ABSTRACT

Code review is an essential part to software development lifecycle since it aims at guaranteeing the quality of codes. Modern code review activities necessitate developers viewing, understanding and even running the programs to assess logic, functionality, latency, style and other factors. It turns out that developers have to spend far too much time reviewing the code of their peers. Accordingly, it is in significant demand to automate the code review process. In this research, we focus on utilizing pre-training techniques for the tasks in the code review scenario. We collect a large-scale dataset of real-world code changes and code reviews from open-source projects in nine of the most popular programming languages. To better understand code diffs and reviews, we propose CodeReviewer, a pre-trained model that utilizes four pre-training tasks tailored specifically for the code review scenario. To evaluate our model, we focus on three key tasks related to code review activities, including code change quality estimation, review comment generation and code refinement. Furthermore, we establish a high-quality benchmark dataset based on our collected data for these three tasks and conduct comprehensive experiments on it. The experimental results demonstrate that our model outperforms the previous state-of-the-art pre-training approaches in all tasks. Further analysis show that our proposed pre-training tasks and the multilingual pre-training dataset benefit the model on the understanding of code changes and reviews.

CCS CONCEPTS

- Software and its engineering → Automatic programming.

KEYWORDS

Code review, deep learning, datasets, pre-training

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ESEC/FSE ’22, November 14–18, 2022, Singapore, Singapore
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9413-0/22/11...$15.00
https://doi.org/10.1145/3540250.3549081

1 INTRODUCTION

Code review, a process of manually inspecting source code by teammates other than the code author, is recognized as a critical part of the software development lifecycle [13]. Studies have shown the huge benefits of code reviews [1, 3, 32, 33]. Compared with the traditional code review process formalized by Fagan in 1976 [12], which avoids introducing errors and defects but is cumbersome, modern code review activities involve fewer formal requirements and aim at fully understanding code changes [4]. Given its benefits, code review has been widely adopted in both open-source and industrial projects. As shown by Yang et al. [45], in open-source projects, such as Qt, there are ten thousand reviews taking place every month (~22k reviews in case of Qt). However, it’s not free
to take all advantages of code reviews. To make a correct judgement whether to accept the pull request, developers have to put much time and effort into code reviewing, checking code from every aspect, including logic, functionality, complexity, code style, documentation, etc. For example, Yang et al. [45] report that only 1,437 reviewers have given more than 1M reviews in 4 years in project Qt. Accordingly, it is in significant demand to automate the code review process.

Many researchers have explored ways to assist reviewers and committers (i.e., code authors) to reduce their workload in the code review process, such as recommending the best reviewer [9, 37], recommending or generating the possible review comments [18, 40, 41] and even revising the code before submitting it for review [40]. This paper shares the same goal to automate some specific tasks related to code review. We target three scenarios from both reviewers’ and committers’ perspectives. The first is called code change quality estimation, aiming to predict whether a code diff needs a review comment. It assists reviewers to pick a code diff chunk that might have issues among a large number of code chunks. The second task is review generation which can dramatically reduce the time cost for the reviewer. The last scenario is for committers, which is code refinement according to the previous code and reviewer’s comment.

Motivated by the wide adoption of deep learning (DL) techniques for software engineering and the rapid development of pre-training techniques, we leverage pre-training for automating code review activities. Recently, researchers have proposed many pre-trained models on source code [7, 14, 17, 43]. However, they can hardly handle the code review process. Chen et al. [7] propose Codex, a large scale pre-trained model based on GPT-3 [6]. Codex has been proven to be the most powerful model for code generation tasks. Since it is trained on the source code files in raw format, it knows very little about code review. We demonstrate that it cannot generate any meaningful comment in the review generation task based on our evaluation. Tufano et al. [40] attempt to use a pre-trained model for code review automation. However, their pre-training dataset is collected from Stack Overflow and CodeSearchNet [22], which is not directly related to code review process. Furthermore, they take the normal form of source code as the model input, ignoring the special format of code diff that can help the model better understand code changes.

To tackle the problems, we pre-train CodeReviewer, an encoder-decoder transformer model. Different from Tufano et al. [40]’s work, CodeReviewer is pre-trained on a large dataset in code review scenario, consisting of code diff hunks and code review comments. We propose four pre-training tasks, including diff tag prediction, denoising code diff, denoising review comment, and review comment generation to make CodeReviewer better understand code diffs and generate review comments. Figure 1 shows the overview of the workflow of our CodeReviewer.

To make CodeReviewer more suitable for the code review process, we build a large-scale dataset of code changes and corresponding review comments. Such code review data is collected from GitHub pull requests in high-quality projects with high stars covering nine of the most popular programming languages. Using GitHub code review data, we create pre-training and benchmark datasets for three tasks related to the code review process. To the best of our knowledge, it is the largest multilingual code review dataset with complete information of code changes and reviews.

We further evaluate our CodeReviewer model on our benchmark dataset. Compare with previous state-of-the-art (SOTA) generative models for code, experimental results show that our model outperforms previous works on all three tasks. Further analysis proves the effectiveness of our proposed pre-training tasks and the multilingual high-quality dataset.

To summarize, the contributions of this work are:

- The first pre-trained model that takes code diffs as input in the code review scenario.
- Novel pre-training tasks for better code changes understanding and generation.
- A large-scale code review related pre-training dataset and a high-quality benchmark dataset for evaluation in nine programming languages.
- A comprehensive evaluation of CodeReviewer and previous SOTA models. Our dataset, code, and model are released 1.

