similarity for it not to be too big. One more restriction on a similar commit search is the timestamp of the retrieved sample. In the training phase, the 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 previous commits from the repository, I constructed the input in the format described above. These modifications 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 transformer-based models. As stated in [1], 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 input. This amount is insufficient to handle a large commit that includes changes in 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 the messages generated 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 model should be high. Additionally, using a larger model could significantly reduce performance in terms of speed. 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 large commits is as follows. In most cases, big commits involve changes in multiple files. Each file in the commit is relatively simple and can be processed using the codeT5+ model. Therefore, if I get the embedding of each file separately, I should have the full context of the commits after the encoder. Suppose that I have a commit with modifications in five code files. Firstly, I separately pass each file to the encoder part of my model. Even if some of the files are too large and cannot fit into the context window, I can 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 leads to truncation of the data. Some files may be excluded from the model input, which may lead to the loss of information about the commit goal. Before passing the separate embeddings of the 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 used 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 the trainable parameters. I get them through the additional module between the encoder and decoder parts of the model.

3.10 Metrics to evaluate CMG results

In the literature review section, I described the most popular metrics to measure the performance of the models in terms of the quality of the generated commit message 2.5. For my experiments, I used the bath metrics described in the literature review. BLEU for syntax similarity and BERTScore for accessing 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 [12]. 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 matches the idea of BLEU. Therefore, the results of this metric properly reflect the quality of the generated commit message.

Taking into account the 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 the brevity penalty (**BP**) and the weighted sum of the precision in n-grams. The bravity penalty is calculated according to the formula 3.3 and penalizes the metric for a too long generated text. In this formula r stands for the length of generated text and c is the length of 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.

The default parameters for the BLEU score are $N = 4$ 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 above explanation, 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 variable, and, therefore, it is hard to compare the results among different works. To overcome this problem, the authors of [20] presented the SacreBLEU package. This Python script helps calculate the BLEU score in a unified way to have reproducible results.

Analyzing the BLEU score specifically for the commit message generation task, the results from [11] show that in this specific code-related task, 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 the prediction and the reference to lowercase, as it is not as important for this task as for translation. The second difference is adding one to both the numerator and the denominator of 3.2

to add smoothness to it.

In my experiments, I decided to use SacreBLEU and B-Norm. This should make the evaluation results more extensive and fair. Counting SacreBLEU should help me compare my results with other approaches to solving CMG tasks. B-Norm is used mainly in my work to compare the results with the work of JetBrains Research [4], as I used their method as a baseline for my work. To make a fair comparison in terms of the B-Norm, the script was used to calculate the metrics from the source code of [4], as it is open and free to use.

3.10.2 BERTScore description

BERTScore is the metric for automatic evaluation of text generation presented in [13]. This metric is based on the BERT [2] - encoder-only transformer-based model. The BERT model was shown to achieve outstanding results in encoding text and getting meaningful hidden representation. Using 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, , *that is,*, 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, and the generated text has the same meaning, but is written in other words.

Consider the situation with the reference text $x = \langle x_1, x_2 \dots x_k \rangle$ and the generated candidate text $\hat{x} = \langle \hat{x_1}, \hat{x_2} \dots \hat{x_k} \rangle$ represented as tokens. The first step to calculate the BERTScore is to obtain embeddings of size h of each token for

both texts, resulting in two embedding 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. The next step is to calculate the cosine similarity for each hidden vector of the matrices. Cosine similarity between the reference token x_i and the 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.

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

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

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

The complete BERTScore matches each token in x with a token in $\hat{x}$ to calculate the recall 3.7, and each token in $\hat{x}$ with 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 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 written above, the most important step in calculating this metric is to obtain representative embedding for each of the tokens with BERT. The choice for this is essential for obtaining 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

In this chapter, my goal is to reveal the details of the implementation for all the experiments described in the methodology 3. My description will include the algorithm to convert the text data into a format for efficient training with the transformer model. It also includes details about the hyperparameters that I used for the training and the generation process. This chapter will also cover the architecture-specific details for the experiments with custom modules. Each section of this chapter concludes with the results of the experiment and a summary of the training process.

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. In addition, I will describe the hyperparameters required for 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 into 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 **??**. 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 is 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 are typically passed in batches rather than as a single sequence. Batched training is utilized to accelerate GPU computations and improve loss convergence. For this reason, the input batch should have the form of a matrix. The issue of constructing this matrix arises due to the varying lengths of the 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 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. The performance of a model 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 to fine-tune 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 V represents a list of all hyper-parameter configurations for CodeT5+ fine-tuning. The **batch size** for training and evaluation was set to 32. According to empirical findings, this batch size is optimal in training time and computing 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 at 2×10^{-5}. The learning rate can be interpreted as a "step size" for the gradient descent method. In general form, it corresponds to γ in the update formula of the model parameters $w_{\text{new}} = w_{\text{old}} - \gamma \nabla f(w_{\text{old}})$. During training,

TABLE V
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

the learning rate is configured using the **linear scheduler**, which monotonically decreases the learning rate to 0 until the end of 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 responsible for the regularization of the parameters of the neural network. 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. The 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 the training of neural networks with bf16 achieves the

same performance as the 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 in this section, except for batch size, is shared among all models trained in my work. This set of parameters was taken from [4], as the 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. However, this greedy strategy may lead to the degradation of the generated text. 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 that I used for my model and explain the intuition behind them to justify my choice.

Table VI lists the set of hyperparameters that I use to generate commit messages. **Max new tokens** configures the maximum length of the generated message. According to the statistics of the training dataset, the commit messages are ~ 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

TABLE VI
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

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, the 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 bigram several times. It is a known issue of large language models that they may hallucinate and start repeating some phrase until 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 **top_k** and prevents the model from hallucinating and predicting irrelevant tokens. All of the generation hyperparameters described in this section help the model generate more relevant text. Moreover, some of the parameters, like sampling, make the generated commit message more human-readable.

4.1.4 Training process

In this section, I will describe the details of the training process for the base version of CodeT5+ [17] with 220 million parameters. As training data, I took the train split from the CommitChronicle data set [4] and for the intermediate evaluation, I took the subset of the CommitChronicle validation set with 1500 random samples. I trained my model for one epoch with the use of 2 NVIDIA A100 80GB. In total, the training process took 3 days and 19 hours.

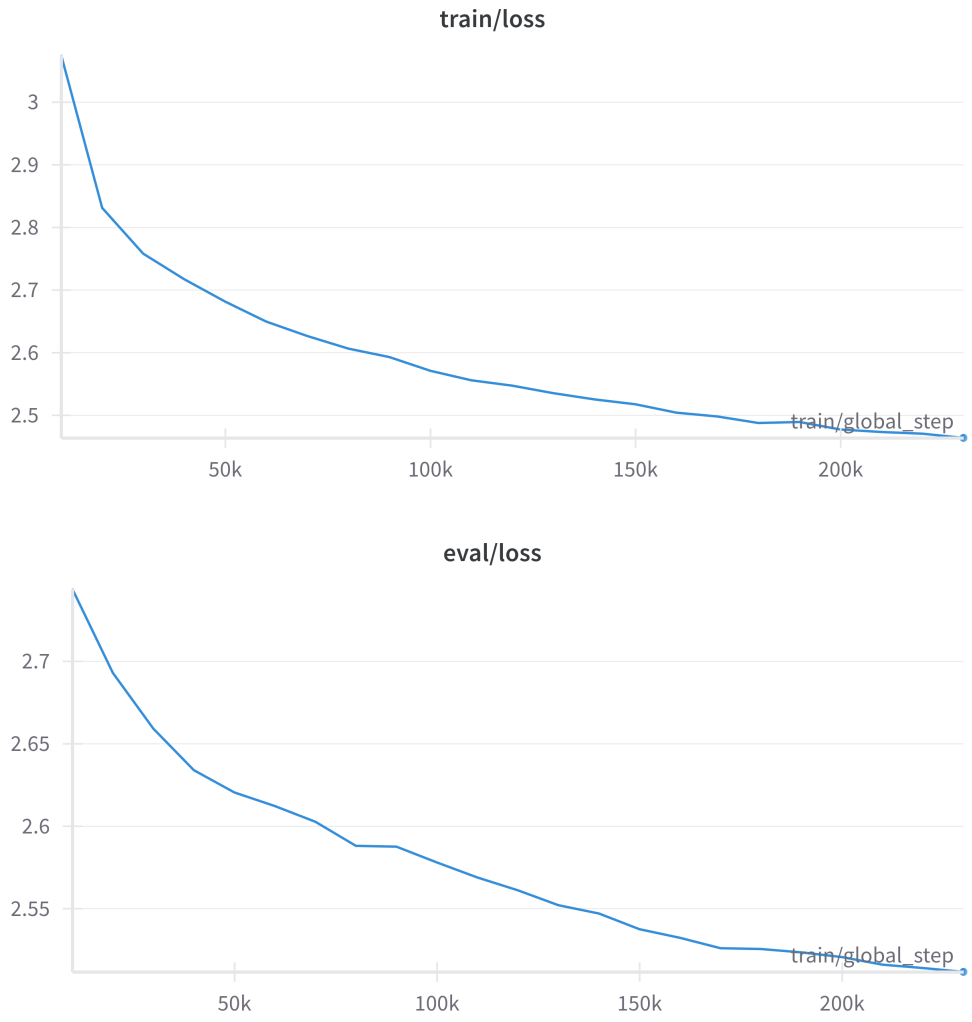

Fig. 4.1. Loss history for the base model.

Results of the loss behaviour during the fine-tuning process are presented in 4.1. From the plots, it is easy to see that the loss decreased during fine-tuning in both the training and validation sets. This means that the fine-tuning process was successful. The model improved its ability to generate commit messages for the training set and generalized this skill on the validation set.

