Using Pre-Trained Models to Boost Code Review Automation

Rosalia Tufano
SEART @ Software Institute
Università della Svizzera italiana
Switzerland

Simone Masiero
SEART @ Software Institute
Università della Svizzera italiana
Switzerland

Antonio Mastropaolo
SEART @ Software Institute
Università della Svizzera italiana
Switzerland

Luca Pascarella
SEART @ Software Institute
Università della Svizzera italiana
Switzerland

Denys Poshyvanyk
SEMERU @ Computer Science Department
William and Mary
USA

Gabriele Bavota
SEART @ Software Institute
Università della Svizzera italiana
Switzerland

ABSTRACT

Code review is a practice widely adopted in open source and industrial projects. Given the non-negligible cost of such a process, researchers started investigating the possibility of automating specific code review tasks. We recently proposed Deep Learning (DL) models targeting the automation of two tasks: the first model takes as input a code submitted for review and implements in it changes likely to be recommended by a reviewer; the second takes as input the submitted code and a reviewer comment posted in natural language and automatically implements the change required by the reviewer. While the preliminary results we achieved are encouraging, both models had been tested in rather simple code review scenarios, substantially simplifying the targeted problem. This was also due to the choices we made when designing both the technique and the experiments. In this paper, we build on top of that work by demonstrating that a pre-trained Text-To-Text Transfer Transformer (T5) model can outperform previous DL models for automating code review tasks. Also, we conducted our experiments on a larger and more realistic (and challenging) dataset of code review activities.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

Code Review, Empirical Study, Machine Learning on Code

1 INTRODUCTION

The benefits of code reviews have been widely recognized, with several studies providing evidence of the higher quality of reviewed code [15, 29, 31]. Also, code reviews help in preventing bugs and foster knowledge transfer among developers [10, 40]. However, studies on code reviews also highlighted an additional cost that such a process entails: Empirical evidence suggests that large software projects can undergo hundreds of code reviews per month. This applies to both open-source (*e.g.,* ∼500 reviews per month in Linux [39]) and industrial (*e.g.,* ∼3k reviews per month in Microsoft Bing [38]) projects. As a result, developers can spend many hours per week reviewing code [16]. Given the non-negligible cost of code review, we recently proposed the automation of specific code review tasks: The goal is not to replace developers, but to help them save time in two scenarios. The first is that of a contributor (*i.e.,* the developer submitting the code for review) who wants to receive a rapid feedback about the code they wrote before submitting it for review. The feedback is provided by a Deep Learning (DL) model trained to take as input the code to submit for review C_s and provide as output a revised version of C_s (*i.e.,* C_r) implementing code changes that are likely to be recommended by a reviewer.

The second scenario concerns the reviewer(s) involved in the process: a DL model is trained to take as input (i) the code C_s submitted for review, and (ii) a comment R_{nl} written by the reviewer in natural language to request a specific change on C_s. The output of the model is a revised version of C_s (*i.e.,* C_r) implementing the changes recommended in R_{nl}. The idea here is that the reviewer can use the model to provide the contributor with a concrete example of the code changes that they would like to see implemented.

In our previous work [46] we trained and experimented with the DL models on a dataset composed of ∼17k triplets $\langle C_s, R_{nl}, C_r \rangle$ extracted from code reviews performed in GitHub [2] and Gerrit [1]. In particular, the model recommending code changes to the contributor is an encoder-decoder model with one encoder taking C_s as input and one decoder generating C_r. Our evaluation shows that this model can recommend a change as a reviewer would do in 3% (single prediction) to 16% of the cases (10 different predictions). The model employed in the second scenario (*i.e.,* the automated implementation of a comment recommended by the reviewer), has instead two encoders taking as input C_s and R_{nl}, respectively, and one decoder generating C_r. This model can successfully implement a change recommended by a reviewer in 12% (single prediction) to 31% (10 different predictions) of the cases.

Despite the encouraging preliminary results, our approach [46] as well as the conducted empirical study suffers of several limitations we try to overcome in this paper. First, in [46] we adopted code abstraction to reduce the vocabulary size and simplify the learning of the DL model: The model did not work on the raw code, but on an abstracted version of it in which, for example, variable identifiers were replaced with a special `VAR_ID` token, where `ID` is a progressive number (*e.g.,* the second variable is represented by `VAR_2`). The possibility to go back to raw code was guaranteed by a map linking abstracted to raw tokens in C_s (*e.g.,* `VAR_1` → `i`).

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510621

While such a procedure simplifies the learning of the model, it poses a strong limitation on the variety of code review tasks that can be supported by such a model. Indeed, the abstraction process forces to exclude from the dataset of triplets $\langle C_s, R_{nl}, C_r \rangle$ all those in which C_r introduces identifiers or literals that were not present in C_s. This is necessary because the abstraction map is built on C_s and, if a new variable VAR_2 is introduced in C_r during the review process, such a variable cannot be mapped back to raw source code, making such an approach unusable in practice. This means that the triplets $\langle C_s, R_{nl}, C_r \rangle$ on which we evaluated our approach [46] were relatively simple changes implemented during code review, not requiring the introduction of new identifiers or literals.

Second, to simplify the learning, we only considered triplets $\langle C_s, R_{nl}, C_r \rangle$ in which both the code submitted for review (C_s) and the revised code (C_r) had no more than 100 tokens [46]. Again, this reduced the complexity of the tackled problem.

Basically, the two above choices resulted in training and experimenting the proposed models on quite simple code review instances only representative of a minority of the code transformations actually implemented during code reviews.

In this paper, we build on top of our previous work [46] experimenting with DL models for code review automation in more realistic and challenging scenarios. We start by training the recently proposed Text-To-Text-Transfer Transformer (T5) model [35] on a dataset similar to the one used in [46]. However, we adopt a tokenizer (*i.e.,* SentencePiece [26]) that allows us to work with raw source code, without the need for code abstraction. Also, we increase the maximum length of the considered code components from 100 "abstracted" tokens to 512 "SentencePiece" tokens (*i.e.,* ∼390 "abstracted" tokens). The absence of an abstraction mechanism and the increased upper bound for input/output length allowed us to build a substantially larger dataset as compared to the one used in [46] (168k instances *vs.* 17k) and, more importantly, to feature in such a dataset a wider variety of code transformations implemented in the code review process, including quite challenging instances such as those requiring the introduction of new identifiers and literals (accounting for 63% of the new dataset we built). Also, we experimented with the automation of a third task related to the code review process: Given the code submitted for review (C_s), generating a natural language comment R_{nl} requesting to the contributor code changes as a reviewer would do (*i.e.,* simulating a reviewer commenting on the submitted code).

We also compare the T5 model with the encoder-decoder model presented in our previous work on the original dataset used in [46]. Our results show the superior performance of T5, which represents a significant step forward in automating code review tasks.

To summarize, the **contributions** of this work are:

(i) A novel approach for code review automation overcoming several limitations of the state-of-the-art technique [46];

(ii) A comprehensive empirical evaluation of such an approach, including a comparison with our previous technique [46];

(iii) The automation of a third task: Given the code submitted for review, automatically generating natural language comments requesting changes as reviewers would do;

(iv) A code review dataset to train and test DL models in more realistic scenarios as compared to the one used in [46];

(v) A comprehensive replication package [8].

2 T5 TO AUTOMATE CODE REVIEW

We describe the DL model we adopt, the construction process of the datasets needed for its training, and the procedure used for hyperparameter search, model training, and generation of predictions.

2.1 Text-to-Text Transfer Transformer (T5)

The Text-to-Text Transfer Transformer, or simply T5, is not merely a model. Raffel *et al.* [35] compare "*pre-training objectives, architectures, unlabeled data sets, transfer approaches, and other factors on dozens of language understanding tasks*".

The result of this exploration is the best combination of architectures and training techniques, namely T5. T5 is based on the Transformer [48] architecture. The proposed implementation differs only in some details (regarding the normalization layer and the embedding scheme) from its original form. Raffel *et al.* proposed several versions of T5, differing from each other in their size (*e.g.,* number of layers) and, as a consequence, training complexity. In this work we adopt the *small* version of T5 consisting of: 8-headed attention, 6 layers in both the encoder and the decoder, each having a dimensionality of 512 and the output dimensionality of 2,048 (∼ 60M parameters).

