Using Transfer Learning for Code-Related Tasks

Antonio Mastropaolo, Nathan Cooper, David Nader Palacio, Simone Scalabrino,
Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota

Abstract—Deep learning (DL) techniques have been used to support several code-related tasks such as code summarization and bug-fixing. In particular, pre-trained transformer models are on the rise, also thanks to the excellent results they achieved in Natural Language Processing (NLP) tasks. The basic idea behind these models is to first pre-train them on a generic dataset using a self-supervised task (*e.g.,* filling masked words in sentences). Then, these models are fine-tuned to support specific tasks of interest (*e.g.,* language translation). A single model can be fine-tuned to support multiple tasks, possibly exploiting the benefits of *transfer learning*. This means that knowledge acquired to solve a specific task (*e.g.,* language translation) can be useful to boost performance on another task (*e.g.,* sentiment classification). While the benefits of transfer learning have been widely studied in NLP, limited empirical evidence is available when it comes to code-related tasks. In this paper, we assess the performance of the Text-To-Text Transfer Transformer (T5) model in supporting four different code-related tasks: (i) automatic bug-fixing, (ii) injection of code mutants, (iii) generation of assert statements, and (iv) code summarization. We pay particular attention in studying the role played by pre-training and multi-task fine-tuning on the model's performance. We show that (i) the T5 can achieve better performance as compared to state-of-the-art baselines; and (ii) while pre-training helps the model, not all tasks benefit from a multi-task fine-tuning.

Index Terms—Deep Learning, Empirical Software Engineering

1 Introduction

Several code-related tasks have been recently automated by researchers exploiting Deep Learning (DL) techniques [81]. Several of these works customize DL models proposed in the Natural Language Processing (NLP) field to support code-related tasks, and most of them share one common characteristic: *They shape the problem at hand as a text-to-text transformation, in which the input and the output of the model are text strings*. For instance, Tufano *et al.* [78] used an encoder-decoder architecture, commonly adopted in Neural Machine Translation (NMT) [16], [33], [69], to predict code changes usually recommended by reviewers in a code review process. Both the input and output are represented as a stream of tokens (*i.e.,* textual format), with the input being the code submitted for review and the output a revised code implementing changes likely to be required in the code review process. While this is only one concrete example, similar observations hold for techniques automating bug fixing [15], [25], [48], [75], learning generic code changes [73], supporting code migration [52], [53], code summarization [24], [32], [39], [42], code reviews [77], [78], pseudo-code generation [55], code deobfuscation [31], [79], injection of code mutants [76], generation of assert statements [82], clone detection [74], [83], traceability [49] and code completion [5], [11], [17], [17], [34], [35], [70], [84].

Recent years have seen the rise of *transfer learning* in the field of natural language processing. The basic idea is to first pre-train a model on a large and generic dataset by using a self-supervised task, *e.g.,* masking tokens in strings and asking the model to guess the masked tokens. Then, the trained model is fine-tuned on smaller and specialized datasets, each one aimed at supporting a specific task. In this context, Raffel *et al.* [60] proposed the T5 (Text-To-Text Transfer Transformer) model, pre-trained on a large natural language corpus and fine-tuned to achieve state-of-the-art performance on many tasks, all characterized by text-to-text transformations.

In our recent work [44] we empirically investigated the potential of a T5 model when pre-trained and fine-tuned to support four code-related tasks also characterized by text-to-text transformations. In particular, we started by pre-training a T5 model using a large dataset consisting of 499,618 English sentences and 1,569,889 source code components (*i.e.,* Java methods). Then, we fine-tuned the model using four datasets from previous work with the goal of supporting four code-related tasks:

Automatic bug-fixing. We used the dataset by Tufano *et al.* [75], composed of instances in which the "input string" is represented by a buggy Java method and the "output string" is the fixed version of the same method.

Injection of code mutants. This dataset is also by Tufano *et al.* [76], and features instances in which the input-output strings are reversed as compared to automatic bug-fixing (*i.e.,* the input is a fixed method, while the output is its buggy version). The model must learn how to inject bugs (mutants) in code instead of fixing bugs.

Generation of assert statements in test methods. We used the

- *A. Mastropaolo is with SEART @ Software Institute, Università della Svizzera italiana, Switzerland. E-mail: antonio.mastropaolo@usi.ch*
- *N. Cooper is with SEMERU @ William & Mary, USA. E-mail: nacooper01@email.wm.edu*
- *D. Nader Palacio is with SEMERU @ William & Mary, USA. E-mail: danaderp@gmail.com*
- *S. Scalabrino is with University of Molise, Italy. E-mail: simone.scalabrino@unimol.it*
- *D. Poshyvanyk is with SEMERU @ William & Mary, USA. E-mail: denys@cs.wm.edu*
- *R. Oliveto is with University of Molise, Italy. E-mail: rocco.oliveto@unimol.it*
- *G. Bavota is with SEART @ Software Institute, Università della Svizzera italiana, Switzerland. E-mail: gabriele.bavota@usi.ch*

dataset by Watson *et al.* [82], composed of instances in which the input string is a representation of a test method without an assert statement and a focal method it tests (*i.e.,* the main production method tested), while the output string encodes an appropriate assert statement for the input test method.

Code Summarization. We used the dataset by Haque *et al.* [24] where input strings are some representations of a Java method to summarize, & an output string is a textual summary.

We fine-tuned a single pre-trained T5 model in a multi-task setting on all four tasks, and showed that it is able to achieve better results as compared to the four referenced baselines in all tasks [24], [75], [76], [82]. However, since we only experimented with a pre-trained model fine-tuned in a multi-task setting, questions about the actual advantage offered by transfer learning remained unanswered. In this work, we aim at overcoming such a limitation that is also typical of several other works in the literature using off-the-shelf pre-trained models like T5 to support code related tasks (*e.g.,* [61], [87]). Indeed, little effort has been spent on understanding the actual benefits (if any) that transfer learning brings when dealing with code-related tasks. Such observation holds for both (i) the pre-training phase, that should provide the model with general knowledge about a language of interest (*e.g.,* Java) being at the core of the tasks to automate (*e.g.,* bug-fixing); and (ii) the multi-task fine-tuning, that should allow the model to exploit knowledge acquired when trained for a specific task (*e.g.,* bug-fixing) also for the automation of other tasks (*e.g.,* generation of assert statements), thus possibly boosting the overall performance in all the tasks. Besides the expected positive impact on performance, pre-training and multi-task fine-tuning are also useful in real-life scenarios in which the training data for a particular task of interest is scarce (*e.g.,* when manually labeled instances are needed) [63]. Pre-training the model in an unsupervised setting and/or fine-tuning it on other related tasks for which more training data is available can unlock the possibility of using deep learning models also for tasks characterized by scarcity of training data.

In this paper, we extend our previous work [45] by carefully assessing the impact of both pre-training and multi-task fine-tuning on the T5 performance. In particular, we assess the performance of the T5 in the following scenarios:

- **No Pre-training:** We do not run any pre-training step. We directly fine-tune four different T5 models, each one supporting one of the four tasks we experiment with.
- **Pre-training single task:** We first pre-train the T5 model on the dataset presented in Table 1. Then, starting from it, we fine-tune four models, one for each single task.
- **Pre-training Multi-Task**: Lastly, we fine-tune the pre-trained model using a multi-task learning framework in which we train a single model to support all four code-related tasks. We experiment with two different multi-task fine-tunings: (i) the first is the one used in our original paper [45], in which the percentage of training instances from each of the four tasks is proportional to the size of their training dataset; (ii) the second in which the percentage of training instances is the same for all four tasks (*i.e.,* 25% per task).

In total, this resulted in the training, hyperparameters tuning, and testing of ten different models. Note that the choice of the four tasks subject of our study (*i.e., bug-fixing*, *mutants injection*, *asserts generation*, and *code summarization*) is dictated by the will of experimenting with tasks that use, represent, and manipulate code in different ways. In particular, we include in our study tasks aimed at (i) transforming the input code with the goal of changing its behavior (*bug-fixing* and *mutants injection*); (ii) "comprehending code" to verify its behavior (*asserts generation*); and (iii) "comprehending code" to summarize it in natural language (*code summarization*). Also, following what has been done in the original datasets from previous work, the four tasks involve abstracted source code (*bug-fixing* [75], *mutants injection* [76], and *asserts generation* [82]), raw source code (*asserts generation* [82] and *code summarization* [24]), and natural language (code summarization [24]). Such a mix of tasks helps in increasing the generalizability of our findings.

We also perform a novel analysis of our dataset aimed at assessing the generalizability of our models by looking at the level of data snooping among our training and test datasets.

Our results confirm that the T5 can substantially boost the performance on all four code-related tasks. For example, when the T5 model is asked to generate assert statements on raw source code, ∼70% of test instances are successfully predicted by the model, against the 18% of the original baseline [82]. Also, we show that the pre-training is beneficial for all tasks, while the multi-task fine-tuning does not consistently help in improving performance. Finally, our datasets analysis confirm the generalizability of the tested models. The code and data used in this work are publicly available [2].

2 Background and Related Work

In recent years, DL techniques have been increasingly used to support software engineering (SE). The activities commonly supported by state-of-the-art approach include software maintenance and software testing [86], and most of the proposed approaches target the source code [81]. While available approaches support a plethora of concrete SE tasks [81], [86], in this section we focus on the ones we target in our study: automated bug-fixing, injection of code mutants, generation of assert statements in test methods, and code summarization. We discuss in detail the techniques we use as baselines for each task. A broader literature review on the topic is available in two recent surveys by Yang *et al.* [86] and Watson *et al.* [81].

2.1 Automatic Bug-Fixing

Many techniques have been proposed for the automatic fixing of software bugs. Several of them [7], [13], [20], [21], [38], [54], [58], [66], [85] rely on the *redundancy assumption*, claiming that large programs contain the seeds of their own repair. Such an assumption has been verified by at least two independent studies [9], [43]. Automated bug-fixing techniques based on DL can rely on different levels of code abstraction. Word tokenization is a commonly used one, even if higher-level abstractions (*e.g.,* AST-based) allow to achieve better results [51].

Mesbah *et al.* [48] focus on build-time compilation failures by presenting DeepDelta, an approach using NMT to fix the

build. The input is represented by features characterizing the compilation failure (*e.g.,* kind of error, AST path, etc.). As output, DeepDelta provides the AST changes needed to fix the error. In the presented empirical evaluation, DeepDelta correctly fixed 19,314 out of 38,788 (50%) compilation errors.

Chen *et al.* [15] present SequenceR, a sequence-to-sequence approach trained on over 35k single-line bug-fixes. SequenceR takes as input the buggy line together with relevant code lines from the buggy class (*abstract buggy context*). The output of the approach is the recommended fix for the buggy line. The approach, tested on a set of 4,711 bugs, was able to automatically fix 950 (∼20%) of them. Similar approaches have been proposed by Hata *et al.* [25] and Tufano *et al.* [75]. The latter is the one we compared our approach with and, thus, we describe it in more details.

Tufano *et al.* [75] investigate the performance of an NMT-based approach in the context of automatic bug-fixing. They train an encoder-decoder model on a set of bug-fix pairs (BFPs), meaning pairs of strings in which the first one (input) represents a Java method that has been subject to a bug-fixing activity, and the second one (target) represents the same Java method once the bug was fixed. To build this dataset, the authors mined ∼787k bug-fixing commits from GitHub, from which they extracted ∼2.3M BFPs. After that, the code of the BFPs is abstracted to make it more suitable for the NMT model (*i.e.,* to reduce the vocabulary of terms used in the source code identifiers and literals). The abstraction process is depicted in Fig. 1.

raw source code

```
public Integer getMinElement(List myList) {
   if(myList.size() >= 0) {
      return ListManager.getFirst(myList);
   }
   return 0;
}
```

abstracted code

```
public TYPE_1 METHOD_1 ( TYPE_2 VAR_1 )
{ if ( VAR_1 . METHOD_2 ( ) >= INT_1 )
{ return TYPE_3 . METHOD_3 ( VAR_1 ) ; }
return INT_1 ; }
```

abstracted code with idioms

```
public TYPE_1 METHOD_1 ( List VAR_1 )
{ if ( VAR_1 . size ( ) >= 0 )
{ return TYPE_2 . METHOD_3 ( VAR_1 ) ; }
return 0 ; }
```

Fig. 1: Abstraction process [75]

The top part of the figure represents the raw source code to abstract. The authors use a Java lexer and a parser to represent each method as a stream of tokens, in which Java keywords and punctuation symbols are preserved and the role of each identifier (*e.g.,* whether it represents a variable, method, etc.) as well as the type of a literal is discerned.