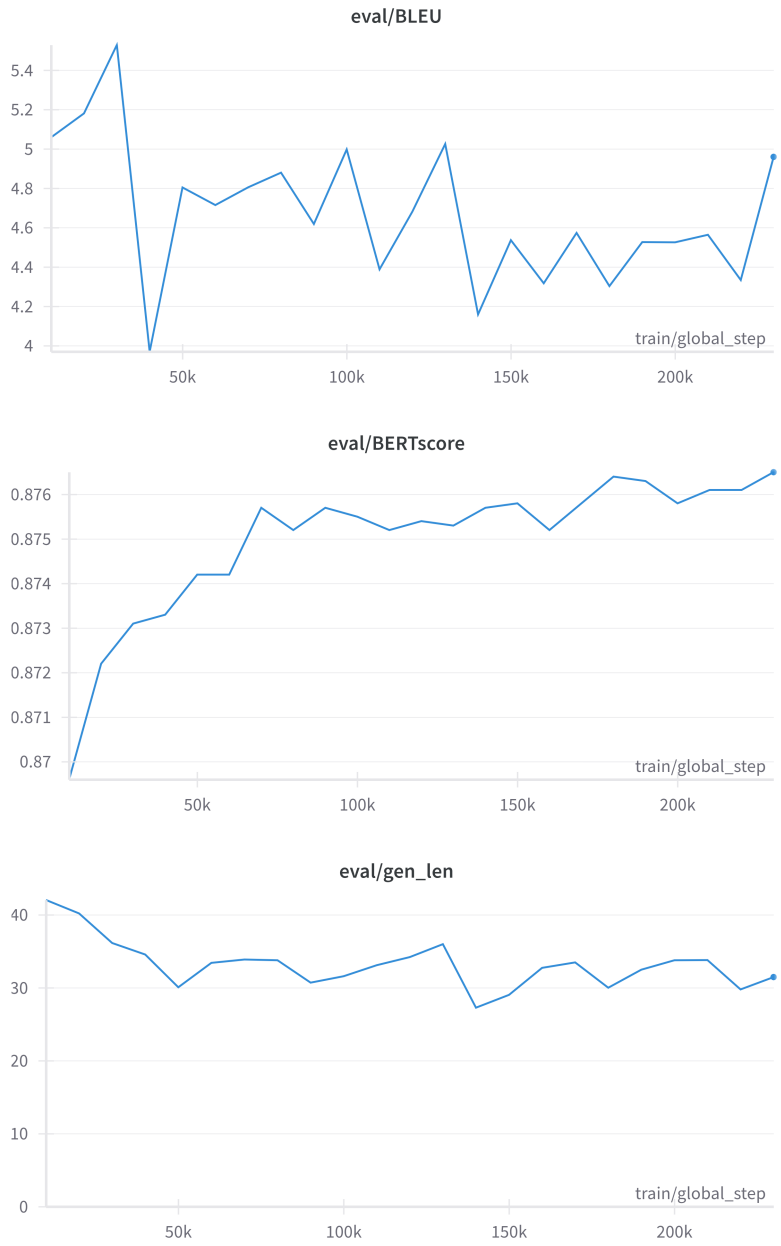

Fig. 4.2. Evaluation results for the base model

The results of the training are presented in Fig. 4.2. From the plots, it is visible that the BERTScore improved during the fine-tuning. This improvement signalled that the quality of the generated commit messages improved. But the range of the BERTScores is not that big. It only grew from 0.87 to 0.876. The value of this metric is close to 1 for the entire process. This is a problem of the

base BERT model in calculating the token embeddings. The default version of this model is not so sensitive to the semantic meaning of texts. Thus, the value of this metric was excessively high and lacked enough variation. I solved the issue by using a more advanced model capable of better capturing the text's meaning. As this was my first experiment, here I can report only this poor BERTScore, but in the results section, I will calculate the single metric for all the trained models. When discussing the behavior of the BLEU score during fine-tuning, it did not show any trends. When comparing the start and the end scores, it changed from 5.05 to 4.96. This result is statistically insignificant, so the metric did not change during training. During the training process, I also tracked the length of the generated messages in tokens. As stated earlier, the "true" messages in the training set are short on average. I expected the model to learn this pattern and produce concise responses. From the plot, it is visible that my expectations have been met, and the mean output length became shorter, from 42 to 31.4 tokens. From all the results presented above, I may conclude that the training was successful. The model learned to construct the commit message from the code modifications. Precise analysis of examples of generated messages and model testing will be covered in the corresponding chapter. The training pipeline was implemented using the PyTorch [26] machine learning framework. To efficiently work with the transformer architecture, I used the transformers [27] library.

4.2 Training results of CodeT5+ with 770M parameters

The process of the CodeT5+ model with 770 million parameters is the same as for the base version of the model. I used the same hyperparameters and the same format as the input data. The only difference from the base model is the results of the training. In this section my goal is to test how the scaling affects the quality of the generated text.

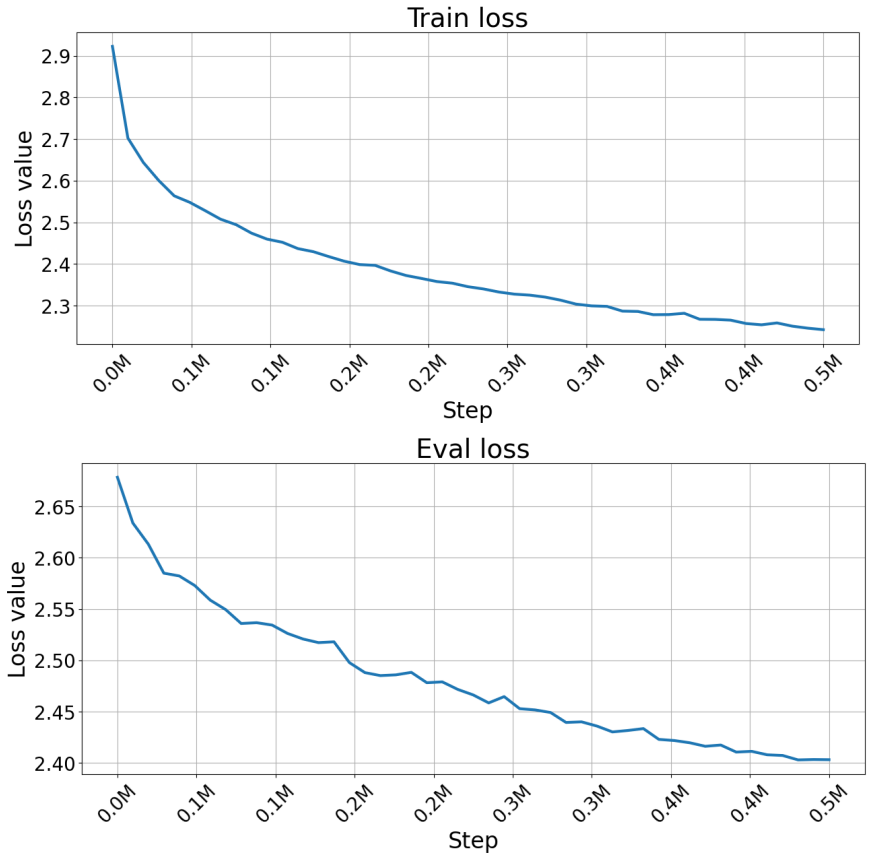

Fig. 4.3. Loss history for CodeT5+ 770M.

From Fig. 4.3, it is easy to see that the training converges for the data splits of the training and evaluation. It should be noted that the model achieves better

loss results across all splits. This means that the scaling transformer-based model in terms of parameters helps to achieve better results for the generation of commit messages.

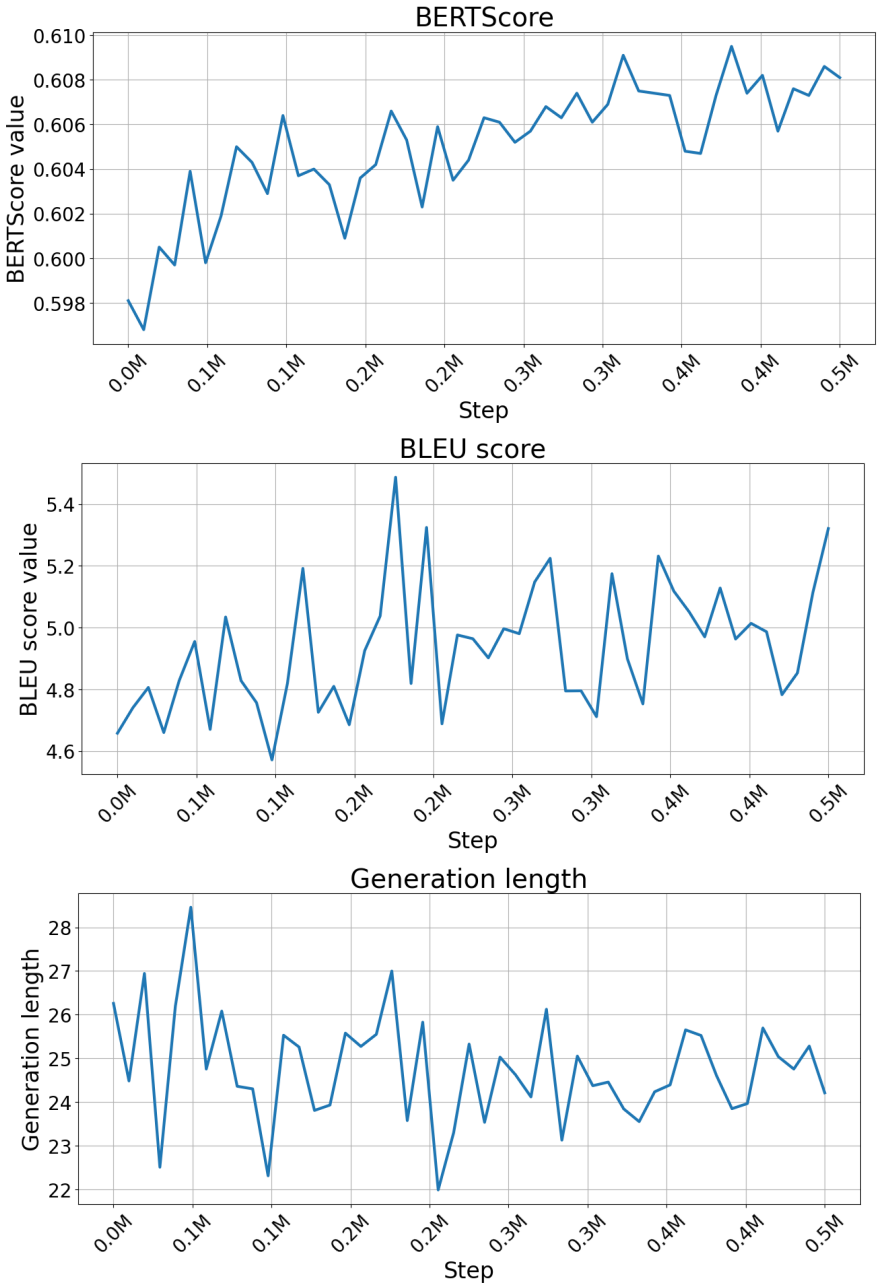

Fig. 4.4. Evaluation results for CodeT5+ 770M.