2 CODE REVIEW AUTOMATION TASKS

In this section, we give a formulated description of the code review process, provide the formal definitions of three key tasks abstracted from the code review process, and clarify the data format of the tasks. The employed symbols are listed in Table 1.

2.1 Code Review

In the code review process, contributors (code change author, \(P_C\)) update the source code to accomplish new features or fix bugs in the old version. The original code and updated code are denoted as \(C_0\) and \(C_1\). Once the code changes (\(D : C_0 \rightarrow C_1\)) are ready for review, the author of these changes creates a pull request to start the code review process. Other peer contributors (reviewers, \(P_R\)) will review the code changes and provide comments or suggestions (\(R_{nl}\)) on them if necessary. Based on the comments, the author makes revisions and provides a newer version of code \(C_2\). We call the activities up to now as a review round. Note that the review process is not finished yet. The reviewers can further give suggestions on the revisions. And the author may also revise the code again. After a few review rounds, a pull request will finally be approved (changes are merged into the main branch) or discarded.

There are usually multiple commits in a pull request, and each commit may be related to multiple code changes across different files. Thus, to model the whole review process is difficult and challenging [19]. As an early step, we focus on automating the review process of a single commit. In the review round, two distinct roles are involved – contributor \(P_C\) who commits a code change (\(D : C_0 \rightarrow C_1\)), and reviewer \(P_R\) who gives advice and comments (\(R_{nl}\)). The goal of code review automation is to help reduce their workload. To that end, we focus on three specific code review automation tasks in both scenarios of different roles. An overview of all three tasks is shown in figure 2.

---

1https://github.com/microsoft/CodeBERT/tree/master/CodeReviewer
2.2 Code Change Quality Estimation

Code change quality estimation task is to predict whether a code change is high-quality and ready to be accepted in the review process. This is a binary classification task, i.e., $Y = \{0, 1\}$. The input is a code change, i.e., $X = \{D(C_0, C_1)\}$. As stated before, there are usually lots of changes across different source code files in a pull request. It takes reviewers a large amount of time to review all the changes. But it always turns out that most of the changes are minor and don’t need a comment or suggestion. To improve the efficiency of code review, we define and advance to automating the code change quality estimation task. Given the estimates, reviewers can give questionable code changes a higher priority to review and pay more attention to them, saving time and effort. Contributors can also leverage the estimates to improve low-quality code changes before submitting them to reviewers.

2.3 Code Review Generation

Code review generation is a sequence generation task. The output is a predicted review comment, i.e., $Y = \{w_1, \cdots, w_n\}$ where $w_i$ is a natural language word and $n \in \mathbb{N}$ is the length of review comment. The input is still a code change, i.e., $X = \{D(C_0, C_1)\}$, with its context. In some previous works [18, 40, 41], researchers use the changed code as input but not the code diff, without taking into account that review comments have to focus on the changed part. It’s not recommended for reviewers to give suggestions to the code context which has not been revised. Considering the naturalness of software[21], language models can capture general statistical properties of programs due to programs tend to be repetitive and predictable. In the code review process, there are also common issues in some code changes. For example, reviewers often write “This method should be private” to suggest the contributor add a private decorator to a method. This gives us a chance to learn the general code review patterns and generate comments automatically to lighten the burden of reviewers. In the real-world scenario, the
model generates few review comment candidates and the reviewers may choose an ideal one from them, free from writing comments manually.

2.4 Code Refinement
In the code refinement task, the model takes as input both the original code $C_1$ written by the contributor and the review comment $R_{nl}$ from the reviewer, and targets to generate the revised version code $C_2$ implementing the requirements mentioned in $R_{nl}$. Different from the code refinement task in other software development scenarios where only source code is taken as input [39], we design specifically for the code review process to utilize the review comment as guidance for refining. Code refinement is a task designed for assisting contributors. When they open pull requests and get review comments as feedback, the model helps to revise the submitted code automatically based on the comments.

2.5 Data Format
All three tasks are related to source code or code changes. In the code review process, there are usually multiple related files and functions [19]. A source code file or a function/method is too large to process, because there may be multiple review comments and code revisions spreading across a file or function/method. Instead, we define the inputs of the three tasks at diff hunk level.

In pull requests, the original version and revised version of code are shown in a special format: diff (Figure 2). The diff file is generated by comparing two files before and after the change. The diff tool finds sequences of lines common to both files, interspersed with groups of differing lines called hunks. Specifically, a diff hunk is a sequence of source code lines surrounded by a few unchanged lines. Diff hunk includes the lines deleted from the original file and added to the new file. The diff format is frequently used by programmers to show the changes in the code clearly. Moreover, the diff format is an efficient representation for code changes because the unchanged lines occur only once, and the changes are aligned with "$-" and "$+" tags at the begging of each line. In the first two tasks, we formulate the input as a diff hunk representing the code change. In the code refinement task, we extract the input lines ($C_1$) and output lines ($C_2$) from the revision diff hunk.

3 CODE REVIEW DATASET
Nowadays, many developers use collaborative software development platforms like GitHub\(^3\) and Gerrit\(^4\) not only to share their code but also to perform code review activities. GitHub platform allows contributors to make all these review details publicly available, including the exact content of code changes, review comments and their authors, and happening time in each pull request, which makes it possible to collect and analyze code review data from open-source projects. Hence, we build our CodeReview dataset based on pull requests data collected from open-source projects, covering nine programming languages, in GitHub.

