Automating Code-Related Tasks Through Transformers: The Impact of Pre-training

Rosalia Tufano, Luca Pascarella, Gabriele Bavota

Software Institute – USI Università della Svizzera italiana, Switzerland

Abstract—**Transformers have gained popularity in the software engineering (SE) literature. These deep learning models are usually pre-trained through a self-supervised objective, meant to provide the model with basic knowledge about a language of interest (*e.g.,* Java). A classic pre-training objective is the *masked language model* (MLM), in which a percentage of tokens from the input (*e.g.,* a Java method) is masked, with the model in charge of predicting them. Once pre-trained, the model is then fine-tuned to support the specific downstream task of interest (*e.g.,* code summarization). While there is evidence suggesting the boost in performance provided by pre-training, little is known about the impact of the specific pre-training objective(s) used. Indeed, MLM is just one of the possible pre-training objectives and recent work from the natural language processing field suggest that pre-training objectives tailored for the specific downstream task of interest may substantially boost the model's performance. For example, in the case of code summarization, a tailored pre-training objective could be the identification of an appropriate name for a given method, considering the method name to generate as an *extreme summary*. In this study, we focus on the impact of pre-training objectives on the performance of transformers when automating code-related tasks. We start with a systematic literature review aimed at identifying the pre-training objectives used in SE. Then, we pre-train 32 transformers using both (i) generic pre-training objectives usually adopted in SE; and (ii) pre-training objectives tailored to specific code-related tasks subject of our experimentation, namely bug-fixing, code summarization, and code completion. We also compare the pre-trained models with non pre-trained ones and show the advantage brought by pre-training in different scenarios, in which more or less fine-tuning data are available. Our results show that: (i) pre-training helps in boosting performance only if the amount of fine-tuning data available is small; (ii) the MLM objective is usually sufficient to maximize the prediction performance of the model, even when comparing it with pre-training objectives specialized for the downstream task at hand.**

Index Terms—**Pre-training, Code Recommenders**

I. INTRODUCTION

Transformers [1] are deep learning (DL) models built on the idea of self-attention, a mechanism able to assign different weights to different parts of the input data. Transformers achieved state-of-the-art performance in several natural language processing (NLP) tasks and, recently, have gained popularity in the software engineering (SE) literature for the automation of code-related tasks (see *e.g.,* [2]–[13]).

Transformers are usually trained in a two-step process. First, they are pre-trained using a self-supervised training objective with the goal of providing the model with general knowledge about a language relevant for the final task to automate. Then, the model is fine-tuned to support the downstream task of interest.

The most adopted pre-training objective is the *Masked Language Model* (MLM), in which a percentage of tokens from the input sentence is masked, with the model in charge of predicting them. For example, given the final goal of building a translator from English to German, a transformer can be firstly pre-trained by feeding as input English and German sentences with masked words. This task is self-supervised, since random words can be automatically masked with the model in charge of predicting them. Then, a labeled dataset mapping English sentences to their corresponding German translation can be used to fine-tune the model. Several works applying DL to SE report boost of performance[1] provided by pre-training in the automation of code-related tasks [11], [13], [14]. However, little is known about (i) the circumstances in which pre-training actually helps, and (ii) the impact of the specific pre-training objective(s) adopted on the performance of transformers when automating code-related tasks.

Concerning the first point, it is known that pre-training is helpful when the fine-tuning dataset is small [15]. To make a concrete example, per-training can be useful when fine-tuning a model for automated bug-fixing. For this task, the fine-tuning dataset is usually composed by pairs of ⟨*buggy, fixed*⟩ functions, which are mined from bug-fixing commits. The amount of training data that can be collected is thus limited, in the order of tens of thousands instances [16], [17]. Pre-training datasets, instead, can be automatically built with virtually "no limitations" in terms of size since the pre-training objective is self-supervised. There are however SE tasks which can leverage very large fine-tuning datasets, for which the boost in performance provided by pre-training is not obvious. For example, in the case of code completion, fine-tuning instances are usually represented by pairs of ⟨*incomplete_code, complete_code*⟩, with the model learning how to "finalize an implementation task". These pairs can be built automatically from any piece of code by removing parts of it.

As for the second point (*i.e.,* the impact of pre-training objectives), the widely used MLM is just one of the possibilities here: Any self-supervised task can be used as pre-training objective. Also, in a recent work from NLP, Zhang *et al.* hypothesized that "*using a pre-training objective that more closely resembles the downstream task leads to better fine-tuning performance*" [18].

---

[1]With "performance" we refer to the quality of the predictions generated by the model (*e.g.,* accuracy) rather than to attributes such as execution time.

We present a large-scale study aimed at (i) investigating whether pre-training is actually useful in code-related tasks for which the fine-tuning dataset can be built without any obvious impediment in collecting large amount of data (*e.g.,* code completion); and (ii) experimenting the impact on transformers' performance of both generic and task-specific pre-training objectives when automating three code-related tasks, namely bug-fixing, code summarization, and code completion.

We start by performing a systematic literature review (SLR) to identify the pre-training objectives used in the SE literature. We inspected a total of 936 papers published in SE venues up to January 2022, ending up with 33 relevant papers, namely those (i) automating a code-related task (*i.e.,* a task involving source code as input and/or output of the model), and (ii) pre-training a transformer or exploiting an already pre-trained transformer from previous work. The output of such a SLR (Table IV) provided us with an overview of pre-training objectives used in the literature.

Based on this analysis, we selected three generic pre-training objectives to experiment based on their popularity and potential impact in SE: MLM, *next sentence prediction*, and *replaced tokens detection*. Moreover, we defined three pre-training objectives tailored for the specific downstream tasks we aim at supporting. For *bug-fixing*, we pre-train the model through the *injected mutants fixing* objective. The latter simulates the bug-fixing downstream task (i) without using real bug-fixing instances that can be kept to increase as much as possible the size of the fine-tuning dataset; and (ii) while being scalable in terms of size, since the objective is self-supervised and the experimenter can create as many pre-training instances as wished. Concerning the *code summarization* task, we consider the *Method Name Generation* as a tailored self-supervised pre-training objective: The model takes as input a method and must guess an appropriate name for it. The name is treated as an *extreme summary* [19], thus preparing the model for the downstream task in which, given a method, it is expected to generate a textual summary. Finally, concerning *code completion*, we devise the *code block selection* objective, which asks the model to guess which among two code blocks is appropriate to complete a code snippet having a code block masked. Such an objective prepares the model for the more challenging downstream task, in which it is asked to generate from scratch the missing block rather than just selecting it among two choices.

We then trained 36 Text-To-Text Transfer Transformer (T5) models [20], accounting for a total of 890 training hours, using different pre-training objectives and fine-tuning datasets. In particular, we study the impact of the fine-tuning dataset size on the "boost in performance" (if any) provided by pre-training. Also, we assess the impact on the achieved performance of different combinations of pre-training objectives. Our findings suggest that:

1) Pre-training is extremely useful when the pre-training dataset is substantially larger than the fine-tuning one, while it does not help when the fine-tuning dataset is of comparable size.

2) The MLM pre-training objective represents a safe choice for all tasks we investigated, being almost always the best-in-class; (iii) specialized pre-training objectives only help if they strictly resemble the fine-tuning task and can provide the model with knowledge that cannot be captured by generic objectives.

We release all code and data used in our study in a comprehensive replication package [21].

II. A SLR ON PRE-TRAINING OBJECTIVES USED TO AUTOMATE CODE-RELATED TASKS

Section II-A describes the design of our SLR following the guidelines by Kitchenham and Charters [22]. The achieved results are discussed in Section II-B.

A. Study Design

Our SLR aims at answering the following research question: *What are the pre-training objectives used in the SE literature exploiting transformers to automate code-related tasks?*

With "code-related tasks" we refer to any task involving source code as input and/or output of the model. For example, code summarization is considered a code-related task, since it takes as input a code component to summarize in natural language. Differently, using transformers to automate "sentiment analysis" on software-related artifacts (*e.g.,* discussions in issue trackers) is not considered relevant for our SLR. Answering this research question will inform the study presented in Section III, in which we experiment with representative pre-training objectives from the state-of-the-art.