Considering the evaluation results for the bigger model presented in Fig. 4.4, it also shows improvement of the generation quality during the fine-tuning. For the BERTScore computation in this experiment, I used a more powerful model for the embedding calculation. Therefore, the metric range is lower than for the base version. However, this metric improved during training and therefore the quality of the generated messages improved. BLEU score shows better results for this model and even has some tendency to grow, in contrast to the base CodeT5+ model. The ability of the model to generate more concise messages is also improved. It is visible from the average generation length plot. CodeT5+ with 770M parameters learned to generate messages with 24 tokens, compared to 31 for the base version. In conclusion, I can say that the use of a deeper model with more stacked transformer blocks improved the performance of the target task. However, there is still a gap in the analysis of the model in terms of computational cost, GPU memory consumption, and inference time for the big model. In the evaluation chapter, I will analyze all the models from the efficiency perspective and make a judgment call on whether or not to use a scalled version of the model for the CMG task.

4.3 Training of CodeT5+ with retrieval

4.3.1 Architecture implementation details

In this section, I will give details of the implementation of the experiment with the retrieved context. The motivation for the usage of the retrieval module and the design of the model architecture was described in 3.8.2, so here I will focus on the practical part. The general pipeline of the model is presented in Fig. 4.5.

In summary, the whole pipeline consists of the following steps: 1)Process the input batch of data, 2)Get the previous commit message from the same repository, 3)Using the embedding of code modifications, find the most similar commit and take its message, and 4)Using all the information construct the new batch and process it with base CodeT5+ like in the first experiment.

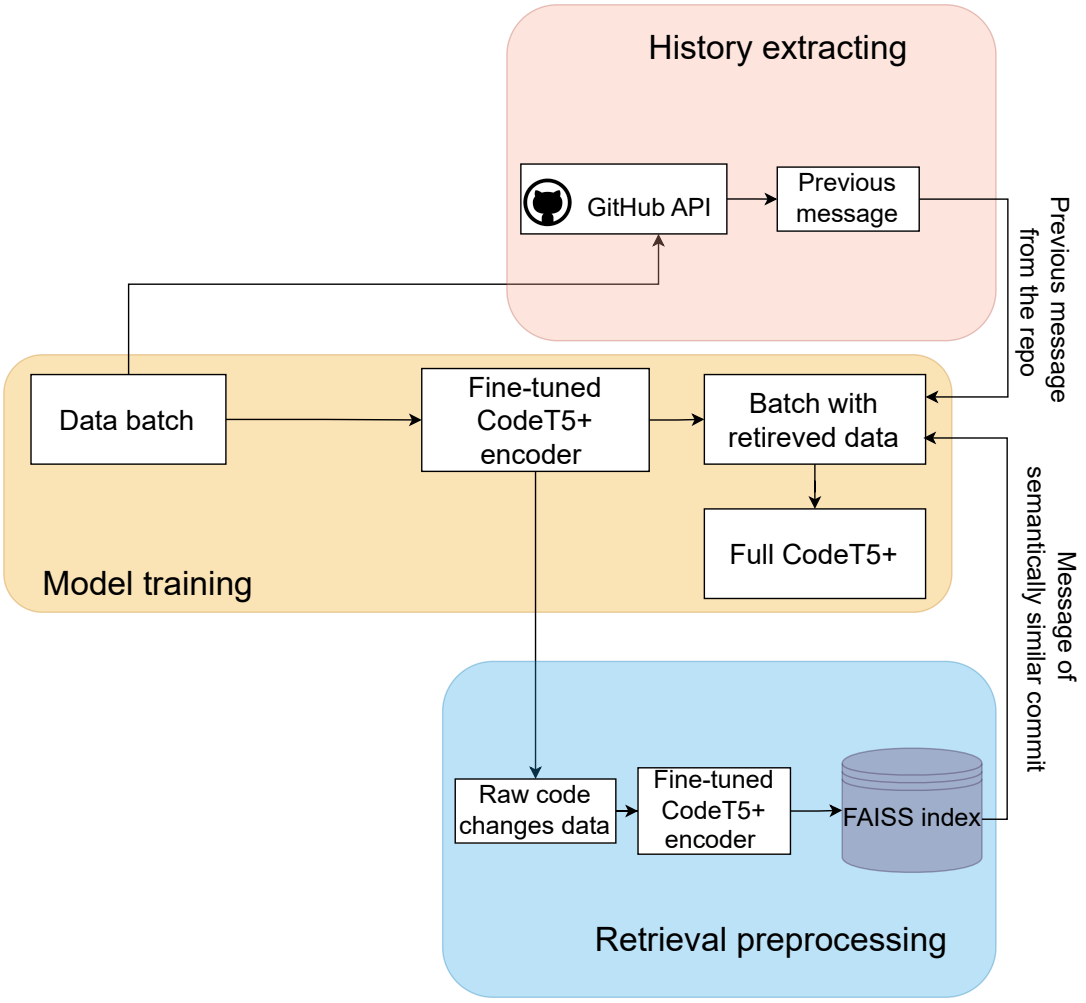

Fig. 4.5. Architecture description for CodeT5+ with retrieval.

To retrieve previous messages from the same repository for a given commit, I utilized two different approaches for training and inference. During training and

validation, I already had all the commits from the repository in the corresponding dataset split. Therefore, all I had to do was sort my dataset in order to map the samples with respect to time. During training, I precomputed all the necessary steps in advance and passed the previous message as an additional input field. On the other hand, during the inference time, I have live-time access only to the GitHub link to the commit. For this reason, I need to get the previous commit during the inference time. I used GitHub API to get the parent commit of the given.

To retrieve the message from the most similar code changes, I was required to have a precomputed database of embeddings from the training dataset. For this purpose, I processed the input split of the CommitChronicle with the encoder part of the pre-trained CodeT5+ from my first experiment. With a straightforward approach to finding the most similar commit at each forward pass, I need to calculate the cosine similarity between the input embedding and all the database embeddings. This approach takes too much computation, so I need an approach to find the closest embedding to the given in a faster way. For this purpose, I used the FAIS [28] retrieval index, which can efficiently find the k closest points to the given in the high-dimensional data. Furthermore, this framework can make all computations on the GPU, thus making them parallelizable and faster. I have also experimented with another popular retrieval index - Annoy [29], and found that FAIS is more precise and efficient.

Table VII shows the hyperparameters for this specific retrieval architecture. The parameter **top_k_search** is responsible for the number of retrieved passages. According to the pipeline architecture, the model takes only one similar commit from the database. However, some passages can be considered irrelevant during the filtration phase. The retrieved commits must have been made after

the given one. This filtration is crucial during model training, as it prevents overfitting by restricting the model from looking into the future and possible commits on the same code. One more filtration parameter is ϵ - cosine similarity threshold. This parameter is responsible for not including too similar commits, avoiding accidental retrieval of the target message. One more hyperparameter is the **max_msg_tokens**. As mentioned earlier, the transformer model has a limited context window size. Therefore, when I expand the input with previous and similar messages as the raw text, I need to control the length of the inserted data. I do not know the size of the retrieved context in advance. To prevent the context window from being overwhelmed by messages, I truncated the retrieved passages after a certain number of tokens.

TABLE VII
Retrieval specific hyperparameters

Hyperparameter	value
top_k_search	20
max_msg_tokens	50
ϵ	0.1

4.3.2 Results of the training

This section includes the results of CodeT5+ training with the retrieval module. Fig. 4.6 represents the loss history on the train and the validation data. From the plots, it is visible that the loss is decreasing during fine-tuning. It is important to note that even if the validation loss is in the same range as in the previous experiments, the loss for the train set is significantly lower than it was before. This behavior of the training loss may signal overfitting, meaning that

the retrieved passage is close to the target message and the model trains to use it instead of looking into the code. However, validation loss is decreasing, so the model is generalized in the ability to generate commit messages.

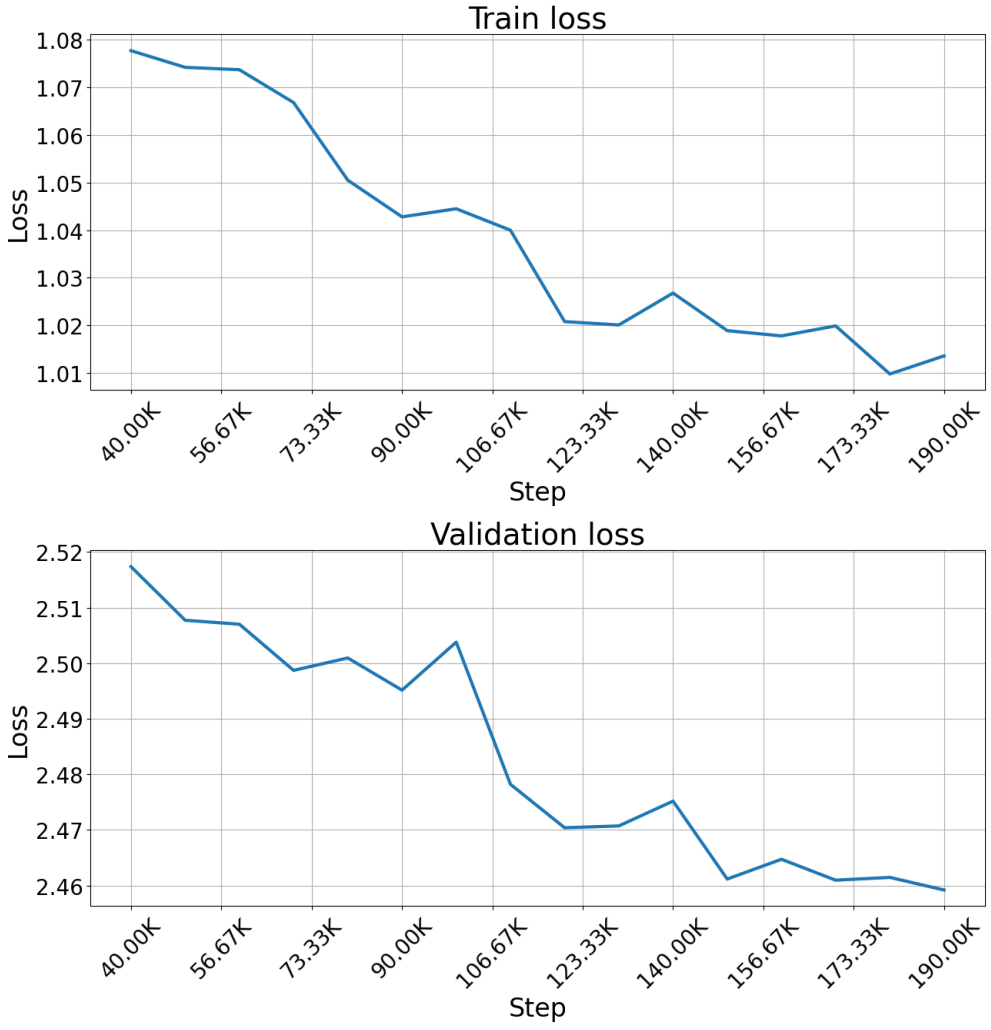

Fig. 4.6. Loss history for CodeT5+ with retrieval.