3.1 Project Selection
To ensure the quality of our dataset, we collect pull request data from publicly available high-quality open-source repositories. We first sort the projects by “popularity” as indicated by the number of stars. To improve the generalizability of the models trained on our dataset, we collect projects in nine of the most popular programming languages in GitHub, including C, C++, C#, Go, Java, JavaScript, PHP, Python, and Ruby. Then, we keep the top 10,000 projects for each of the nine programming languages and remove those projects that do not explicitly permit the re-distribution of their data. To be specific, all repositories with popular licenses such as Apache-2.0 license and MIT license are kept. If the contributors write in the license file that they allow for re-distribution of their data, their projects are also collected. In order to ensure the quality of the project and acquire code review data as much as possible, we further sort the projects by the number of pull requests and filter out projects with less than 1,500 pull requests. This process keeps only active projects with many contributors and removes repositories that are forked from other projects as the pull request number is not inherited. Then we start crawling pull request information of the projects.

3.2 Review Data Collection
We use GitHub REST API to collect pull requests data from the projects. GitHub API offers an HTTP request-based API for accessing repository information. By sending HTTP requests to GitHub, we can access the branches, commits, pull requests, code diff, review comments, etc in JSON format conveniently. Many researchers have used GitHub API to collect and analyze pull request data in their work [20, 35]. To advance the research of code review, Heumüller et al. [20] develop the ETCR infrastructure for mining code review datasets from any GitHub project which practices pull-request-based development. ETCR tool can be set up requiring only a GitHub API key and repository name. The ETCR uses a Kotlin-based web crawler and works in four stages to collect data of pull requests, commits, comments and source files via the GitHub API. All the data is stored in a relational database and thus can be accessed conveniently. We first use the ETCR tool [20] to collect meta-information of pull requests and review comments, including the git commit hash and changed file name related to the comments. Using the meta-information, we can further query GitHub API to get code changes (including the original file, new file, and the code diff) corresponding to the review comments. These code changes and comments make up our CodeReview dataset.

3.3 Dataset Construction
The pre-training dataset is a set of code changes with or without review comments. However, it requires further data processing to build datasets for the three downstream tasks.

\(^3\)https://github.com/
\(^4\)https://www.gerritcodereview.com/

Apache-2.0, GPL-3.0, MIT, BSD-2.0, BSD-3.0, BSL-1.0, GPL-2.0 license, etc.
We use the CodeReview data to build the pre-training dataset and Appendix (attached as supplementary material). All these samples are removed from the dataset because modeling the Transformer (T5) model [31]. The CodeReviewer model consists of 12 Transformer encoder layers and 12 decoder layers. There are 12 attention heads in each layer and the hidden size is 768. The total parameter size of the model is 223M.

We initialize CodeReview with the parameters of CodeT5 [43]. We further pre-train the model with our four pre-training tasks. Once pre-trained, CodeReviewer is finetuned and evaluated on the downstream tasks respectively.

### 4.2 Input-Output Representations

CodeReviewer takes different inputs and outputs for different tasks. For code understanding tasks or comment generation tasks, the model takes code diff as input. For the code refinement task, the model takes as input both the original source code and the review comment.

Following the standard way of input processing in Transformer, the input is treated as a token sequence. We use the same RoBERTa [26] tokenizer as CodeT5 to split the source code and review comment to the tokens. A special token [CLS] is prepended to the sequence, forming the input as \{[CLS], c_1, c_2, \ldots, c_n\} where \(c_i\) is source code token and \(n\) is the sequence length. To help our model understand the diff format better, the special line tags “-” and “+” in diff file indicating line deletion and line insertion are replaced with special tokens [DEL] and [ADD]. We also insert a [KEEP] before each unchanged line. When there are both source code tokens and review comment tokens in the input, a [MSG] token is inserted to separate the two sequences. Thus the input is \{[CLS], c_1, c_2, \ldots, c_n, [MSG], w_1, w_2, \ldots, w_m\}.

The model outputs consist of two components: token representations from the encoder and generated token sequence from the decoder. For classification tasks, the representation of [CLS] token is used to generate the predictions. For sequence generation tasks, the decoder outputs the predicted token sequence.

### 4.3 Pre-training Tasks

The key challenge of automating code review activities is to understand code changes and capture the relationship between code changes and corresponding review comments. Therefore, we design four pre-training tasks to improve the abilities of CodeReviewer.

#### 4.3.1 Diff Tag Prediction
As mentioned in Section 2.5, diff is a special format containing rich information in code changes. Compared with encoding both the original and new version code, the diff format prevents duplicates of unchanged lines. Thus, learning code diff as a special format of source code is critical for the model in the code review scenario.

We use Diff Tag Prediction (DTP) task to equip the model with the ability to understand the special line tags in code diff. Note that we replace “+” and “-” tags at the beginning of diff lines by special tags. We set up a dictionary of these special tokens and replace the special tokens in the input by these tokens at the beginning of the diff lines. For the diff file indicating line deletion and line insertion, we replace the special tokens in the input by these tokens at the beginning of the diff lines.

#### 4.3.2 Code Refinement

We use Code Refinement task to equip the model with the ability to understand the interactions between multiple code changes. In the code review scenario, the code change can be associated with multiple code changes. This is because the commit of the code change is updated in the later commit, and the reviewer comments on the code diff as a special format of source code is critical for the model to understand the code changes.

We set up a dictionary of these special tokens and replace the special tokens in the input by these tokens at the beginning of the diff lines. For the diff file indicating line deletion and line insertion, we replace the special tokens in the input by these tokens at the beginning of the diff lines.