1) Relevant Study Identification: We used the following digital libraries to search for primary studies: ACM Digital Library [23], IEEE Xplore Digital Library [24], Springer Link Online Library [25], Wiley Online Library [26], Elsevier ScienceDirect [27], and Scopus [28]. Google Scholar was not considered as an option due to the lack of quality control, clear indexing guidelines, and missing support for data download [29]. The following search query has been run on the search engines integrated in each of these online databases:

full text CONTAINS
    ("*pretrain*" OR "*pretrained*" OR "*pretraining*" OR
    "*pre-train*" OR "*pre-trained*" OR "*pre-training*" OR
    "*transfer learning*") AND
publication date IS FROM *01.01.2007* TO *02.02.2022* AND
publication venue CONTAINS
    ("*software*" OR "*program*" OR "*code*")

The composition of the query is the result of a trial-and-error procedure performed by the three authors. The query searches for the listed terms (*e.g.,* pretrain, pretrained) in the full text of the articles (*i.e.,* title, keywords, abstract, main text, references). The date interval has been defined by conservatively collecting papers starting from 2007, year in which we found a first mention to the notion of "transfer learning" in a SE-related article [30], and using the date in which the search has been performed as the end of the interval (02.02.2022).

Finally, based on the authors' knowledge of the existing SE publication venues, we only searched for articles published in venues containing at least one of three keywords: software, program, and code. We acknowledge that there might be relevant articles published in related fields (*e.g.,* artificial intelligence) that our query would exclude. However, our focus was indeed on the SE research community and these keywords should capture most of the relevant venues.

Some of the search engines (*i.e.,* Springer, Wiley, Elsevier, and Scopus) allow to specify a *discipline* of interest. Such a feature is useful to limit the retrieved false positive instances.

In all online libraries we selected "Computer Science" as discipline. In addition, Springer also allows to specify sub-disciplines, for which we selected "Software Engineering/Programming" and "Operating Systems". While the latter might not be fully relevant, we decided to include it to be more conservative. Links with the exact queries we have run are publicly available [21].

TABLE I
ARTICLES RETURNED BY THE QUERIED DIGITAL LIBRARIES

Source	Returned Articles
ACM Digital Library	623
IEEE Xplore Digital Library	850
Springer Link Online Library	1,167
Wiley Online Library	57
Elsevier ScienceDirect	288
Scopus	1,139
Total (including duplicates)	4,124
Total (excluding duplicates)	**2,343**

Table I reports the number of articles returned from each digital library (complete list in [21]). Overall, 4,124 articles were returned, which were reduced to 2,343 by excluding duplicates. Given the very high number of articles, we decided to perform a further cleaning step before starting looking into the papers. We extracted the set of 302 venues in which the articles have been published and two of the authors independently validated them deciding whether to include or exclude them. We excluded venues unrelated to SE or not being international conferences/journals. An open discussion was performed to reach an agreement on the 53 cases of conflict (17%). As output of this process we kept 163 publication venues as valid, excluding 1,407 papers published in the excluded venues. Examples of excluded publication venues are "*Computer Methods and Programs in Biomedicine*" and the "*Brazilian Symposium on Programming Languages*". As output we obtained 936 candidate primary studies.

Study Filtering. The 936 papers were equally distributed among the three authors. Each author was in charge of inspecting the paper and decide whether to include or exclude it. Inclusion and exclusion criteria are listed in Table II. As a guideline, the authors agreed on including the paper in case of doubt, since a double-check was foreseen in the study filtering process. Indeed, despite the availability of the selection criteria as reference, such a process still remains highly subjective.

A total of 77 papers survived this first analysis. Then, to at least partially address the subjectivity issue, we applied the following procedure. First, we randomly selected 30 papers excluded by each author, asking one of the other two authors to double check whether the papers were actually to be excluded. For all 90 randomly selected papers (30 × 3 authors), no conflicts arisen, showing consistency in the exclusion criteria applied by the authors. The papers included by each author were also all double-checked by one of the other two authors. Out of the 77 papers included in the first round, 30 made it into the final list of papers, including a SLR that we kept as secondary study for the subsequent snowballing step.

Cases of disagreement have been discussed among all authors to reach consensus. Note that the decrease brought by the double-check we performed (77 → 30) was expected, considering that in the first pass on the papers we decided to be inclusive in case of doubts.

Backward Snowballing. The included papers were split among the authors, with each of them in charge of reading the reference list and identify possible relevant papers. At this stage we relaxed one of our inclusion criteria (IC1): We agreed to include papers published in venues outside of SE as long as they were presenting pre-trained models that have then been exploited to automate code-related tasks in publications appeared in SE venues. Eight papers were added through snowballing. Also in this case, a second author double-checked each of them and, through open discussion among all authors, we finally agreed to include four of them as relevant. This led to the final list of 33 primary studies included (30 - 1 secondary study + 4 output of the snowballing).

2) Data Extraction and Analysis: The data extraction was performed following the questionnaire in Table III. While most of the questions are self-explanatory, it is worthwhile to clarify 3 and 4. Our focus is only on papers using pre-trained transformers to automate code-related tasks. However, the pre-training could have been done by the authors of the papers or being the result of a pre-trained model made available in previous work (question 3). If the authors reused an already pre-trained model, then by answering question 4 we expect to know from which reference the pre-trained model has been taken. Question 5 is the ultimate goal of our SLR, which will inform our subsequent experiments.

B. Results Discussion

Table IV lists the pre-training objectives we identified. Each pre-training objective is identified by an acronym we will use to refer to it in the text. If the acronym has a </> icon close to it, this indicates that the pre-training task is specific for code, otherwise the pre-training objective is "generic" and can be applied to any sort of data that can be fed to the model as a stream of tokens. For each pre-training objective Table IV also reports a short description and references to primary studies in the SLR having a pre-trained model using it.

TABLE II
INCLUSION AND EXCLUSION CRITERIA

	Inclusion Criteria
IC1	The paper must be peer-reviewed, published at SE conferences, workshops, or journals. Such a criterion is particularly important in the snowballing phase described later, in which we ignore all referenced preprints (*e.g.,* those published on arXiv.org).
IC2	The PDF of the paper must be available online. If the PDF was not available in the online library of interest, we tried to search it on Google.
IC3	The paper must present and/or evaluate technique(s) to automate a code-related task.
IC4	The proposed/experimented technique(s) must be built on top of a pre-trained transformers model. The pre-training of the model can either have been done directly in the paper or the authors may have used an already existing pre-trained model.

	Exclusion Criteria
EC1	The paper is not written in English.
EC2	The paper has been published in a conference/workshop and later on extended to a journal. We only keep the journal paper to avoid redundancy.
EC3	The paper is not a full research publication (*e.g.,* doctoral symposium articles, posters, ERA track). We exclude all papers having less than six pages. The rationale for such a filter is to remove papers that may not have been subject to the same peer-review process typical of full research papers.
EC4	It is unclear from the paper what the adopted pre-training objective is. Such information is instrumental for the goal of our SLR.

TABLE III
DATA EXTRACTION QUESTIONNAIRE

No.	Question
1	Which code-related task has been automated?
2	Which specific transformer-based model has been used?
3	Has the model been pre-training in the paper?
4	If "no" to question 3: Which already pre-trained model has been exploited by the authors?
5	Which pre-training objectives have been used?

Without surprise, the most used pre-training objectives are those that the SE community inherited from the NLP community, such as the classic MLM, randomly masking $X\%$ of tokens in an instance that, in the case of SE research, could be for example a code function. MLM is used in 21 of the papers included in our SLR. Variations of this pre-training objective are TI, ULM, TD, and RTD (see Table IV for their description) that are, however, less popular in SE. Among them, the *Replaced Token Detection* (RTD) objective has been proposed in the paper introducing the pre-trained CodeBERT model [56]. CodeBERT is gaining substantial popularity in the SE literature especially when it comes to papers published in 2022 that, due to the time in which we run the search query, are not included in our SLR.

