SHOULD BE REPLACED ON REQUIRED TITLE PAGE

Instruction

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

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

3. Put current year (last row).

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

Contents

List of Tables

List of Figures

Abstract

Background: The task of automatically generating commit messages is the perspective research topic. Writing concise commit messages is a crucial task in the software development process. They serve as a valuable tool for developers to track changes in the project and to collaborate effectively. Despite this, a significant proportion of commit messages in open source are under-informative. State-Of-The-Art approaches to commit message generation (CMG) show promising results. However, most current research proposes methods that involve training encoder-decoder transformer-based models for this task. And one of the limitations of current CMG methods is the inability to handle the full context of code changes for the big commits, due to the limited context window of the model.

Methods: The given task can be formulated as code2text language modelling. And the most popular method for this at the moment is fine-tuning transformer-based encoder-only Large Language Models (LLM) for the given task. So in my work I have fine-tuned LLM, pre-trained it on the massive code data and compared this solution with the existing ones. And to deal with the problem of long context, I used methods that allow to extract useful information from the long data sequences and pass it to the model input instead of the raw code context.

Results: Using encoder-only model with much more parameters improved the result of generating commit messages. Methods for extracting embeddings from the code and using them instead of the raw code give an improvement in terms of the meaning of the result. However, the computation time for commit message generation increased due to the size of the model.

Contribution and applicability: The methods proposed in this work increase the performance in the CMG task, but at the same time the time efficiency of the solution became worse. And the problem of finding a balance between running time and prediction quality requires future work.

Chapter 1

Introduction

1.1 What is commit message

1.2 Why it is important to wrtite descriptive commit messages?

1.3 Why the system for automatic commit messages generation might be helpfull?

Chapter 2

Literature Review

2.1 Introduction

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

2.2 Background of the Deep Learning in the NLP

Deep learning techniques in NLP have brought remarkable progress and widespread adoption. Deep learning models, particularly those based on transformer architectures [1], have revolutionized the field since 2018. Models like **BERT** (Bidirectional Encoder Representations from Transformers) [2], **GPT** (Generative Pre-trained Transformer) [3], and their derivatives have set new standards for NLP tasks, achieving state-of-the-art results for most of the NLP tasks. Transfer learning is a dominant paradigm, where pre-trained models are fine-tuned for specific tasks, reducing the need for large labeled datasets. These models have made NLP more accessible, enabling researchers and practitioners to build language understanding systems with relatively modest computational resources. Despite these advancements, several challenges remain: mitigating biases, improving the model's understanding of context and semantics, and hallucinations in the model's answers. Nevertheless, deep learning in NLP has revolutionized this research area, offering unprecedented capabilities for processing and generating natural language.

2.3 Approaches to Commit Message Generation

Almost all modern commit message generation solutions use **Language modeling** to get the result.
In this approach, CMG is represented as a code2text **Neural Machine Translation** (NMT) task. In this setting, the input and output of the model are considered as a sequence of tokens. Input tokens are represented as $X = (x_1, x_2, \ldots x_n)$, which is information about the certain commit. Output tokens are the generated commit

message $Y = (y_1, y_2, \ldots, y_m)$. Language Modeling model is training to learn a distribution of conditional probabilities of the Commit Message tokens, given the information about the commit $P(y_1, \ldots y_m \mid x_1, \ldots x_n)$. Modern CMG solutions mostly use encoder-decoder neural network architecture [4]–[7]. In this approach, the encoder part transforms input X into a hidden representation $h \in \mathbb{R}^d$, and the decoder then, using this h generates an output Y. At this point, the structure of the model may vary. The approach might based on the pure transformer model [6], [7] or **Graph Neural Network** (GNN) combined with the transformer model [8]. Some approaches combine the transformer model with the retrieval mechanism [9], and this helps to increase the performance of the method.