The model is subjected to a first training (pre-training) whose purpose is to provide it with a general knowledge useful to solve a set of related tasks. Suppose, for example, that we want to train a model able to (i) translate English to German, and (ii) summarize English text. Instead of starting by training the model for these two tasks, T5 can be pre-trained in an unsupervised manner by using the *denoising objective* (or *masked language modeling*): The model is fed with sentences having 15% of their tokens (*e.g.,* words in English sentences or code tokens in Java statements) randomly masked and it is asked to predict them. By learning how to predict the masked tokens, the model can acquire general knowledge about the language of interest. In our example, we could pre-train the model on English and German sentences.

Once pre-trained, T5 is fine-tuned on the downstream tasks in a supervised fashion. Each task is formulated in a "text-to-text" format (*i.e.,* both the input and the output of the model are represented as text). For example, for the translation task a dataset composed of pairs of English and German sentences allows to fine-tune the model. Similarly, the summarization task requires the input English text and a corresponding summary. In the next sections we explain how we pre-train and fine-tune T5 to support code review tasks.

2.2 Training Data

We describe the process used to build the datasets needed for the pre-training (Section 2.2.1) and fine-tuning (Section 2.2.2) of T5. Part of the fine-tuning dataset has been used for hyperparameter search (Section 2.3) and for testing the performance of T5 (Section 3).

2.2.1 Pre-training Dataset. Given the goal of the pre-training phase (*i.e.,* providing the model with general knowledge about the languages of the downstream tasks) we built a dataset allowing to train T5 on Java and technical English.

Indeed, besides source code, technical English is instrumental in a code review process in which reviewers post natural language comments about code.

We start from two datasets featuring instances including both source code and technical English: the official Stack Overflow dump (SOD) [7] and CodeSearchNet (CSN) [25]. Stack Overflow is a Q&A website for programmers. The data dump we used collects all the questions and relative answers between 2006 and 2020 for a total of roughly 51M posts (where a post is a single question or answer). A post includes English text (as per the SO guidelines) and/or code snippets. Posts are usually accompanied by tags characterizing their topic (*e.g., Java*, *Android*) and can be rated with *up-*/*down-votes* and, for what concerns the answers, they can be marked as the "accepted answer" from the question's author.

We extracted from the SOD all the answers (i) having a *Java* tag; (ii) containing at least one `<pre><code>` HTML tag to ensure the presence of at least one code snippet in the answer; and (iii) having at least 5 up-votes and/or being the accepted answer. These filters are justified by the goal of our pre-training. Indeed, we want the model to acquire knowledge about technical English and Java: focusing on answers containing at least one code snippet increases the chances that their natural language text refers to an implementation task, similarly to what happens in code review. Also, the up-votes/accepted answer filter aims at discarding low-quality instances containing, for example, wrong code solutions. This is also the reason why we focused on high-quality answers likely to contain working solutions rather than on questions that, even if up-voted (*e.g.,* because they are relevant for many users) may contain wrong implementations. From this step we obtained 1,018,163 candidate instances from the SOD.

On each selected answer a, we performed the following cleaning steps: We remove emojis, non-latin characters, control characters, trailing spaces and multiple white spaces. Some special symbols are replaced using latin characters having the same meaning, *e.g.,* "≥" is replaced with ">=". Moreover, we replace any embedded link with a special tag "<LINK_i>", with i being an integer ranging from 0 to $n-1$, where n is the number of links in a. Finally, we removed all the instances having less than ten tokens or more than 512 (40,491). This left us with 977,379 valid instances.

The CSN [25] Java dataset features 1.5M unique Java methods, some of which containing their Javadoc. We filtered out all those in which a Javadoc was not available or it did not contain any letter, removing 1,034,755 of them. Unlike the SOD, CSN can contain instances in which the "textual part" (*i.e.,* the method comment) is not in English. To partially address this issue, we exclude pairs in which no Latin characters were found. While this does not exclude all non-English comments, at least identifies and removes those written in specific languages (*e.g.,* Russian, Chinese) (15,229). We decided to accept some level of noise in the pre-training dataset (*e.g.,* comments written in French) since (i) given the size of this dataset, this little amount of noise should not substantially affect the model's performance, and (ii) the pre-training dataset is not used as test set to assess the performance of the approach. As we will explain later, a more fine-grained cleaning has been performed for the fine-tuning dataset that, instead, is used for performance evaluation. On the 519,905 remaining instances, we performed the same cleaning steps described for the SOD (*e.g.,* remove emojis). Finally, from each pair we obtain a single string concatenating the Javadoc comment and the code, retaining the ones having more than ten and less than 512 tokens (507,947 instances left).

By putting together the instances collected from the SOD and CSN we obtained the pre-training dataset consisting of 1,485,326 instances. To perform the pre-training, we randomly mask in each instance 15% of its tokens. The masked tokens are replaced with *sentinel tokens* `<extra_id_i>`, where i is an increasing number ranging from 0 up to $n-1$, where n is the number of tokens masked in a given instance. If several contiguous tokens are masked they are replaced by a single sentinel token. These "masked instances" represent the input of the model during the pre-training. The target (*i.e.,* the string the model is expected to generate) is built concatenating the sentinel tokens and the token(s) they are masking. An extra sentinel token is added to indicate the end of the string.

Our pre-training dataset is publicly available [8].

2.2.2 Fine-tuning Datasets. To create the fine-tuning dataset we mined Java open source projects from GitHub using the web application by Dabic *et al.* [19]. Using the querying interface [5], we selected all Java projects having at least 50 pull requests (PRs), ten contributors, ten stars, and not being forks. The filters aim at (i) ensuring that enough "code review" material is contained in the projects (*i.e.,* at least 50 PRs); (ii) discarding personal/toy projects (at least ten contributors and stars); and (iii) reducing the chance of mining duplicated code. This resulted in a list of 4,901 projects. We also mined the six Gerrit [1] installations used in [46] containing code review data about 6,388 projects.

From both the GitHub and the Gerrit datasets we extract triplets $< m_s, c_{nl}, m_r >$, where m_s is a method submitted for the review; c_{nl} is a single reviewer's comment suggesting code changes for m_s; and m_r is the revised version of m_s implementing the reviewer's recommendation c_{nl}. Note that (i) we only looked for PRs that are accepted at the end of the code review, since we want to learn how to recommend changes that, at the end, can lead to code considered good from a reviewer's perspective; and (ii) a single PR in GitHub and Gerrit can result in several triplets for our dataset. Indeed, we mine the different review rounds in each PR. For example, a method m_s can be submitted for review, receiving a comment c_{nl} asking for changes (first round). The revised version of m_s addressing c_{nl} is then resubmitted (m_r), resulting in the second review round (possibly leading to additional comments and revisions of the method). We stop when the code is formally accepted.

Overall, we mined 382,955 valid triplets from GitHub and Gerrit using the pipeline from [46] that we summarize in the following (see [46] for additional details). We target triplets in which a comment c_{nl} has been posted by a **reviewer** on a method m_s. We can identify these cases since both GitHub and Gerrit (i) provide information about the developers submitting the code and posting comments in the review process; and (ii) allow to retrieve the specific code line(s) c_{nl} refers to (*i.e.,* the code in m_s that has been highlighted by the reviewer when posting the comment).

We exclude all the comments posted by the authors of the code (*e.g.,* to reply to reviewers), since they do not represent a review of the code. Thus, the triplets in our dataset have c_{nl} being a single comment posted by a reviewer. Also, we exclude c_{nl} linked to inline comments (rather than code lines) in m_s, since we target the fixing of code-related issues. To consider a triplet as valid, c_{nl} must be the only comment posted by a reviewer on m_s in that specific review round.

In this way, we can be confident that the revised version submitted later on by the author (m_r) actually aimed at implementing c_{nl}. Also, m_r must differ from m_s (*i.e.,* a change must have been implemented in the code to address c_{nl}). From the technical point of view, the parsing of the methods from the patches submitted for review has been done using the lizard library [4]. Note that, the removal of triplets in which c_{nl} include more than one comment has been done later in the processing pipeline (we will get back to this point). Indeed, before we had to clean comments possibly just representing noise.

As done for the pre-training dataset, we performed some cleaning steps. We replaced any link with the numbered token `<LINK_i>`, with i being an integer ranging from 0 to $n - 1$, where n is the total number of links in c_{nl}, m_s and m_r. If the same link appears in different parts (*e.g.,* in c_{nl} and m_r), it is replaced with the same token. We also removed any emoji and non-ascii characters from the comments, extra spaces and control characters from both the comments and the methods, and inline comments from the methods (we are not interested in addressing issues related to internal comments).