#### 4.3.3 Code Understanding

We use Code Understanding task to equip the model with the ability to understand the interactions between multiple code changes. In the code review scenario, the code change can be associated with multiple code changes. This is because the commit of the code change is updated in the later commit, and the reviewer comments on the code diff as a special format of source code is critical for the model to understand the code changes.

We set up a dictionary of these special tokens and replace the special tokens in the input by these tokens at the beginning of the diff lines. For the diff file indicating line deletion and line insertion, we replace the special tokens in the input by these tokens at the beginning of the diff lines.

#### 4.3.4 Review Comment Generation

We use Review Comment Generation task to equip the model with the ability to understand the interactions between multiple code changes. In the code review scenario, the code change can be associated with multiple code changes. This is because the commit of the code change is updated in the later commit, and the reviewer comments on the code diff as a special format of source code is critical for the model to understand the code changes.

We set up a dictionary of these special tokens and replace the special tokens in the input by these tokens at the beginning of the diff lines. For the diff file indicating line deletion and line insertion, we replace the special tokens in the input by these tokens at the beginning of the diff lines.
We randomly mask spans with 20% corrupted rate and train the (DCD) and denoising review comment (DRC). In the DCD task, we 15% and predicts these spans. By learning to predict masked spans, in the input and is required to recover the masked spans. Specifically, in DRC task, the model is given a corrupted review comment as input and asked to generate the review comment written by the human reviewer. We use the negative log-likelihood loss:

\[ L_{DCD} = \sum_{i=1}^{k} - \log P_0(c_i | e^{\text{mask}}, c_{<t}) \]  

where \( c_{<t} \) is the comment generated so far. \( e^{\text{mask}}, w^{\text{mask}} \) are masked code diff and masked review comment, and \( c_{<t}, w_{<t} \) are the span sequence of code diff and review comment generated so far.

4.3.3 Review Comment Generation. In each pre-training task mentioned above, only one modal is involved, which means that the model only learns to understand source code or review comments in one task. However, the most challenging part of code review is to capture the relationship between code changes and review comments. To equip our model with this capability, we utilize bi-modal data (code changes in programming languages and related review comments in natural language) in the pre-training task. For generalizability, we use a simple conditional generation task: review comment generation (RCG). In RCG, the model is given a code change as input and asked to generate the review comment written by the human reviewer. We use the negative log-likelihood loss:

\[ L_{RCG} = \sum_{i=1}^{k} - \log P(w_i | c, \ w_{<t}) \]  

where \( c \) is code change and \( w_{<t} \) is the comment generated so far.

4.4 Fine-Tuning

We group all downstream tasks into classification tasks and generation tasks. For classification tasks such as code change quality estimation, we only use the pre-trained encoder. The representation in the last layer of the special token [CLS] at the beginning \( h_0 \) is
fed into a linear classifier to produce the prediction. For generation tasks such as code refinement, the entire pre-trained encoder-decoder model is used. A standard token negative log-likelihood loss is leveraged to optimize the probability of the target sequence.

5 STUDY DESIGN

To investigate the performance of our CodeReviewer model on the code review tasks, we perform a large-scale study to answer the following research questions (RQ):

RQ1: How does CodeReviewer perform on the code change quality estimation task? We provide as input to CodeReviewer a code change $D_1 : C_0 \rightarrow C_1$, and ask the model to give a binary prediction whether the code change needs a review.

RQ3: How does CodeReviewer perform on the code refinement task? Given a few lines of source code submitted for code review and the feedback from reviewers written in natural language, the model is required to refine the submitted code to implement the requirements of the review comments.

In RQ1-RQ3, we evaluate CodeReviewer on different code review tasks. To assess the contribution of each pre-training task and our multilingual dataset, we further investigate RQ4 and RQ5.

RQ4: What role does each pre-training task play in CodeReviewer? In RQ1-RQ3, the evaluated CodeReviewer model is pre-trained on all four pre-training tasks before being applied to the downstream tasks. We remove the pre-training tasks one by one and evaluate the resulting models again to expose the influence of each pre-training task on different downstream tasks.

RQ5: Can multilingual dataset benefit model performance on understanding single programming language? Existing research show that multilingual training dataset can benefit for model performance compared with monolingual dataset on neural machine translation and code translation [8, 48], especially for low-resource languages. We evaluate the performance of CodeReviewer pre-trained and fine-tuned on monolingual dataset and compare it with the model trained on the full dataset to show the influence of multilingual dataset.

Dataset We build our pre-training dataset and 3 downstream task datasets as described in Section 3.3. Table 2 summarizes the statistics of the pre-training dataset. For benchmark datasets, details are shown in Table 3.

5.1 Baseline Models

To demonstrate the superiority of our multilingual code review related pre-training dataset and carefully designed pre-training tasks, we compare our CodeReviewer model with three baselines, including a state-of-the-art (SOTA) model architecture Transformer [42] trained from scratch and two pre-trained models: T5 for code review [40] and CodeT5 [43].

Table 2: Statistics of pre-training dataset.

| Language | Meta Info | Data Size | w/o comment | w/ comment |
|----------|-----------|-----------|-------------|------------|
| Project | PRs† | Data Size | w/o comment | w/ comment |
| Python | 195 | 1,451k | 72.8G | 887k | 518k |
| Java | 175 | 1,073k | 54.8G | 876k | 467k |
| Go | 146 | 951k | 40.4G | 728k | 410k |
| C++ | 133 | 999k | 82.1G | 474k | 202k |
| JavaScript | 194 | 1,354k | 30.6G | 425k | 293k |
| C | 77 | 441k | 135.4G | 292k | 110k |
| C# | 77 | 463k | 28.2G | 324k | 19k |
| PHP | 92 | 574k | 16.0G | 215k | 157k |
| Ruby | 72 | 626k | 3.8G | 90k | 120k |

Total 1,161 7,933k 463.2G 4,311k 2,481k

† Pull request numbers of the projects in total.