IDs are assigned to identifiers and literals by considering their position in the method to abstract: The first variable name found will be assigned the ID of VAR_1, likewise the second variable name will receive the ID of VAR_2. This process continues for all identifiers as well as for the literals (*e.g.,* STRING_X, INT_X, FLOAT_X). The output of this stage is the code reported in the middle of Fig. 1 (*i.e.,* abstracted code). Since some identifiers and literals appear very often in the code (*e.g.,* variables `i`, `j`, literals `0`, `1`, method names such as `size`), those are treated as "idioms" and are not abstracted (see bottom part of Fig. 1, idioms are in bold). Tufano *et al.* consider as idioms the top 0.005% frequent words in their dataset. During the abstraction a mapping between the raw and the abstracted tokens is maintained, thus allowing to reconstruct the concrete code from the abstract code generated by the model.

The set of abstracted BFPs has been used to train and test the approach. The authors build two different sets, namely BFP_{small}, only including methods having a maximum length of 50 tokens (for a total of 58,350 instances), and BFP_{medium}, including methods up to 100 tokens (65,455). The model was able to correctly predict the patch for the buggy code in 9% and 3% of cases in the BFP_{small} and BFP_{medium} dataset, respectively.

While other works have tackled the automatic bug-fixing problem, the approach by Tufano *et al.* has been tested on a variety of different bugs, rather than on specific types of bugs/warnings (*e.g.,* only single-line bugs are considered in [15], while compilation failures are addressed in [48]).

Thus, we picked it as representative DL technique for automatic bug-fixing and we use the two datasets by Tufano *et al.* [75] to fine-tune the T5 model for the "automatic bug-fixing" problem, comparing the achieved performance with the one reported in the original paper.

2.2 Injection of Code Mutants

Brown *et al.* [12] were the first to propose a data-driven approach for generating code mutants, leveraging bug-fixes performed in software systems to extract syntactic-mutation patterns from the diffs of patches. Tufano *et al.* [76] built on this concept by presenting an approach using NMT to inject mutants that mirror real bugs. The idea is to reverse the learning process used for fixing bugs [75]: The model is trained to transform correct methods (*i.e.,* the method obtained after the bug-fixing activity) into buggy methods (before the bug-fix). Indeed, the methodology used by the authors is the same used for the bug-fixing task (previously described), including the abstraction process.

This is, to date, the only DL-based technique for injecting code mutants. Thus, we use the dataset exploited by Tufano *et al.* [76] to fine-tune the T5 model for the problem of "injecting code mutants", comparing the achieved results with the ones reported in the original paper. Specifically, we reused their largest dataset, referred to as GM_{ident} in the paper[1], featuring 92,476 training instances, 11,560 used for hyperparameter tuning (evaluation set), and 11,559 used for testing. On this data, the approach by Tufano *et al.* was able to correctly predict the bug to inject in 17% of cases (1,991).

2.3 Generation of Assert Statements in Test Methods

Watson *et al.* [82] start from the work by Shamshiri *et al.* [65], who observed that tools for the automatic generation of test cases such as Evosuite [19], Randoop [56] and Agitar [3] exhibit insufficiencies in the automatically generated assert statements.

Thus, they propose ATLAS, an approach for generating syntactically and semantically correct unit test assert

1. A subset of this dataset named $GM_{ident-lit}$ has also been used in the original paper [76] to avoid including in the study bugs requiring the generation of previously unseen literals. We decided to test the T5 model on the most complex and complete dataset.

statements using NMT. To train ATLAS, the authors mined 2.5M test methods from GitHub with their corresponding `assert` statement. For each of those test methods, they also identified the focal method, meaning the main production code method exercised by the test. A preprocessing of the dataset has been performed to remove all test methods longer than 1K tokens. Also, test methods requiring the synthesis of one or more unknown tokens for generating the appropriate assert statements have been removed. Indeed, if the required tokens cannot be found in the vocabulary of the test method they cannot be synthesized when the model attempts to generate the prediction. Finally, all duplicates have been removed from the dataset, leading to a final set of 158,096 Test-Assert Pairs (TAPs). Each method left in the dataset has then been abstracted using the same approach previously described by Tufano *et al.* [75]. However, in this case the authors experiment with two datasets, one containing raw source code and one abstracted code. ATLAS was able to generate asserts identical to the ones written by developers in 31.42% of cases (4,968 perfectly predicted assert statements) when only considering the top-1 prediction, and 49.69% (7,857) when looking at the top-5 in the abstracted dataset, while performance is lower on the raw dataset (17.66% for top-1 and 23.33% for top-5).

We use the datasets by Watson *et al.* [82] to fine-tune our T5 model for the "generation of assert statements" problem, and compare the achieved performance with the one in the original paper. Recently, Tufano *et al.* [72] proposed an approach based on transformers to achieve a the same goal. Their results show that such an approach achieves better results than ATLAS [82]. We did not use the approach proposed by Tufano *et al.* [72] as the main baseline because it is very similar to the one we presented in the our conference paper that this paper extends [45].

2.4 Code Summarization

Code summarization is one of the mainstream methods for automatic documentation of source code. The proposed summarization techniques fall into two categories. *Extractive* summarization techniques generate summaries by extracting information from the code components being summarized [23], [50], [64], [68]. On the other hand, *abstractive* summarization techniques aim at including in the summaries information not directly available in the source code [24], [28], [32], [46], [67]. DL techniques have been used to support for the latter.

Hu *et al.* [28] use a Deep Neural Network (DNN) to automatically generate comments for a given Java method. The authors mine ∼9k Java projects hosted on GitHub to collect pairs of ⟨method, comment⟩, where "comment" is the first sentence of the Javadoc linked to the method. These pairs, properly processed, are used to train and test the DNN. The authors assess the effectiveness of their technique by using the BLEU-4 score [57], showing the superiority of their approach with respect to the competitive technique presented in [30].

Allamanis *et al.* [4] use attention mechanisms in neural networks to suggest a descriptive method name starting from an arbitrary snippet of code. Their approach can name a code snippet exactly as a developer would do in ∼25% of cases.

LeClair *et al.* [39] present a neural model combining the AST source code structure and words from code to generate coherent summaries of Java methods. The approach, tested on 2.1M methods, showed its superiority as compared to the previous works by Hu *et al.* [28] and Iyer *et al.* [30].

The approach by Haque *et al.* [24] is the most recent in the area of DL-aided source code summarization, and it is an improvement of the work by LeClair *et al.* [39].

It still aims at documenting Java methods through an encoder-decoder architecture but, in this case, three inputs are provided to the model to generate the summary: (i) the source code of the method, as a flattened sequence of tokens representing the method; (ii) its AST representation; and (iii) the "file context", meaning the code of every other method in the same file. The authors show that adding the contextual information as one of the inputs substantially improves the BLEU score obtained by deep learning techniques. The dataset used in the evaluation is composed of 2.1M Java methods paired with summaries. We reuse this dataset for the fine-tuning of the T5 model for the code summarization problem, and compare its performance to the state-of-the-art approach proposed by Haque *et al.* [24].

3 Text-to-Text-Transfer-Transformer

The T5 model has been introduced by Raffel *et al.* [60] to support multitask learning in Natural Language Processing (NLP). The idea is to reframe NLP tasks in a unified text-to-text format in which the input and output are always text strings. For example, a single model can be trained to translate across languages and to autocomplete sentences. This is possible since both tasks can be represented in a text-to-text format (*e.g.*, in the case of translation, the input is a sentence in a given language, while the output is the translated sentence). T5 is trained in two phases: *pre-training*, which allows defining a shared knowledge-base useful for a large class of sequence-to-sequence tasks (*e.g.*, guessing masked words in English sentences to learn about the language), and *fine-tuning*, which specializes the model on a specific downstream task (*e.g.*, learning the translation of sentences from English to German). We briefly overview the T5 model and explain how we pre-trained and fine-tuned it to support the four said code-related tasks. Finally, we describe the decoding strategy for generating the predictions.

3.1 An Overview of T5

T5 is based on the transformer model architecture that allows handling a variable-sized input using stacks of self-attention layers. When an input sequence is provided, it is mapped into a sequence of embeddings passed into the encoder. The T5, in particular, and a transformer model [80], in general, offer two main advantages over other state-of-the-art models: (i) it is more efficient than RNNs since it allows to compute the output layers in parallel, and (ii) it is able to detect hidden and long-ranged dependencies among tokens, without assuming that nearest tokens are more related than distant ones. This last property is particularly relevant in code-related tasks since a variable declaration may be distant from its usage. Five different versions of T5 have been proposed [60]: *small*, *base*, *large*, *3 Billion*, and *11 Billion*. These

variants differ in terms of complexity, with the smaller model ($T5_{small}$) having 60M parameters against the 11B of the largest one ($T5_{11B}$). As acknowledged by the authors [60], even if the accuracy of the most complex variants is higher than the less complex models, the training complexity increases with the number of parameters. Considering the available computational resources, we decided to use the simplest $T5_{small}$ model.

$T5_{small}$ architectural details. The $T5_{small}$ architecture is characterized by six blocks for encoders and decoders. The feed-forward networks in each block consist of a dense layer with an output dimensionality (d_{ff}) of 2,048. The *key* and *value* matrices of all attention mechanisms have an inner dimensionality (d_{kv}) of 64, and all attention mechanisms have eight heads. All the other sub-layers and embeddings have a dimensionality (d_{model}) of 512.

3.2 Pre-training of T5

In the *pre-training* phase we use a self-supervised task similar to the one used by Raffel *et al.* [60], consisting of masking tokens in natural language sentences and asking the model to guess the masked tokens. However, we did not perform the pre-training by only using natural language sentences, since all the tasks we target involve source code. We use a dataset composed of both (technical) natural language (*i.e.,* code comments) and source code. To obtain the dataset for the pre-training we start from the CodeSearchNet dataset [29] which provides 6M functions from open-source code. We only focus on the ∼1.5M methods written in Java, since the four tasks we aim at supporting are all related to Java code and work at method-level granularity (*e.g.,* fixing a bug in a method, generating the summary of a method, etc.).

Then, since for three of the four tasks we support (*i.e., automatic bug-fixing* [75], *generation of assert statements* [82], and *injection of code mutants* [76]) the authors of the original papers used an abstracted version of source code (see Section 2), we used the `src2abs` tool by Tufano [75] to create an abstracted version of each mined Java method. In the abstraction process, special tokens are used to represent identifiers and literals of the input method. For example, the first method name found (usually the one in the method signature) will be assigned the `METHOD_1` token, likewise the second method name (*e.g.,* a method invocation) will be represented by `METHOD_2`. This process continues for all the method and variable names (`VAR_X`) as well as the literals (`STRING_X`, `INT_X`, `FLOAT_X`). Basically, the abstract method consists of language keywords (*e.g.,* `for`, `if`), separators (*e.g.,* "(", ";", "}") and special tokens representing identifiers and literals. Comments and annotations are removed during abstraction. Note that, since the tool was run on Java methods in isolation (*i.e.,* without providing it the whole code of the projects they belong to), `src2abs` raised a parsing error in ∼600k of the ∼1.5M methods (due *e.g.,* to missing references), leaving us with ∼900k abstracted methods. We still consider such a dataset as sufficient for the pre-training.