After the cleaning process we obtained some triplets in which c_{nl} became an empty string or where m_s and m_r became equal (*e.g.,* they only differed for some spaces before the cleaning). We removed these instances (-33,005) as well as those having c_{nl} + m_s or m_r longer than 512 tokens (-61,233). We considered the sum of c_{nl} and m_s in terms of length because, for one of the tasks (*i.e.,* the automated implementation of a comment posted by a reviewer), they will be concatenated to form the input for the model.

Then, we removed from our triplets non-relevant comments (-28,581), *i.e.,* comments not recommending code change suggestions (*e.g.,* "looks good to me"). In [46] we manually crafted a set of natural language patterns to spot non-relevant comments (*e.g.,* single-word comments containing words such as "thanks", "nice", etc.). We have extended this set since we noticed that in our richer dataset several non-relevant comments were left by these patterns. Such analysis has been done by one of the authors by manually inspecting all the triplets having c_{nl} consisting of less than six words. The updated heuristics are available in our replication package [8].

We also excluded triplets including non-English c_{nl} comments (-4,815) through a pipeline composed by three language detector tools. A preliminary classification has been performed using the Python libraries langdetect [3] and pycld3 [6]. If both of these tools classify the comment as non-English, we relied on the Google language detection API for a final decision. Such a process was needed since we noticed that the Google API was the most accurate in detecting the language, especially when the comments also featured code constructs in them. In this scenario, the Python libraries often generated false negatives (*i.e.,* classifying an English sentence as non-English). However, we had a limited number of requests available for the Google API. Thus, we performed a pre-filtering using the Python libraries and, when they both reported the comment as being not in English, we double checked using the Google API.

After this cleaning process, we excluded all triplets featuring more than one comment in c_{nl} (-86,604). Finally, we removed all the duplicates from the fine-tuning dataset (-918). To be conservative, we identify as duplicates two triplets having the same m_s (thus, even triplets having the same m_s but different c_{nl}/m_r have been removed).

The resulting dataset features 167,799 triplets that have been used to build the three fine-tuning datasets needed for the three tasks we aim at automating. In the first task (*code-to-code*) the model takes as input m_s with the goal of automatically generating its revised version m_r, implementing code changes that may be required in the code review process. Thus, the fine-tuning dataset is represented by pairs $m_s \rightarrow m_r$.

In the second task (*code&comment-to-code*) the model takes as input both m_s and a comment c_{nl} posted by the reviewer and targets the generation of m_r, the revised version of m_s implementing the code changes recommended in c_{nl}.

The m_s code contains two special tags `<START>`, `<END>` marking the portion of the code c_{nl} refers to. The fine-tuning dataset of this second task is represented by pairs $< m_s, c_{nl} >\longrightarrow m_r$.

Finally, in the third task (*code-to-comment*) the model takes as input m_s and aims at generating a natural language comment (c_{nl}) suggesting code changes as a reviewer would do. The fine-tuning dataset is represented by pairs $m_s \rightarrow c_{nl}$.

Table 1: Pre-training and fine-tuning datasets (# instances)

Dataset	train	evaluation	test
Pre-training			
Stack Overflow	977,379	-	-
CodeSearchNet	507,947	-	-
Fine-tuning	134,239	16,780	16,780

All three fine-tuning datasets have been split into 80% training, 10% evaluation, and 10% test. Table 1 summarizes the number of instances in the datasets: The pre-training is only used for training, while the fine-tuning datasets are exploited also for the hyperparameter tuning (*evaluation*) and for assessing the performance of the model (*test*). In Table 1 we only report information for a single fine-tuning dataset (rather than for the three previously described), since all three fine-tuning datasets contain the same number of instances. Indeed, they are all derived from the same set of triplets.

2.3 Training and Hyperparameter Search

Raffel *et al.* [35] showed the major role pre-training plays on the performance of T5 models. The importance of pre-training has also been confirmed (for other Transformer-based models) in the context of code-related tasks such as test case generation [44]. To further study this aspect, we decided to experiment with both a pre-trained and a non pre-trained model, both of which have been subject to a hyperparameter tuning process.

Since we adopted the *small* version of T5 presented by Raffel *et al.* [35], we did not experiment with variations related to its architecture (*e.g.,* changing the number of layers or the number of hidden units). Though, as also done by Mastropaolo *et al.* [28], we experimented with different learning rate configurations: (i) *Costant Learning Rate* (C-LR), in which the learning rate value is fixed during the training; (ii) *Inverse Square Root Learning Rate* (ISR-LR), in which the learning rate value decays as the inverse square root of the training step; (iii) *Slanted Triangular Learning Rate* (ST-LR) in which first the learning rate linearly increases and then it linearly decays returning to the starting value; (iv) *Polynomial Decay Learning Rate* (PD-LR), in which the learning rate polynomially decays to a fixed value in a given number of steps.

The hyperparameter tuning has been done for the fine-tuning phase only. Indeed, even though we just focus on one hyperparameter, such a process still remains quite expensive, requiring the training of eight different T5 models (*i.e.,* pre-trained and non pre-trained each with four different learning rates).

For pre-training we use the same configuration proposed by Raffel *et al.* in [35]. We pre-trainied the model on the pre-training dataset (Table 1) for 200k steps (~34 epochs). Starting from the pre-trained model, we fine-tuned for 75k steps four different models, each using one of the experimented learning rates.

Since the goal of this procedure is to find the best learning rate for the three code review tasks, we fine-tuned each of these models using a mixture of the three tasks: A single model is trained to support all three tasks using the union of their training sets. This is one of the characteristics of T5, the possibility to train a single model for multiple tasks. The same approach has been used for the non pre-trained model: In this case four T5 models (one per learning rate) have been directly fine-tuned.

We assessed the performance of the eight models on the evaluation set of each task in terms of "perfect predictions", namely cases in which the generated output was identical to the target (expected) string. Table 2 reports the achieved results. As it can be seen, no learning rate achieves the best results in all the tasks. Nevertheless, ST-LR shows better overall performance and, for this reason, is the one we adopt in our experiments.

Table 2: Hyperparameter tuning results

Task	Learining Rate Strategy			
	C-LR	ISR-LR	ST-LR	PD-LR
		Pre-Trained		
code-to-code	2.68%	3.68%	**4.64%**	2.53%
code&comment-to-code	**10.39%**	9.23%	8.46%	9.89%
code-to-comment	0.15%	0.32%	**0.60%**	0.15%
		Non Pre-Trained		
code-to-code	1.23%	3.71%	**4.16%**	1.22%
code&comment-to-code	5.05%	**6.41%**	6.24%	5.18%
code-to-comment	0.09%	0.44%	**0.49%**	0.03%

Given the best configuration for both the pre-trained and the non pre-trained models, we fine-tuned them for a maximum of 300k steps using an *early stop strategy*. This means that we saved a checkpoint of the model every 10k steps computing its performance in terms of "perfect predictions" on the evaluation set and stopped the training if the performance of the model did not increase for three consecutive checkpoints (to avoid overfitting).

2.4 Generating Predictions

Once the models are trained, they can be used to generate predictions. As done in previous work, we adopt a beam search strategy [36] to generate multiple predictions given a single input. For example, in the case of the *code-to-code* task, for a single m_s method provided as input multiple m_r candidates can be generated. When we ask the model to generate k predictions, it generates the k most probable sequences of tokens given the input sequence; k is known as the *beam size* and we experiment with $k = 1, 3, 5, 10$.

For each prediction generated by T5, we also exploited its `score` function to assess the model's confidence on the provided input. The value returned by this function ranges from minus infinity to 0 and it is the log-likelihood (ln) of the prediction. Thus, if it is 0, it means that the likelihood of the prediction is 1 (*i.e.,* the maximum confidence, since $ln(1) = 0$), while when it goes towards minus infinity, the confidence tends to be 0. In our empirical study (Section 3) we assess the reliability of the confidence level as a proxy for the quality of the predictions.

3 STUDY DESIGN

The *goal* of our evaluation is to empirically assess the performance of the T5 model in code review automation tasks. The *context* consists of (i) the datasets we presented in Section 2; and (ii) the dataset from our previous work [46]. From now on we refer to our previously presented approach as the *baseline*. The study aims at tackling five research questions (RQs).