Table 3: Statistics of benchmark datasets.

| Dataset | Train # | Valid # | Test # | LOC |
|---------|---------|---------|--------|-----|
| Quality Estimation | ~266k | ~31k | ~31k | ~11M |
| Review Comment Generation | ~118k | ~10k | ~10k | ~1.8M |
| Code Refinement | ~150k | ~13k | ~13k | ~1.3M |

Transformer. Transformer [42] is a SOTA model architecture for many classification and generation tasks. The key component of Transformer is the multi-head attention module and the parameterized linear transformation layers. We use the same model setting as CodeT5-base, with a 12-layer encoder and a 12-layer decoder.

T5 for code review. Tufano et al. [40] attempt to use pre-trained model for code review automation. They pre-train a small version of T5 model with the denoising objective proposed by Raffel et al. [31] on their own dataset including Stack Overflow dumps and CodeSearchNet [22]. The model consists of 61M parameters, with 6 layers for both encoder and decoder. We refer to this model with T5 for convenience later.

CodeT5-base. CodeT5 is a SOTA unified pre-trained encoder-decoder model for code understanding and generation tasks proposed by Wang et al. [43]. They propose to pre-train CodeT5 with identifier-aware denoising tasks and bimodal dual generation objective, which makes CodeT5 the SOTA model for multiple code-related downstream tasks including code summarization, code generation, code translation and code refinement. The base version of CodeT5 consists of 12 encoder layers and decoder layers with the parameter size 220M. We use CodeT5 to refer to this model.

5.2 Evaluation Metrics

We provide a brief description of the evaluation metrics used for the three downstream tasks in this section.

Code change quality estimation. It is a binary classification task. We use accuracy, precision, recall, and F1 to evaluate the model predictions. Note that when computing the later 3 metrics, the code changes with issues (requires for comments and updates) are treated as the positive class.

Review comment generation. We compute the BLEU (Bilingual Evaluation Understudy) [30] score of the predictions. BLEU is
widely used to assess the quality of automatically generated text in generation tasks such as machine translation and code generation. We use the BLEU-4 variant, which computes the overlap of $n$-grams between the generated text and the reference text with $n$ from 1 up to 4. As review comments are diverse and non-unique, we further apply human evaluation on generated comments from two perspectives: information and relevance. ❶ Information: in this metric, we evaluate how informative the comment is for the contributor to revise the source code. Comments like “declare this variable as private” are more informative than those like “why do we need this?”; ❷ Relevance: in this metric, we evaluate to what extent the review comment is related to the corresponding code change. The comments point out the issues in the code changes will get high scores, while those not related or disobey the logic in the code changes should get low scores. The comments are labeled with a 1-5 score for each metric. We describe in detail of the two metrics in the Appendix.

**Code refinement.** For this task, we compute the BLEU score between generated code and target code and the exact match (EM) rate. BLEU only evaluates the similarity between the prediction and the target, while a small change in source code can result in compile error and execution error. So exact match is a more important metric in this task. Only if a prediction is exactly the same as the target, will it be considered as correct.

### 5.3 Implementation Details

We implement our model with the popular deep learning development framework PyTorch $^5$ and the python package transformers developed by HuggingFace $^6$. We pre-train our CodeReviewer model with 2 DGX-2 servers with 16 NVIDIA V100-32G GPUs on each server. The learning rate and batch size in the pre-training stage are set to 0.0002 and 768. We use AdamW optimizer with linear warmup to optimize the model for 250k steps. The warmup step number is set to 2500. When fine-tuning our CodeReviewer and baseline models on the three downstream tasks, we use batch size 72 and learning rate 0.0003. Each experiment on downstream tasks is performed on a server with 4 V100 GPUs. When evaluating the review comment generation task and the code refinement task, we use beam search with size 10.

### 6 RESULTS ANALYSIS

In this section, we answer each research question based on our experiment results. We first focus on three research questions concerning our model’s performance in each task and how it compares to other state-of-the-art baseline models. We further demonstrate the effects of our pre-training tasks and multilingual dataset.

#### 6.1 RQ1: Performance on Code Change Quality Estimation

Table 4 shows the results on the code change quality estimation task. From the table, we can see that whether the baselines are trained from scratch or pre-trained models, CodeReviewer outperforms them significantly on all four metrics. Specifically, our CodeReviewer model improves F1 and accuracy by 8.24% and 7.07% compared with T5. The improvement over CodeT5 is also over/about 7%, which demonstrates that our pre-training tasks help CodeReviewer understand code changes better. Besides, the performance of Transformer trained from scratch is inferior to the other three models, indicating the importance of pre-training.

#### 6.2 RQ2: Performance on Review Generation

Table 5 shows the results on the review generation task. CodeReviewer produces a higher BLEU score than the baseline models. However, the BLEU score of our model is still lower than 10, indicating it is a hard task. As mentioned before, review comments are diverse and non-unique. Referring to the example illustrated in Figure 5, our model prediction conveys similar intent as the ground truth. But their words differ greatly so that the BLEU score is lower than Codex. To investigate the models’ performance better, we perform a human evaluation on their predictions.