The CodeSearchNet dataset does also provide, for a subset of the considered Java source code methods, the first sentence in their Javadoc. We extracted such a documentation using the `docstring_tokens` field in CodeSearchNet, obtaining it for 499,618 of the considered methods. We added these sentences to the pre-training dataset. This whole process resulted in a total of 2,984,627 pre-training instances, including raw source code methods, abstracted methods, and code comment sentences. In the obtained dataset there could be duplicates between (i) different raw methods that become equal once abstracted, and (ii) comments re-used across different methods. Thus, we remove duplicates, obtaining the final set of 2,672,423 instances reported in Table 1. This is the dataset we use for pre-training the T5 model, using the BERT-style objective function Raffel *et al.* used in their experiments and consisting of randomly masking 15% of tokens (*i.e.,* words in comments and code tokens in the raw and abstracted code).

TABLE 1: Datasets used for the pre-training of T5.

Data sources	Instances
Source code	1,569,773
Abstracted source code	766,126
Technical natural language	336,524
Total	2,672,423

Finally, since we pre-train and fine-tune the models on a software-specific dataset, we create a new *SentencePiece* model [37] (*i.e.,* a tokenizer for neural text processing) by training it on the entire pre-training dataset so that the T5 model can properly handle the *Java* language and its abstraction. This model implements subword units (*e.g.,* byte-pair-encoding BPE) and unigram language model [36] to alleviate the open vocabulary problem in neural machine translation. The pre-training of the models has been performed for 250k steps which, using a batch size of 128 results in ∼32M of masked code instances processed that, given the size of the pre-training dataset (see Table 1) correspond to ∼12 epochs.

3.3 Fine-tuning of T5

We detail the process used to fine-tune the T5 model. Before explaining how the training instances are represented within each fine-tuning dataset, it is important to clarify that both in the pre-training and in the fine tuning the T5 can handle any sort of training instance as long as it can be formulated as a text-to-text transformation. Indeed, the T5 represents each training dataset as a $N \times 2$ matrix, where N is the number of instances in the dataset and the 2 dimensions allow to express the input text and the expected output text. In the case of pre-training, the input text is an instance (*i.e.,* a raw method, an abstract method, or a Javadoc comment) in which 15% of tokens have been masked, while the output text represents the correct predictions for the masked tokens. In the four downstream tasks, instead, the text-to-text pairs are represented as explained in the following.

3.3.1 Fine-tuning dataset

We describe the datasets we use for *fine-tuning* the model for the four targeted tasks. The datasets are summarized in Table 2. The number of training steps performed for the different tasks is proportional to the size of their training dataset. Indeed, we aim at ensuring that the same number of "epochs" is performed on each training dataset. Thus, smaller training datasets require a lower number of steps

to reach the same number of epochs of larger datasets. In particular, we used 1.75M fine-tuning steps for the *multi-task* setting ∼90 epochs) and we scaled the others proportionally to reach the same number of epochs (*e.g.,* ∼1.41M for the *code summarization* task).

Automatic Bug Fixing (BF). We use the dataset by Tufano *et al.* [75] composed by triplets $BF_m = \langle m_b, m_f, M \rangle$, where m_b and m_f are the abstracted version of the buggy and fixed version of Java method, respectively, and M represents the mapping between the abstracted tokens and the raw code tokens (*e.g.,* `VAR_1` → `webServerPort`), which allows to track back the output of the model to source code. The triplets refer to methods with at most 100 tokens and they are split into two sub-datasets: (i) the *small* version, containing methods with up to 50 tokens, and a *medium* version, with methods with at most 100 tokens. We train the model to predict the fixed versions, m_f, given the buggy versions, m_b. Given the presence of two datasets, we divide the BF task in two sub-tasks, BF_{small} and BF_{medium}, depending on the size of the involved methods [75].

Injection of Code Mutants (MG). For the MG task we exploited one of the two datasets provided by Tufano *et al.* [73]: MG_{ident} and $MG_{ident-lit}$. In both datasets each instance is represented by a triple $\langle m_f, m_b, M \rangle$, where, similarly to the BF datasets, m_b and m_f are the buggy and fixed version of the snippet, respectively, and M represents the mapping between the abstracted tokens and the code tokens. The first dataset (MG_{ident}) represents the most general (and challenging) case, in which the mutated version, m_b, can also contain new tokens (*i.e.,* identifiers, types, or method names) not contained in the version provided as input (m_f). $MG_{ident-lit}$, instead, only contains samples in which the mutated version contains a subset of the tokens in the non-mutated code. In other words, $MG_{ident-lit}$ represents a simplified version of the task. For this reason, we decided to focus on the most general scenario and we only use the MG_{ident} dataset.

Generation of Assertions in Test Methods (AG). For the AG task we used the dataset provided by Watson *et al.* [82] containing triplets $\langle T, TM_n, A \rangle$, where T is a given test case, TM_n is the *focal* method tested by T, *i.e.,* the last method called in T before the assert [59], and A is the assertion that must be generated (output). For such a task, we use two versions of the dataset: AG_{raw}, which contains the raw source code for the input ($T + TM_n$) and the output (A), and AG_{abs}, which contains the abstracted version of input and output, *i.e.,* $src2abs(T + TM_n)$ and $src2abs(A)$, respectively. These are the same datasets used in the original paper.

Code Summarization (CS). For code summarization, we exploited the dataset provided by Haque *et al.* [24] containing 2,149,120 instances, in which each instance is represented by a tuple $\langle S, A_S, C_S, D \rangle$, where S represents the raw source code of the method, A_S is its AST representation, C_S is the code of other methods in the same file, and D is the summary of the method, *i.e.,* the textual description that the model should generate [24]. For this specific task, we consider a variation of the original dataset to make it more coherent with the performed pre-training. In particular, since in the pre-training we did not use any AST representation of code, we decided to experiment with the T5 model in a more challenging scenario in which only the raw source code to summarize (*i.e.,* S) is available to the model. Therefore, the instances of our dataset are represented by tuples $\langle S, D \rangle$: We train our model to predict D given only S.

3.3.2 Decoding Strategy

Once the models have been trained, different decoding strategies can be used to generate the output token streams. T5 allows to use both *greedy decoding* and *Beam-search*. When generating an output sequence, the greedy decoding selects, at each time step t, the symbol having the highest probability. The main limitation of greedy decoding is that it only allows the model to generate one possible output sequence (*e.g.,* one possible bug fix) for a given input (*e.g.,* the buggy method).

Beam-search is an alternative decoding strategy previously used in many DL applications [8], [10], [22], [62]. Unlike greedy decoding, which keeps only a single hypothesis during decoding, beam-search of order K, with $K > 1$, allows the decoder to keep K hypotheses in parallel: At each time step t, beam-search picks the K hypotheses (*i.e.,* sequences of tokens up to t) with the highest probability, allowing the model to output K possible output sequences.

We used Beam-search to provide several output sequences given a single input, and report results with different K values. It is worth noting that having a large K increases the probability that one of the output sequences is correct, but, on the other hand, it also increases the cost of manually analyzing the output for a user (*i.e.,* a developer, in our context).

3.3.3 Data Balancing for the multi-task model

The datasets we use for fine-tuning have different sizes, with the one for code summarization dominating the others (see Table 2). This could result in an unbalanced effectiveness of the model on the different tasks. In our case, the model could become very effective in summarizing code and less in the other three tasks. However, as pointed out by Arivazhagan *et al.* [6], there is no free lunch in choosing the balancing strategy when training a multi-task model, with each strategy having its pros and cons (*e.g.,* oversampling of less represented datasets negatively impacts the performance of the most representative task). For this reason, we decide to experiment with both strategies. In the first strategy, we follow the true data distribution when creating each batch. In other words, we sample instances from the tasks in such a way that each batch during the training has a proportional number of samples accordingly to the size of the training dataset. For the second strategy, we train a multi-task pre-trained model using a balanced sampling strategy. In other words, we feed the T5 model with batches of data having exactly the same number of samples per task randomly selected during the fine-tuning.

The results we obtained confirm the findings of Arivazhagan *et al.* [6]. In particular, when using the first training sampling strategy (*i.e.,* proportional sampling), the performance of the tasks having a large training dataset (*i.e.,* AG_{abs}, AG_{raw}, CS) had a boost. In contrast, when using the second strategy (*i.e.,* balanced sampling), the performance increases for those tasks whose training dataset is small with, however, a price to pay for the other three tasks. Nonetheless, since the observed differences in performance are not major and each strategy has its pros and cons, we decided to discuss

TABLE 2: Task-specific datasets used for fine-tuning T5.

Task	Dataset	Training-set	Evaluation-set	Test-set
Automatic Bug-Fixing	BF_{small} [75]	46,680	5,835	5,835
	BF_{medium} [75]	52,364	6,546	6,545
Injection of Code Mutants	MG_{ident} [76]	92,476	11,560	11,559
Generation of Asserts in Test	AG_{abs} [82]	126,477	15,809	15,810
	AG_{raw} [82]	150,523	18,816	18,815
Code Summarization	CS [24]	1,953,940	104,272	90,908
Total		2,422,460	162,838	149,472

in this paper the results achieved using the proportional sampling schema, as we did in [45].

The results of the proportional sampling are available in our replication package [2].

4 Study Design

We aim at investigating the performance of the T5 model on four code-related tasks: *Automatic bug-fixing, Injection of code mutants, Generation of Asserts in Tests and Code Summarization*. The focus of our evaluation is on (i) investigating the extent to which *transfer learning* is beneficial when dealing with code-related tasks, studying the impact on performance of both pre-training and multi-task learning; and (ii) comparing the obtained results with representative state-of-the-art techniques. The *context* is represented by the datasets introduced in Section 2, *i.e.,* the ones by Tufano *et al.* for bug fixing [75] and injection of mutants [76], by Watson *et al.* for assert statement generation [82], and by Haque *et al.* for code summarization [24]. We aim at answering the following research questions (RQs):

- **RQ$_1$:** *What are the performances of the T5 model when supporting code-related tasks?* With RQ$_1$ we aim at understanding the extent to which T5 can be used to automate code-related tasks, investigating the performance achieved by the model on the four experimented tasks. In the context of RQ$_1$, we also investigate the impact of transfer learning on performance:
  - **RQ$_{1.1}$:** *What is the role of pre-training on the performances of the T5 model for the experimented code-related tasks?* With RQ$_{1.1}$ we aim at investigating the boost in performance (if any) brought by pre-training the models on a software-specific dataset.
  - **RQ$_{1.2}$:** *What is the role of multi-task learning on the performances of the T5 model for the experimented code-related tasks?* RQ$_{1.2}$ analyzes the influence of the *multi-task learning* (*i.e.,* training a single model for all four tasks) on the model's performance.
- **RQ$_2$:** *What are the performances of T5 as compared with state-of-the-art baselines?* In RQ$_2$ we compare the performances achieved by the T5 model against the ones achieved by the baseline approaches. In this regard, we run T5 on the same test sets used in the four original papers presenting automated solutions for the code-related tasks we target.

4.1 Data Collection and Analysis

As explained in Section 3.3, we experimented with different variants of the T5: (i) *no pre-training* (*i.e.,* four models each fine-tuned for one of the supported tasks, without any pre-training); (ii) *pre-training single task* (*i.e.,* four models each fine-tuned for one of the supported tasks, with pre-training); and (iii) *pre-training multi-task* (*i.e.,* one model pre-trained and fine-tuned for all four tasks). These nine models have all been run on the test sets made available in the works presenting our four baselines and summarized in Table 2. Once obtained the predictions of the T5 models on the test sets related to the four tasks, we compute the evaluation metrics reported in Table 3. We use different metrics for the different tasks, depending on the metrics reported in the papers that introduced our baselines.