RQ$_1$: *To what extent is T5 able to automatically recommend code changes to developers as reviewers would do?* We provide as input to T5 a Java method m_s submitted for review and assess the extent to which the model is able to provide as output a revised version of m_s (m_r) implementing code changes that will be likely requested during the code review process. The idea here is that such a model could be used *before* the code is submitted for review as an automated check for the contributor.

RQ$_2$: *To what extent is T5 able to automatically implement code changes recommended by reviewers?* Given a Java method submitted for review (m_s) and a natural language comment (c_{nl}) in which a reviewer asks to implement specific code changes in m_s, we assess the ability of T5 to automatically revise m_s to address c_{nl} (thus obtaining a revised method m_r).

The third RQ focuses on the novel code review-related task we introduce in this paper:

RQ$_3$: *To what extent is T5 able to automatically recommend changes in natural language as reviewers would do?* In this RQ T5 is provided as input with a Java method submitted for review (m_s) and it is required to generate a natural language comment (c_{nl}) requesting code changes as reviewers would do.

For RQ$_1$-RQ$_3$, we experiment with different variants of the T5 model. In particular, we assess the quality of T5 predictions for all three tasks when (i) the model is pre-trained or not; and (ii) the predictions have different confidence levels. Thanks to these analyses, we can answer our fourth RQ:

RQ$_4$: *What is the role played by the model pre-training on the performance of T5? How does the confidence of the predictions affects their quality?* As explained in Section 2.3, we perform an ablation study in which T5 is fine-tuned without any pre-training (*i.e.,* by starting from random weights in the neural network). This allows to assess the contribution of the pre-training to the performance of the model. As for the confidence of the predictions, we assess whether it can be used as a reliable proxy for the quality of the predictions (*i.e.,* the higher the confidence, the higher the likelihood the prediction is correct). If this is the case, such a finding would have implications for the usage of the T5 model in practice: A developer using the model could decide to receive recommendations having confidence higher than t, reducing the chances of receiving meaningless predictions.

Finally, the last RQ compares the performance of the T5 model with that of the approach we presented in [46]:

RQ$_5$: *What is the performance of T5 as compared to the state-of-the-art technique?* We use the implementation and datasets from our previous work to compare the performance of the T5 model with the baseline [46].

3.1 Data Collection and Analysis

To answer the first four research questions, we experiment with the best configuration of both the pre-trained and non pre-trained T5 model on the test set of the fine-tuning dataset reported in Table 1.

Remember that for each of the three tasks we support (*i.e.,* the ones that map to RQ$_1$, RQ$_2$, and RQ$_3$) the 16,779 test set instances are the same triplets $< m_s, c_{nl}, m_r >$. The only difference is that: in RQ$_1$ the model has been trained (and is tested) to take as input m_s and produce m_r; in RQ$_2$ it takes as input m_s and c_{nl} and produces m_r; in RQ$_3$ it takes as input m_s and produces c_{nl}.

By running the models on the test sets, we report for each of the three tasks the percentage of "perfect predictions", namely the cases in which the output of the model is the expected one. For example, in the case of RQ$_3$, this means that the model was able, given m_s as input, to generate a comment c_{nl} identical to the one manually written by the reviewer who inspected m_s.

Besides computing the perfect predictions, in RQ$_3$ (*i.e.,* the task in which the model is required to generate natural language text), we also compute the BLEU (Bilingual Evaluation Understudy) score of the predictions [32]. BLEU assesses the quality of the automatically generated text. The BLEU score ranges between 0 and 1, with 1 indicating, in our case, that the natural language comment generated by the model is identical to the one manually written by the reviewer. We use the BLEU-4 variant, that computes the overlap in terms of 4-grams between the generated and the reference text.

In RQ$_1$ and RQ$_2$ (*i.e.,* in the tasks in which the model is required to generate code), we adopt instead the CodeBLEU [37], a recently proposed similarity metric inspired by the BLEU score but tailored to assess the quality of automatically generated code.

Differently from BLEU, CodeBLEU computes not only an "n-gram based similarity" but it also considers how similar the abstract syntax tree and the data-flow of the generated and the reference code are. Ren *et al.* [37], who proposed the CodeBLEU, showed that their metric better correlates with developers' perception of code similarity as compared to the BLEU metric.

Concerning RQ$_4$, we compare the results (*i.e.,* perfect predictions, BLEU, CodeBLEU) achieved by the T5 model with and without pre-training. We also statistically compare the two models (*i.e.,* with/without pre-training) using the McNemar's test [30] and Odds Ratios (ORs) on the perfect predictions they can generate. As for the confidence of the predictions, we take the best performing model (*i.e.,* the one with pre-training) and split its predictions into ten buckets based on their confidence c going from 0.0 to 1.0 at steps of 0.1 (*i.e.,* the first interval includes all predictions having a confidence c with $0 < c \leq 0.1$, the last interval has $0.9 < c \leq 1$). Then, we report for each interval the percentage of perfect predictions.

Finally, in RQ$_5$, we compare T5 with the baseline [46] on the two tasks automated in our previous work (*i.e.,* the ones related to our RQ$_1$ and RQ$_2$).

As metrics for the comparisons, we used the percentage of perfect predictions and the CodeBLEU of the predictions. We compared the two techniques in several scenarios. First, we used the dataset from [46] featuring 17,194 triplets $< m_s, c_{nl}, m_r >$. By performing some checks on this dataset, we noticed that a few instances (97) had comments (c_{nl}) not written in English or containing invalid unicode characters that did not allow our tokenizer to work. Thus, we excluded those instances from the training and the test sets shared by the authors. The training set has then been used to (i) train the baseline [46]; and (ii) fine-tune the T5 model without any pre-training. In this way, we can compare the performance of the two models on the test set when trained on exactly the same data. Important to notice is that the baseline has been trained and tested on abstracted code (as done in [46]), while T5 worked directly with the raw source code.

On top of this, we also report the performance of the pre-trained T5 model when run on the test set from [46]. This pre-trained model has been fine-tuned using the training dataset in [46]. Clearly, this analysis favors T5 since it has been trained on more data (*i.e.,* the pre-training dataset). However, it provides additional hints into the role played by the pre-training and on the effectiveness of the T5 model in general. Besides reporting descriptive statistics, we statistically compare the two models using the McNemar's test [30] and Odds Ratios (ORs) on the perfect predictions they can generate. Since multiple comparisons are involved (*e.g.,* comparing the pre-trained and the non pre-trained model to the baseline), we adjust the p-values using the Holm's correction [24].

4 RESULTS DISCUSSION

We start by answering **RQ$_1$-RQ$_3$** (Section 4.1), presenting the performance of T5 in the three tasks we aim at automating. Then, we discuss the impact on the performance of the pre-training and the reliability of the confidence level as a proxy for the quality of the predictions (Section 4.2). Finally, we compare T5 with the baseline [46] (Section 4.3).

4.1 RQ$_1$-RQ$_3$: Performance of T5

Fig. 1 reports two graphs for each task. The line chart on top shows the percentage of perfect predictions (y-axis) achieved by T5 for different beam sizes (x-axis); the continuous line represents the pre-trained version of the model, while the dashed line the non pre-trained one. The boxplots at the bottom report the CodeBLEU for the two code-generation tasks (*i.e., code-to-code* and *code&comment-to-code*) and the BLEU score for the *code-to-comment* task in which text is generated. Lighter blue represents the pre-trained model.

We start by commenting on the perfect predictions (line charts). At a first sight, the performance of the model might seem quite low. For example, in the case of *code-to-code* at $k = 1$ (*i.e.,* a single prediction is proposed by T5), both the pre-trained and the non pre-trained models achieve ∼5% of perfect predictions (751 and 863 instances correctly predicted with and without pre-training, respectively). However, such a result should be considered in the context of what was reported by the state-of-the-art technique [46] that, on a much simpler test dataset, achieved for the same task and same beam size 2.91% of perfect predictions.