We firstly choose 300 random samples in Java and Python from the test set and manually select 100 of them with high-quality code changes and review comments. For the selected samples, we invite 6 professional sophisticated programmers to score the reference comment and generated comments of each model as described in Section 5.2. Results are listed in Table 5. Our model improves the information score and relevance score of the generated comments by about 20% relatively compared with the baseline models, which means comments generated by CodeReviewer are more useful and adequate. The performance of T5 is inferior to the other models, including the Transformer baseline. Because the parameter size of T5 (61M) is about 1/4 of the other models (223M). We conjecture that it is too challenging for a small model to generate the review comments.

In this task, we also conduct a qualitative analysis of the world’s largest language model on source code, Codex [7], which has been proven to be the most powerful code generation model and capable of zero-shot task transfer. Since we don’t get access to the model and can’t fine-tune it, we alternately use GitHub Copilot [15] which is powered by Codex. By providing several code diff and review comment pairs as prompt, we find that Codex can’t generate any meaningful reviews but copying comments in the given examples. Figure 5 shows two examples of outputs generated by different models including Codex.
public class DockerOptions {
    for (int i = 0; i < maxContainerCount; i++) {
        node.add(caps, new DockerSessionFactory(clientFactory, docker, image, caps));
    }
    LOG.info(String.format(
        "Mapping %s to docker image %s %d times", caps, name,
    )
    LOG.finest(String.format(
        "Mapping %s to docker image %s %d times", caps, name,
    )
    public class MockExecutorLoader implements ExecutorLoader {
        public void unassignExecutor(int executionId) throws ExecutorManagerException {
            executionExecutorMapping.remove(executionId);
        }
    }
    public class DockerOptions {
        for (int i = 0; i < maxContainerCount; i++) {
            node.add(caps, new DockerSessionFactory(clientFactory, docker, image, caps));
        }
    }
}

public class MockExecutorLoader implements ExecutorLoader {
    public void unassignExecutor(int executionId) throws ExecutorManagerException {
        executionExecutorMapping.remove(executionId);
    }
    @Override
    public List<ExecutableFlow> fetchRecentlyFinishedFlows(long lifeTimeMs) throws ExecutorManagerException {
        return null;
    }
}

public class DockerOptions {
    for (int i = 0; i < maxContainerCount; i++) {
        node.add(caps, new DockerSessionFactory(clientFactory, docker, image, caps));
    }
    LOG.info(String.format(
        "Mapping %s to docker image %s %d times", caps, name,
    )
    LOG.finest(String.format(
        "Mapping %s to docker image %s %d times", caps, name,
    )
    public class MockExecutorLoader implements ExecutorLoader {
        public void unassignExecutor(int executionId) throws ExecutorManagerException {
            executionExecutorMapping.remove(executionId);
        }
    }
    public class DockerOptions {
        for (int i = 0; i < maxContainerCount; i++) {
            node.add(caps, new DockerSessionFactory(clientFactory, docker, image, caps));
        }
    }
}

Review comment: flip the conditions, null check should be done before
T5:
try {
    wallpaperManager.setBitmap(bitmap);
    ViewUtil.showLongToast(context, context.getString(R.string.wallpaper_set_successfully));
    if (progressDialog.isShowing() && progressDialog != null) {
        progressDialog.dismiss();
    }
} catch (IOException e) { ... }

CodeT5:
try {
    wallpaperManager.setBitmap(bitmap);
    ViewUtil.showLongToast(context, context.getString(R.string.wallpaper_set_successfully));
    if (progressDialog.isShowing()) {
        progressDialog.dismiss();
    }
} catch (IOException e) { ... }

CodeReviewer: return empty list instead?

Figure 5: Two examples of the review generation task. The Codex output is obtained by Copilot.

Figure 6: An example of the code refinement task. To make clear, we show the code diff between the inputs and model outputs instead of their original form.

6.3 RQ3: Performance on Code Refinement
Table 6 shows the results on the code refinement task. The Naïve-Copy method directly copies the input code as the refinement result. It produces a decent BLEU score but 0.00 Exact Match (EM).

CodeReviewer successfully generates the repaired code exactly the same as ground truth for more than 30% cases, which is two times as the result of T5 and 25% more than CodeT5 relatively, demonstrating the superior ability of CodeReviewer to understand the
We demonstrate the superior performance of CodeReviewer on Table 5: Results on review comment generation. The perfect scores of two metrics in human evaluation set are both 5.

| Model (Layers #) | Test set | Human evaluation set |
|------------------|----------|-----------------------|
|                  | BLEU     | Information | Relevance |
| Transformer (12) | 4.76     | 3.08        | 2.50      |
| T5 (6)           | 4.39     | 2.54        | 1.62      |
| CodeT5 (12)      | 4.83     | 3.03        | 2.40      |
| CodeReviewer (12)| 5.32     | 3.60        | 3.20      |

Table 6: Results on code refinement.

| Model (Layers #) | BLEU | EM |
|------------------|------|----|
| NaïveCopy        | 58.75| 0.00 |
| T5 (6)           | 77.03| 15.08|
| CodeT5 (12)      | 80.82| 24.41|
| CodeReviewer (12)| 82.61| 30.32|

Table 7: Ablation study on code change quality estimation.

| Model            | Precision | Recall | F1  | Accuracy |
|------------------|-----------|--------|-----|----------|
| CodeReviewer     | 78.60     | 65.63  | 71.53| 73.89    |
| w/o CMT          | 77.14     | 65.35  | 71.34| 73.35    |
| w/o DTP          | 79.59     | 62.64  | 70.10| 73.29    |
| w/o DCD          | 79.87     | 60.94  | 69.13| 72.80    |