Accuracy@K measures the percentage of cases (*i.e.,* instances in the test set) in which the sequence predicted by the model equals the oracle sequence (*i.e.,* perfect prediction). Since we use beam-search, we report the results for different K values (*i.e.,* 1, 5, 10, 25, and 50), as done in [75] (BF) and [82] (AG). Tufano *et al.* [73] do not report results for $K > 1$ for the MG task. Thus, we only compute $K = 1$.

BLEU score (Bilingual Evaluation Understudy) [57] measures how similar the candidate (predicted) and reference (oracle) texts are. Given a size n, the candidate and reference texts are broken into n-grams and the algorithm determines how many n-grams of the candidate text appear in the reference text. The BLEU score ranges between 0 (the sequences are completely different) and 1 (the sequences are identical). We use different BLEU-n scores, depending on the ones used in the reference paper of the baseline (see Table 3). For the CS task, we report BLEU-{1, 2, 3, 4} and their geometric mean (*i.e.,* BLEU-A); for the MG task we only report BLEU-A.

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) is a set of metrics for evaluating both automatic summarization of texts and machine translation techniques in general [41]. ROUGE metrics compare an automatically generated summary or translation with a set of reference summaries (typically, human-produced). We use the ROUGE LCS metrics based on the Longest Common Subsequence for the CS task [24]. Given two token sequences, X and Y, and their respective length, m and n, it is possible to compute three ROUGE LCS metrics: R (recall), computed as $\frac{LCS(X,Y)}{m}$, P (precision), computed as $\frac{LCS(X,Y)}{n}$, and F (F-measure), computed as the harmonic mean of P and R.

The computed metrics are used to select what the best training strategy for the T5 is (*i.e., no pre-training*, *pre-training single task*, or *pre-training multi-task*). We also statistically compare the performance of these three strategies for each task using the McNemar's test [47], which is a proportion

TABLE 3: Baselines and evaluation metrics for the tasks.

Task	Baseline	Accuracy@K	BLEU-n	ROUGE LCS
Automatic Bug-Fixing	[75]	$\{1, 5, 10, 25, 50\}$	-	-
Injection of Code Mutants	[76]	$\{1\}$	$\{A\}$	-
Generation of Asserts in Test	[82]	$\{1, 5, 10, 25, 50\}$	-	-
Code Summarization	[24]	-	$\{1, 2, 3, 4, A\}$	$\{P, R, F\}$

test suitable to pairwise compare dichotomous results of two different treatments. We statistically compare each pair of training strategy in our study (*i.e., no pre-training vs pre-training single task*, *no pre-training vs pre-training multi-task*, *pre-training single task vs pre-training multi-task*) in terms of their Accuracy@1 (*i.e.,* perfect predictions) for each of the four experimented tasks. To compute the test results for two training strategies T_1 and T_2, we create a confusion matrix counting the number of cases in which (i) both T_1 and T_2 provide a correct prediction, (ii) only T_1 provides a correct prediction, (iii) only T_2 provides a correct prediction, and (iv) neither T_1 nor T_2 provide a correct prediction. We complement the McNemar's test with the Odds Ratio (OR) effect size. Also, since we performed multiple comparisons, we adjusted the obtained p-values using the Holm's correction [26].

The best model output of this analysis has then been used to compare the best T5 model with the four baselines by using the performance metrics reported in Table 3. Moreover, we also statistically compare the Accuracy@1 of the T5 and of the baselines using the same procedure previously described (*i.e.,* McNemar's test with the OR effect size). We also perform a complementarity analysis: We define the sets of perfect predictions generated by the T5 (PP_{T5_d}) and by the baseline (PP_{BL_d}) with a beam size $K = 1$. Then, for each task and dataset we compute three metrics:

$$Shared_d = \frac{|PP_{T5_d} \cap PP_{BL_d}|}{|PP_{T5_d} \cup PP_{BL_d}|}$$

$$OnlyT5_d = \frac{|PP_{T5_d} \setminus PP_{BL_d}|}{|PP_{T5_d} \cup PP_{BL_d}|} \qquad OnlyBL_d = \frac{|PP_{BL_d} \setminus PP_{T5_d}|}{|PP_{T5_d} \cup PP_{BL_d}|}$$

$Shared_d$ measures the percentage of perfect predictions shared between the two compared approaches on the dataset d, while $OnlyT5_d$ and $OnlyBL_d$ measure the percentage of cases in which the perfect prediction is only generated by T5 or the baseline, respectively, on the dataset d.

We also present an "inference time" analysis: we compute the time needed to run T5 on a given input. We run such an experiment on a laptop equipped with a 2.3GHz 8-core 9th-generation Intel Core i9 and 16 GB of RAM, using the CPU to run the DL model. We do this for different beam search sizes, with $K \in \{1, 5, 10, 25, 50\}$. For each K, we report the average inference time (in seconds) on all the instances of each task. Besides that, we also report the training time (in hours) for the nine different models involved in our study, *i.e., no pre-training* (four models, one for each task), *pre-training single task* (+4 models), and *pre-training multi-task* (one model pre-trained and fine-tuned for all four tasks). For the training we used a 2x2 TPU topology (8 cores) from Google Colab with a batch size of 128, with a sequence length (for both inputs and targets) of 512 tokens.

Finally, we discuss qualitative examples of predictions generated by T5 and by the baselines to give a better idea to the reader about the capabilities of these models in supporting the four code-related tasks.

4.2 Hyperparameter Tuning

Before running the T5 models on the test sets, we performed a hyperparameter tuning on the evaluation sets from Table 2, to decide the best configuration to run. This was done for all nine models we built (*e.g.,* with/without pre-training, with/without multi-task learning).

For the *pre-training* phase, we use the default parameters defined for the T5 model [60]. Such a phase, indeed, is task-agnostic, and hyperparameter tuning would provide limited benefits. Instead, we tried different learning rate strategies for the *fine-tuning* phase. Especially, we tested four different learning rates: (i) *Constant Learning Rate* (C-LR): the learning rate is fixed during the whole training; (ii) *Inverse Square Root Learning Rate* (ISR-LR): the learning rate decays as the inverse square root of the training step; (iii) *Slanted Triangular Learning Rate* [27] (ST-LR): the learning rate first linearly increases and then linearly decays to the starting learning rate; (iv) *Polynomial Decay Learning Rate* (PD-LR): the learning rate decays polynomially from an initial value to an ending value in the given decay steps. Table 4 reports the specific parameters we use for each scheduling strategy.

TABLE 4: Learning-rates tested for hyperparameter tuning.

Learning Rate Type	Parameters
Constant	$LR = 0.001$
Inverse Square Root	$LR_{starting} = 0.01$ $Warmup = 10,000$
Slanted Triangular	$LR_{starting} = 0.001$ $LR_{max} = 0.01$ $Ratio = 32$ $Cut = 0.1$
Polynomial Decay	$LR_{starting} = 0.01$ $LR_{end} = 0.001$ $Power = 0.5$

In total, we fine-tuned 36 models (*i.e.,* nine models with four different schedulers) for 100k steps each. To select the best configuration for each training strategy, we compute the following metrics: for BF and AG, we compute the percentage of perfect predictions achieved on the evaluation set with the greedy decoding strategy (Accuracy@1); for MG, we compute the BLEU score [57]; for CS, we compute BLEU-A, the geometric average of the BLEU-{1,2,3,4} scores [57]. Basically, for each task we adopt one of the evaluation metrics used in the original paper. The complete results of the

hyperparameters tuning phase are reported in our replication package [2].

5 Results Discussion

We discuss our results accordingly to the formulated RQs.

5.1 Performance of T5 (RQ_1) and impact of transfer learning on performance ($RQ_{1.1}$-$RQ_{1.2}$)

Table 5 reports the performance achieved by the different variants of the T5 model that we experimented with. For each task (*e.g.,* Automatic Bug-Fixing) and for each dataset (*e.g.,* BFsmall), performance metrics are reported for the three adopted training strategies (*i.e.,* no pre-training, pre-training single task, and pre-training multi-task). For readability reasons, we only report the BLEU-A, but the results of the other BLEU scores (*e.g.,* BLEU-4) are available in our online appendix [2]. The pre-training multi-task setting is the same as used in our ICSE'21 paper [45] that this work extends. Note that for some tasks (*e.g.,* AG_{raw}) the results reported in Table 5 are different as compared to the ones reported in the ICSE paper. This is due to two changes we performed in our experimental pipeline. First, as compared to the ICSE paper, we updated our scripts to exploit the latest T5 version available as of today (*i.e.,* T5 0.9.2 - https://libraries.io/pypi/t5/0.9.2) and re-executed all of our experiments. Second, in our ICSE paper we lower-cased the source code before providing it as input to the T5. However, we realized that when working with Java raw code (see *e.g.,* the AG_{raw} task), it is important to keep such information considering the wide adoption of the camelCase naming convention in such a language.

Table 6 reports the results of the statistical analysis we performed using the McNemar's test [47] to identify (if any) statistical differences in terms of Accuracy@1 when using different training strategies.

Focusing on the Accuracy@1, it is evident that there is no training strategy being the best one across all tasks and datasets. In particular: *no pre-training* works better on the BF_{small} dataset for automatic bug-fixing; *pre-training single task* works better on the BF_{medium} dataset for automatic bug-fixing, on both datasets related to the generation fo assert statements, and for the code summarization task; finally, *pre-training multi-task* works better for the injection of code mutants. Overall, the *pre-training single task* strategy seems to be the best performing strategy. Indeed, even when it is not the first choice for a given task/dataset, it is the second best-performing training strategy. Also, by looking at Table 6 we can observe that:

1) When *pre-training single task* is the best strategy, its performance in terms of Accuracy@1 are significantly better (p-value < 0.001) than the second best-performing strategy, with ORs going from 1.13 (for CS) to 3.39 (AG_{raw}). This means that chances of getting a perfect predictions using this strategy are 13% to 339% higher when using this strategy as compared to the second choice.
2) When *pre-training single task* is not the best strategy, but the second choice, the difference in Accuracy@1 is not significant when compared to *pre-training multi-task* for MG_{ident}. The only significant difference is the one in favor of *no pre-training* on BF_{small}, with an OR of 0.77.

For these reasons, in our RQ_2 we will compare the T5 using the *pre-training single task* strategy against the baselines.

A few observations can be made based on the findings in Table 5. First, the additional pre-training is, as expected, beneficial. Indeed, on five out of the six datasets the T5 performs better with pre-training. Second, the multi-task setting did not help in most of cases. Indeed, with the exception of MG_{ident} in which the performance of *pre-training single task* and *pre-training multi-task* are basically the same, the *single task* setting performs always better. Such a result, while surprising at a first sight, can be explained by diverse types of input/output handled by the models across the four tasks. Indeed, (i) the datasets related to automatic bug-fixing and AG_{abs} include abstracted code instances as input/output; (ii) the dataset used for code mutants and AG_{raw} feature raw code instances as input/output; and (iii) the one for code summarization has raw source code as input and natural language text as output. Basically, given the different formats, the transfer learning across different tasks is likely to hinder the model rather than helping it.

Differently, the pre-training dataset features all three input/output representations and, thus, provides the model with a basic knowledge about all of them that, as a result, boosts performance.

While we will discuss more in depth the performance of the T5 model when comparing it to the considered baselines (Section 5.2), it is also worth commenting on the ability of the T5 to generate correct predictions, namely outputs that are identical to the reference ones (*e.g.,* a method summary equal to the one manually written by developers). Quite impressive are the performances achieved on the generation of assert statements, especially on the dataset dealing with raw source code, in which the T5 correctly predicts 68.93% of assert statements with a single guess (75.95% when using five guesses). The Accuracy@1 is instead much lower for the other tasks, ranging between 11.85% (fixing bugs in the most challenging BF_{medium} dataset) up to 28.72% when injecting mutants. Also worth noticing is the 12.02% of code summaries generated by the T5 that are identical to the manually written ones. In the next subsection, together with a comparison of our model with the baselines, we present qualitative examples of predictions generated by the T5.