2.3.1 Transformer based models

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

2.3.2 Graph Neural Network approach

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

2.3.3 Transformer-based architecture with the retrieval

The transformer-based architecture with retrieval mechanisms is a sophisticated approach that enhances CMG by combining the strengths of the transformer model with retrieval techniques. In this context, the retrieval mechanism allows the model to access and incorporate relevant information from a database of previous commits and corresponding commit messages. This approach is particularly valuable because it addresses the challenge of generating commit messages that are not only informative but also consistent with past practices and project-specific terminology. The retrieval mechanism can be used to identify and incorporate snippets of text or phrases from historical commit messages that are relevant to the current code changes. This helps in producing commit messages that maintain consistency in terminology and style, which is essential for codebase documentation and understanding. The transformer-based architecture, with its ability to model code-related information effectively, is well-suited for this task. It can combine the retrieved information with its understanding of the code changes to generate commit messages that are both contextually rich and in line with the development team's conventions. In summary, the combination of a transformer-based architecture with retrieval mechanisms in CMG results in commit messages that are not only accurate and informative but also consistent with the project's history and established practices, contributing to more effective code documentation and collaboration.

2.4 Datasets for CMG

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

- The **CommitBERT$_{\mathbf{data}}$** dataset, presented in [7], encompasses a collection of 345,000 code modification instances paired with corresponding commit messages. These instances come from 52,000 repositories representing six distinct programming languages (Python, PHP, Go, Java, JavaScript, and Ruby), parsed from Github. It is worth noticing that this dataset exclusively contains information about the altered code segments and does not include any supplementary details regarding the associated commit actions. Unlike other datasets, CommitBERT$_{\text{data}}$ is specifically tailored to focus only on modified lines within a commit, potentially resulting in the omission of crucial contextual information that may be present in the unaltered sections of the code. Other strict filterings on the data: limitation on the number of files in the code changes; usage of only the first line of the original commit message as a prediction target; collecting the only commit messages, beginning from the verb.

- **ATOM$_{\textbf{data}}$** - Liu *et al.* in [5] 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 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 [10] 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 [10], but also significant alterations to the original commit history and stringent data filtering. In the CommitChronnicle dataset, the majority of the filters previously applied in prior datasets are eliminated, resulting in increased data diversity and enhanced dataset generality. Notably, CommitChronnicle encompasses 10.7 million commits after the application of various filtration steps. Additionally, this dataset incorporates supplementary information about commits, facilitating the tracking of commit history for specific users and projects during the training process. Leveraging this supplementary information, it becomes possible to generate commit messages that exhibit greater alignment with the project's historical context. In conclusion, CommitChronnicle stands out as the most comprehensive and diverse dataset available for the CMG task, successfully addressing the issues present in previous datasets while offering a significantly larger volume of data.

2.5 Evaluation metrics for CMG

Numerous metrics are commonly used to assess the performance of CMG methods. Foremost among these is BLEU [11], a metric widely adopted in the evaluation of machine translation models. Given that CMG can be conceptualized as a code-to-text task analogous to a neural machine translation, BLEU, and related metrics find relevance in evaluating CMG methods. B-Norm, a variant of BLEU,

has been demonstrated to align most closely with human judgments regarding the quality of commit messages, as observed by Tao *et al.* [10]. BLEU, reliant on the precision concerning shared n-grams between generated and reference sequences, has been a staple in prior commit message generation studies.
Nevertheless, BLEU presents limitations in assessing the quality of the generated text. This metric lacks sensitivity to the semantic nuances embedded within generated text and is incapable of capturing the semantic similarity between the generated output and reference text. To address these limitations, more advanced metrics rooted in neural networks have been developed. Notably, BERTScore [12], which leverages the BERT model, exhibits heightened sensitivity to the semantic nuances of generated text. Despite these advantages, BERTScore has seen relatively limited adoption in the evaluation of CMG methods.

2.6 Research gaps

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

2.6.1 Limited context window

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

2.6.2 Stalling performance on the out-of-filter data

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

2.6.3 Lack of the evaluation metrics

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

2.6.4 Lack of the research on CMG using the LLM

Current SOTA for most of the NLP tasks uses the encoder-based transformer models. For example - CodeLLaMa [13] - is the current SOTA for the code generation task. However, LLM usage for the CMG task does not have enough exploration. Authors in [4] attempt to approach the CMG using the ChatGPT with a well-chosen prompt. However, this approach performs worse than the fine-tined encoder-decoder models. The results are the following since ChatGPT is not trained for this task. It is a significant limitation, as the LLMs are the current SOTA for most NLP tasks.