While the above-described objectives work at token-level granularity, *Next Sentence Prediction* (NSP) and *Sentence Ordering* (SO) aim at providing the model with knowledge related to sentence-level relationships. In SE, both can be used for example to "teach" the model the correct order of code statements in a given function. NSP is the second most-popular pre-training objective in our set of papers, with 9 articles exploiting it. The popularity of NSP is mostly due to the adoption of the BERT pre-trained model [57] in SE.

The bottom of Table IV lists the objectives specifically designed for code. Despite being pre-training objectives, some of them are targeting a specific task, like for example *Code Summarization* (CS). The latter has been instantiated in [6] as a task in which the model was asked, given a Java method, to generate its textual summary (*i.e.,* first Javadoc sentence).

The final code-related task that the authors wanted to automate after the fine-tuning phase was a natural language description of smart contract. Thus, the CS objective started providing the model with knowledge about the code-to-NL translation task. This is a concrete example of pre-training objective already tailored for the specific downstream task of interest. Overall, as it can be seen from Table IV, code-specific pre-training objectives have been usually adopted only in the paper proposing them.

Section III describes how we use the findings of our SLR to select the pre-training objectives for our experiments.

III. STUDYING THE IMPACT OF PRE-TRAINING

We aim at answering the following research questions:

RQ$_1$: *To what extent is the effectiveness of pre-training influenced by the size of the fine-tuning dataset?* RQ$_1$ investigates if pre-training is still useful when abundant fine-tuning instances are available (*e.g.,* can be automatically generated).

We assess the impact of pre-training when the model is fine-tuned on a dataset (i) being substantially smaller and (ii) having a size comparable to the pre-training dataset.

RQ$_2$: *To what extent does the choice of the pre-training objective impact the performance of transformer models?* Our second research question focuses on the impact on the model's performance of the used pre-training objectives, with a particular focus on comparing general *vs* task-specific objectives, also investigating combinations of multiple objectives.

The context of our experiments are three code-related tasks for which DL-based solutions have been proposed in the past. For reasons we will explain later, RQ$_1$ focuses on *code summarization* and *code completion*, while RQ$_2$ also includes bug-fixing. All our experiments are run on Java datasets defined at method-level granularity (*i.e.,* fixing a bug in a method, summarizing a method in natural language, and complete missing code statements in a method).

A. Transformer Model

As representative of transformers [1], we adopt the T5 proposed by Raffel *et al.* [20], that has been already used in SE to automate code-related tasks [9], [13], [14], [58], [59].

TABLE IV
PRE-TRAINING OBJECTIVES IDENTIFIED IN THE SLR

Acronym	Name	Description	References
MLM	Masked Language Model	Masks $X\%$ of tokens (usually 15%) in the instance (*e.g.,* a function) and asks the model to guess the masked tokens based on their bidirectional context. The model knows how many tokens have been masked, since each of them is replaced with a special token (*e.g.,* <MASK>).	[4], [12], [31]–[48]
NSP	Next Sentence Prediction	Given two sentences (or two statements) asks the model to guess whether they follow each other.	[32], [37]–[39], [42], [44], [45], [49]
ULM	Unidirectional Language Model	A left-to-right language modeling task, asking the model to guess one masked token in an instance by only considering the leftward tokens (*i.e.,* the tokens preceding the masked one).	[50]–[52]
TI	Token Infilling	Masks a random number of contiguous tokens and asks the model to predict them. Differently from MLM, TI does not suggest to the model how many tokens have been masked, since the sequence of masked tokens is replaced with a single special token (*e.g.,* <MASK>).	[33]
TD	Token Deletion	Deletes random tokens from the instance expecting the model to reintroduce them where needed. TD is similar to MLM, but without suggesting the model where tokens have been masked.	[33]
RTD	Replaced Tokens Detection	Replaces random tokens in the instance with other tokens. The model must guess which are the non-original tokens (*i.e.,* those that have been replaced).	[34]
SO	Sentence Ordering	Given two sentences (or two statements) asks the model to guess whether they order.	[53]
IM </>	Identifiers Masking	Masks the identifiers in the code instance and asks the model to guess the masked identifiers.	[50]
PLC </>	Prog. Language Classification	Given a sequence of code tokens asks the model to identify its programming language.	[12]
GSM </>	Generative State Modeling	Given assembly code and a small subset of its execution states (*e.g.,* register values), asks the model to reconstruct the complete set of its execution states.	[54]
EP </>	Edge Prediction	Masks edges in data-flow graph belonging to 20% of nodes randomly selected and asks the model to predict them.	[47]
NA </>	Node Alignment	Similar to data flow EP. Instead of predicting edges between nodes, the model is asked to predict edges between code tokens and nodes. Such a task is performed to align the source code-data flow representations.	[47]
CS </>	Code Summarization	Provides as input to the model a function and asks to summarize it in natural language.	[55]

Raffel *et al.* proposed several variants of T5, differing in number of trainable parameters. We use the *small* variant, featuring a total of ∼60M parameters resulting from 6 layers in both the encoder and the decoder each having a dimensionality of 512, 8-headed attention, and an output dimensionality of 2,048. While larger T5 versions are likely to achieve better performance, the training cost increases with the number of parameters. Considering the number of models that we need to train in our study (*i.e.,* 36 different models), we opted for the smaller T5 version. Indeed, our goal is not to achieve state-of-the-art performance in the automated code-related tasks, but rather to study the impact of pre-training on the model's performance in different circumstances.

B. Pre-training Objectives

We describe the pre-training objectives used in our RQs. We anticipate that they have been applied on a pre-training dataset (detailed in Section III-C) which is composed by 1M pairs of Java ⟨*method*, *javadoc*⟩.

Concerning RQ_1, we only use the *Masked Language Model* (MLM) objective, the one mostly used in the literature. Indeed, the focus of RQ_1 is not on the impact of different pre-training objectives (RQ_2), but rather on the boost provided by pre-training when the fine-tuning dataset is substantially smaller than the pre-training one or, instead, has a comparable size.

For RQ_2, based on the findings of our SLR, we selected three generic pre-training objectives to experiment with.

First, we picked the two currently being the most popular in SE, namely the MLM (with 15% of masked tokens) and the *Next Sentence Prediction* (NSP). As third "generic" objective, we selected the *Replaced Tokens Detection* (RTD) that, as previously explained, is gaining popularity since used in CodeBERT [56], adopted in several recent studies (see *e.g.,* [5], [60], [61]). While MLM and NSP are straightforward to understand, a clarification is needed on RTD. The latter starts by randomly selecting 15% of tokens to be replaced. However, the replacements for these tokens are not randomly selected from a vocabulary, but picked based on the recommendation of an n-gram model ($n = 3$) trained on the pre-training dataset. An n-gram model can predict a single token likely to follow the $n-1$ tokens preceding it. As suggested in [56], we used it to identify suitable alternatives for the tokens to be replaced, thus making the pre-training task (*i.e.,* identify which tokens in an instance have been replaced) more challenging. Given a token to replace T_i, we run the n-gram model by providing it as input with the two tokens preceding it (T_{i-2}, T_{i-1}) and collect the ranked list of candidate tokens that, accordingly to the n-gram model, is likely to follow T_{i-2} and T_{i-1}.

The ranked list features on top the most likely token T_c: If T_c is different from the token to replace T_i, we use T_c for the replacement. Otherwise, we take the token in second position.

On top of the three generic objectives, we also experiment with three pre-training objectives tailored for the downstream tasks at hand. For **bug-fixing**, we pre-train the model through the *Injected-Mutants Fixing* (IMF) objective.