5.2 Competitiveness of the T5 model compared to the baselines (RQ_2)

We compare the results achieved by the T5 model when using the *pre-training single task* strategy with the baseline we consider for each task (Table 3). The comparison is depicted in Fig. 2, while Table 8 shows the results of the statistical tests, and Table 10 shows the overlap metrics described in Section 4.1.

5.2.1 Automatic Bug Fixing (BF)

When using T5 for automatically fixing bugs, the accuracy achieved using a greedy decoding strategy ($K = 1$) differs according to the dataset we consider. For example, the T5 model achieves 15% of perfect predictions on the BF_{small}

TABLE 5: Overall results achieved by the T5 model for each tasks. The best configuration is highlighted in bold

Task	Dataset	Model Configuration	Accuracy@1	Accuracy@5	Accuracy@10	Accuracy@25	Accuracy@50	BLEU-A
Automatic Bug-Fixing	BF_{small}	*no pre-training*	**16.70%**	29.88%	34.37%	39.57%	42.86%	-
		pre-training single task	15.08%	32.08%	37.01%	42.51%	45.94%	-
		pre-training multi-task	11.61%	**35.64%**	**43.87%**	**52.88%**	**57.70%**	-
	BF_{medium}	*no pre-training*	10.50%	17.60%	20.53%	24.38%	27.62%	-
		pre-training single task	**11.85%**	**19.41%**	23.28%	28.60%	32.43%	-
		pre-training multi-task	3.65%	19.17%	**24.66%**	**30.52%**	**35.56%**	-
Injection of Code Mutants	MG_{ident}	*no pre-training*	25.78%	-	-	-	-	78.26%
		pre-training single task	28.72%	-	-	-	-	**78.69%**
		pre-training multi-task	**28.92%**	-	-	-	-	78.29%
Generation of Asserts in Test	AG_{raw}	*no pre-training*	60.95%	59.14%	62.41%	69.05%	71.97%	-
		pre-training single task	**68.93%**	**75.95%**	**77.70%**	**79.24%**	**80.22%**	-
		pre-training multi-task	58.60%	66.90%	70.31%	73.19%	74.58%	-
	AG_{abs}	*no pre-training*	47.81%	49.60%	55.04%	64.28%	68.57%	-
		pre-training single task	**56.11%**	**71.26%**	**74.32%**	**76.67%**	**78.02%**	-
		pre-training multi-task	44.90%	63.40%	68.23%	73.04%	73.12%	-
Code Summarization	CS	*no pre-training*	11.80%	-	-	-	-	24.67%
		pre-training single task	**12.02%**	-	-	-	-	**25.21%**
		pre-training multi-task	11.45%	-	-	-	-	24.90%

TABLE 6: McNemar's test (adj. p-value and OR) considering only accuracy@1 matches as correct predictions

Task	Dataset	Model Configuration	p-value	OR
Automatic Bug-Fixing	BF_{small}	*no pre-training vs pre-training single task*	< 0.001	0.77
		no pre-training vs pre-training multi-task	< 0.001	0.46
		pre-training multi-task vs pre-training single task	< 0.001	1.67
	BF_{medium}	*no pre-training vs pre-training single task*	< 0.001	1.56
		no pre-training vs pre-training multi-task	< 0.001	0.12
		pre-training multi-task vs pre-training single task	< 0.001	8.56
Injection of Code Mutants	MG_{ident}	*no pre-training vs pre-training single task*	< 0.001	1.51
		no pre-training vs pre-training multi-task	< 0.001	1.38
		pre-training multi-task vs pre-training single task	0.75	0.99
Generation of Asserts in Test	AG_{raw}	*no pre-training vs pre-training single task*	< 0.001	3.39
		no pre-training vs pre-training multi-task	< 0.001	0.71
		pre-training multi-task vs pre-training single task	< 0.001	4.95
	AG_{abs}	*no pre-training vs pre-training single task*	< 0.001	2.55
		no pre-training vs pre-training multi-task	< 0.001	0.74
		pre-training multi-task vs pre-training single task	< 0.001	2.93
Code Summarization	CS	*no pre-training vs pre-training single task*	< 0.001	1.13
		no pre-training vs pre-training multi-task	< 0.001	0.83
		pre-training multi-task vs pre-training single task	< 0.001	1.40

dataset against 9% achieved by the baseline, with an improvement of 6 percentage points, while in the most challenging scenario, (*i.e.*, BF_{medium}) our model obtains an improvement of 8 percentage points over the baseline (11% *vs* 3%). Such improvements are statistically significant (Table 8) with ORs of 2.39 (BF_{small}) and 6.88 (BF_{medium}), indicating higher chance of observing a perfect prediction when using the T5 as compared to the baseline. Worth noticing is that as the beam width increases, the performance of the T5 and of the baseline gets closer, with the baseline performing better for $K = 25$ and $K = 50$ on BF_{small}.

Looking at the overlap metrics (Table 10), 25.90% of perfect predictions on BF_{small} and 28.78% on BF_{medium} are shared by the two techniques. The remaining are perfect predictions only with T5 (53.20% on BF_{small} and 36% on BF_{medium}) or only with the baseline (20.90% on BF_{small} and 35.16% on BF_{medium}). This indicates that the two approaches are complementary for the bug fixing task suggesting that further improvements could be possible by exploiting customized ML-based bug-fixing techniques. To further look into this finding, we analyzed the type of "code transformation" that T5 and the baseline were able to learn. With "code transformation" we refer to Abstract Syntax Tree (AST) operations needed to correctly transform the input code into the target prediction (*i.e.*, the AST operations performed by developers to transform the buggy code into the fixed code). In particular, we used the Gumtree Spoon AST Diff [18] to collect the *Delete*, *Insert*, *Move* and *Update* operations performed on the AST nodes when fixing bugs. Then, for each of these operations, we extracted the 5 most popular ones (*e.g.*, the five most popular *Delete* node operations). These 20 AST-level operations (4 types of operations $\times$ 5 most popular for each type) characterize the successful fixing of bugs/injection of code mutants in the three datasets. The column "Oracle" of (Table 7) reports such numbers. Then, we took the correct predictions generated by T5 and by the baselines and checked the extent to which those predictions feature the "popular" AST operations that, accordingly to our oracles, are needed to properly fix bugs. Table 7 reports for both techniques and both datasets (BF_{small} and BF_{medium}) the number of times the different AST operations were performed by the models.

Given the previously discussed superior performance of T5, it is expected to see that it managed to correctly perform the needed AST operations more often than the baseline. However, what is interesting is that there are specific types of operations that are not learned by the baseline while they are successfully implemented by T5. This is especially true for less popular operations such as the *Insert* ones, that require to synthesize new nodes that were not present in the input AST.

TABLE 7: Top-20 AST operations needed to fix bugs in our dataset (see "Oracle" column) and their presence in correct predictions generated by T5 and the baseline

	BF_{small}			BF_{medium}		
Delete	**Oracle**	**Baseline [75]**	**T5**	**Oracle**	**Baseline [75]**	**T5**
Delete TypeAccess at Invocation	2,016	402	450	1,926	125	250
Delete Invocation at Block	1,444	294	326	1,315	159	240
Delete TypeAccess at ThisAccess	821	92	134	598	32	81
Delete VariableRead at Invocation	818	51	106	1,106	61	126
Delete FieldRead at BinaryOperator	479	92	100	651	66	116
Insert						
	BF_{small}			BF_{medium}		
	Oracle	**Baseline [75]**	**T5**	**Oracle**	**Baseline [75]**	**T5**
Insert Block at If	486	3	28	828	3	48
Insert Literal at BinaryOperator	468	5	27	736	0	37
Insert If at Block	406	2	22	659	0	33
Insert BinaryOperator at If	380	3	23	634	0	36
Insert VariableRead at Invocation	328	10	33	675	0	38
Move						
	BF_{small}			BF_{medium}		
	Oracle	**Baseline [75]**	**T5**	**Oracle**	**Baseline [75]**	**T5**
Move Invocation from Block to Invocation	633	17	61	1,005	4	86
Move VariableRead from Invocation to VariableRead	158	7	11	281	2	19
Move Assignment from Block to Assignment	120	0	13	209	1	19
Move Invocation from BinaryOperator to Invocation	95	7	11	183	1	14
Move BinaryOperator from BinaryOperator to BinaryOperator	68	0	2	174	0	9
Update						
	BF_{small}			BF_{medium}		
	Oracle	**Baseline [75]**	**T5**	**Oracle**	**Baseline [75]**	**T5**
Update Wra at Method	280	15	37	191	1	22
Update TypeAccess at Invocation	201	17	41	404	18	115
Update Invocation at Block	115	0	8	153	2	21
Update VariableRead at Invocation	101	1	12	226	0	19
Update BinaryOperator at If	56	3	8	148	1	12

TABLE 8: McNemer's test considering the correct predictions achieved by the T5 model and the baselines when both techniques generate only one prediction (*i.e., accuracy@1*)

Task	*Dataset* (d)	***p*-value**	**OR**
Automatic Bug-Fixing	BF_{small}	< 0.001	2.39
	BF_{medium}	< 0.001	6.88
Injection of Code Mutants	MG_{ident}	< 0.001	2.95
Generation of Asserts in Test	AG_{abs}	< 0.001	6.19
	AG_{raw}	< 0.001	43.12
Code Summarization	CS	< 0.001	35.56

In BF_{medium}, four of the top-five AST Insert operations are never applied by the baseline (see Table 7). Similar results are also obtained for the *Update* operations, while both models work similarly well when the bug-fix mostly requires the deletion of existing AST nodes.

5.2.2 Injection of Code Mutants (MG)

Looking at Fig. 2 we can observe that using T5 to generate mutants allows to obtain more accurate results than the baseline, with the Accuracy@1 improving by 12 percentage points, with 1,336 additional perfect predictions. The average BLEU score also improves by ∼0.02 on top of the very good results already obtained by the baseline (*i.e.,* 0.77). Minor improvements in BLEU score can still indicate major advances in the quality of the generated solutions [14]. Also in this case differences in terms of Accuracy@1 are statistically significant, with the T5 model being more likely to generate correct solutions ($OR = 2.95$) as compared to the baseline approach [76] (Table 8).

Differently from the bug-fixing task, for the injection of code mutants the percentage of shared perfect predictions (Table 10) is slightly higher (33%) with, however, T5 being the only one generating 50.52% of perfect predictions as compared to the 16.48% generated exclusively by the baseline.

Similarly to what has been done in the context of the bug-fixing task, Table 9 reports the top-20 AST-level operations needed to correctly inject mutants in our dataset. Note that, differently from what observed for the bug-fixing task, injecting mutants mostly requires the insertion of new AST nodes. The trend that we observe is, as expected, the opposite of what we found for the bug-fixing task because the task is the same but with reversed input/output. Indeed, the baseline seems to correctly predict the most popular *Insert* operations in the AST, while it almost ignores the more rare *Delete* ones. T5 instead, covers all top-20 operations.