Since pre-trained models can address several downstream tasks by fine-tuning, previous works target at different scopes of code-related tasks with their models. Kanade et al. [23] pre-train CuBERT in a Python code corpus mostly for classification tasks like variable-misuse classification, wrong binary operator, etc. CodeBERT [14] and GraphCodeBERT [17] are bi-directional transformer models pre-trained on NL-PL pairs in six programming languages, the latter introduces new pre-training tasks designed for source code data flow. Both models have shown effectiveness on code-text crossing tasks like NL code search and code summarization, and other code understanding tasks like code clone detection and defect detection thanks to bi-directional attention mechanism. CodeGPT [27], GPT-C [36] and Codex [7] focus on generative tasks like code completion and code generation because they are pre-trained with decoder-only transformer architecture. As for encoder-decoder models like PLBART [2] and CodeT5 [43] and unified models like UniXcoder

### 6.5 RQ5: Influence of Multilingual Dataset

In the previous experiments, CodeReviewer is trained on the datasets consisting of nine programming languages. To investigate whether multilingual datasets help our model better understand a single programming language, we build monolingual datasets for Java, C# and Ruby languages, respectively. Java represents popular languages and Ruby represents the low-resource languages as shown in Table 2. For each of the three languages, we pre-train and fine-tune the CodeReviewer on the monolingual datasets and compare the performance on the code change quality estimation task with the original CodeReviewer pre-trained and fine-tuned on full multilingual datasets. The results are listed in Table 8.

The multilingual CodeReviewer outperforms the three monolingual models consistently, improving the accuracy by 2.32% and the F1 score by 1.10% on average. We conclude that our multilingual dataset benefits the CodeReviewer for understanding specific languages significantly. This reveals the superiority of our multilingual dataset over the datasets for single programming language. It also proves the broad applicability of CodeReviewer in different programming languages.

### 7 RELATED WORKS

#### 7.1 Pre-training for Code-related Tasks

Deep Learning techniques have been widely adopted in software engineering research [10, 44]. In recent years, motivated by the great impact of pre-training technique in natural language (NL) processing area [6, 11, 24, 31], researchers have made attempts to investigate whether pre-trained models for programming language (PL) can further facilitate software development.

Since pre-trained models can address several downstream tasks by fine-tuning, previous works target at different scopes of code-related tasks with their models. Kanade et al. [23] pre-train CuBERT in a Python code corpus mostly for classification tasks like variable-misuse classification, wrong binary operator, etc. CodeBERT [14] and GraphCodeBERT [17] are bi-directional transformer models pre-trained on NL-PL pairs in six programming languages, the latter introduces new pre-training tasks designed for source code data flow. Both models have shown effectiveness on code-text crossing tasks like NL code search and code summarization, and other code understanding tasks like code clone detection and defect detection thanks to bi-directional attention mechanism. CodeGPT [27], GPT-C [36] and Codex [7] focus on generative tasks like code completion and code generation because they are pre-trained with decoder-only transformer architecture. As for encoder-decoder models like PLBART [2] and CodeT5 [43] and unified models like UniXcoder
As indicated by previous empirical studies, code review is an important part of the software development lifecycle and involves a significant amount of effort and time of reviewers [5, 33]. Researchers are paying more attention to automating code review activities, including reviewer recommendation [38, 47], comment location prediction [19, 34], review comment recommendation [18, 35] and code refinement [41].

Thongtanunam et al. [38] reveal that 4%-30% of reviews have code reviewer assignment problems. Thus, they propose a file location-based tool RevFinder to recommend appropriate code reviewers. To tackle the same problem, Zanjani et al. [47] design the system chRev which utilizes code review histories to recommend reviewers for a code change. While these researchers aim at improving code review at an early stage, others are devoted to solving the more challenging tasks during the code review process. Shi et al. [34] propose the DACE framework based on CNN and LSTM to predict whether a hunk in the code change will be accepted by the reviewers. Hellendoorn et al. [19] use the Transformer architecture to solve this task. Furthermore, they also attempt to capture the relation of different hunks in a pull request by encoding each hunk and computing attention scores across diff hunks to fuse the information. Li et al. [25] formalize automatic code review as a multi-instance learning task, in which each hunk is an instance and the target is to predict whether a pull request will be accepted.

Other researchers focus on tasks related to review comments. To save the time reviewers spent on writing reviews related to common issues such as coding style and documentations, Gupta and Sundaresan [18] propose the LSTM-based model DeepMem. DeepMem learns the relations between code changes and review comments, and recommends review comments automatically based on existing code reviews and code changes. Siow et al. [35] also recommend code reviews in a retrieval-based manner. They propose an LSTM-based multi-level embedding attentional model CORE aiming at capturing the semantic information in both source code and reviews. Tufano et al. [41] use deep learning techniques to automate a different task in code review. They train a Transformer model to revise the contributors’ code to implement the requirements from the review comments. In the previous work, the researchers usually train a small model for a specific task in the code review process. Differently, we propose a large model pre-trained on four general pre-training tasks for code review, producing superior performance across three different code review tasks.

### 7.2 Automating Code Review Activities

As indicated by previous empirical studies, code review is an important part of the software development lifecycle and involves a significant amount of effort and time of reviewers [5, 33]. Researchers are paying more attention to automating code review activities, including reviewer recommendation [38, 47], comment location prediction [19, 34], review comment recommendation [18, 35] and code refinement [41].