Fig. 4.7 represents the results of validation for CodeT5+ with the retrieval. BERTScore metric improved during training, but not significantly, from 0.6 to 0.604. BLEU score did not improve, even though it was the main goal to follow the style of the human-written message. However, the validation set is only a small subset of the original data. Thus, I need to have a full set evaluation to

make sure that the experiment did not succeed. From the perspective of the generated sequence length, the model learned to construct shorter messages. Concluding this section I can say that the results from the small validation subset are unpersuasive, but to see the full picture and make a final decision about the experiment's importance I need to have a full evaluation.

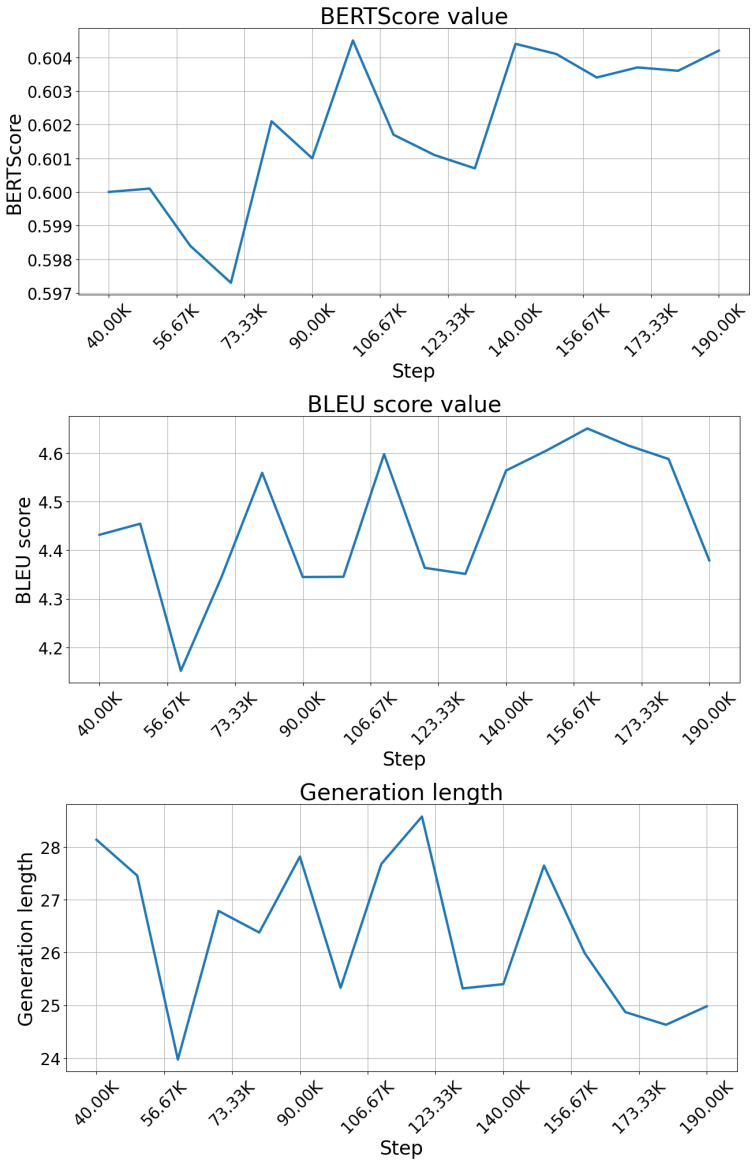

Fig. 4.7. Loss history for CodeT5+ with retrieval.

4.4 Training of CodeT5+ with file attention

4.4.1 Model architecture details

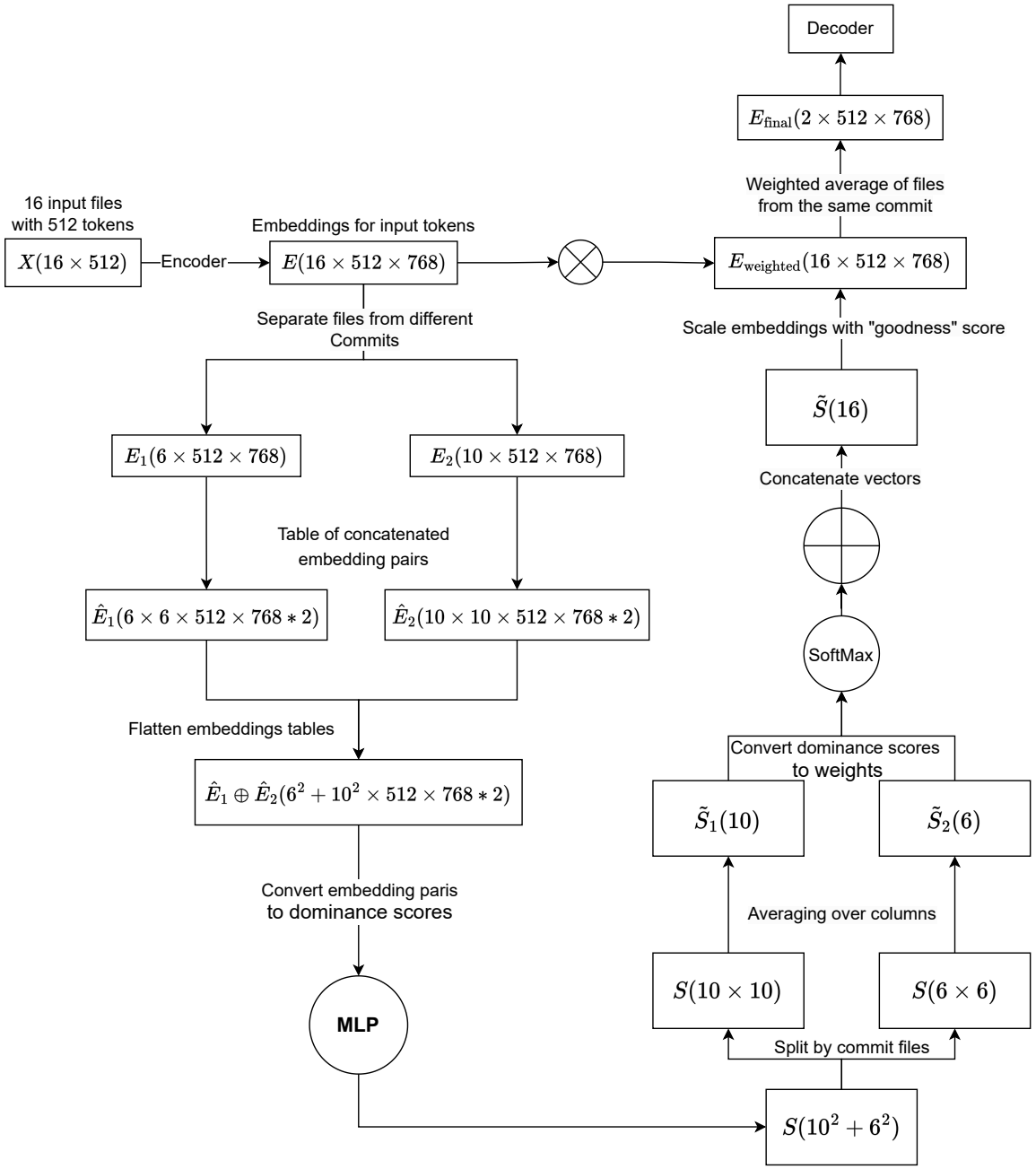

Fig. 4.8. Architecture description for CodeT5+ with File attention.

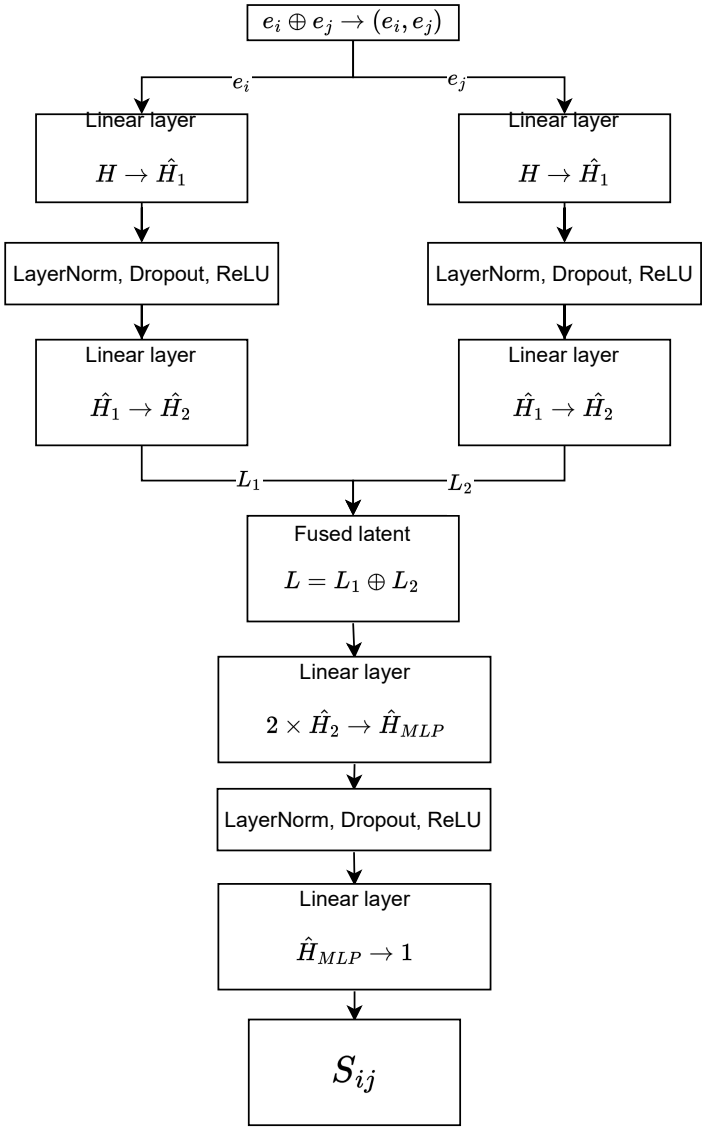

Fig. 4.9. Architecture of the MLP block.