2.7 Conclusion

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

Chapter 3

Methodology

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

3.1 Experiment design

The main objective of this study is to develop a system for automatically generating commit messages that rival state-of-the-art (SOTA) methods. To achieve this, I first need to construct my dataset. The next step involves analyzing this data and that from other studies, crucial for understanding the data structure and extracting relevant statistics. This analysis also involves filtering to ensure data quality. Subsequently, my research will examine the metrics used in Neural Machine Translation (NMT) tasks, requiring an understanding of their calculation methods and underlying intuition. Finally, I will train various models, providing a detailed description of each model's architecture and the rationale for its expected success in the commit message generation (CMG) task.

3.2 Data retrieval

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

3.2.1 Retrieval process

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

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

3.2.2 Retrieved components

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

TABLE 3.1
Commit Data Attributes

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

3.2.3 Structure of the data

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

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

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

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

3.3 Collected data analysis and filtering

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

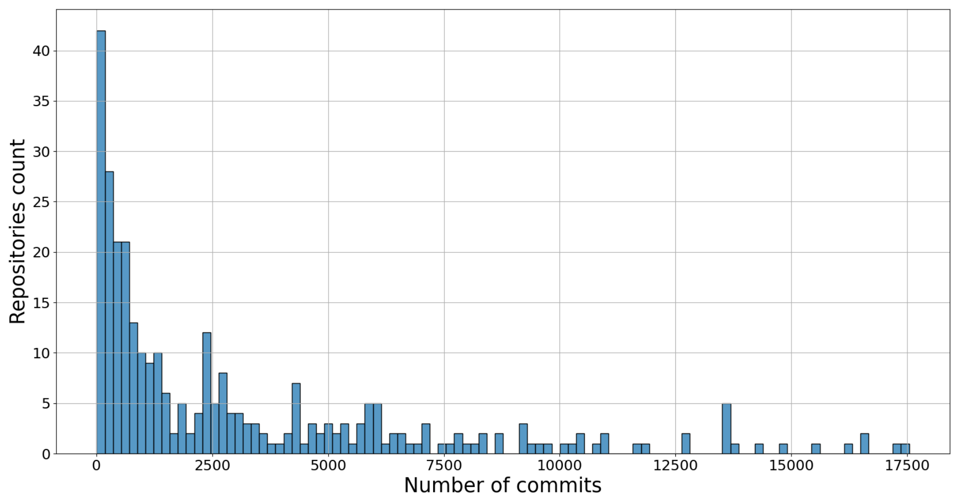

Fig. 3.1. Distribution of the number of commits in the repositories.

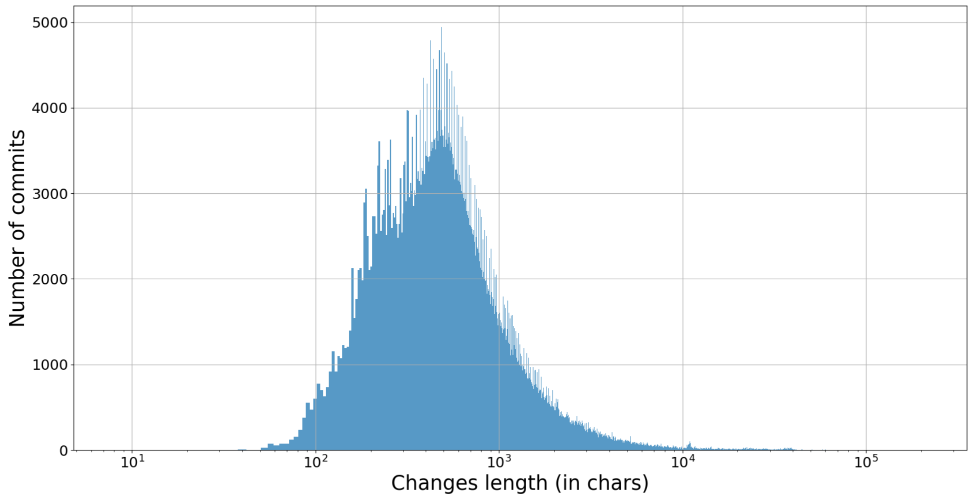

Fig. 3.2. Distribution length of the commits in the dataset.