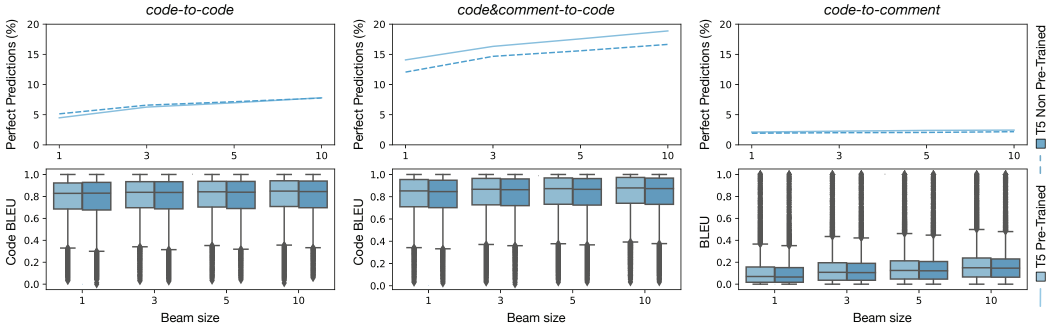


Figure 1: Results T5 dataset large

Similar observations can be made for the *code&comment-to-code* task, where at $k = 1$ T5 can generate 14.08% (2,363 instances) and 12.06% (2,024) perfect predictions when pre-trained and not, respectively. For this task, in our previous work [46], we achieved on a simpler dataset 12.16% perfect predictions. We directly compare the two approaches in RQ_5.

Interestingly, increasing the beam size from 1 to 10 does only result in marginal improvements for all tasks. The largest improvement is obtained for the *code&comment-to-code*, where we move from 14.08% ($k = 1$) to 18.88% ($k = 10$) of perfect predictions for the pre-trained model. Given the goal of our approach, we believe that the most relevant performance are those achieved at $k = 1$.

Indeed, providing several recommendations to inspect to a developer might be counterproductive, especially considering that the recommendations are entire methods in the case of the two code-generation tasks.

Moving to the *code-to-comment* task, T5 struggles in formulating natural language comments identical to the ones written by reviewers. The pre-trained model, at $k = 1$, generates 356 correct comments (2.12%) against the 324 (1.93%) of the non pre-trained model. These numbers only slightly increase at $k = 10$, with a maximum of 2.44% perfect predictions achieved with pre-training.

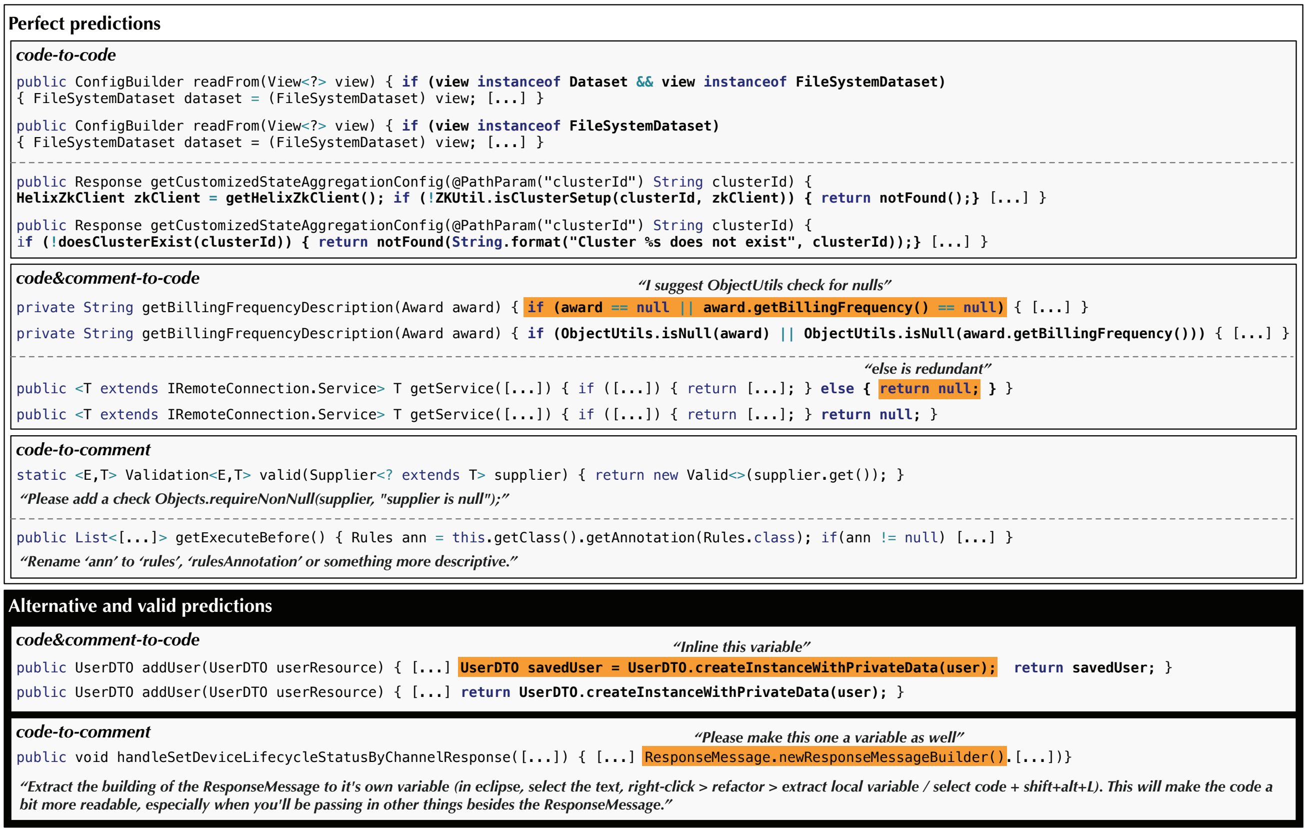
The top part of Fig. 2 shows two examples of perfect predictions generated by the model for each task. A dashed line separates the two examples within each task. For the *code-to-code* task, the first code in each example represents the input of the model, while the second its output. We highlighted in bold the parts of code changed by the model and replaced irrelevant parts of the methods with [...] to save space. In the first *code-to-code* example, T5 removes an unneeded `instanceof` check, since `FileSystemDataset` is a subclass of `Dataset`. Instead, the second example simplifies the checking for the existence of a cluster, providing a meaningful error message. This second case cannot be supported by the baseline [46], since it requires the introduction of new code tokens that were not present in the input code. Remember that, these being perfect predictions, the implemented changes are identical to those performed by developers during code review.

For the *code&comment-to-code* task, the input provided by the model includes the comment written by the reviewer and requiring a specific change to the part of code highlighted in orange. In the first example, the reviewer suggests to use a specific object to perform the `null` check and T5 correctly implements the change. The second one is interesting because, despite the reviewer highlighting `return null` as the relevant code for their comment ("*else is redundant*"), the model correctly understands that the action to take is the removal of the unneeded `else` statement.

Finally, for the *code-to-comment* task, we report the code provided as input to the model (first line) with the comment it generated as output (second line). In the first example, T5 suggests (as done by the real reviewer) to add a `null` check, also showing the code needed for its implementation. This code is not just a template, but it is suitable for the provided input code (it refers to the `supplier` object). In the second example, T5 suggests to rename an identifier, providing valid recommendations for the renaming.

Looking at the bottom of Fig. 1, the results in terms of CodeBLEU show a median higher than 0.80 for all beam sizes and for both code-generation tasks. However, while we report these values for completeness and for being consistent with what done in similar works [45, 46, 50], they say little about the quality of the predictions and they are mostly useful for future work that wants to compare with our approach (complete distributions are available in our replication package [8]). Indeed, it is difficult to properly interpret these values for two reasons. First, there is no accepted threshold above which good performance can be claimed. Second, as also done in previous works proposing models taking as input a code snippet and providing as output the same code "revised" in some way (*e.g.,* with a fixed bug [45], with a single statement added [50], or with review-related changes implemented [46]), we computed the CodeBLEU between the predicted and the target code (two methods in our case). However, the input provided to the model is already quite similar to the target output, which means that a model taking as input a method and not implementing any change on it, is likely to obtain high values of CodeBLEU. For this reason, we mostly focus our discussion on perfect predictions. Concerning the BLEU score achieved in the *code-to-comment* task, the median ranges around 0.10 (see Fig. 1). Such a result is expected given the low percentage of perfect predictions achieved for this task.

Going back to the perfect predictions, the results reported in the line charts in Fig. 1 represent a lower bound for the performance of our approach. Indeed, we consider a prediction as "perfect" only if it is identical to the reference one. For example, in the case of the *code-to-comment* task, the natural language comment generated by T5 is classified as correct only if it is equal to the reference one, including punctuation.