This section includes the model implementation and training details for CodeT5+ with file attention. The model architecture follows the idea described in the methodology section 3.9. The overall pipeline of the model forward pass consists of the following steps:

1. Taking the input in the format of the matrix of tokens of size (n, m) where m is the context window of the CodeT5+ and n corresponds to the number files with the code changes in the batch. Each commit is separated by the files at the beginning. For example, the model takes a batch of 16 files, where the first 6 files correspond to the first commit and the remaining 10 to the second.

2. Each file is passed through the encoder and produces embeddings matrix E of the shape (n, m, h), where h is the hidden dimension of the model. For this experiment I used the fine-tuned version of the codeT5+ model, thus the encoder part is frozen and does not train.

3. The embedding matrix is separated by the commits. For each commit in the batch, I took the corresponding files and considered their embeddings separately.

4. For each embedding matrix, the model constructs the tensor of pairwise concatenated embeddings for each file from commit. It has a shape of $(n, n, m, h \times 2)$. It consists of all possible variants of embedding concatenations.

5. Concatenated embeddings for all commits are flattened and then passed to the multi-layer perceptron **(MLP)** block. Flattening is used to parallelize the computation of the **MLP**. The feedforward block of the neural network aims to determine the superiority score of code change file *i* over file *j* for each file pair. The architecture of the MLP block is presented in Fig. 4.9. Initially, the model splits back the concatenated embeddings for files *i* and *j* to consider them separately. Both embeddings are further mapped into

the latent space using two linear layers with normalization activation and dropout between them. In the next step, latent vectors are fused and passed to the two linear layers resulting in one scalar - dominance score S_{ij}. This value represents how much file i is "better" than file j.

6. In the previous step, the model produced vector S of the shape $(n_1^2 + n_2^2)$, where n_1 and n_2 are numbers of files in the first and the second commit from the batch respectively. Each element in the S represents the comparison of all possible pairs of files for every commit in the batch. In the next step, the model breaks this sequence to obtain a separate matrix with dominance scores for each commit. These matrices are then averaged by columns resulting with vector $\tilde{S}_i$ with size n_i for each commit. For the given commit this vector represents how useful is the certain file with code changes within the scope of this commit.

7. The main purpose of this method is to have a trainable weighted sum of the embeddings over the files. To convert $\tilde{S}_i$ into weights it is passed through the Softmax function to have a distribution of usefulness among the files.

8. On the next step all the $\tilde{S}_i$ are fused into one vector $\tilde{S}$. This vector represents coefficients for the weighted average of the files embeddings for each commit. Applying weighted average to the initial embeddings matrix results with an embeddings matrix for the commits, which can be passed to the decoder in a regular way.

This procedure of averaging separate file embeddings instead of truncating the input increase the model performance in the case of long commits with multiple files. Moreover, all the operations performed between the encoder and decoder

parts of the model are parallelized and does not affect the speed of the generation. Detailed analysis of the performance and time efficiency for all the model used in my work is presented in the evaluation chapter 5

4.4.2 Training setup details

The training setup for this model is different from other experiments. The input batch for the file attention model consists of a fixed number of code files. Therefore, the number of commits in a single batch may vary. The dataset entry contains a single commit that may include up to 16 changed files. For training, I decided to have the data in batches with 16 files. For batch construction, I iterated over the dataset and included the commit in the batch if the number of files did not exceed the fixed maximum number. Thus, before the start of the training, I had pre-computed batches. Commit data may vary among the samples. They may contain different programming languages or project semantics. The backward pass through the MLP block has a united gradient for the whole batch. This property of the unified gradient may break the correct training of the weights for the importance of the files. To handle this possible drawback, I did an additional experiment with a single commit in the batch. This setup reduces the parallelism of the training, but it can improve the model's ability to determine the relevance of files within a commit.

4.4.3 Training results

This section presents the results of the training for both setups described above. As in the previous experiments, it includes the loss convergence in training and validation sets. During the training phase, the implementation of inference

mechanisms for this architecture was not ready. Consequently, this section will not encompass a discussion of the metric evaluations for the training checkpoints. A comprehensive analysis of the quality of the generated commits will be included in the subsequent evaluation chapter.

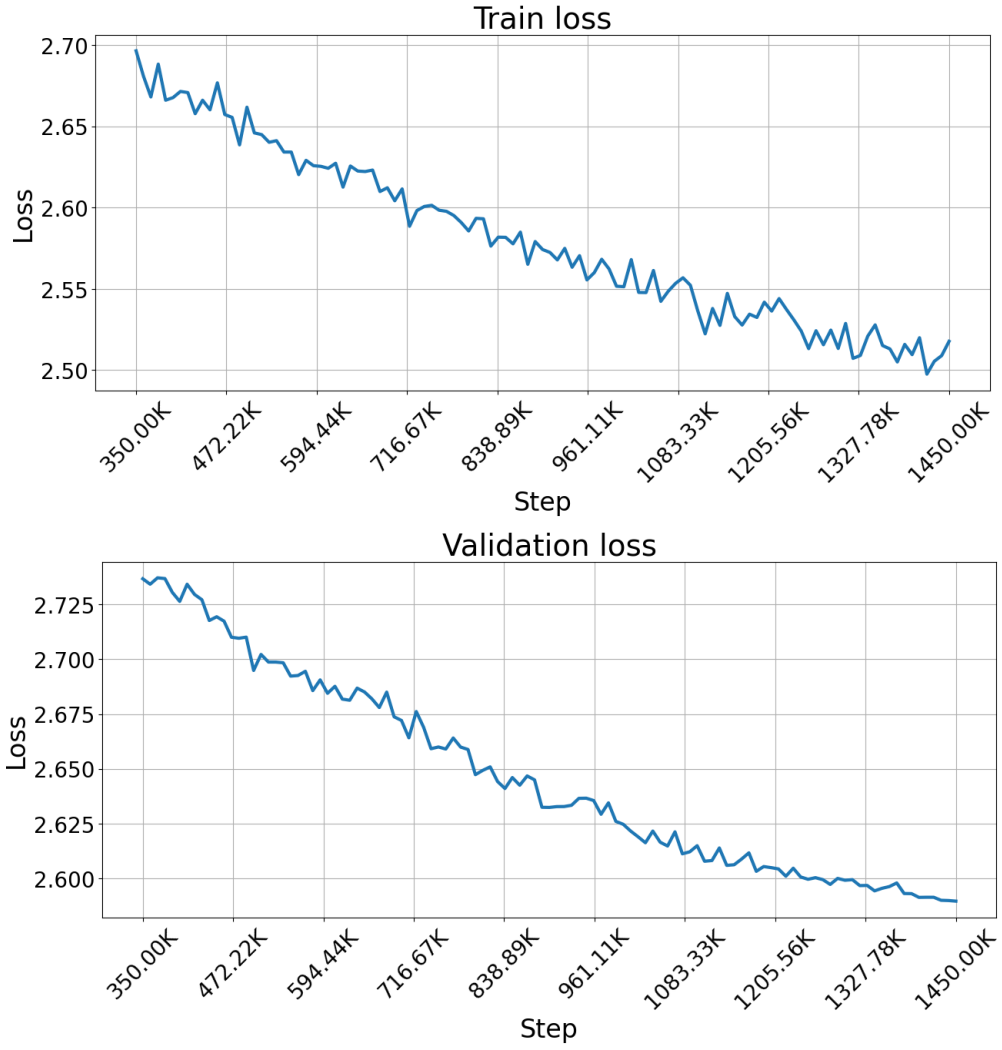

Fig. 4.10. Loss history for CodeT5+ with file attention.

Fig. 4.10 represents the trajectory of loss convergence for the CodeT5+ model incorporating the file attention mechanism. The plots clearly illustrate that the loss trajectories for both datasets are monotonically decreasing, indicating

that the fine-tuning process has been successful and the model has effectively acquired the capability to generate commit messages. However, considering only the intermediate loss values is not sufficient to draw a conclusion about the model performance.

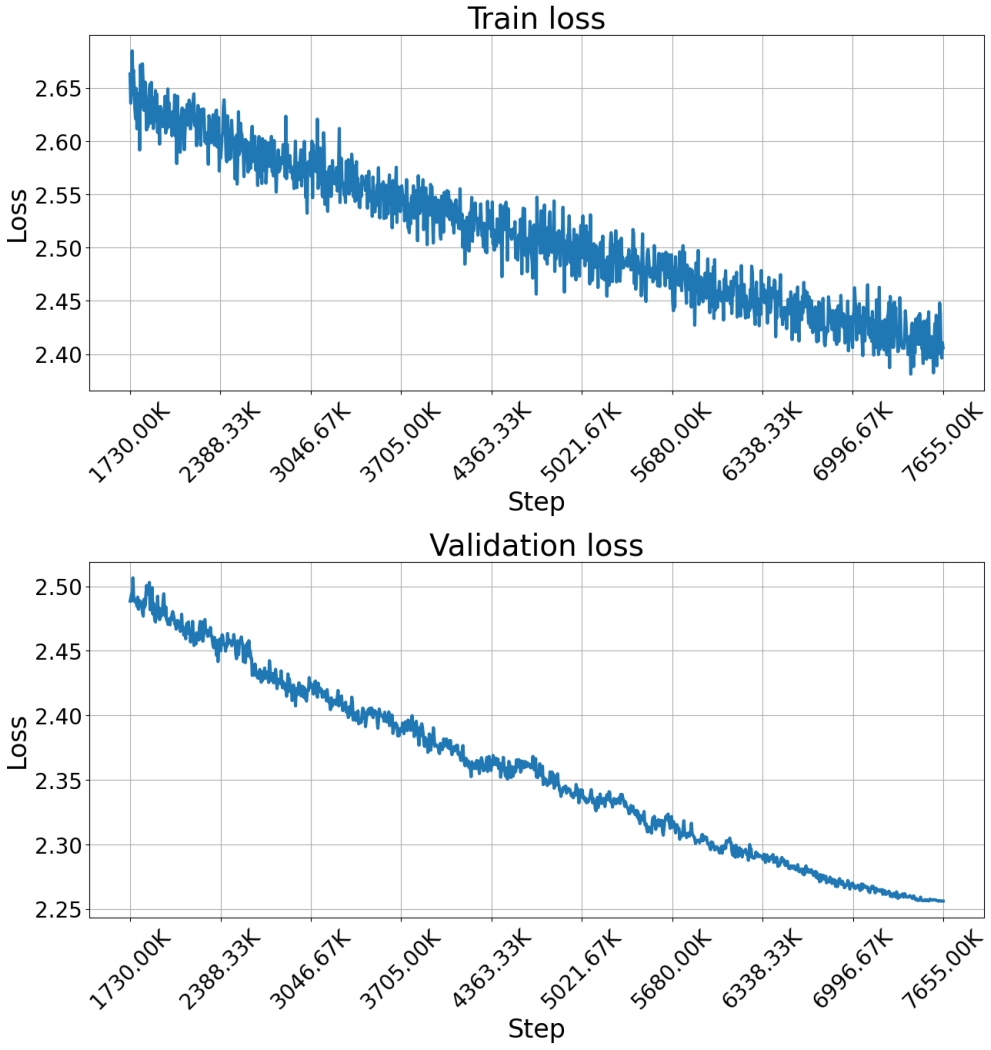

Fig. 4.11. Loss history for CodeT5+ with file attention.