Fig 3.2 represents the distribution of the code changes in commits. From this distribution, we can see that the average length of the code changes is ~5000 characters. From this, we can conclude that the code changes on average in a very long sequence. Most of the modern Deep Learning models are transformer-based and therefore experience significant degradation of the performance due to the quadratic complexity of the self-attention mechanism as mentioned in [14]. Also in this step, I decided to trim my dataset, to have only samples with less than 6000 characters. With this truncation, we lose only 15% of the data but have much shorter sequences on average.

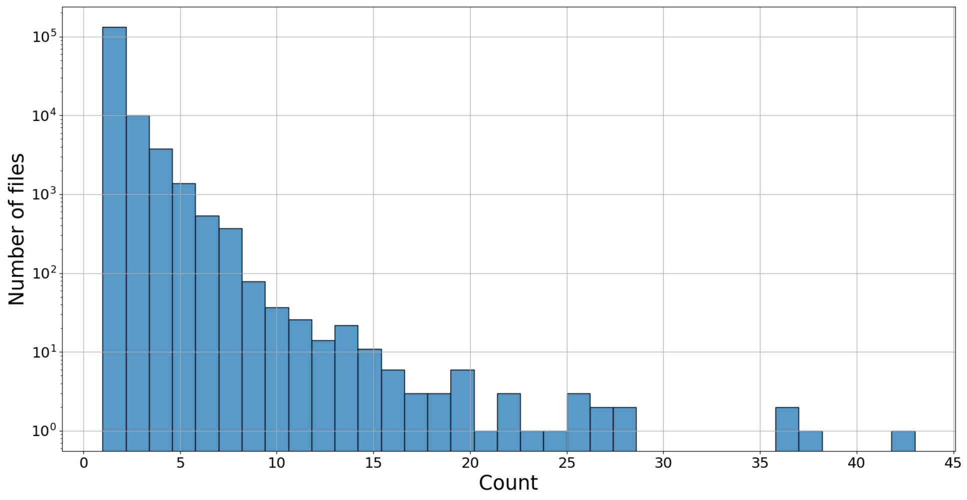

Fig. 3.3. Distribution of the number of files in commits.

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

One more insight, that we can get from the collected data is the most popular beginning word of commit messages. This part of the data analysis is similar to the one from [7]. The authors of this work state that commit typically states with the common verbs. From the collected dataset I extracted the 10 most used starting words for commits and got the results, presented in the table 3.2. From this table, it's visible that there exist some common words for starting a commit message. Even among these 10 words, there are several repetitive ones with modified writing or form. These words cover ~35% of the total number of samples in the dataset.

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

TABLE 3.2
Distribution of commit messages

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

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

- Samples without Python code changes.

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

3.4 Analysis of datasets from other works

In the literature review chapter (2) I described most of the existing datasets for the commit messages generation task. But in this section, I would like to focus only on the CommitChronicle dataset, presented in [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 lots 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: Authors of this dataset dropped examples out of the $[5\%, 95\%]$ percentile range of the length of code difference and samples with more than 16 files changed. They also get rid of the commits with non-ASCII symbols in the message, merge and revert commits, and samples with trivial messages [15], which does not provide any useful information about code updates.

For this dataset, I performed the same analysis to compare it with my data. Figure 3.4 displays the distribution of files changed within a single commit. Comparing CommitChronicle with my dataset in this regard we can observe, that the dataset presented in [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 code changes length in characters. From this side, the statistics are similar, so we can say that datasets are the same from this perspective. The distribution of the number of commits among the repositories from the CommitChronicle is shown in Fig 3.6. In this figure, I displayed only information for the repositories with less than 5000 commits to make the plot more representable. From this side, CommitChronicle has more balanced data, as the distribution is smoother, but it is mostly connected with the much fewer repositories in my dataset. The last thing I would like to mention about the statistics of the CommitChronicle is the most popular first words of commit messages. They are presented in the Table 3.3. As was mentioned above, merge commits were filtered out by the authors, and in all other respects, the results are the same as for my dataset. So datasets are identical from this side.