The idea is to mutate each method M in the pre-training dataset by injecting artificial bugs in it, creating a mutant M_m. During pre-training the IMF objective provides T5 with M_m as input and asks it to generate M (*i.e.,* to fix the bug). One challenge we faced was the selection of the mutation testing framework to use. We considered tools such as *μJava* [62], *PIT* [63], *javaLanche* [64], and *Jester* [65]. PIT was the only one supporting recent versions of Java. However, since it works at Byte code level, PIT requires the input code to mutate to be compilable. This is problematic in our context since the 1M methods in our pre-training dataset come from thousands of software projects, several of which are likely to be unbuildable [66]. For this reason, we built a source code-level mutation tool using Javaparser [67]. Our tool implements the 11 mutation operators belonging to the "default group" in PIT [68] (*i.e., invert negatives, empty returns,* etc.). Given a Java method M, our tool builds its AST and, using it, identifies the set of mutation operators that can be applied on M. For example, the *empty returns* operator replaces return values with "empty" values (*e.g.,* `""` if M returns a `String`, `Collections.emnptyList()` if M returns a `Collection`, etc.), and can only be applied to methods returning a value.

Finally, assuming that n operators can be applied to M, n mutants (*i.e., n* versions of M) are generated, each implementing one of the applicable operators.

Concerning the **code summarization** task, we consider the *Method Name Generation* (MNG) as a tailored pre-training objective. During pre-training, T5 takes as input a Java method and it is required to synthesize an appropriate name for it, based on the idea that the method name represents an *extreme summary* of the method [19].

Finally, concerning **code completion**, we focus on the challenging task of predicting entire code blocks, as recently attempted by Ciniselli *et al.* [13]. A code block is defined as the code enclosed between two curly brackets (*e.g.,* the code executed when an `if`/`else`/`else if` condition is satisfied). To prepare the model for such a downstream task, we devised *Code Block Selection* (CBS) as a tailored pre-training objective. Given a Java method in the pre-training dataset, we randomly mask a code block in it, and ask the model to decide which of two candidate code blocks is the correct one to complete the method. This pre-training is expected to prepare the model for the more challenging downstream task of generating masked code blocks from scratch.

C. Pre-training Dataset

The same pre-training dataset is used across both RQs. To build it, we used the GitHub search tool by Dabić *et al.* [69] to identify GitHub Java projects having at least 5 contributors, 50 commits, 10 stars and not being forks of other projects. These selection criteria aimed at removing personal projects from our selection. We ended up with 14,645 valid repositories, that we parsed to extract 64,546,432 Java methods.

Since among the downstream tasks we experiment with there is *code summarization*, we wanted to make sure that each pre-training instance was composed by both source code (instrumental for all three tasks) and natural language (useful for *code summarization*). Also, recent studies [70] showed that pre-training models on both natural language and code (as opposed to code only) is beneficial when dealing with code-related tasks. For these reasons, methods without Javadoc have been excluded, leading to 17,758,579 ⟨*method, javadoc*⟩ pairs.

We then started processing our dataset to clean it and remove problematic instances. We excluded all pairs meeting one of the following conditions: (i) the Javadoc, while present, is an empty string; (ii) the method has an empty body; (iii) the method is annotated with `@Test`; (iv) the method does not end with a } (this may happen in case of parsing errors when we extract the methods). We excluded test methods since none of our tasks is test-related, and we preferred to create a more cohesive pre-training dataset featuring only methods from production code. We only consider in our dataset the first part of the Javadoc comment (*i.e.,* the one summarizing the method in natural language) excluding the Javadoc tags (*e.g.,* `@param`, `@author`). Once done with this basic filtering, the first author manually inspected hundreds of instances in the dataset to identify other sources of noise.

Four main issues were identified: (i) non-English Javadoc comments; (ii) instances containing non-ASCII characters; (iii) comments containing special symbols/tags which may not help with learning textual patterns in Javadoc; and (iv) comments not representing code summaries, but rather notes written by developers (*e.g.,* TODOs). We removed all non-English Javadocs using two Python libraries: *langid* [71] and *cld3* [72]. We keep an instance in the dataset only if both libraries classified the Javadoc as English text. We replaced all non-ASCII math characters with their corresponding ASCII representation (*e.g.,* we replaced "±", with "+-") and removed all instances featuring non-Latin characters. Additional cleaning aimed at removing from the Javadoc sequences of characters used for formatting (*e.g.,* "-----") or special markdown tags (*e.g.,* we replace {@class ClassName} with ClassName). We also replace any embedded link in the Java method and/or in the Javadoc 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 the instance. If the same link is found both in the method and in its Javadoc, they are replaced with the same special tag with the same index. Finally, since the collected methods came from different projects possibly using different coding styles, we formatted all instances using the Javaparser [67] library. We also performed additional (minor) cleaning steps that we do not document here due to lack of space. However, we publicly release our cleaning script as part of our replication package [21].

After this process, we removed all instances longer than 512 tokens (*i.e.,* the number of tokens used to represent both the method and its Javadoc was higher than 512), as also done by previous work using DL to automate code-related tasks (see *e.g.,* [70], [73], [74]): 4,821,922 instances were left.

As a last step, we excluded instances that are not suitable for one or more of our pre-training objectives. The two objectives which are not applicable to all possible instances are the *Injected Mutants Fixing* (IMF) and the *Code Block Selection* (CBS): We removed (i) 219,863 instances for which none of the 11 mutation operators we support can be applied; and (ii) 2,994,723 which did not have any code block to mask.

From the set of remaining instances, we randomly pick 1M of them to create our pre-training dataset. While, in theory, all ~1.6M remaining instances were valid, we capped the size of the pre-training dataset to limit the time needed to perform several training epochs.

D. Fine-tuning Datasets

A different fine-tuning dataset has been built for each of the three subject downstream tasks. Due to its limited size, the bug-fixing dataset has only be used in the context of RQ_2, since it was not possible to create a version of it large enough (*i.e.,* having a size comparable to our pre-training dataset) to answer RQ_1 as well.

1) Bug-fixing: We exploit the dataset used by Chen *et al.* [17] when presenting SequenceR, a sequence-to-sequence model trained on 35,578 one-line Java bug fixes (*i.e.,* commits fixing a bug by only changing a single line of code).

The training set consists of pairs featuring the buggy code and the corresponding fixed code, and it is accompanied by a validation and a testing set featuring additional 4,711 one-line bug fixes each. The buggy code includes a "buggy line" explicitly marked with two special tokens (*i.e.,* ⟨START_BUG⟩ and ⟨END_BUG⟩) and being part of a "buggy method". In addition to that, the buggy code also includes contextual information extracted from the "buggy class" (*e.g.,* its constructor). The fixed code the model is expected to generate includes, instead, only the "fixed line" (*i.e.,* revised version of the "buggy line").

Before using this dataset, we pre-processed it to make it more "aligned" to our pre-training dataset, and in particular to the tailored pre-training objective we devised for the *bug-fixing* task (*i.e., Injected-Mutants Fixing*). First, our pre-training dataset only features Java methods with its related Javadoc, excluding any class-related information. Similarly, the IMF objective provides as input to the model a mutated Java method without any additional contextual information nor special tag signaling the injected bug. Thus, we processed the buggy code of the dataset by Chen *et al.* [17] to only include the buggy method without the special tokens marking the buggy line. Second, as done for the pre-training instances, we formatted the code using Javaparser [67], to have a coherent code representation. Finally, we removed any duplicated method already present in our pre-training dataset. This process left us with 25,901 instances that we split into training (80%), validation (10%) and test (10%) set.

2) Code summarization: We use the FunCom dataset [73], [75], featuring 2,149,120 instances, with each of them being composed by a Java method and its associated Javadoc comment. FunCom has been curated to only include English comments and exclude auto-generated files.