Perfect predictions

code-to-code

```
public ConfigBuilder readFrom(View<?> view) { if (view instanceof Dataset && view instanceof FileSystemDataset)
{ FileSystemDataset dataset = (FileSystemDataset) view; [...] }

public ConfigBuilder readFrom(View<?> view) { if (view instanceof FileSystemDataset)
{ FileSystemDataset dataset = (FileSystemDataset) view; [...] }
```

```
public Response getCustomizedStateAggregationConfig(@PathParam("clusterId") String clusterId) {
HelixZkClient zkClient = getHelixZkClient(); if (!ZKUtil.isClusterSetup(clusterId, zkClient)) { return notFound();} [...] }

public Response getCustomizedStateAggregationConfig(@PathParam("clusterId") String clusterId) {
if (!doesClusterExist(clusterId)) { return notFound(String.format("Cluster %s does not exist", clusterId));} [...] }
```

code&comment-to-code

"I suggest ObjectUtils check for nulls"

```
private String getBillingFrequencyDescription(Award award) { if (award == null || award.getBillingFrequency() == null) { [...] }
private String getBillingFrequencyDescription(Award award) { if (ObjectUtils.isNull(award) || ObjectUtils.isNull(award.getBillingFrequency())) { [...] }
```

"else is redundant"

```
public <T extends IRemoteConnection.Service> T getService([...]) { if ([...]) { return [...]; } else { return null; } }
public <T extends IRemoteConnection.Service> T getService([...]) { if ([...]) { return [...]; } return null; }
```

code-to-comment

```
static <E,T> Validation<E,T> valid(Supplier<? extends T> supplier) { return new Valid<>(supplier.get()); }
```

"Please add a check Objects.requireNonNull(supplier, "supplier is null");"

```
public List<[...]> getExecuteBefore() { Rules ann = this.getClass().getAnnotation(Rules.class); if(ann != null) [...] }
```

"Rename 'ann' to 'rules', 'rulesAnnotation' or something more descriptive."

Alternative and valid predictions

code&comment-to-code

"Inline this variable"

```
public UserDTO addUser(UserDTO userResource) { [...] UserDTO savedUser = UserDTO.createInstanceWithPrivateData(user);  return savedUser; }
public UserDTO addUser(UserDTO userResource) { [...] return UserDTO.createInstanceWithPrivateData(user); }
```

code-to-comment

"Please make this one a variable as well"

```
public void handleSetDeviceLifecycleStatusByChannelResponse([...]) { [...] ResponseMessage.newResponseMessageBuilder().[...])}
```

"Extract the building of the ResponseMessage to it's own variable (in eclipse, select the text, right-click > refactor > extract local variable / select code + shift+alt+L). This will make the code a bit more readable, especially when you'll be passing in other things besides the ResponseMessage."

Figure 2: Examples of perfect and alternative predictions

However, it is possible that a natural language comment generated by T5 is different but semantically equivalent to the one written by the developer (*e.g.,* "variable v should be private" *vs* "change v visibility to private"). Similar observations hold for the two code-generation tasks (*e.g.,* a reviewer's comment could be addressed in different but semantically equivalent ways).

To have an idea on the number of valuable predictions present among those classified as "wrong" (*i.e.,* the non-perfect predictions), three authors manually analyzed a sample of 100 "wrong" predictions for each task (300 in total). The analysis was done in two meetings in which each instance was discussed by all three authors. The goal was to classify each instance into one of three categories: (i) "semantically equivalent" (*i.e.,* the generated code/-comment is different but semantically equivalent to the reference one); (ii) "alternative solution" (*i.e.,* the generated code/comment is not semantically equivalent, but valuable); or (iii) "wrong" (*i.e.,* the generated code/comment is not meaningful for the provided input). Since we also computed the confidence for each of the predictions generated by T5, rather than randomly selecting the 300 instances to inspect, we decided to target for each task the top-100 wrong predictions generated by the model in terms of confidence. Indeed, those cases are particularly interesting, since they represent wrong predictions for which, however, the model is quite confident.

Table 3: Manual analysis of 100 "wrong" predictions per task

Task	Semantically Equivalent	Alternative Solution	Wrong
code-to-code	1	10	89
code&comment-to-code	6	56	38
code-to-comment	36	10	54

Table 3 shows the results of our manual analysis. For the *code-to-code* we observed that, in most cases (89%) the model actually generates wrong predictions that are not inline with the changes implemented by the developer. There are few exceptions to these cases, mostly related to small changes in which the model made a decision different from that one of the developer but still valid (*e.g.,* extracting a string into a variable and using a different name for the extracted variable). More interesting are the results for the other two tasks.

In the case of *code&comment-to-code*, we found that 62 out of the 100 "wrong" predictions we inspected were actually valid implementations of the change recommended by the reviewer. One example is presented at the bottom of Fig. 2 (black background), where we show the input provided to the model (*i.e.,* the code in the first line and the reviewer's comment "*Inline this variable*") and the output of the model right below. T5 successfully addressed the reviewer's comment.

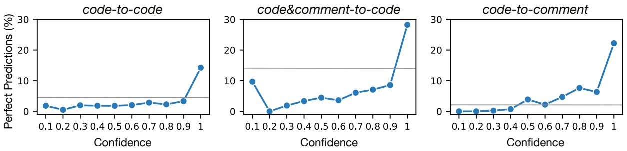


Figure 3: Perfect predictions by confidence of the model

However, the prediction is different from the target implementation, since the latter also includes another change that was not explicitly required in the code review. This case is representative of all 56 instances we classified as "alternative solutions" for this task and, given the goal of the *code&comment-to-code*, we believe they represent good predictions.

Finally, also for the *code-to-comment* task, we found a large number of "wrong" predictions that are actually valuable, with 36 of them even being semantically equivalent (*i.e.,* T5 formulated a comment asking the same changes required by the reviewer, but using a different wording). One example is reported at the very bottom of Fig. 2. While the model only received the code as input we also show the original reviewer's comment (*i.e.,* "*Please make this one a variable as well*") to make it easier to assess the relevance of the comment generated by T5 (*i.e.,* "*Extract the building ...*").

Overall, our analysis showed that the perfect predictions really represent a lower bound for the performance of T5, especially for the two tasks in which natural language comments are involved.

4.2 RQ$_4$: Pre-training and confidence

In Fig. 1 we observed better performance for the pre-trained model in the *code&comment-to-code* and in the *code-to-comment* task, while the non pre-trained model performed better in the *code-to-code* task. The results of the McNemar's test on the predictions at k=1, confirm such findings: besides the significant difference confirmed for all tasks (p-value < 0.01), the ORs indicate 85% and 59% higher odds of obtaining a perfect prediction using the pre-trained model in the *code&comment-to-code* (OR=1.85) and in the *code-to-comment* (OR=1.59) task, while odds are 34% lower in the *code-to-code* task (OR=0.66). Two observations are worth to be made. First, overall, the pre-trained model seems to represent a more valuable solution. Second, the lack of improvement in the *code-to-code* task can be explained by the pre-training and fine-tuning we performed. Indeed, the *code-to-code* task only focuses on source code, with no natural language in the input nor in the output. The fine-tuning stage, focused on source code, was probably sufficient to the model to learn about the code syntax and the possible transformations to perform. The additional pre-training, also including technical English, did not benefit the model for the *code-to-code* task. The other two tasks, instead, either include natural language as input (*code&comment-to-code*) or require its generation as output (*code-to-comment*), obtaining a boost of performance from the pre-training.

Fig. 3 depicts the percentage of perfect predictions (y-axis) within each confidence interval (from 0.0-0.1 up to 0.9-1.0, x-axis) when using the pre-trained model and k=1. To better interpret the reported results, the gray line represents the overall performance of the model when considering all predictions (*e.g.,* 4.48% of perfect predictions for the *code-to-code* task).

In all three tasks, we observe a clear trend, with the predictions in the highest confidence bucket (0.9-1.0) ensuring substantially better performance than the overall trend. When only considering the predictions in this bucket, the percentage of perfect predictions increases to: 14.24% for *code-to-code* (from an overall 4.48%), 28.23% for *code&comment-to-code* (overall=14.08%), and 22.23% for *code-to-comment* (overall=2.12%). Considering the complexity of the addressed tasks, the jump in performance is substantial and indicates the usability of the confidence level as a proxy for the prediction quality. Also, while the percentage of perfect predictions is quite limited, with seven out of ten predictions being wrong in the best-case scenario (28.23% for *code&comment-to-code*), it is worth considering what previously observed in our manual analysis, with "valuable" predictions which are classified as "wrong" in our quantitative analysis.