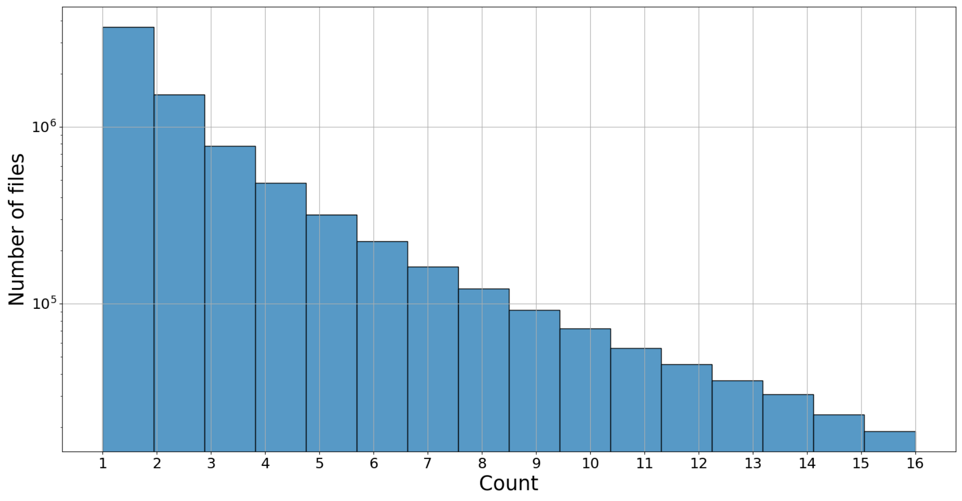

Fig. 3.4. Distribution of the number of files in commits in CommitChronicle.

TABLE 3.3
Distribution of commit messages

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

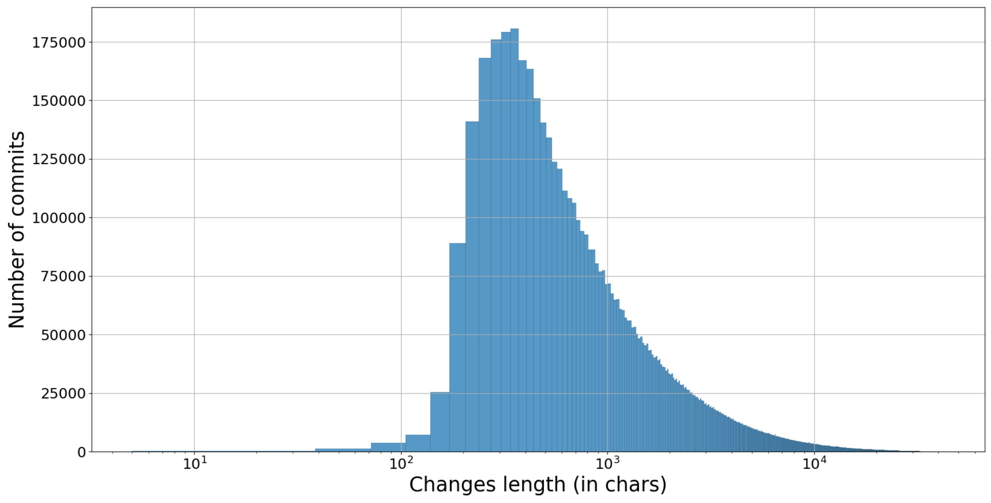

Fig. 3.5. Distribution of length of commits in CommitChronicle.