5.2.3 Generation of Assertions in Test Methods (AG)

T5 achieve much better performance in this task as compared to the baseline. The gap is substantial both with (AG_{abs}) and without (AG_{raw}) code abstraction (Fig. 2). With abstraction,

TABLE 9: Top-20 AST operations needed to inject mutants in our dataset (see "Oracle" column) and their presence in correct predictions generated by T5 and the baseline

	MG_{ident}		
Delete	**Oracle**	**Baseline [76]**	**T5**
Delete TypeAccess at Invocation	387	1	30
Delete Return at Block	327	20	64
Delete FieldRead at BinaryOperator	283	0	7
Delete FieldRead at Invocation	242	0	19
Delete Invocation at Block	236	0	15
Insert			
Insert TypeAccess at Invocation	6,230	1,125	1,744
Insert Invocation at Block	3,979	860	1,183
Insert TypeAccess at ThisAccess	2,219	479	722
Insert VariableRead at Invocation	2,061	245	466
Insert Block at If	1,795	485	671
Move			
Move Invocation from Block to Invocation	1,154	225	356
Move Invocation from Return to Invocation	283	55	105
Move Return from Block to Return	224	58	100
Move Assignment from Block to Assignment	190	26	56
Move Invocation from Invocation to Invocation	129	1	27
Update			
Update TypeAccess at Invocation	923	67	220
Update FieldRead at BinaryOperator	408	14	63
Update Wra at Method	264	1	31
Update TypeAccess at ThisAccess	228	10	73
Update TypeReference at Method	208	0	25

the T5 achieves a 56% accuracy at $K = 1$ against the 31% achieved by *ATLAS* [82]. When both approaches are asked to generate multiple assert statements (*i.e.*, $K = 5, 10, 25, 50$) the gap in performance ranges between 13-25 percentage points. When using the more challenging non-abstracted dataset AG_{raw}, T5 achieves even better results. In this regard, when T5 is asked to generate only one assert statement ($K = 1$), the reported accuracy is 51 percentage points higher compared to the baseline , while for larger K values the gap in performance ranges between 51-53 percentage points. The McNemar's test confirms the huge gap in performance between the two techniques, with ORs ranging between 6.19 (AG_{abs}) and 43.12 (AG_{raw}).

In terms of overlap, we found a trend similar to the previously discussed task (mutants injection): On AG_{abs} we have 34.92% of perfect predictions shared between the two approaches, while the remaining instances are distributed between the ones only predicted by T5 (58.87%) and the ones only predicted by the baseline (6.21%). The overlap is much smaller on the AG_{raw} dataset, with only 9.56% of the instances correctly predicted by both the approaches, 89.65% of them correctly predicted only by T5, and 0.79% only by the baseline.

5.2.4 Code Summarization (CS)

On this task, T5 achieves a substantial increase in BLEU score as compared to the baseline. When considering the average BLEU (BLEU-A), the improvement is of ∼5 percentage points. On the other hand, it can be noticed that the ROUGE-LCS scores achieved when using T5 are lower than the ones achieved by the baseline (∼5 percentage points lower on the F-measure score). Thus, looking at these metrics, there is no clear winner, but T5 seems to be at least comparable to the baseline. To have something easier to interpret, we compared the two approaches in terms of the number of perfect predictions they generate, despite the fact that such a metric was not used in the original paper [24]. This means counting the comments generated by a technique that are exactly equal to the ones manually written by humans. T5 managed to generate 12.02% of perfect predictions (10,929 instances) against the 3.4% (3,048) of the baseline technique (over 3 × better). As expected from previous results, the majority of the perfect predictions for this task can be done only using T5 (93.79%). A limited percentage of perfect predictions is shared (4.79%), and a minority of instances can be only predicted through the baseline (1.42%). The McNemar's test highlights a statistical significance in terms of Accuracy@1, with an OR of 35.56.

TABLE 10: Overlap metrics for correct predictions generated by the T5 model and the baselines.

Task	*Dataset (d)*	$Shared_d$	$OnlyT5_d$	$OnlyBL_d$
Automatic Bug-Fixing	BF_{small}	25.90%	53.20%	20.90%
	BF_{medium}	28.78%	36.06%	35.16%
Injection of Code Mutants	MG_{ident}	33.00%	50.52%	16.48%
Generation of Asserts in Test	AG_{abs}	34.92%	58.87%	6.21%
	AG_{raw}	9.56%	89.65%	0.79%
Code Summarization	CS	4.79%	93.79%	1.42%

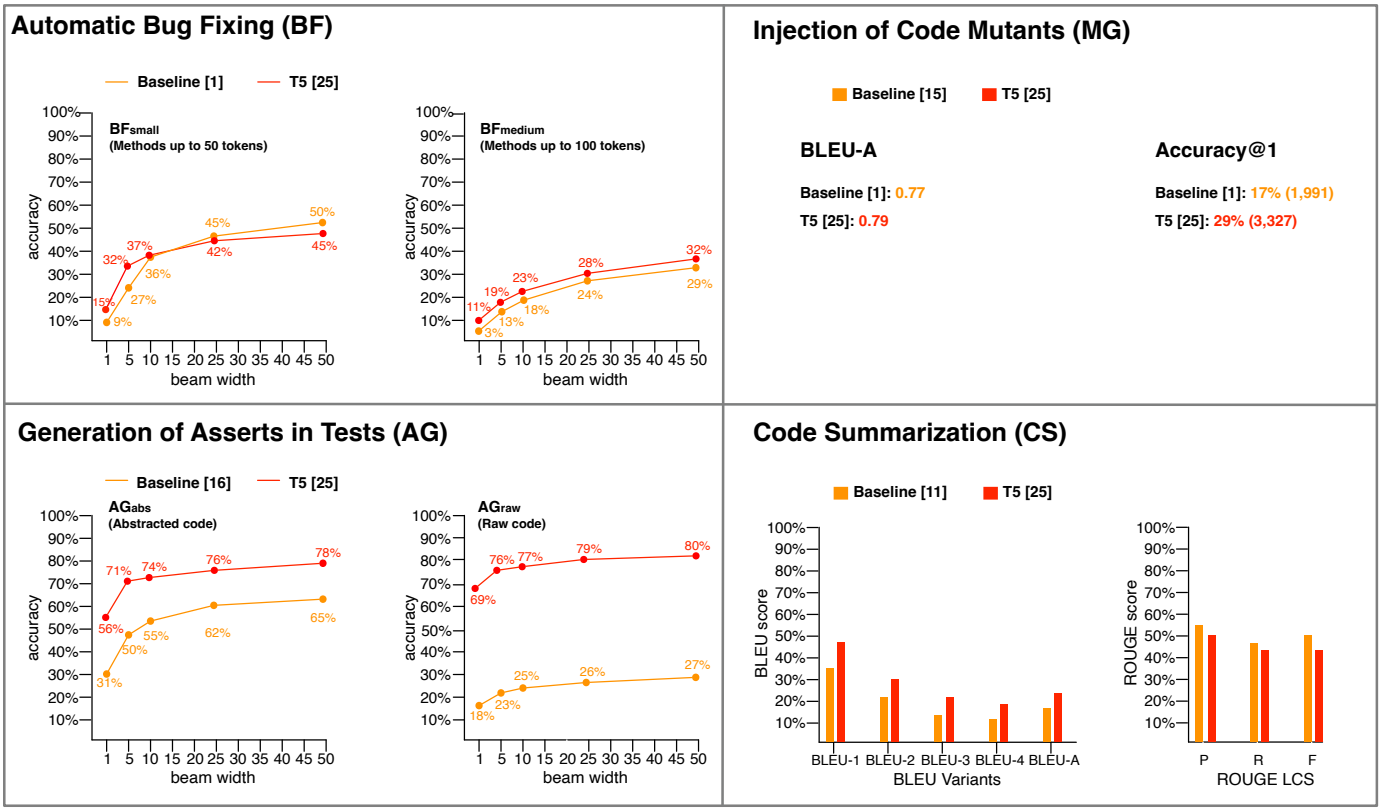
5.2.5 Qualitative Analysis

To give a better idea to the reader about the capabilities of the T5 model in supporting the four code-related tasks, Fig. 3 shows two examples of perfect predictions made by T5 for each task. Each example is bordered with a dashed line and shows (i) the input provided by the model, and (ii) the generated output. In particular, in the case of the bug-fixing, mutants injection, and code summarization tasks, the first line shows the input and the second the output. Concerning the generation of assert statements, the first two lines (*i.e.*, those marked with "//Test method" and "//Focal method") represent the input, while the third line shows the generated assert statement. We highlighted in bold the most relevant parts of the output generated by the model. The bottom part of Fig. 3 also shows some "wrong" predictions (*i.e.*, the output of the model is different from the expected target) for the code summarization task, that we will discuss later on.

Concerning the bug-fixing task, in the first example the model adds the `break` statement to each `case` of the `switch` block, thus allowing the program to break out of the `switch` block after one `case` block is executed. In the second example, instead, it changes the execution order of a statement as done by developers to fix the bug.

As per the mutants injection, the first example represents an *arithmetic operator deletion*, while the second is a *non void method call mutation* [1]. While these transformations might look trivial, it is worth remembering that they are considered as correct since they reproduce real bugs that used to affect these methods. Thus, the model is basically choosing where to mutate and what to mutate in such a way to simulate

Fig. 2: Performance of the T5 model against the experimented baselines.



real bugs (accomplishing one of the main goals of mutation testing).

Both examples of correct prediction we report involve the generation of an assert statement including an invocation to the focal method (*i.e.,* the main method tested by the test method). While the first is a rather "simple" `assertFalse` statement, the second required the guessing of the expected value (*i.e.,* `assertEquals`).

Finally, for the code summarization, the two reported examples showcase the ability of T5 to generate meaningful summaries equivalent to the ones manually written by developers. For this task, we also reported in the bottom part of the figure some wrong but still meaningful predictions. In this case, the grey text represent the original summary written by developers, while the bold one has been generated by T5. In both cases, the generated summary is semantically equivalent and even more detailed that the manually written one.

These two examples help in discussing an important limitation of our analysis: While we assume the correct predictions to be the *only* valuable outputs of T5 and of the experimented baselines, they actually represent a lower-bound for their performance. Indeed, there are other predictions that, even if wrong, could still be valuable for developers, such as the two shown for the code summarization task.

5.3 Training and Inference Time

Table 11 reports the training time (in hours) for the nine models we trained. On average, the infrastructure we used for training requires 31.5 seconds every 100 training steps which, given our batch size = 128, means that 12,800 training instances can be processed in 31.5 seconds. Of course, multiple passes (usually referred to as epochs) are needed on the dataset during the training. Table 11 shows that (i) the pre-training has a cost of ∼22h that should be added on top of the fine-tuning cost shown for each task; (ii) as expected, the training time increases with the increase in size of the training dataset, with the *code summarization* task being the most expensive in terms of training time; (iii) clearly, the *multi-task* setting requires to train the model on all tasks, resulting in the highest training time (175h).

TABLE 11: Training time (hours) for the trained T5 models

Training	Bug-fixing	Mutants generation	Generation of assert statements	Code summarization	Multi-Task
No pre-training	6.26	5.85	17.51	123.55	-
Pre-training	28.10	27.72	39.40	145.42	175.00

Table 12 presents, instead, the results of the inference time analysis (*i.e.,* the time needed to run the model on a given input and obtain the prediction). Such analysis allows to understand the extent to which such a model can be used in practice. Table 12 reports the inference time in seconds for different K values (*e.g.,* with $K = 10$ the reported time is the one required by the model to generate 10 possible solutions).