4.3 RQ$_5$: Comparison with the baseline [46]

Fig. 4 compares the performance achieved by the T5 model with those obtained by the baseline [46].

In the line charts the continuous lines represent the pre-trained T5, the dashed lines non pre-trained T5, and the dotted lines the baseline. Two important points are worth remembering: First, the results in Fig. 4 have been computed on the test set used in [46]. Indeed, the performance in terms of perfect predictions are substantially higher as compared to those in Fig. 1 (see values on the y-axis), due to the simpler instances featured in this dataset. Second, the baseline has been trained and tested on abstracted code (as in the original paper), while T5 worked on raw source code.

When k=1, T5 achieves substantially better performance. The results of the statistical test in Table 4 always show a significant difference in favor of T5 (adjusted p-value < 0.01), with ORs ranging from 1.69 (non pre-trained T5 *vs* [46] in the *code-to-code* task) to 11.48 (pre-trained T5 *vs* [46] in the *code&comment-to-code* task). The pre-trained T5 in this case performs better than the non pre-trained one for both tasks. This is likely due to the limited size of the fine-tuning dataset used in this comparison. Indeed, to have a fair comparison with [46], we fine-tuned T5 on the training set we used in [46] and composed by ~13.5k instances (*vs* the ~134k we had in our fine-tuning dataset when answering RQ$_1$-RQ$_4$). This is probably not sufficient to effectively train a large model such as T5, and makes the instances used in the pre-training fundamental to further learn about the language. Still, even without pre-training, T5 outperforms the baseline when k=1. For example, in the *code&comment-to-code* task, the baseline achieves 9.48% perfect predictions, against the 15.46% of the non pre-trained T5, and the 29.74% of the pre-trained T5. The baseline observes a stronger improvement with the increasing of k (*i.e.,* the beam size) as compared to T5 (see Fig. 4). We believe this is due to usage of the abstraction. Indeed, when working with abstracted code the "search space" (*i.e.,* the number of possible solutions that can be generated with the given vocabulary) is much more limited since the model does not deal with identifiers and literals. Attempting ten predictions in a smaller search space is more likely to result in correct predictions. The results of the CodeBLEU confirm the trend observed with the perfect predictions, with the pre-trained T5 being the best model.

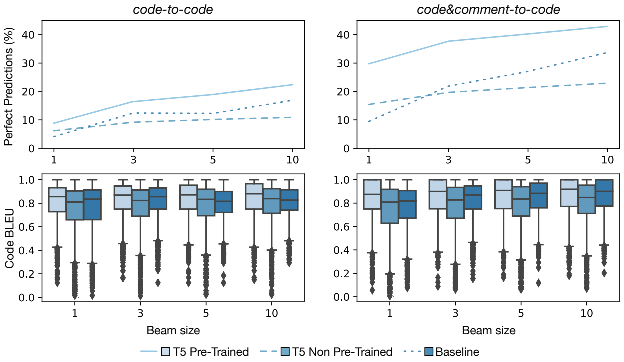


Figure 4: T5 *vs.* baseline [46]

We also looked at the union of perfect predictions generated by the two approaches on the test set to verify the complementarity of the techniques. On the *code-to-code* (*code&comment-to-code*) task we observed that 15% (24%) of perfect predictions are shared by both approaches (*i.e.,* both succeed), 65% (70%) are perfect predictions only for T5, and 20% (6%) only for the baseline.

Table 4: RQ$_5$: McNemar’s test (adj. *p*-value and OR)

Task	Test	*p*-value	OR
code-to-code	T5 pre-trained *vs* [46]	<0.01	2.90
	T5 non pre-trained *vs* [46]	<0.01	1.69
	T5 pre-trained *vs* T5 non pre-trained	<0.01	2.50
code&comment-to-code	T5 pre-trained *vs* [46]	<0.01	11.48
	T5 non pre-trained *vs* [46]	<0.01	2.38
	T5 pre-trained *vs* T5 non pre-trained	<0.01	5.69

5 THREATS TO VALIDITY

Construct validity. As explained in Section 2 we took care of cleaning the datasets used in our study by removing duplicates and noisy data points to the extent possible. Still, we are aware that problematic instances may be present, especially in the new (large) dataset we built. This manifests, for example, in non-English comments, or in some wrong “links” between comments and implementation (*e.g.,* we assume that m_r implemented a change described in c_nl while, in fact, it implemented another change).

Internal validity. We did not fully explore the role played by the T5 parameters on its performance. Indeed, our hyperparameter tuning was limited to variations in the learning rate, as done in previous work [28]. For the other parameters we relied on the best architecture identified by Raffel *et al.* [35]. We acknowledge that additional tuning can result in improved performance.

External validity. RQ$_1$-RQ$_4$ have been answered using a dataset being one order of magnitude larger as compared to our previous work on automating code review tasks [46]. However, our findings are limited to Java. Concerning RQ$_5$ in which we compare with the baseline [46], we only used the dataset presented in [46]. This is due to the fact that our previous approach [46] requires code abstraction and, as previously explained, cannot work on instances having new identifiers and literals inserted during the code review process. The new dataset used in this paper has not been built with such a constraint in mind and, thus, it is not suitable for direct comparison.

6 RELATED WORK

Our work relates to three research areas: (i) DL techniques to automate software-related tasks, (ii) empirical studies on code review, and (iii) works providing recommendations on how to optimize the code review process and/or presenting techniques to partially automate it. Here we focus on the third research area, while for the first two we point the reader to the systematic literature reviews by Watson *et al.* [49] (deep learning in software engineering) and by Davila and Nunes [20] (modern code review).

Optimizing/automating the code review process. By studying tools and techniques supporting code review, Tymchuk *et al.* [47] concluded that popular code review platforms (*e.g.,* Gerrit, Code Flow, Phabricator) mostly offer the same basic functionalities with little support for automating tasks. Such a finding has been confirmed by Pascarella *et al.* [34]. Also, in a study performed by Lewis *et al.* [27] at Google, the authors show that while developers are excited by the idea of embracing automated solutions for code review, they find current solutions not to be ready for daily use. Starting from these observations, researchers studied possible optimizations of the review process: Baum *et al.* [14] investigate the effect of ordering submitted changes in alternative ways rather than in alphabetical order that, as shown by Barnett *et al.* [12] and Baum and Schneider [13], is sub-optimal. Baum *et al.* [14] concluded that smarter ordering is needed as the size of the patch increases, and suggest to aggregate changed parts by relatedness.

Di Biase *et al.* [21] studied the impact of the patch size on the review’s effectiveness, finding that smaller patches, while not increasing the defects found, affect how reviewers approach their task. Spadini *et al.* [43] compared the effectiveness of a standard code review process with test-driven code review (TDR), *i.e.,* the reviewer inspects the changed test code before the production code. They show that TDR does not boost the code review effectiveness.

Several researchers [23, 33, 51] suggest exploiting defect prediction models during code review. Similarly, Balachandran [11] and Singh *et al.* [42] suggest the use of static analysis tools to automatically spot coding standard violations and common defects.

Concerning the automation of specific code review tasks, authors proposed techniques to optimize the reviewers’ assignment. For example, Al-Zubaidi *et al.* [9] in open source and Chouchen *et al.* [18] in industrial contexts show how a multi-objective search-based approach can simplify the code review triaging process.

Shi *et al.* [41] and Chouchen *et al.* [18] look at the automation of code review from a similar perspective. Shi *et al.* [41] present a DL model taking as input the code submitted for review and the revised code implementing the changes recommended by reviewers and providing as output whether the change can be accepted or not. Note that the change(s) required by the reviewer(s) are not considered by the model. Chouchen *et al.* [18] use instead a set of quality metrics as features for machine learning algorithms to classify the quality of the code submitted for review. Recently, Hellendoorn *et al.* [22] focus on the prediction of the location of a possible reviewer’s comment, showing that even this simple task is challenging to automate.

The above discussed techniques [18, 22, 41] are complementary to the approach we presented in [46] (and, as a consequence, to the models experimented in this work).