The convergence of the loss function for the secondary configuration, with a single commit within the batch, is depicted in Fig. 4.11. Plots shows that this setup shows better convergence for the validation loss. In contrast to the initial

configuration, which has multiple commits per batch and ends up with a final validation loss approximately equal to 2.60, the second setup employing a single commit shows a better loss, achieving approximately 2.25. Furthermore, the loss metrics for the training data are also improved in the second configuration. However, the possible reason for this may be the greater number of backward steps. As visible from the plots, in the setup with a batched input, 1.45M steps were made during fine tuning. At the same time, using a single commit as an input, the model takes 7.655M steps to fine-tune on the whole dataset. This significant difference in the training samples also affected the fine-tuning time. While the first version took approximately 5 days to train, the second one required almost 11 days for the full fine-tuning process. In summarizing this comparative analysis, it is evident that relying only on the values of the loss function is insufficient to make a judgment regarding the performance of the model in different setups. A comprehensive evaluation of training methodologies requires the analysis of metrics derived from the test data to ensure a fair comparison.

Chapter 5

Evaluation and Discussion

The main objective of this chapter is to describe the final results of the trained models. It includes a comparison of the results of all the experiments described in the implementation chapter 4. The analysis includes statistical evaluations for the quality of generated commit messages with BERTScore and BLEU metrics. In addition to evaluating the quality of generation. I also assessed the length of the messages, emphasizing the necessity for the model to produce brief and concise text comparable to that authored by humans. One more important aspect of trained models is the time and memory efficiency at inference time. Taking into account the task context, it is important that commit messages be generated quickly and with minimal computation. Therefore, comparing the efficiency of different models is essential when deciding on the best model to use in production. In discussing the data used for the evaluation, I used two distinct datasets. One is the test split from the CommitChronicle dataset, and another is the dataset parsed from GitHub. Considering the evaluation using the parsed data, I did not include the timestamps for commits while collecting the data. Therefore, it is impossible to access the quality of CodeT5+ with the retrieval on this dataset, as it requires

previous commits from the same repository.

5.1 BERTScore analysis

TABLE VIII
Mean BERTScore Metric With Std

Model	Mean	Std	Mean parsed	Std parsed
CodeT5+ 220M	0.583	0.110	0.558	0.108
CodeT5+ 770M	**0.608**	0.116	**0.581**	0.119
CodeT5+ with file attention	0.600	0.112	0.572	0.112
CodeT5+ with file attention single commit train	0.595	0.112	0.570	0.114
CodeT5+ with retrieval	0.603	0.115	-	-
JetBrains CodeT5	0.600	0.112	0.569	0.114

Table VIII represents the mean value and standard deviation of the BERTScore. It was measured for the test set of Commit Chronicle and the manually parsed data. As demonstrated in the table, CodeT5+ with 770 million parameters achieves the highest performance on both datasets. This leads to the conclusion that model scaling implies better semantic extraction from the code changes. Considering the comparison of two different CodeT5+ training setups with file attention, the metric reports that the use of a single commit per batch during training leads to a slight degradation of the metric. Both experiments with the addition of an additional module to the base CodeT5+ increased the quality of the generated messages. The results for the parsed data on average are worse than in the results on the CommitChronicle. One of the possible reasons for this is the difference

in the filtration process. The authors in [4] implemented rigorous filtering on the data. In contrast, I did not apply any quality-based filters to my data, with the exception of setting an upper bound on the length of the message. This may cause the presence of messages with questionable meaning semantics in my data. Therefore, the quality of the BERTScore decreased.

5.2 BLEU score analysis

TABLE IX
Mean BLEU Score Metric With Std

Model	**Mean**	**Std**	**Mean parsed**	**Std parsed**
CodeT5+ 220M	3.697	2.992	2.437	2.368
CodeT5+ 770M	**4.737**	3.472	**3.291**	2.918
CodeT5+ with file attention	3.729	2.761	2.438	2.166
CodeT5+ with file attention single commit train	3.367	2.610	2.117	2.096
CodeT5+ with retrieval	4.409	1.985	-	-
JetBrains CodeT5	3.994	3.103	2.610	2.732

Table IX reports the BLEU metric for all models trained in my work. The performance of models with BLEU is correlated with the BERTScore. CodeT5+ with 770 million parameters shows the best results in the CommitChronicle test set and parsed data. Alterations in this metric were observed in the CodeT5+ with file attention, where its performance declined and became less than that of the JetBrains model. Given the configuration of CodeT5+ with file attention, where each batch consisted of only one commit, its performance was even lower than

that of CodeT5 220M, which served as its backbone model. An important detail about the statistics collected for the BLEU metric is the standard deviation of the data. The metric is too variable and therefore not robust. In some cases, the value of std is almost equal to the mean value. This instability in BLEU scores raises doubts about its relevance for model comparisons. Therefore, to make a fair comparison of the abilities of the models to imitate human-written commit messages, I need to collect another metric.

TABLE X
Mean B-Norm Score Metric With Std

Model	**Mean**	**Std**	**Mean parsed**	**Std parsed**
CodeT5+ 220M	0.129	0.035	0.099	0.031
CodeT5+ 770M	**0.157**	0.040	**0.128**	0.037
CodeT5+ with file attention	0.146	0.038	0.114	0.034
CodeT5+ with file attention single commit train	0.141	0.038	0.113	0.035
CodeT5+ with retrieval	0.151	0.022	-	-
JetBrains CodeT5	0.147	0.038	0.118	0.036

As a more robust version of the BLEU score, I took a B-norm metric, which is a normalized version of BLEU. This metric was used in the original work of JetBrains [4] and is considered the best version of BLEU specifically for the task of generating commit messages. From the perspective of this metric, a scaled version of CodeT5+ outperforms all other models. Every experiment involving additional modules for the base CodeT5+ model enhanced the metric. As described in the methodology section 3.10.1, the BLEU metric in the calculation focuses only on the exact match of n-grams in the generated text. Thus, from the point of this metric

the most important model to consider is CodeT5+ with the retrieval. The model with retrieval additionally gets the most similar commit from the database and the previous commit from the same repository to the input. Using this additional information model should implicitly learn how to generate the message in the same style. Therefore, the retrieval model should have better BLEU results than others, as it was trained to write the messages in the same words as humans with the help of additional human-written samples. The results of the table indicate that the retrieval model achieves the top 2 score with B-norm 0.151. Moreover, the variability of the metric for this model is considerably less compared to all other models. This makes this method the most robust for generating qualitative messages. Based on these findings, it can be inferred that the retrieval model was effectively trained and achieved the desirable results in the evaluation.

5.3 Generation length analysis

Original messages from the CommitChronicle dataset are short and concise. Therefore, after the fine-tuning the models should be able to describe all the code modifications in a short text. To access the ability of the models to do this, I measured the number of tokens in the models responses. Table XI summarizes the results of the collected statistics. From the table is visible that almost all the models have approximately identical lengths of the generated messages. The results for the base CodeT5+ are longer on average. This may mean that the model did not learn to construct the concise messages. The JetBrains-trained model produces more concise messages compared to other models. This could be attributed to the special tokens added to the input data and the output designed for my models.

TABLE XI
Mean Message Length With Std

Model	Mean	Std	Mean parsed	Std parsed
CodeT5+ 220M	13.268	6.298	12.598	5.594
CodeT5+ 770M	12.392	5.269	12.450	4.864
CodeT5+ with file attention	12.298	5.383	12.154	4.221
CodeT5+ with file attention single commit train	11.705	5.049	11.402	3.612
CodeT5+ with retrieval	12.272	5.410	-	-
JetBrains CodeT5	10.913	4.22	10.510	4.269

5.4 Models speed analysis

To access the model generation speed, I measured the time to generate the commit messages for a batch. Batched measurements provide a comprehensive view of the efficiency with which the model processes multiple inputs simultaneously. This measurement correlates with real-world usage scenarios where models often handle numerous tasks concurrently. This approach captures the model's ability to handle workload variations and optimize computational resources effectively. The batch for all models includes 20 samples, except for the retrieval model, where the batch consists of 64 sequences. The results of the measurement of the generation time are presented in Table XII The primary focus when evaluating the generation speed of the models is to explore the impact of additional modules added to the base version of CodeT5+ on the generation speed. As indicated in the table, the time to process a single batch did not increase too much for all the experiments with additional modules. The computational over-

TABLE XII
Mean Evaluation Time With Std

Model	Mean	Std	Mean parsed	Std parsed
CodeT5+ 220M	1.508	0.565	1.409	0.604
CodeT5+ 770M	2.919	0.873	2.756	1.057
CodeT5+ with file attention	1.626	0.421	1.353	0.459
CodeT5+ with file attention single commit train	1.461	0.413	1.168	3.451
CodeT5+ with retrieval	5.147	1.626	-	-
JetBrains CodeT5	**1.081**	0.300	**1.042**	0.314

head is less than 150 milliseconds for the batch of 20 samples. Considering the model with the best performance, JetBrains CodeT5 showed significantly faster batch processing. One of the reasons for this is the number of parameters, as the JetBrains model has only 220M of them. When comparing the JB model to the base code T5+, it shows superior performance due to the lower average number of tokens generated by the model. Two models considered in the table showed the result that is significantly different from all the others, the scaled version of CodeT5+ and the version with the retrieval. The problem of the retrieval model is it's bottleneck on the retrieval from the FAISS index. Big CodeT5+ performs worse because of its number of parameters. However, a duration of 2.9 seconds to generate messages for 20 commits is considered efficient, since a typical software developer takes about 15-20 seconds to compose one.