We start from the "Filtered dataset" version [75] consisting of not processed instances. We perform on them the same cleaning process used for the pre-training dataset (*e.g.,* removing instances containing non-Latin characters) and remove any duplicate with between FunCom and our pre-training dataset. From the 1,898,437 instances left, we create two fine-tuning datasets needed to answer RQ_1. For the first (*large-ft*), we randomly select 1M instances, splitting them into training (80%), validation (10%), and test (10%). This fine-tuning dataset will be used in RQ_1 as representative of a fine-tuning dataset having a size of the same order of magnitude of the pre-training dataset. For the second (*small-ft*), we randomly select 25,901 instances, the same number of instances in our *bug-fixing* fine-tuning dataset. The idea is indeed to create a second fine-tuning dataset being substantially smaller than the pre-training one, as it usually happens when working on tasks characterized by a scarcity of training data. Also *small-ft* followed the usual training (80%), validation (10%), and test (10%) split. Both datasets are used to answer RQ_1, while only *small-ft* is used in RQ_2. Indeed, as our RQ_1's findings will show, pre-training is mostly useful when a small fine-tuning dataset is available. Thus, we experiment the impact on performance of the pre-training objectives (*i.e.,* RQ_2) when using the *small-ft* dataset.

3) Block-level code completion: Following Ciniselli *et al.* [13], we aim at building a fine-tuning dataset in which Java methods having a masked block of up to three statements are provided as input to T5, which is in charge of generating the masked block. We start from the 1,569,889 Java methods in the CodeSearchNet dataset [76]. We applied a cleaning process similar to the one described for the pre-training dataset (*e.g.,* checking that the method body is not empty, that the method is not a test method, etc.), removed methods not containing at least one code block composed by at most three code statements (2,847). Then, we followed the training procedure by Ciniselli *et al.* [13]: Given k the number of blocks identified in a method M, we create k versions of M, each one having a specific code block masked. As previously said, only blocks composed by at most three statements are masked. We then remove instances longer than 512 tokens (333,955). Such a process resulted in a dataset composed by 1,823,977 instances. Finally, similarly to what explained for code summarization, we create two versions of the *code completion* fine-tuning dataset: *large-ft* featuring a total of 1M randomly selected instances, and *small-ft* featuring 25,901 instances.

Table V summarizes the pre-training/fine-tuning datasets, also indicating which dataset has been used in each RQ.

E. Experimental Procedure

The training of the models has been performed using a 2x2 TPU topology (8 cores) from Google Colab with a batch size of 128 and the *Inverse Square Root* learning rate.

1) Answering RQ_1: . We start by fine-tuning (without pre-training) four models, two for the *code summarization* and two for the *code completion* task.

Dataset	Training	Evaluation	Test	RQ$_1$	RQ$_2$
Pre-training	1,000,000	-	-	✓	✓
Fine-tuning					
Bug-fixing	22,321	2,790	2,790	✗	✓
Code summarization					
large-ft	800,000	100,000	100,000	✓	✗
small-ft	22,321	2,790	2,790	✓	✓
Code completion					
large-ft	800,000	100,000	100,000	✓	✗
small-ft	22,321	2,790	2,790	✓	✓

The models trained within each task differ for the fine-tuning dataset used, being either the *large-ft* (800k training instances) or the *small-ft* (∼22.3k). The fine-tuning has been performed using an early-stopping training strategy by exploiting the evaluation set. In particular, we saved a checkpoint of the model every epoch computing its performance in terms of correct 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). With "correct predictions" we refer to cases in which the generated prediction is identical to the target. For code summarization, a correct prediction implies that the summary generated by the model is equal to the one written by developers. In the case code completion, the predicted code block matches the one we masked.

Then, we pre-train a T5 model for 40 epochs on the 1M instances featured in the pre-training dataset using the MLM objective. We fine-tune four versions of it, again two for each task (*i.e., code summarization* and *code completion*) differing for the used fine-tuning dataset (*large-ft* or *small-ft*). We used the same early-stopping procedure described above.

This process resulted in an overall of eight models, four being pre-trained and four not being pre-trained. These eight models have been run on the corresponding test sets collecting their predictions. This allows to compare the pre-trained and the not pre-trained models both when using a fine-tuning dataset having a size comparable to that of the pre-training (*large-ft*) or being substantially smaller than it (*small-ft*).

2) Answering RQ$_2$: . We experiment with four possible pre-training objectives (and their combinations) for each of the three tasks subject of our study: three "generic" pre-training objectives (*i.e.,* MLM, NSP, and RTD) that can be applied to any task and the pre-training objective specifically tailored for the given task (*e.g., Method Name Generation* for code summarization). Thus, for each task, we start by pre-training and fine-tuning four T5 models, each one using a specific pre-training objective. To avoid confounding factors, we made sure that all pre-trainings (i) exploit exactly the same pre-training instances (*i.e.,* the 1M instances in Table V), and (ii) are run for 40 training epochs. Concerning the fine-tuning, we adopt the same early-stopping training strategy described for RQ$_1$.

The above-described process results in 12 different models being pre-trained and fine-tuned (4 × 3 tasks).

For each task, we evaluate the four fine-tuned models on the corresponding test set (see Table V) in terms of correct predictions, identifying the best performing pre-training objective pt_o. The latter has then been combined in pairs with the remaining three objectives. For example, assuming that *Method Name Generation* (MNG) results the best pre-training objective among the four experimented for the *code summarization* task, we create three pairs of pre-training objectives including <MNG, MLM>, <MNG, NSP>, and <MNG, RTD>. This provides us with additional three models pre-trained and fine-tuned for each task (9 models overall — 3 × 3 tasks). Again, each of these models has then be evaluated on the corresponding test set, identifying the best performing "pair" of pre-training objectives for each task. The latter has been used to generate two triplets of pre-training objectives by combining it with the remaining two objectives. For example, assuming <MNG, NSP> to be the best pair for *code summarization*, we create <MNG, NSP, MLM>, and <MNG, NSP, RTD>. This results in the training and testing of two additional models for each task (6 overall). Finally, we test for each task the full combination of the four corresponding pre-training objectives, thus training one additional model for each task (3 overall). In total, we pre-trained and fine-tuned 12+9+6+3=30 T5 models in RQ$_2$.

The output of such a process is, for each task, the set of predictions generated by the 10 models trained for it (4 using a single pre-training objective, 3 using pairs of objectives, 2 using triplets, and 1 using all four objectives).

F. Data Analysis

Using the generated predictions, in both our RQs we assess the performance of the models by computing the percentage of correct predictions generated (*i.e.,* predicted output identical to the expected one). We statistically compare the results achieved by the different models for each task using the McNemar's test [77], which is a proportion test suitable to pairwise compare dichotomous results of two different treatments. To account for running multiple test instances (*e.g.,* in RQ$_2$ comparing the results of the model pre-trained with MLM with those pre-trained using RTD, NSP, etc.), we adjust *p*-values using the Holm's correction [78]. We complement the McNemar's test with the Odds Ratio (OR) effect size.

|FT| *vs* **|PT|**	Task	Non Pre-trained	Pre-trained (MLM)
<<	Code Summarization	1.94%	4.73%
	Code Completion	2.37%	5.05%
∼	Code Summarization	16.60%	15.98%
	Code Completion	30.41%	29.11%

IV. RESULTS DISCUSSION

We discuss the achieved results by research question. We highlight the main take-aways of our study using the ⚲ icon.

A. RQ₁: Effectiveness of pre-training when dealing with fine-tuning datasets of different sizes

Table VI reports the percentage of correct predictions achieved by the non pre-trained T5 and the T5 pre-trained using the MLM objective. Results are reported for the two tasks involved in RQ₁ (*i.e., code summarization* and *code completion*) in the scenario in which the fine-tuning dataset is (i) substantially smaller ($22.3k << 1M$) than the pre-training dataset (*i.e.,* the *small-ft* dataset has been used — top part of Table VI), and (ii) of a size similar ($800k \sim 1M$) to the pre-training dataset (*i.e.,* the *large-ft* dataset has been used — bottom part of Table VI).