Concerning the bug-fixing task, the time needed to generate a fix depends on the dataset, since the complexity of the instances they feature is different. In the BF_{small} dataset, the average inference time ranges between 0.72s ($K = 1$) and 5.99s ($K = 50$), while it is larger on the BF_{medium} dataset (1.86s for $K = 1$ and 20.90s for $K = 50$). For the

Fig. 3: Examples of perfect and alternative predictions

Bug-fixing

```
private void METHOD_1 ( int VAR_1 , int offset ) { switch ( VAR_1 ) { case 0 : this . VAR_2 = offset ; case 1 : this . VAR_3 = offset ; case INT_1 :
this . VAR_4 = offset ; case INT_2 : this . VAR_5 = offset ; } }
```

```
private void METHOD_1 ( int VAR_1 , int offset ) { switch ( VAR_1 ) { case 0 : this . VAR_2 = offset ; break ; case 1 : this . VAR_3 = offset ; break ;
case INT_1 : this . VAR_4 = offset ; break ; case INT_2 : this . VAR_5 = offset ; break ; } }
```

```
public void METHOD_1 ( final long [ ] data , boolean length ) { int VAR_1 = ( data . length ) * ( VAR_2 ) ; if ( length ) METHOD_2 ( VAR_1 ) ; METHOD_3
( ( ( position ) + VAR_1 ) ) ; VAR_3 . METHOD_4 ( data , VAR_4 , null , ( ( VAR_5 ) + ( position ) ) , VAR_1 ) ; position += VAR_1 ;
end = ( ( position ) > ( end ) ) ? position : end ; }
```

```
public void METHOD_1 ( final long [ ] data , boolean length ) { int VAR_1 = data . length ; if ( length ) METHOD_2 ( VAR_1 ) ; VAR_1 *= VAR_2 ;
METHOD_3 ( ( ( position ) + VAR_1 ) ) ; VAR_3 . METHOD_4 ( data , VAR_4 , null , ( ( VAR_5 ) + ( position ) ) , VAR_1 ) ; position += VAR_1 ;
end = ( ( position ) > ( end ) ) ? position : end ; }
```

Mutants generation

```
public int METHOD_1 ( int a ) { int index = VAR_1 [ ( a + 1 ) ] ; return VAR_2 . METHOD_2 ( index ) ; }
```

```
public int METHOD_1 ( int a ) { int index = VAR_1 [ a ] ; return VAR_2 . METHOD_2 ( index ) ; }
```

```
public java.lang.String METHOD_1 ( final java.lang.String VAR_1 ) { return STRING_1 . METHOD_2 ( VAR_2 . METHOD_3 ( ) . METHOD_4 ( ) ) ; }
```

```
public java.lang.String METHOD_1 ( final java.lang.String VAR_1 ) { return VAR_2 . METHOD_3 ( ) . METHOD_4 ( ) ; }
```

Generation of assert statements

```
// Test method
shouldNeverValidateNullUserIV ( ) { final uk . gov . gchq . gaffer . federatedstore . FederatedAccess access = new uk . gov . gchq . gaffer .
federatedstore . FederatedAccess . Builder ( ) . addingUserId ( null ) . build ( ) ; "<AssertPlaceHolder>" ; }
```

```
// Focal method
isValidToExecute ( uk . gov . gchq . gaffer . user . User ) { return ( isPublic ) || ( ( null != user ) && ( ( isAddingUser ( user ) )
|| ( ( ! ( isAuthsNullOrEmpty ( ) ) ) && ( isUserHasASharedAuth ( user ) ) ) ) ) ; }
```

org . junit . Assert . assertFalse (access . isValidToExecute (null))

```
// Test method
testClone ( ) { org . apache . flink . api . common . accumulators . DoubleMinimum min = new org . apache . flink . api . common . accumulators .
DoubleMinimum ( ) ; double value = 3.14159265359 ; min . add ( value ) ; org . apache . flink . api . common . accumulators . DoubleMinimum clone =
min . clone ( ) ; "<AssertPlaceHolder>" ; }
```

```
// Focal method
getLocalValue ( ) { return null ; }
```

org . junit . Assert . assertEquals (value , clone . getLocalValue ( ) , 0.0)

Code summarization

```
public void update() { check Widget ( ) ; Utils . paintComponentImmediately ( handle ) ; update ( false ) ; }
```

forces all outstanding paint requests for the widget

```
public void setWordWrap(int row, int column, boolean wrap) { prepareCell ( row , column ) ; String wrapValue = wrap ? "" : "nowrap" ; DOM .
setStyleAttribute ( getElement ( row, column ) , "whiteSpace" , wrapValue ) ; }
```

sets whether the specified cell will allow word wrapping of its contents

Wrong but meaningful predictions for the code summarization task

```
testCase getTestCase ( String implementationNumber ) int index = Integer . valueOf ( implementationNumber ) ; int value = return getTestCase ( index ) ;
```

return the specific test case

returns the test case with the given implementation number

```
protected void doConfigure(HierarchicalConfiguration config) throws ConfigurationException {}
```

override to handle config

subclasses can override this method to perform custom configuration

TABLE 12: Inference time with different beam size values.

K	BF_{small}	BF_{medium}	MG_{ident}	AG_{abs}	AG_{raw}	CS
1	0.72	1.86	0.94	0.73	0.53	0.20
5	1.47	3.69	1.70	1.59	1.04	0.36
10	1.91	5.26	2.20	2.64	1.52	0.48
25	3.54	11.10	4.32	5.45	3.15	0.81
50	5.99	20.90	7.60	10.24	5.45	1.45

injection of code mutants, we observed results comparable to those of BF_{small}: with $K = 1$ the average inference time is 0.94s, while for $K = 50$ it is 7.60s. The generation of assert statement is very fast for low values of K (0.73s for AG_{abs} and 0.53s for AG_{raw} with $K = 1$), while it gets slower for higher values of K (10.24 for AG_{abs} and 5.45 for AG_{raw} with $K = 50$). Finally, concerning the code summarization task, T5 takes only 0.20s for $K = 1$ and 1.45s for $K = 50$ to output code summaries for a method given as input.

Overall, considering that all the targeted tasks do not have strong real-time constraints (*e.g.*, a developer can wait a few seconds for the automated fixing of a bug), the inference times should not hinder the model applicability in

practice. Also, the reported inference times were obtained by running the model on a consumer-level device and by only using CPUs. We also computed the inference time using an Nvidia Tesla P100 GPU equipped with 16GB of VRAM. The achieved results are available in our replication package [2]. In summary, we observed an average decrease of inference time of ∼70% as compared to the one obtained using the CPU.

6 Threats to Validity

Construct validity. Threats to construct validity concern the relationship between theory and observation. We used existing datasets that are popular and used in the community for both pre-training and fine-tuning our model with minimal additional processing (*e.g.,* removal of duplicates after abstraction in the dataset used for the pre-training). Additionally, we have released all of our code and models in our replication study [2] for verification.

Internal validity. Threats to internal validity concern factors, internal to our study, that could influence its results. Many factors can influence our results, from model architecture, hyperparameter choices, data processing, the data itself, etc. For mitigating these issues, we have adopted methodologies usually employed in DL-based research. Specifically, we performed a detailed analysis of hyperparameter choices as discussed in Section 4.2. Concerning the pre-training phase, we used the default T5 parameters selected in the original paper [60] since we expect little margin of improvement for such a task-agnostic phase. For the fine-tuning, due to computational feasibility reasons, we did not change the model architecture (*e.g.,* number of layers), but we experiment with different learning rates schedulers. We are aware that a more extensive calibration would likely produce better results. Finally, we pre-trained the model by masking 15% of tokens (*i.e.,* words in comments and code tokens in the raw and abstracted code) in the ∼2.7M instances from the pre-training dataset. However, we did not experiment with the model after pre-training to verify whether it actually learned the languages of interest (*i.e.,* raw source code, abstracted source code, and technical natural language). To address this limitation, we randomly selected 3k instances from the BF_{medium} test set, both in their abstract and raw representation (6k in total). We also selected 3k code summaries from the CS dataset obtaining a dataset of 9k instances, equally split across raw source code, abstracted source code, and technical natural language. Note that these are instances that have not been used to pre-train the model and, thus, are unseen for a model only subject to pre-training. We randomly masked 15% of tokens in each of those instances, asking the pre-trained model to predict them. T5 correctly predicted 87,139 out of the 102,711 masked tokens (*i.e.,* 84.8% accuracy). As expected, given the different complexity of the three "languages", T5 achieved a higher accuracy of 90.8% when working on abstracted code, 82.7% on raw code, and 64.6% when guessing tokens from technical language. Overall, such results indicate that the model successfully gathered knowledge about the languages of interest during the pre-training.

Also the quality of the employed datasets can dramatically impact the achieved results. This is because there may be biases making the dataset not representative of the real world. To assess the quality of our datasets we conducted various analyses around sampling bias and data snooping as recommended by Watson *et al.* [81].

To this end, we conducted an exploratory data analysis (EDA), which helps answering questions related to the reliability and quality of our datasets. To accomplish this, we performed a two-fold statistical procedure: complexity size and token distributions. In the complexity size procedure, we count the number of tokens per dataset and data partition. Then, we present the relative distribution in log scale. While in the token procedure, we concentrated on counting specific tokens by popularity or special interest (*e.g.,* if, $assert$, or $public$). The purpose of the EDA is to monitor the size of datasets and its impact in the model performance. EDA's results can be found in our web appendix [2].

Conclusion validity. Threats to conclusion validity concern the relationship between evaluation and outcome. To this extent, we used appropriate statistical procedures, also adopting p-value adjustment when multiple tests were used within the same analysis.

External validity. Threats to external validity are related to the generalizability of our findings. Our study focused on the T5 model on four tasks using six datasets, all of which only involved Java code. While it is unclear how our model would perform if trained on other programming languages, excluding the abstraction component, the whole pipeline is language agnostic and can be easily adapted to other languages for evaluating this.

We also performed an analysis of our dataset aimed at finding out the generalizability of our models. This analysis assessed the level of data snooping among our datasets' training and test sets and how this impacts our model's results. To accomplish this we calculate the overlap between our fine-tuning datasets' training and test sets by computing the pairwise Levenshtein Distance [40] between the two sets. With these distances calculated, we computed the correlation between the distances and the performance of our model on the different test sets.

Specifically, we selected a statistically representative sample (confidence level = 95% and confidence interval = 5%) of each training set and calculated the pairwise Levenshtein Distance [40] between it and the entirety of the test set for each fine-tuning dataset. Next, depending on the type of performance metric (Perfect Prediction or BLEU Score), we calculate the correlation between the minimum, median, and maximum distances of all sampled training examples to each test example and the performance of our model on the test set. For the perfect prediction, we use Point Biserial Correlation (PBC) [71] as it allows to compare binary and continuous data. For the BLEU score, we use Pearson Correlation [71] since both are continuous values.

Table 13 shows the correlation for each dataset. As shown, there exists a negative correlation between the minimum and median distances and model performance, *i.e.,* the model tends to perform worse as the distance between the training and test examples increases. For the maximum distance case, there is instead a positive correlation for perfect prediction performance, *i.e.,* the model tends to perform better the further away the maximum training examples are from the test examples. Such a result may be simply due to

TABLE 13: Correlation between training-test set similarity and test set performance.

Dataset	Min	Median	Max
BF_{small}	-0.15	-0.03	0.04
BF_{medium}	-0.05	-0.03	0.01
MG_{ident}	0.21	0.03	-0.23
AG_{abs}	-0.21	-0.14	0.29
AG_{raw}	-0.21	-0.14	0.19
CS	-0.38	-0.17	-0.09

specific outliers present in the test set (*i.e.*, an instances being very far from the ones in the training set). However, all the correlations we observed are quite low, supporting the generalizability of our models.

7 Conclusion

We presented an empirical study aimed at investigating the usage of transfer learning for code-related tasks. In particular, we pre-trained and fine-tuned several variants of the Text-To-Text Transfer Transformer (T5) model with the goal of supporting four code-related tasks, namely *automatic bug-fixing*, *injection of code mutants*, *generation of assert statements in test methods*, and *code summarization*. We compared the performance achieved by the T5 against state-of-the-art baselines that proposed DL-based solutions to these four tasks.

The achieved results showed that: (i) the pre-training process of the T5, as expected, boosts its performance across all tasks; (ii) the multi-task fine-tuning (*i.e.*, a single model trained for different tasks) instead, does not consistently help in improving performance, possibly due to the different types of "data" manipulated in the four tasks (*i.e.*, raw code, abstracted code, natural language); (iii) in its best configuration, the T5 performs better that the baselines across all four tasks. When looking at the latter finding it is important to remember that the baselines used for comparison are not pre-trained and, thus, they (i) exploited less training data, and (ii) did not need the additional ∼22 hours of computation required by the pre-training.