TABLE XIII
Performance for the specific programming language

	BERT_220M	BERT_770M	BLEU_220M	BLEU_770M
C	0.583	0.604	4.082	4.807
C#	0.575	0.595	2.065	3.077
C++	0.582	0.603	3.271	4.448
Dart	0.6	0.615	5.356	5.971
Elixir	0.583	0.612	4.234	5.883
Go	0.577	0.602	2.224	3.2
Groovy	0.607	0.636	5.79	7.829
Java	0.585	0.606	2.972	4.003
JavaScript	0.586	0.61	3.677	4.743
Kotlin	0.572	0.595	1.45	2.11
Nix	0.639	0.663	3.323	4.697
Objective-C	0.569	0.599	1.442	2.411
PHP	0.583	0.611	3.855	5.282
Python	0.586	0.609	4.143	5.334
Ruby	0.581	0.607	3.837	5.071
Rust	0.579	0.603	3.768	5.051
Shell	0.591	0.623	4.246	5.617
Smalltalk	0.543	0.571	1.047	1.857
Swift	0.579	0.603	2.457	3.5
TypeScript	0.589	0.611	3.95	4.878

5.5 Results for the specific programming languages

The CommitChronicle dataset collects the commits in the 20 popular programming languages. To access the performance for any specific language, I collected separate metrics with BLEU and BERTScore for any language. As models for comparison, I took the base version of the CodeT5+ model and its scaled version. The results of the metric collection are presented in Table XIII. The table demonstrates that the scaled version of CodeT5+ outperforms the base version in all programming languages. The worst results are achieved for the

Smaltalk programming language. The unique syntax of this language and the small number of examples in the training data make it difficult for the models to learn to extract the semantics from the source code. Considering the programming languages with which the model performed best, Nix shows the best results in terms of BERTScore. One of the possible reasons for this may be the declarative nature of this language. However, all conclusions about the quality of models in certain languages are just guesses. The number of examples for each language in the test set is insufficient to draw any conclusions.

5.6 Results discussion

After considering different aspects of the model performance and collecting the metrics results, I would like to finalize the results of my work and assess the effectiveness and outcomes of the experiments carried out.

5.6.1 Result of adding the special tokens

My experiment with special tokens involved changing the format of input and output sequences according to the format specified in Section 3.2.3. According to my initial idea, the model should better understand the format of code changes with special tokens, representing the start and the end of the modified code line. The result of this experiment can be observed by comparing the base CodeT5+ and the CodeT5 trained in [4]. These models were fine-tuned using the same training data and utilized the same set of training hyperparameters. The only difference between these approaches is that the model trained by JetBrains received the code changes in a raw format. According to the metrics statistics, the JetBrains

model works better for all cases. This leads to the conclusion that the addition of special tokens does not boost the model's understanding of the code changes. Furthermore, this modification leads to the degradation of the quality of generated commit messages.

5.6.2 Results of the model scaling

In modern deep learning research, it is a well-known fact that increasing the number of parameters in the transformer model increases its performance in various tasks. In my work, I experimented with scaling CodeT5+. For this purpose, I tested the version of the model with 770 million parameters and accessed performance improvements compared to the base version. The evaluation results indicate that this scaling enables the model to outperform all other experiments. The main drawback of this model is the increased inference time. The batch processing time nearly doubled after model scaling. However, even three seconds to process the batch of 20 samples is efficient enough to use this solution in real-world scenarios. Therefore, I can conclude that this experiment achieved the stated objective. Scaling the model took advantage of its ability to capture code modification semantics and improved the quality of generated commit messages.

5.6.3 Results of adding the retrieval module

The main goal of this experiment was to extend the ability of the model to adapt to the commit messaging style specific to a certain field of software development or a particular repository. For this purpose, I retrieved the previous commit message from the same repository and the most similar message from the database. The primary measure used to evaluate the model's adaptation to the

specific messaging style is the BLEU score. This metric quantifies the exact match of n-grams, thus indicating how well the style of the produced message matches the original one. The results of Section 5.2 show that the retrieval model outperforms all models except the CodeT5+ with 770 million parameters. Furthermore, the same situation holds for the BERTScore metric statistics described in Section 5.1. Therefore, it is evident that the model's training was effective, leading to improved performance. However, there is still room for improvement in this method. For example, there is still a bottleneck in the retrieval part of the model. Another possible drawback is that the retrieved messages are used only as raw text, however, there exist some advanced methods like RACE [10] where the retrieved context is used on the embeddings level.

5.6.4 Result of training file attention model with single commit

The main aim of this experiment was to investigate how batched training affects gradient flow through the specific module responsible for estimating the utility of files. While constructing the architecture of the file attention model presented in Section 4.4.1, I came up with the possible problem of gradient backpropagation. Due to the heterogeneous nature of commits, there may be a problem with properly training the MLP block, which calculates the dominance score among the commit files. To determine whether this issue affects the training of this architecture, I also conducted training on a model variant that included only a single commit per training batch. This approach ensures data homogeneity. By evaluating the performance of both model variants, I will be able to make a conclusive decision. By analyzing the results of the BERTScore and BLEU score, it is visible that the model trained in a batched format performs better than the

variant with a single commit. This leads to the conclusion that the batches do not break the gradient flow for the intermediate MLP block. The underperformance of the variant trained with a single commit could be caused by a lack of generality, as the model has seen only one sample per gradient step.

5.6.5 Result of adding the file attention module

Goal of this section is to indicate the overall performance of the model with a file attention mechanism. When comparing the file attention model with the base model, it outperforms it in all metrics. But since the file attention model was initially based on the fine-tuned CodeT5 +, this may be just because of the second epoch training. Therefore, this statistic is insufficient for a definitive conclusion. When comparing the file attention model with the CodeT5 trained in the Jetbrains work, the difference between them is statistically insignificant for all metrics. The base CodeT5 was trained for one epoch, and the file attention model was trained for one epoch as a base CodeT5+ model and one more epoch with a file attention mechanism added and a frozen encoder. Taking this into account, base CodeT5 outperforms the file attention module. One potential explanation could be the ineffective use of additional special tokens, and another could be the poor performance of the file attention module. In discussing the concept of the file attention mechanism, it was stated that the primary objective of this approach is to enhance the model's performance for commits involving multiple files. To access how well this method performed, I collected the BERTScore statistics for the commit sample data sets with several files. CommitChronicle dataset includes commits with up to 16 modified files. To have only big commits I filtered out the samples with less than 14 files changed in the commits and came up with

approximately 3000 samples. For the parsed data, I took the commits with more than six files changed to get nearly the same amount of data. BERTScore results for these data subsets are presented in Table XIV. The table indicates that, for the CommitChronicle subset, file attention outperforms the JetBrains CodeT5. For the parsed data, the situation is the opposite. Thus, a definitive conclusion cannot be drawn regarding the efficacy of the file attention model for large commits. Summarizing the evaluation of the file attention model, the outcomes of this experiment are unclear and require further research.

TABLE XIV
BERTScore for big commits

Experiment	**Mean Value**	**Std**	**Mean parsed**	**Std parsed**
codeT5+ 220M	0.559	0.105	0.584	0.130
codeT5+ 770M	0.587	0.113	0.614	0.143
codeT5+ with file attention	0.581	0.104	0.587	0.123
codeT5+ with file attention single commit train	0.569	0.102	0.583	0.127
JetBrains codeT5	0.577	0.108	0.602	0.139

Chapter 6

Conclusion

The purpose of this chapter is to summarize all the work done in my thesis. This chapter finalizes the results of all the experiments conducted. Furthermore, it underscores the broad impact of my research in the domain of automatic commit message generation. In addition, this chapter considers the possible extension of the experiments performed in this work.

6.1 Summary and main findings

Regarding the models training, my work includes three different experiments: (I) Scaling the base model to access the performance improvement, (II) Addition of the retrieval mechanism to the base model, (III) Addition of the file attention mechanism to the base model. The results of these experiments can be summarized in the following way. Scaling the model from 220 million to 770 million parameters improved the quality of the generated messages and demonstrated the best performance across all experiments. This modification of the model increased the average time to process the input; however, the time efficiency is

still at a level sufficient for effective utilization. Adding the retrieval mechanism to the base model also increased the processing time due to the bottleneck in the step of retrieving from the index and receiving a response from the GitHub API. However, this method helped the model to adapt to the style of the commit messages and improved the performance of the model. Analyzing the mechanism of file attention invented in the course of this work, this method has not lived up to initial expectations. Despite the degradation in efficiency and the more computational power spent training, this model achieves the same results as a base CodeT5 trained in another work. Initial assumptions about efficient management of large commits were not confirmed as there was no change in the distribution of the metric when analyzing the group of commits that included multiple files.

Another research question considered in my work is the specific format of the input and output sequences. More precisely, I tested special tokens that represent the logic of file names, modified lines, and commit messages. According to the original hypothesis, modifying the data in this way is expected to enhance the model's understanding of the semantics involved in code changes. During the fine-tuning, the model should learn how to construct a representative embedding for the special tokens, which will get the model's attention in the right direction. However, the evaluation results showed that this method leads to performance degradation of the model. Taking into account the fact that the training set is approximately three billion tokens, which is enough to train several special token representations, there is no problem with underfitting in this method. Therefore, the conclusion about this method is that the transformer architecture is powerful enough to learn the code modification representation without explicit instructions. In this scenario, special tokens are redundant elements that reduce the available space for code data within the context window.

One more novelty presented in my work is the set of metrics used to assess the performance of the models. In the section describing research gaps, a particular issue was highlighted: the absence of semantic metrics used to evaluate commit message generation models. To address this problem, my research incorporates the BERTScore metric, described in 3.10.2. The authors of this method in [13] state that this method outperforms the BLEU score and is more correlated with human judgement. In the course of my work, I ensured that the BERTScore correlates with the quality of the commit messages. To do this, I conducted an experiment by comparing the BERTScore of the generated answer and a random irrelevant commit message. The results of this experiment are shown in the Appendix A. This metric calculates the semantic similarity of tokens instead of the exact match for the BLEU score and, therefore, is more robust for the considered task.

All the points described above show the importance of this work in the sphere of commit messages generation task. In general, this research contributes valuable insights and methodologies to this field. This work highlights the importance of model scalability, semantic evaluation metrics, and the limitations of certain architecture modifications.