Thongtanunam et al. [38] reveal that 4%-30% of reviews have code reviewer assignment problems. Thus, they propose a file location-based tool RevFinder to recommend appropriate code reviewers. To tackle the same problem, Zanjani et al. [47] design the system chRev which utilizes code review histories to recommend reviewers for a code change. While these researchers aim at improving code review at an early stage, others are devoted to solving the more challenging tasks during the code review process. Shi et al. [34] propose the DACE framework based on CNN and LSTM to predict whether a hunk in the code change will be accepted by the reviewers. Hellendoorn et al. [19] use the Transformer architecture to solve this task. Furthermore, they also attempt to capture the relation of different hunks in a pull request by encoding each hunk and computing attention scores across diff hunks to fuse the information. Li et al. [25] formalize automatic code review as a multi-instance learning task, in which each hunk is an instance and the target is to predict whether a pull request will be accepted.

Other researchers focus on tasks related to review comments. To save the time reviewers spent on writing reviews related to common issues such as coding style and documentations, Gupta and Sundaresan [18] propose the LSTM-based model DeepMem. DeepMem learns the relations between code changes and review comments, and recommends review comments automatically based on existing code reviews and code changes. Siow et al. [35] also recommend code reviews in a retrieval-based manner. They propose an LSTM-based multi-level embedding attentional model CORE aiming at capturing the semantic information in both source code and reviews. Tufano et al. [41] use deep learning techniques to automate a different task in code review. They train a Transformer model to revise the contributors’ code to implement the requirements from the review comments. In the previous work, the researchers usually train a small model for a specific task in the code review process. Differently, we propose a large model pre-trained on four general pre-training tasks for code review, producing superior performance across three different code review tasks.

### 7.3 Code Review Datasets

To advance the research of automating code reviews activities, researchers also make effort to collect and publish code review datasets. Tufano et al. [41] create two datasets with 17k abstracted code changes for predicting method-level code changes. The Code Review Open Platform (CROP) by Paixao et al. [29] is a dataset based on specific Gerrit projects consisting of 507k comments. The dataset from Mukadam et al. [28] has fewer comments (106k) and provides only metadata of source code. Yang et al. [46] propose the dataset with the largest number of comments. But they also provide only metadata of source code. Compared with them, we propose the largest, multilingual code review dataset with completed information collected from over 1k GitHub repositories with more than 7.9M pull requests in total.

### 8 Threats to Validity

**Threats to internal validity** relate to the roles played by the model architecture and hyper-parameters setting. We do a small-scale grid search on learning rate and batch size settings, leaving other hyper-parameters the same as those in CodeT5 [43]. It is expected that more hyper-parameter tuning would bring more improvements.

**Threats to external validity** are mostly related to the dataset we collect and use in this paper. Since the dataset is collected from GitHub, it is built upon only open-source projects, not industrial projects. Besides, code review is often not done by a single reviewer but multiple reviewers, and they could give different comments from different perspectives. But we only collect a single comment for a code change, leading a bias in our dataset.

**Threats to construct validity** include the rationality of evaluation metrics. We argue that the BLEU score which is widely used in text generation tasks is not suitable for the review generation task. So we conduct a human annotation to better evaluate the results.

### 9 Conclusion

In this paper, we focus on pre-training techniques for automating code review activities. We start by formulating three tasks in the code review process with the code change format. We collect and organize a large-scale dataset from GitHub for code review pre-training and a benchmark for evaluation on the three tasks. It is the largest dataset in code review scenario, covering nine of the most popular programming languages in GitHub. Based on this, we introduce CodeReviewer, a transformer-based encoder-decoder model pre-trained on our dataset with four designed pre-training tasks for code review. The experimental results demonstrate that our model outperforms the state-of-the-art models pre-trained with source code in all three tasks.

### References

[1] A Frank Ackerman, Priscilla J Fowler, and Robert G Ebenau. 1984. Software inspections and the industrial production of software. In *Proc. of a symposium on Software validation: inspection-testing-verification-alternatives*. 13–40.

[2] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2655–2668.
[42] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In NIPS. 5998–6008.

[43] Yue Wang, Wenshi Wang, Shafiq R. Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. CoRR abs/2109.00859 (2021).

[44] Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, and Denys Poshyvanyk. 2022. A Systematic Literature Review on the Use of Deep Learning in Software Engineering Research. ACM Trans. Softw. Eng. Methodol. 31, 2, Article 32 (mar 2022), 58 pages. https://doi.org/10.1145/3485275

[45] Xin Yang, Raula Gaikovina Kula, Norihiro Yoshida, and Hajimu Iida. 2016. Mining the modern code review repositories: a dataset of people, process and product. In Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016, Miryung Kim, Romain Robbes, and Christian Bird (Eds.). ACM, 460–463. https://doi.org/10.1145/2901739.2903504

[46] Xin Yang, Raula Gaikovina Kula, Norihiro Yoshida, and Hajimu Iida. 2016. Mining the Modern Code Review Repositories: A Dataset of People, Process and Product. In Proceedings of the 13th International Conference on Mining Software Repositories (Austin, Texas) (MSR ’16). Association for Computing Machinery, New York, NY, USA, 460–463. https://doi.org/10.1145/2901739.2903504

[47] Motahareh Bahrami Zanjani, Huzea Kagdi, and Christian Bird. 2015. Automatically recommending peer reviewers in modern code review. IEEE Transactions on Software Engineering 42, 6 (2015), 530–543.

[48] Ming Zhu, Kartik Suresh, and Chandan K Reddy. 2022. Multilingual Code Snippets Training for Program Translation. (2022).