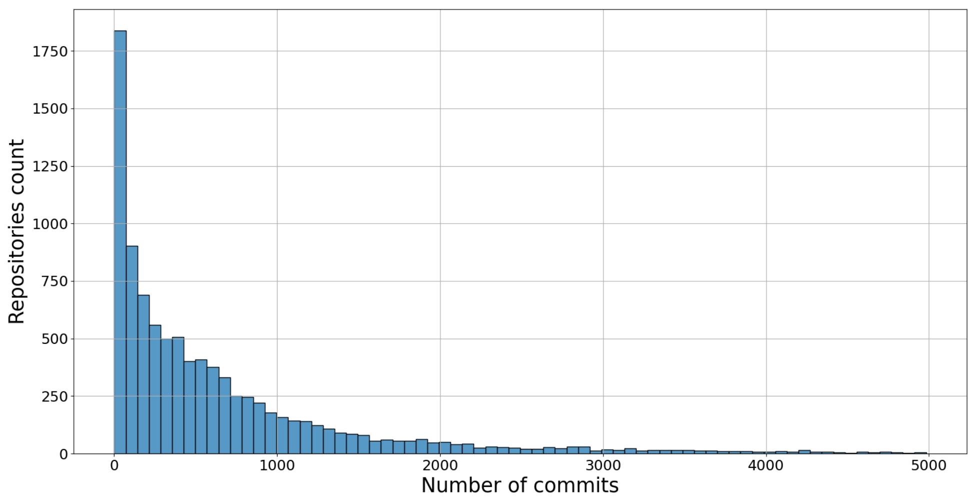

Fig. 3.6. Number of commits in the repositories in CommitChronicle.

3.5 Conclusion about the data

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

3.6 CodeT5+ training

My first attempt at approaching the task of commit message generation is straightforward. I will consider this task as a simple NMT (Neural Machine Translation) problem. In this setting, the model aims to translate the input code into a natural language description of the code changes. The 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 a 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 that time, the authors of CodeT5 have introduced an updated version of the model, called CodeT5+ [16]. This new model outperforms CodeT5 in performance across the majority of code-related benchmarks. So, I decided to train CodeT5+ on CMG task as the first experiment in my work. In this setting, I will use 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 the input and output as a sequence of numerical tokens. In Paragraph 3.2.3, I described the specific format of the input and output data for my task. Therefore, before training the model, it is necessary to add special tokens to the tokenizer vocabulary. The model will learn the embedded representation of these special tokens during training. This step will aid the model in better understanding the structure of code changes.

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

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

3.7 Experiments with larger CodeT5+

To assess the impact of scaling the model in terms of parameters on the quality of generated commits, I trained the CodeT5+ model with 770 million parameters. The differences between the base version and this model are presented in Table 3.4. From this table, it is clear that the distinctions between these models are primarily in the hidden representations and the number of blocks in both the encoder and decoder parts. A larger model version does not process longer sequences but has better internal representation, potentially enabling the generation of better commit messages. The training objective I utilized and the data format are identical to those used for training the base CodeT5+. This experiment demonstrated how scaling the depth of the neural network and improving representation can affect model performance. Additionally, this phase allowed me to determine whether 220 million parameters are sufficient for constructing commit messages that are relatively simple sequences. The final aspect requiring analysis in this section is how this scaling influences the model's inference time since efficiency in terms of time is as crucial as message quality when considering real-world applications.

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

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

3.8 CodeT5+ with retrieval components training

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

3.8.1 Retrieval approaches

To mitigate issue with a message style, works like [9], [5], and [17] adopt additinal retrieval module to the standard endcoder-decoder architecture. These methods are mostly uses the database of commits and corresponding commit message. And then new commit is given as an input to the model, this method firstly use encoder to get a embedded vector representation of code modifications. This representation is further used to find the most similar code changes from the dataset, and get both modifications and commit message. This additional information is then passed to the model to get the result. Analysis of these method shows, that retrieval mechanism leverages the results of commit generation.

One more way to use the historical data in generation of commit messages is to retrieve previous commits from the same repository, or previous commits of the same person. This method way used in the [4], and showed improvement of model performance. If previous methods are mostly used to improve the understanding of code changes, then this historical data retrieval method is used only for message style adaptation.