6.2 Future work

The work done leaves great room for further development of the presented methods. To address the shortcomings identified in this study, it is recommended to retrain all models without the use of special tokens. An additional possible direction for future research could involve altering the file attention mechanism. Given that the method did not yield satisfactory results in this study, it requires

a further in-depth analysis. There are also some possible improvements to the architecture of this model. For example, a combination of the file attention and retrieval models can be implemented. In this setup, the retrieved passages will be considered as separate entries of the commit and involved in the process of weighted averaging of embeddings. This approach could potentially improve the quality of the messages generated without the efficiency losses observed with model scaling.

Bibliography cited

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

[2] 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.

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

[4] 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.

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

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

[7] А. М. Елисеева and А. Шпильман, "Автодополнение сообщений к коммитам в среде разработки intellij idea,"

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

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

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

[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.

[26] A. Paszke, S. Gross, F. Massa, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[27] T. Wolf, L. Debut, V. Sanh, *et al.*, "Huggingface's transformers: State-of-the-art natural language processing," *arXiv preprint arXiv:1910.03771*, 2019.

[28] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with GPUs," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2019.

[29] Spotify. "Annoy: Approximate Nearest Neighbors in C++/Python optimized for memory usage and loading/saving to disk." (2023), [Online]. Available: https://github.com/spotify/annoy.

Appendix A

BERTScore analysis

To evaluate the effectiveness of BERTScore in accurately assessing message quality, I analyzed the responses produced by the model alongside random commit messages. As a dataset for this experiment I used the CommitChronicle validation set. As a model to generate the messages, I used the base version of CodeT5+ to generate the messages. As a base BERT model for the BERTScore, i used the default BERT base uncased to enhance the calculation speed. The results of the comparison are presented in Fig. A.1. Y axis of this plot shows the BERTScore of a certain sample, and the x axis represents the length of code changes in this sample in characters. As is visible from the plot, the results of the generated messages strongly outperform the irrelevant random commit messages from the dataset. Therefore, I can conclude that the metric correctly measures the quality of the generated commit messages even with the base model.

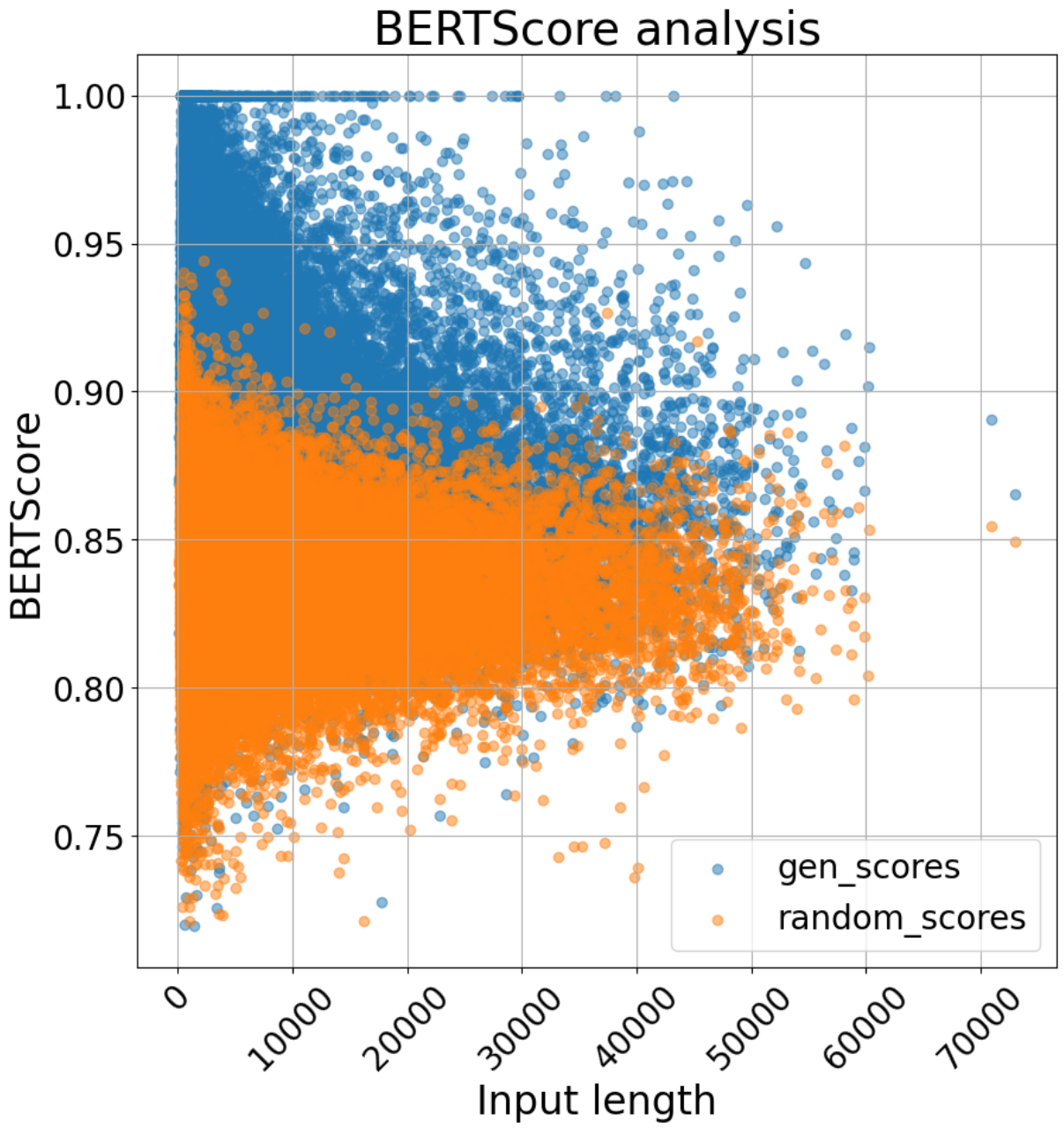

Fig. A.1. Comparing the BERTScore of generated messages and random ones.

Appendix B

Inference examples

In this section, I present the inference results of my models. As an example for the inference, I used the random commit from the official PyTorch repository. The source of this commit can be found in the following GitHub link. This commit changes the changes the content of some Dockefiles and shell scripts in the repository. It covers six different files and makes seven insertions and nine deletions. The inference results for all models and the true commit message for this commit are presented in Table XV. The source code for this commit is presented below.

```
.ci/docker/centos-rocm/Dockerfile
@@ -62,7 +62,7 @@ RUN if [ -n "${DB}" ]; then
bash ./install_db.sh; fi
 RUN rm install_db.sh
 ENV INSTALLED_DB ${DB}

-# (optional) Install vision packages like
OpenCV and ffmpeg
+# (optional) Install vision packages like
OpenCV
 ARG VISION
 COPY ./common/install_vision.sh
./common/cache_vision_models.sh
./common/common_utils.sh ./
 RUN if [ -n "${VISION}" ]; then bash
./install_vision.sh; fi
```

```
.ci/docker/common/install_vision.sh
@@ -5,8 +5,7 @@ set -ex
 install_ubuntu() {
   apt-get update
   apt-get install -y --no-install-recommends
\
-          libopencv-dev \
-          libavcodec-dev
+          libopencv-dev
```

```
    # Cleanup
    apt-get autoclean && apt-get clean
@@ -19,8 +18,7 @@ install_centos() {
    yum --enablerepo=extras install -y epel-
release

    yum install -y \
-       opencv-devel \
-       ffmpeg-devel
+       opencv-devel

    # Cleanup
    yum clean all
```

```
.ci/docker/ubuntu-cuda/Dockerfile
@@ -56,7 +56,7 @@ RUN if [ -n "${DB}" ]; then
bash ./install_db.sh; fi
 RUN rm install_db.sh
 ENV INSTALLED_DB ${DB}

-# (optional) Install vision packages like
OpenCV and ffmpeg
+# (optional) Install vision packages like
OpenCV
 ARG VISION
 COPY ./common/install_vision.sh
./common/cache_vision_models.sh
./common/common_utils.sh ./
```

```
 RUN if [ -n "${VISION}" ]; then bash
./install_vision.sh; fi
```

```
.ci/docker/ubuntu-rocm/Dockerfile
@@ -53,7 +53,7 @@ RUN if [ -n "${DB}" ]; then
bash ./install_db.sh; fi
 RUN rm install_db.sh
 ENV INSTALLED_DB ${DB}

-# (optional) Install vision packages like
OpenCV and ffmpeg
+# (optional) Install vision packages like
OpenCV
 ARG VISION
 COPY ./common/install_vision.sh
./common/cache_vision_models.sh
./common/common_utils.sh ./
 RUN if [ -n "${VISION}" ]; then bash
./install_vision.sh; fi
```

```
.ci/docker/ubuntu-xpu/Dockerfile
@@ -83,7 +83,7 @@ RUN if [ -n "${DB}" ]; then
bash ./install_db.sh; fi
 RUN rm install_db.sh
 ENV INSTALLED_DB ${DB}

-# (optional) Install vision packages like
```

```
OpenCV and ffmpeg
+# (optional) Install vision packages like
OpenCV
 ARG VISION
 COPY ./common/install_vision.sh
./common/cache_vision_models.sh
./common/common_utils.sh ./
 RUN if [ -n "${VISION}" ]; then bash
./install_vision.sh; fi
```

```
.ci/docker/ubuntu/Dockerfile
@@ -80,7 +80,7 @@ RUN if [ -n "${DB}" ]; then
bash ./install_db.sh; fi
 RUN rm install_db.sh
 ENV INSTALLED_DB ${DB}

-# (optional) Install vision packages like
OpenCV and ffmpeg
+# (optional) Install vision packages like
OpenCV
 ARG VISION
 COPY ./common/install_vision.sh
./common/cache_vision_models.sh
./common/common_utils.sh ./
 RUN if [ -n "${VISION}" ]; then bash
./install_vision.sh; fi
```

TABLE XV
Inference results

Model	**Message**
CodeT5+ 220M	"[CI] Install libopencv-dev and ffmpeg-devel in Ubuntu Dockerfiles"
CodeT5+ 770M	"[CI] Remove ffmpeg and libavcodec from docker images"
CodeT5+ with file attention	"ci: remove ffmpeg from docker image"
CodeT5+ with file attention single commit train	"ci: install ffmpeg and libopencv"
CodeT5+ with retrieval	"Remove ffmpeg from vision docker images"
JetBrains CodeT5	"Remove ffmpeg from Dockerfiles"
Original message	Remove FFMPEG from CI scripts (#125546) Because FFMPEG was solely used by Caffe2.