💡 When the fine-tuning dataset is small, the pre-training, as expected, helps the learning of the model. For *code summarization*, the boost in terms of perfect predictions goes from 1.94% up to 4.73%, resulting in a statistically significant difference (p-value < 0.001) with an OR=10.3, indicating ten times higher odds of obtaining a correct prediction from the pre-trained model as compared to the non pre-trained one. Similar findings hold for the *code completion* task, with correct predictions growing from 2.37% to 5.05% thanks to the pre-training (p-value < 0.001, OR=11.7).

Moving to the bottom part of Table VI, two observations can be made. First, with the fine-tuning dataset being 36 times larger ($22.3k$ *vs* $800k$) the performance of the model improves dramatically both for the pre-trained and for the non pre-trained model. This is kind of expected considering the larger amount of data from which the model can learn useful patterns.

Second, 💡 when the fine-tuning dataset size is similar to that of the pre-training dataset, we do not observe any boost provided by pre-training, with performance slightly in favor of the non pre-trained model in both tasks (p-value < 0.001, OR=1.2 for *code summarization*, and p-value < 0.001, OR=1.3 for *code completion*). While such a result may look surprising, it might be partially explained by the well-known "catastrophic forgetting" phenomenon affecting neural networks [79]: DL models tend to forget previously learned information once new information is provided. Having a large fine-tuning dataset may lead to "override" what the model learned during pre-training, making the latter basically useless.

B. RQ₂: Impact of pre-training objectives on performance

Fig. 1 summarizes the results in terms of correct predictions achieved by the 30 models pre-trained using different objectives (10 for each task). These experiments have been run in the scenario in which pre-training helps the most (*i.e.,* when using small fine-tuning datasets). From the left to the right of Fig. 1 we move from pre-trainings performed with a single objective (*e.g.,* MLM) to those involving all objectives relevant for a given task (*e.g.,* IMF + MLM + NSP + RTD). Within each task and each pre-training group (*i.e.,* those involving only one objective, those involving two objectives, etc.), the objectives are sorted from the top to the bottom based on the performance they ensured.
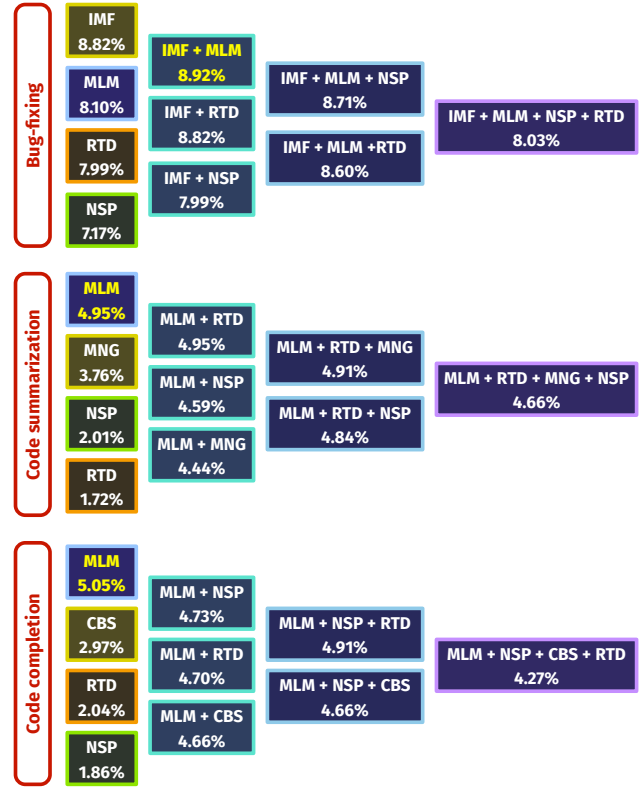

Fig. 1. RQ₂: Results with different combinations of pre-training objectives

For example, for *bug-fixing*, when experimenting with a single pre-training objective, the task-specific objective we devised (*i.e., Injected-Mutants Fixing* — IMF) is the best one, followed by MLM, RTD, and NSP. The overall best performing combination of objectives for each task is highlighted with yellow text (*e.g.,* IMF + MLM for bug-fixing). Given the high number of statistical tests performed (*i.e.,* each combination of pre-training objectives has been contrasted against all others, resulting in 45 tests per task), we provide full tables with the adjusted p-values and ORs in our replication package [21].

Three main lessons can be learned from our results. First, 💡 the choice of the pre-training objective can make a substantial difference in the performance of transformers. Within each task, contrasting the best-performing combination with the worst-performing one results in statistically significant differences (p-value < 0.001) with ORs of 2.9 (*bug-fixing*), 38 (*code summarization*), and 29 (*code completion*). For example, in the case of *code summarization*, we move from the 4.73% of correct predictions ensured by MLM, down to the 1.94% achieved with the NSP objective.

Second, the 💡 high effectiveness of the MLM pre-training objective: MLM is involved, alone or in combination with other objectives, in all best configurations we found (*i.e.,* IMF + MLM for *bug-fixing*, MLM for *code summarization* and *code completion*).

Moreover, even in the *bug-fixing* task in which the MLM objective in isolation is not the best-in-class, the difference in performance with respect to the best configuration (*i.e.,* IMF + MLM) is not statistically significant.

Third, concerning the task-specific objectives we devised, we only observed a (not statistically significant) improvement over MLM for the *bug-fixing* task with the IMF objective. The latter provides the model with information substantially different from those that can be captured by MLM and it closely resembles the fine-tuning task. This might not be the case for the other two task-specific objectives. For example, the *Method Name Generation* MNG objective devised for *code summarization* is a sort of more specific version of MLM: rather than randomly masking 15% of tokens in the training instance, the method name is the only masked token the model has to predict. We conclude that ⚲ task-specific pre-training objectives might boost performance if they (i) capture orthogonal information as compared to non-specific objectives such as MLM; and (ii) strictly simulate the downstream task. However, even in this case, the gain over the classic MLM objective may be limited and should be assessed empirically.

V. THREATS TO VALIDITY

Construct validity. The usage of the correct predictions as metric only provides a limited view about the performance of the experimented models. For example, for *code summarization*, the model might generate a code summary that, while different from the one written by developers, may be semantically equivalent. To partially address this threat, for the *code summarization* task we also computed the BLEU-4 score [80] of the predictions, being the overlap in terms of 4-grams between the predicted and the reference summary.

Similarly, for the *code completion* task, we compute the normalized token-level Levenshtein distance [81] between the predicted code block and the target one. The latter indicates the percentage of code tokens in the prediction that should be changed to obtain the target code block. The results of this analysis are available in our replication package [21]. In short, our findings hold by considering these metrics instead of the correct predictions (*e.g.,* the best pre-training objective for *code summarization* in RQ$_2$ is the same both by considering the percentage of correct predictions or the BLEU-4).

Internal validity. As a design choice, we did not perform any hyperparameter tuning of T5, using the architecture proposed by the original authors [20]. However, we compared the different models (*e.g.,* with/without pre-training) when using the same exact configuration. Thus, we expect no impact of this choice on our findings.

When we employed the small versions of the fine-tuning datasets, we observed a substantial drop of the models' performance in RQ$_1$. This may pose questions on how realistic is the size of the *small-ft* datasets we used. However, we just mirrored the size we had for the *bug fixing* task, for which the scarcity of training data is a real problem.

For the SLR, we did not consider papers published in 2022, since we started working on this paper in December 2021 running the search queries in January 2022. We acknowledge that several works relying on pre-trained transformers have been published in the meanwhile. However, the goal of our work was not to be comprehensive in terms of all pre-training objectives ever used in SE, but to get a good overview to inform our main study (Section III).

Finally, due to the stochastic nature of neural networks, small variations are possible when re-running our trained models on the same test set. This is something we observed in our experiments with a few predictions changing from one run to another. The overall findings are not affected by such minor changes.