While Shi *et al.* [41] and Chouchen *et al.* [18] assess the code under review through a "boolean answer" (*i.e.,* accepted/rejected or well-written/badly-written), we attempt the automation of code changes implemented in code review. Also, the approach by Hellendoorn *et al.* could be combined with the automation of the *code-to-comment* task we presented.

7 CONCLUSION AND FUTURE WORK

Our paper starts by discussing limitations in the approach we recently proposed to automate code review tasks [46]. We highlighted that the usage of code abstraction does not allow to support non-trivial code review scenarios requiring code changes resulting in the introduction of new identifiers/literals. Hence, we proposed the usage of a pre-trained T5 model [35] relying on a SentencePiece [26] tokenizer to overcome such a limitation and work directly on raw source code. Our empirical evaluation, performed on a much larger and realistic code review dataset, shows the improvements brought by the T5 model that represents a step forward as compared to the state-of-the-art [46] both in terms of applicability (*i.e.,* scenarios in which it can be applied) and performance. Still, the level of actual performance observed makes these techniques far from being deployable in practice, calling for more research in code review automation.

Our future research agenda will be focused on designing improved solutions to boost the prediction accuracy of these techniques (*e.g.,* by combining different representations of code [17] and/or by exploiting the model's confidence as a possible filter to select only high-quality recommendations).

The code and data used in our study are publicly available [8].

ACKNOWLEDGMENT

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 851720). W&M has been supported in part by the NSF CCF-1955853 and CCF-2007246 grants. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

REFERENCES

[1] [n.d.]. Gerrit. https://www.gerritcodereview.com/.

[2] [n.d.]. GitHub. https://github.com/.

[3] [n.d.]. langdetect. https://pypi.org/project/langdetect/.

[4] [n.d.]. Lizard. https://github.com/terryyin/lizard/.

[5] [n.d.]. MSR mining platform. https://seart-ghs.si.usi.ch.

[6] [n.d.]. pycld3. https://pypi.org/project/pycld3/.

[7] [n.d.]. Stack Exchange Dumps. https://archive.org/details/stackexchange.

[8] 2021. Replication Package. https://github.com/RosaliaTufano/code_review_automation.

[9] Wisam Haitham Abbood Al-Zubaidi, Patanamon Thongtanunam, Hoa Khanh Dam, Chakkrit Tantithamthavorn, and Aditya Ghose. 2020. Workload-aware reviewer recommendation using a multi-objective search-based approach. In *Proceedings of the 16th ACM International Conference on Predictive Models and Data Analytics in Software Engineering*. 21–30.

[10] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 international conference on software engineering*. IEEE Press, 712–721.

[11] Vipin Balachandran. 2013. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *2013 35th International Conference on Software Engineering (ICSE)*. 931–940. https://doi.org/10.1109/ICSE.2013.6606642

[12] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K. Lahiri. 2015. Helping Developers Help Themselves: Automatic Decomposition of Code Review Changesets. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. 134–144.

[13] Tobias Baum and Kurt Schneider. 2016. On the need for a new generation of code review tools. In *International Conference on Product-Focused Software Process Improvement*. Springer, 301–308.

[14] Tobias Baum, Kurt Schneider, and Alberto Bacchelli. 2017. On the optimal order of reading source code changes for review. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 329–340.

[15] Gabriele Bavota and Barbara Russo. 2015. Four eyes are better than two: On the impact of code reviews on software quality. In *IEEE International Conference on Software Maintenance and Evolution, (ICSME)*. 81–90.

[16] A. Bosu and J. C. Carver. 2013. Impact of Peer Code Review on Peer Impression Formation: A Survey. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. 133–142.

[17] Saikat Chakraborty and Baishakhi Ray. 2021. On Multi-Modal Learning of Editing Source Code. arXiv:2108.06645 [cs.SE]

[18] Moataz Chouchen, Ali Ouni, Mohamed Wiem Mkaouer, Raula Gaikovina Kula, and Katsuro Inoue. 2021. WhoReview: A multi-objective search-based approach for code reviewers recommendation in modern code review. *Applied Soft Computing* 100 (2021), 106908.

[19] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. Sampling Projects in GitHub for MSR Studies. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*. IEEE, 560–564.

[20] Nicole Davila and Ingrid Nunes. 2021. A systematic literature review and taxonomy of modern code review. *Journal of Systems and Software* (2021), 110951.

[21] Marco di Biase, Magiel Bruntink, Arie van Deursen, and Alberto Bacchelli. 2019. The effects of change decomposition on code review—a controlled experiment. *PeerJ Computer Science* 5 (2019), e193.

[22] Vincent J Hellendoorn, Jason Tsay, Manisha Mukherjee, and Martin Hirzel. 2021. Towards automating code review at scale. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1479–1482.

[23] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. 2019. DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 34–45.

[24] Sture Holm. 1979. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics* (1979), 65–70.

[25] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *CoRR* abs/1909.09436 (2019). http://arxiv.org/abs/1909.09436

[26] Taku Kudo and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing. *CoRR* (2018). arXiv:1808.06226

[27] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E James Whitehead. 2013. Does bug prediction support human developers? findings from a google case study. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 372–381.

[28] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the Usage of Text-To-Text Transfer Transformer to Support Code-Related Tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 336–347.

[29] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. 2014. The Impact of Code Review Coverage and Code Review Participation on Software Quality: A Case Study of the Qt, VTK, and ITK Projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014)*. 192–201.

[30] Quinn McNemar. 1947. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika* 12, 2 (1947), 153–157.

[31] Rodrigo Morales, Shane McIntosh, and Foutse Khomh. 2015. Do Code Review Practices Impact Design Quality? A Case Study of the Qt, VTK, and ITK Projects. In *Proc. of the 22nd Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER)*. 171–180.

[32] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics (ACL '02)*. 311–318.

[33] Luca Pascarella, Fabio Palomba, and Alberto Bacchelli. 2019. Fine-grained just-in-time defect prediction. *Journal of Systems and Software* 150 (2019), 22–36.

[34] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. 2018. Information needs in contemporary code review. *Proceedings of the ACM on Human-Computer Interaction* 2, CSCW (2018), 1–27.

[35] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67. http://jmlr.org/papers/v21/20-074.html

[36] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, 419–428.

[37] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. arXiv:2009.10297 [cs.SE]

[38] Peter C. Rigby and Christian Bird. 2013. Convergent Contemporary Software Peer Review Practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. 202–212.

[39] Peter C. Rigby, Daniel M. German, Laura Cowen, and Margaret-Anne Storey. 2014. Peer Review on Open-Source Software Projects: Parameters, Statistical Models, and Theory. *ACM Trans. Softw. Eng. Methodol.* 23, 4 (2014).

[40] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern Code Review: A Case Study at Google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '18)*. 181?190.

[41] Shu-Ting Shi, Ming Li, David Lo, Ferdian Thung, and Xuan Huo. 2019. Automatic code review by learning the revision of source code. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 4910–4917.

[42] Devarshi Singh, Varun Ramachandra Sekar, Kathryn T Stolee, and Brittany Johnson. 2017. Evaluating how static analysis tools can reduce code review effort. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 101–105.

[43] Davide Spadini, Fabio Palomba, Tobias Baum, Stefan Hanenberg, Magiel Bruntink, and Alberto Bacchelli. 2019. Test-driven code review: an empirical study. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1061–1072.

[44] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit Test Case Generation with Transformers. *CoRR* abs/2009.05617 (2020). https://arxiv.org/abs/2009.05617

[45] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Trans. Softw. Eng. Methodol.* 28, 4 (2019), 19:1–19:29.

[46] Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. 2021. Towards Automating Code Review Activities. In *43rd International Conference on Software Engineering, ICSE'21*. https://arxiv.org/abs/2101.02518

[47] Yuriy Tymchuk, Andrea Mocci, and Michele Lanza. 2015. Code review: Veni, vidi, vici. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 151–160.

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[49] Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, and Denys Poshyvanyk. 2021. A Systematic Literature Review on the Use of Deep Learning in Software Engineering Research. *ACM Transactions on Software Engineering and Methodology* (2021).

[50] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1398–1409.

[51] Supatsara Wattanakriengkrai, Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Hideaki Hata, and Kenichi Matsumoto. 2020. Predicting Defective Lines Using a Model-Agnostic Technique. *CoRR* (2020).