3.8.2 My experiment with retrieval

For my experiment, I decided on the method combining retrieving the most similar code modifications and getting historical data from the previous commits. My experiment is close to the one provided in [4] with a combination of the RACE method from [9] and the history of messages added to the model input. The main difference between my approach and the RACE is that the RACE leverage the commit generation by passing the combination of the input code changes with the most similar one from the database. At the same time, my approach is utilizing only the message from the most similar commit to adapt to the style of the commit message of the same code theme. Additionally, I pass to the model input history samples of the messages from the same repository. In the end, my input to the model have the following format:

$<$commit_msg$>$ most simmilar message $<$/commit_msg$>$

$<$commit_msg$>$ previous message $<$/commit_msg$>$

code modifications

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

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

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

3.9 Possible way to resolve limited context window problem

Another problem with generating commit messages is the limited context window of the transformer-based models. As stated in [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 the input. This amount is insufficient to handle a large commit that includes changes of multiple files. One possible way to mitigate this limitation is to use a larger model. For example, the mistral model presented in [18] has a maximum context length of 8192 tokens. However, the problem with this solution is that this model has 7 billion parameters, compared with 220 million in CodeT5+. CMG is a relatively simple task since generated messages are not that varied. According to my analysis of the data from 3.3 in general, most of the commit messages have almost the same structure. Thus, the inference speed of the used model should be high. And with the larger model, I will lose the performance in terms of the speed too much. That is why I decided to handle big commits more efficiently in another way.

3.9.1 Efficient handling of big commits

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

3.9.2 Model architecture

3.10 Metrics to evaluate CMG results

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

3.10.1 BLEU score description

Bilingual evaluation understudity (**Bleu**) is the metric invented in 2002 and presented at [11]. The original goal of this metric was to evaluate the quality of machine translation. The task of generating commit messages from code modifications is considered a translation from code to natural language. The original commit message in this setup is the reference translation. A comparison of the generated commit message with the original one matches the idea of the BLEU. Therefore, the results of this metric properly reflect the quality of the generated commit message.

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

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

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

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

From the explanation above, it is clear that the formula for BLEU strongly depends on configured parameters, n-grams N, and importance weights w_n. That makes this metric variative, and therefore, it is hard to compare the results among different works. To overcome this problem, the authors of [19] presented the package SacreBLEU. This Python script helps calculate the BLEU score in a unified way to have reproducible results.

Analyzing the BLEU score specifically for the commit messages generation task, results from [10] show that in this specific task related to code, it is more representative to use a normalized version of BLEU called B-Norm. The difference from the original method is to first convert both prediction and reference to lowercase, as it is not as important for this task, as for translation. The second difference is to add one to both the numerator and denominator of 3.2 to add smoothness to it.

In my experiments, I decided to use both SacreBLEU and B-Norm. This will make the results of the evaluation more extensive and fair. Counting SacreBLEU will help me compare my results with other approaches to solving CMG tasks. B-Norm is mostly used in my work to compare 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 B-Norm, I used the script 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 at [12]. This metric is based on the BERT [2] - encoder-only transformer-based model. It was shown that the BERT model achieves outstanding results in encoding text and getting meaningful hidden representation. Utilizing this fact, BERTScore gets embeddings for each token of both generated text and reference. Due to the transformer-based architecture, the representation of the tokens may differ depending on the surrounding context. This feature makes this metric more representative than the BLEU score, which considers only matching n-grams, *i.e.*, based only on the syntax similarity between generated text and reference. At the same time, BERTScore depends on the context and meaning of the word, therefore can catch the situation, then we have generated text with the same meaning, but written in other words.

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

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

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

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

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

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

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

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

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

The last important point about the BERTScore metric is the sensitivity to the base BERT model. As was written before, the most important step in calculating this metric is to get representative embedding for each of the tokens with BERT. And the choice for this is crucial to get an informative metric. For my experiments, I got deberta-xlarge-mnli from [20]. The reason I choose this exact model is that authors of the evaluate[1] library, have analyzed[1] 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.
[1]Documentation for BERTScore from evaluate

Chapter 4

Implementation

4.1 Vanila CodeT5+ training details

4.2 CodeT5+ with retrieval training results

4.3 CodeT5+ with file attention training results

4.4 Analysis of BERTscore metric

Chapter 5

Evaluation and Discussion

5.1 Models performance analysis

5.2 Models speed analysis

Chapter 6

Conclusion

6.1 Future work

Bibliography cited

[1] 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, 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.

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

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

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

[10] 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.

[11] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

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

[20] 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.