External validity. We used T5 as representative of transformers. Other DL models may lead to different results. Also, all our experiments are focused on Java and executed at method-level granularity. Our findings may not generalize to other settings.

VI. RELATED WORK

Given the space constraints, we do not discuss the extensive literature related to the usage of pre-trained models in SE, partially documented in our SLR. We focus on (i) related literature reviews and (ii) empirical studies investigating specific aspects concerning the usage of DL models in SE.

Related Literature Reviews. At the time of writing, we found three SLRs related to the use of DL models in SE. Le [82] *et al.* investigated practices and challenges of using DL for source code modeling and generation, classifying the used architectures, the strategies for dealing with issues such as the large vocabulary of source code, and the targeted tasks.

Ferreira [83] *et al.* conducted a mapping study to understand the usage of DL models in SE. The authors focus the attention on the tasks addressed through DL and the architectures used.

Watson [84] *et al.* also presented a SLR aimed at documenting the use of DL in SE. The authors summarize 151 manuscripts to answer five research questions, related to (i) the tasks addressed via DL, (ii) the data pre-processing and datasets creation pipeline, (iii) the DL models used, (iv) the reported performance, and (v) the replicability of the studies.

Our SLR has a different goal: We investigated the objectives used to pre-train transformers in the SE literature as a starting point to inform our main study (Section III).

Empirical Studies on DL Models in SE. By considering the non-trivial effort needed to create labeled datasets, Robbes and Janes [15] suggested the use of pre-trained models in SE. They showed that pre-trained models can substantially boost performance when small fine-tuning datasets are available.

Zhou *et al.* [46] investigated the ability of CodeBERT [56] to generalize beyond its pre-trained data and to reuse embedded knowledge in various SE tasks. The results confirmed the superiority of CodeBERT compared to specialized models. Both pre-training objectives exploited by CodeBERT (*i.e.,* MLM and RTD) have been considered in our study.

Paltenghi and Pradel [85] studied if the artificial attention mechanisms of DL models behave similarly to skilled developers comprehending code. The results showed that models and humans tend to focus on different parts of code.

Chen *et al.* [86] investigated the proposal by Ahmed and Devanbu [87] to pre-train DL models on multiple programming languages. The authors reported that multilingual models have worst performance as compared to monolingual ones.

To the best of our knowledge, our work is the first extensively studying the impact of pre-training on the performance of transformers when automating code-related tasks.

VII. Conclusions

We investigated the impact on the performance of transformers [1] of the pre-training phase, nowadays adopted in most of the applications of these models to SE tasks. Two aspects have been investigated: (i) the extent to which pre-training helps the learning even when the task at hand allows to build very large datasets for fine-tuning; and (ii) the impact on the model's performance the choice of the pre-training objective(s) can have.

We found that when the size of the fine-tuning dataset is large enough, approaching that of the pre-training dataset, the pre-training phase is unlikely to help. Instead, it provides a substantial boost of performance for tasks in which the scarcity of training data leads to small fine-tuning datasets. We also observed the major role played by the choice of the pre-training objectives, with different combinations of objectives providing substantially different performance. Pre-training objectives specifically tailored for the downstream tasks can help but, at least in our study, did not result in a significant improvement of performance as compared to the classic *Masked Language Model* task.

Acknowledgment

References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[2] D. Drain, C. Wu, A. Svyatkovskiy, and N. Sundaresan, "Generating bug-fixes using pretrained transformers," ser. MAPS 2021, 2021, p. 1?8.

[3] B. Berabi, J. He, V. Raychev, and M. Vechev, "Tfix: Learning to fix coding errors with a text-to-text transformer," in *Proceedings of the 38th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139, 2021, pp. 780–791.

[4] A. Mastropaolo, S. Scalabrino, N. Cooper, D. Nader Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota, "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)*, 2021, pp. 336–347.

[5] E. Mashhadi and H. Hemmati, "Applying codebert for automated program repair of java simple bugs," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 505–509.

[6] Z. Yang, J. Keung, X. Yu, X. Gu, Z. Wei, X. Ma, and M. Zhang, "A multi-modal transformer-based code summarization approach for smart contracts," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, 2021, pp. 1–12.

[7] H. Peng, G. Li, W. Wang, Y. Zhao, and Z. Jin, "Integrating tree path in transformer for code representation," in *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., vol. 34. Curran Associates, Inc., 2021, pp. 9343–9354.

[8] Z. Tang, C. Li, J. Ge, X. Shen, Z. Zhu, and B. Luo, "Ast-transformer: Encoding abstract syntax trees efficiently for code summarization," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 1193–1195.

[9] R. Tufano, S. Masiero, A. Mastropaolo, L. Pascarella, D. Poshyvanyk, and G. Bavota, "Using pre-trained models to boost code review automation," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 2291–2302.

[10] P. Thongtanunam, C. Pornprasit, and C. Tantithamthavorn, "Autotransform: Automated code transformation to support modern code review process," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 237–248.

[11] L. Zhiyu, L. Shuai, G. Daya, D. Nan, J. Shailesh, J. Grant, M. Deep, G. Jared, S. Alexey, F. Shengyu, and N. Sundaresan, "Automating code review activities by large-scale pre-training," in *2022 ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022.

[12] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," ser. ESEC/FSE 2020, 2020, pp. 1433–1443.

[13] M. Ciniselli, N. Cooper, L. Pascarella, A. Mastropaolo, E. Aghajani, D. Poshyvanyk, M. Di Penta, and G. Bavota, "An empirical study on the usage of transformer models for code completion," *IEEE Transactions on Software Engineering*, p. To appear, 2021.

[14] A. Mastropaolo, N. Cooper, D. Nader-Palacio, S. Scalabrino, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Using transfer learning for code-related tasks," *IEEE Transactions on Software Engineering*, 2022.

[15] R. Robbes and A. Janes, "Leveraging small software engineering data sets with pre-trained neural networks," in *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, 2019, pp. 29–32.

[16] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, pp. 19:1–19:29, 2019.

[17] Z. Chen, S. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 09, pp. 1943–1959, 2021.

[18] J. Zhang, Y. Zhao, M. Saleh, and P. J. Liu, "PEGASUS: pre-training with extracted gap-sentences for abstractive summarization," in *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, ser. Proceedings of Machine Learning Research, vol. 119. PMLR, 2020, pp. 11 328–11 339.

[19] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *International Conference on Machine Learning (ICML)*, 2016.

[20] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of Machine Learning Research*, vol. 21, no. 140, pp. 1–67, 2020. [Online]. Available: http://jmlr.org/papers/v21/20-074.html

[21] "Replication Package," https://github.com/RosaliaTufano/impact_pre-training, 2022.

[22] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," 2007.

[23] "Acm digital library," https://dl.acm.org/.

[24] "Ieee xplore digital library," https://ieeexplore.ieee.org/.

[25] "Springer link online library," https://link.springer.com/.

[26] "Wiley online library," https://onlinelibrary.wiley.com/.

[27] "Elsevier sciencedirect," https://www.sciencedirect.com/.

[28] "Scopus," https://www.scopus.com/.

[29] G. Halevi, H. Moed, and J. Bar-Ilan, "Suitability of google scholar as a source of scientific information and as a source of data

for scientific evaluation: Review of the literature," *Journal of Informetrics*, vol. 11, no. 3, pp. 823–834, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1751157717300676

[30] W. Alhoshan, L. Zhao, A. Ferrari, and K. J. Letsholo, "A zero-shot learning approach to classifying requirements: A preliminary study," in *Requirements Engineering: Foundation for Software Quality: 28th International Working Conference, REFSQ 2022, Proceedings*, 2022, p. 52?59.

[31] T. Ahmed, P. Devanbu, and A. A. Sawant, "Learning to find usage of library functions in optimized binaries," *IEEE Transactions on Software Engineering*, 2021.

[32] S. Zafar, M. Z. Malik, and G. S. Walia, "Towards standardizing and improving classification of bug-fix commits," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019, pp. 1–6.

[33] S. Chakraborty and B. Ray, "On multi-modal learning of editing source code," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 443–455.

[34] J. Zhou, M. Pacheco, Z. Wan, X. Xia, D. Lo, Y. Wang, and A. E. Hassan, "Finding a needle in a haystack: Automated mining of silent vulnerability fixes," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 705–716.

[35] P. He, C. Meister, and Z. Su, "Structure-invariant testing for machine translation," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 961–973.

[36] Y. Qu, J. Chi, and H. Yin, "Leveraging developer information for efficient effort-aware bug prediction," *Information and Software Technology*, vol. 137, p. 106605, 2021.

[37] Z. Gao, X. Xia, D. Lo, J. Grundy, and T. Zimmermann, "Automating the removal of obsolete todo comments," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 218–229.

[38] M. Liu, X. Peng, A. Marcus, C. Treude, X. Bai, G. Lyu, J. Xie, and X. Zhang, "Learning-based extraction of first-order logic representations of api directives," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 491–502.

[39] L. Mariani, A. Mohebbi, M. Pezzè, and V. Terragni, "Semantic matching of gui events for test reuse: are we there yet?" in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 177–190.

[40] L. Ghadhab, I. Jenhani, M. W. Mkaouer, and M. B. Messaoud, "Augmenting commit classification by using fine-grained source code changes and a pre-trained deep neural language model," *Information and Software Technology*, vol. 135, p. 106566, 2021.

[41] M. Ciniselli, N. Cooper, L. Pascarella, D. Poshyvanyk, M. Di Penta, and G. Bavota, "An empirical study on the usage of bert models for code completion," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 108–119.

[42] J. Lin, Y. Liu, Q. Zeng, M. Jiang, and J. Cleland-Huang, "Traceability transformed: Generating more accurate links with pre-trained bert models," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 324–335.

[43] A. Mastropaolo, E. Aghajani, L. Pascarella, and G. Bavota, "An empirical study on code comment completion," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 159–170.

[44] J. Sun, Z. Xing, X. Peng, X. Xu, and L. Zhu, "Task-oriented api usage examples prompting powered by programming task knowledge graph," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 448–459.

[45] T. Zhou, K. Liu, L. Li, Z. Liu, J. Klein, and T. F. Bissyandé, "Smartgift: Learning to generate practical inputs for testing smart contracts," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 23–34.

[46] X. Zhou, D. Han, and D. Lo, "Assessing generalizability of codebert," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 425–436.

[47] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[48] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," *arXiv preprint arXiv:2103.06333*, 2021.

[49] J. Henkel, D. Silva, L. Teixeira, M. d'Amorim, and T. Reps, "Shipwright: A human-in-the-loop system for dockerfile repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1148–1160.

[50] F. Liu, G. Li, Y. Zhao, and Z. Jin, "Multi-task learning based pre-trained language model for code completion," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 473–485.

[51] G. Heyman, R. Huysegems, P. Justen, and T. Van Cutsem, "Natural language-guided programming," in *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2021, pp. 39–55.

[52] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1161–1173.

[53] M. H. Asyrofi, F. Thung, D. Lo, and L. Jiang, "Crossasr: Efficient differential testing of automatic speech recognition via text-to-speech," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 640–650.

[54] K. Pei, J. Guan, M. Broughton, Z. Chen, S. Yao, D. Williams-King, V. Ummadisetty, J. Yang, B. Ray, and S. Jana, "Stateformer: fine-grained type recovery from binaries using generative state modeling," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 690–702.

[55] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, "Code generation as a dual task of code summarization," *Advances in neural information processing systems*, vol. 32, 2019.

[56] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139

[57] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT*, J. Burstein, C. Doran, and T. Solorio, Eds. Association for Computational Linguistics, 2019, pp. 4171–4186.

[58] I. Ganguli, R. S. Bhowmick, S. Biswas, and J. Sil, "Empirical auto-evaluation of python code for performance analysis of transformer network using t5 architecture," in *2021 8th International Conference on Smart Computing and Communications (ICSCC)*, 2021, pp. 75–79.

[59] B. Berabi, J. He, V. Raychev, and M. Vechev, "Tfix: Learning to fix coding errors with a text-to-text transformer," in *Proceedings of the 38th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 18–24 Jul 2021, pp. 780–791.

[60] F. Zhang, X. Yu, J. Keung, F. Li, Z. Xie, Z. Yang, C. Ma, and Z. Zhang, "Improving stack overflow question title generation with copying enhanced codebert model and bi-modal information," *Information and Software Technology*, vol. 148, p. 106922, 2022.

[61] C. Pan, M. Lu, and B. Xu, "An empirical study on software defect prediction using codebert model," *Applied Sciences*, vol. 11, no. 11, 2021.

[62] "μjava," https://cs.gmu.edu/~offutt/mujava/.

[63] "Pit," https://pitest.org.

[64] "javalanche," https://www.st.cs.uni-saarland.de/mutation/.

[65] "Jester," http://jester.sourceforge.net.

[66] M. Tufano, F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia, and D. Poshyvanyk, "There and back again: Can you compile that snapshot?" *J. Softw. Evol. Process.*, vol. 29, no. 4, 2017.

[67] "Javaparser," https://javaparser.org.

[68] "Pit mutation operators," https://pitest.org/quickstart/mutators/.

[69] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in github for MSR studies," in *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*. IEEE, 2021, pp. 560–564.

[70] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers," *CoRR*, vol. abs/2009.05617, 2020. [Online]. Available: https://arxiv.org/abs/2009.05617

[71] "langid.py," https://github.com/saffsd/langid.py, accessed:2022-05-22.

[72] "pycld3," https://pypi.org/project/pycld3/, accessed: 2022-05-22.

[73] A. LeClair and C. McMillan, "Recommendations for datasets for source code summarization," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Jun. 2019, pp. 3931–3937.

[74] Z. Chen, S. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *CoRR*, vol. abs/1901.01808, 2019.

[75] "FunCom dataset," http://leclair.tech/data/funcom/.

[76] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[77] Q. McNemar, "Note on the sampling error of the difference between correlated proportions or percentages," *Psychometrika*, vol. 12, no. 2, pp. 153–157, 1947.

[78] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian journal of statistics*, pp. 65–70, 1979.

[79] A. Robins, "Catastrophic forgetting, rehearsal and pseudorehearsal," *Connection Science*, vol. 7, no. 2, pp. 123–146, 1995.

[80] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: A method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ser. ACL '02, 2002, pp. 311–318.

[81] V. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals," *Soviet Physics Doklady*, vol. 10, p. 707, 1966.

[82] T. H. Le, H. Chen, and M. A. Babar, "Deep learning for source code modeling and generation: Models, applications, and challenges," *ACM Computing Surveys (CSUR)*, vol. 53, no. 3, pp. 1–38, 2020.

[83] F. Ferreira, L. L. Silva, and M. T. Valente, "Software engineering meets deep learning: a mapping study," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, 2021, pp. 1542–1549.

[84] C. Watson, N. Cooper, D. N. Palacio, K. Moran, and D. Poshyvanyk, "A systematic literature review on the use of deep learning in software engineering research," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 2, pp. 1–58, 2022.

[85] M. Paltenghi and M. Pradel, "Thinking like a developer? comparing the attention of humans with neural models of code," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 867–879.

[86] F. Chen, F. Fard, D. Lo, and T. Bryksin, "On the transferability of pretrained language models for low-resource programming languages," in *2022 IEEE/ACM International Conference on Program Comprehension (ICPC)*, 2022, p. To appear.

[87] T. Ahmed and P. T. Devanbu, "Multilingual training for software engineering," *CoRR*, vol. abs/2112.02043, 2021. [Online]. Available: https://arxiv.org/abs/2112.02043