Future work will aim at further advancing performance by employing larger versions of the T5. Also, while our results do not support the usage of multi-task learning in code-related tasks, we believe additional investigations are needed on this side. For example, by only considering a set of tasks all manipulating the same type of data (*e.g.*, all working on raw code), it is possible that the benefits of multi-task learning would emerge.

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 team has been supported in part by the NSF CCF-1955853, CCF-1815186 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] "Pit - real world mutation testing https://pitest.org."
[2] "Replication package https://github.com/antonio-mastropaolo/TransferLearning4Code."
[3] "Utilizing fast testing to transform java development into an agile, quick release, low risk process." [Online]. Available: http://www.agitar.com/
[4] M. Allamanis, H. Peng, and C. A. Sutton, "A convolutional attention network for extreme summarization of source code," *CoRR*, vol. abs/1602.03001, 2016. [Online]. Available: http://arxiv.org/abs/1602.03001
[5] U. Alon, R. Sadaka, O. Levy, and E. Yahav, "Structural language models of code," *arXiv*, pp. arXiv–1910, 2019.
[6] N. Arivazhagan, A. Bapna, O. Firat, D. Lepikhin, M. Johnson, M. Krikun, M. X. Chen, Y. Cao, G. F. Foster, C. Cherry, W. Macherey, Z. Chen, and Y. Wu, "Massively multilingual neural machine translation in the wild: Findings and challenges," *CoRR*, vol. abs/1907.05019, 2019. [Online]. Available: http://arxiv.org/abs/1907.05019
[7] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: learning to fix bugs automatically," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 159:1–159:27, 2019.
[8] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *CoRR*, vol. abs/1409.0473, 2014.
[9] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 306–317.
[10] N. Boulanger-Lewandowski, Y. Bengio, and P. Vincent, "Audio chord recognition with recurrent neural networks." in *ISMIR*. Citeseer, 2013, pp. 335–340.
[11] S. Brody, U. Alon, and E. Yahav, "Neural edit completion," *arXiv preprint arXiv:2005.13209*, 2020.
[12] D. B. Brown, M. Vaughn, B. Liblit, and T. Reps, "The care and feeding of wild-caught mutants," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 511–522. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106280
[13] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezzè, "Automatic recovery from runtime failures," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 782–791.
[14] I. Caswell and B. Liang, "Recent advances in google translate," https://ai.googleblog.com/2020/06/recent-advances-in-google-translate.html, 2020.
[15] 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*, 2019. [Online]. Available: http://arxiv.org/abs/1901.01808
[16] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," *CoRR*, vol. abs/1406.1078, 2014.
[17] 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*, 2021.
[18] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the International Conference on Automated Software Engineering*, 2014, pp. 313–324. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01054552/file/main.pdf
[19] G. Fraser and A. Arcuri, "EvoSuite: Automatic Test Suite Generation for Object-oriented Software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. ACM, 2011, pp. 416–419.
[20] M. Gabel and Z. Su, "A study of the uniqueness of source code," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10. New York, NY, USA: ACM, 2010, pp. 147–156.
[21] C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," ser. ICSE'12.
[22] A. Graves, "Sequence transduction with recurrent neural networks," *CoRR*, vol. abs/1211.3711, 2012. [Online]. Available: http://arxiv.org/abs/1211.3711

[23] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *2010 17th Working Conference on Reverse Engineering*, 2010, pp. 35–44.

[24] S. Haque, A. LeClair, L. Wu, and C. McMillan, "Improved automatic summarization of subroutines via attention to file context," in *MSR '20: 17th International Conference on Mining Software Repositories, 2020*. ACM, 2020, pp. 300–310.

[25] H. Hata, E. Shihab, and G. Neubig, "Learning to generate corrective patches using neural machine translation," *CoRR*, vol. abs/1812.07170, 2018. [Online]. Available: http://arxiv.org/abs/1812.07170

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

[27] J. Howard and S. Ruder, "Universal language model fine-tuning for text classification," *arXiv preprint arXiv:1801.06146*, 2018.

[28] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC ?18. Association for Computing Machinery, 2018, p. 200?210.

[29] 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.

[30] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 2073–2083. [Online]. Available: https://www.aclweb.org/anthology/P16-1195

[31] A. Jaffe, J. Lacomis, E. J. Schwartz, C. L. Goues, and B. Vasilescu, "Meaningful variable names for decompiled code: A machine translation approach," in *Proceedings of the 26th Conference on Program Comprehension*, ser. ICPC '18, 2018, pp. 20–30.

[32] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE'17, Oct. 2017, pp. 135–146, iSSN:.

[33] N. Kalchbrenner and P. Blunsom, "Recurrent continuous translation models," in *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. Seattle, Washington, USA: Association for Computational Linguistics, October 2013, pp. 1700–1709.

[34] R. Karampatsis and C. A. Sutton, "Maybe deep neural networks are the best choice for modeling source code," *CoRR*, vol. abs/1903.05734, 2019. [Online]. Available: http://arxiv.org/abs/1903.05734

[35] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," *arXiv preprint arXiv:2003.13848*, 2020.

[36] T. Kudo, "Subword regularization: Improving neural network translation models with multiple subword candidates," *arXiv preprint arXiv:1804.10959*, 2018.

[37] T. Kudo and J. Richardson, "Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing," *CoRR*, vol. abs/1808.06226, 2018. [Online]. Available: http://arxiv.org/abs/1808.06226

[38] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 54–72, 2012.

[39] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19, 2019, pp. 795–806.

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

[41] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Text summarization branches out*, 2004, pp. 74–81.

[42] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, "Neural-machine-translation-based commit message generation: How far are we?" in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018, 2018, pp. 373–384.

[43] M. Martinez, W. Weimer, and M. Monperrus, "Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014, pp. 492–495.

[44] 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.

[45] 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 *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021*. IEEE, 2021, pp. 336–347.

[46] P. W. McBurney and C. McMillan, "Automatic source code summarization of context for java methods," *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 103–119, 2016.

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

[48] A. Mesbah, A. Rice, E. Johnston, N. Glorioso, and E. Aftandilian, "Deepdelta: Learning to repair compilation errors," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019, 2019, pp. 925–936.

[49] K. Moran, D. N. Palacio, C. Bernal-Cardenas, D. McCrystal, D. Poshyvanyk, C. Shenefiel, and J. Johnson, "Improving the effectiveness of traceability link recovery using hierarchical bayesian networks," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, oct 2020, pp. 873–885. [Online]. Available: https://doi.ieeecomputersociety.org/

[50] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *2013 21st International Conference on Program Comprehension (ICPC)*, 2013, pp. 23–32.

[51] M. Namavar, N. Nashid, and A. Mesbah, "A controlled experiment of different code representations for learning-based bug repair," *arXiv preprint arXiv:2110.14081*, 2021.

[52] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Lexical statistical machine translation for language migration," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, 2013, pp. 651–654.

[53] ——, "Migrating code with statistical machine translation," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014, 2014, pp. 544–547.

[54] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan, "A study of repetitiveness of code changes in software evolution," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE'13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 180–190.

[55] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15, 2015, pp. 574–584.

[56] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for java," in *OOPSLA'07*, 01 2007, pp. 815–816.

[57] 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.

[58] D. Pierret and D. Poshyvanyk, "An empirical exploration of regularities in open-source software lexicons," in *The 17th IEEE International Conference on Program Comprehension, ICPC 2009, Vancouver, British Columbia, Canada, May 17-19, 2009*, 2009, pp. 228–232.

[59] A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, and D. Binkley, "Recovering test-to-code traceability using slicing and textual analysis," *J. Syst. Softw.*, vol. 88, no. C, p. 147–168, Feb. 2014.

[60] 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," 2019.

[61] K. Rahmani, M. Raza, S. Gulwani, V. Le, D. Morris, A. Radhakrishna, G. Soares, and A. Tiwari, "Multi-modal program inference: A marriage of pre-trained language models and component-based synthesis," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021.

[62] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 419–428. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594321

[63] R. Robbes and A. Janes, "Leveraging small software engineering data sets with pre-trained neural networks," in *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2019, Montreal, QC, Canada, May 29-31, 2019*, A. Sarma and L. Murta, Eds. IEEE / ACM, 2019, pp. 29–32.

[64] P. Rodeghero, S. Jiang, A. Armaly, and C. McMillan, "Detecting user story information in developer-client conversations to generate extractive summaries," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE ?17, 2017, p. 49?59.

[65] S. Shamshiri, "Automated Unit Test Generation for Evolving Software," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. FSE'15. Bergamo, Italy: ACM, 2015, pp. 1038–1041.

[66] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," *SIGPLAN Not.*, vol. 50, no. 6, pp. 43–54, Jun. 2015.

[67] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *2011 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 101–110.

[68] ——, "Generating parameter comments and integrating with method summaries," in *2011 IEEE 19th International Conference on Program Comprehension*, 2011, pp. 71–80.

[69] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *CoRR*, vol. abs/1409.3215, 2014.

[70] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," *arXiv preprint arXiv:2005.08025*, 2020.

[71] R. F. Tate, "Correlation between a discrete and a continuous variable. point-biserial correlation," *The Annals of mathematical statistics*, vol. 25, no. 3, pp. 603–607, 1954.

[72] M. Tufano, D. Drain, A. Svyatkovskiy, and N. Sundaresan, "Generating accurate assert statements for unit test cases using pretrained transformers," *arXiv preprint arXiv:2009.05634*, 2020.

[73] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, 2019, pp. 25–36.

[74] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "Deep learning similarities from different representations of source code," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, 2018, pp. 542–553.

[75] 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.

[76] ——, "Learning how to mutate source code from bug-fixes," in *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019*, 2019, pp. 301–312.

[77] R. Tufano, S. Masiero, A. Mastropaolo, L. Pascarella, D. Poshyvanyk, and G. Bavota, "Automating code review activities 2.0," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE.

[78] R. Tufano, L. Pascarella, M. Tufano, D. Poshyvanyk, and G. Bavota, "Towards automating code review activities," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 163–174.

[79] B. Vasilescu, C. Casalnuovo, and P. Devanbu, "Recovering clear, natural identifiers from obfuscated js names," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, 2017, pp. 683–693.

[80] 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.

[81] C. Watson, N. Cooper, D. 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*.

[82] C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On learning meaningful assert statements for unit test cases," in *Proceedings of the 42nd International Conference on Software Engineering, ICSE 2020*, 2020, p. To Appear.

[83] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 87–98.

[84] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward Deep Learning Software Repositories," in *Proceedings of the 12th IEEE Working Conference on Mining Software Repositories (MSR'15)*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 334–345. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820518.2820559

[85] M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk, "Sorting and transforming program repair ingredients via deep learning code similarities," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, p. to appear.

[86] Y. Yang, X. Xia, D. Lo, and J. Grundy, "A survey on deep learning for software engineering," *arXiv preprint arXiv:2011.14597*, 2020.

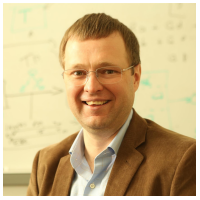
[87] S. Zafar, M. Z. Malik, and G. S. Walia, "Towards standardizing and improving classification of bug-fix commits," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2019, pp. 1–6.

Antonio Mastropaolo is a Ph.D. student in the Faculty of Informatics at the Università della Svizzera italiana (USI), Switzerland, where he is part of the Software Institute. He received his MSc. in Software System Security from Università degli studi del Molise, Italy, in July 2020. His research interests include the study and the application of deep-learning techniques to foster code-related tasks. More information available at: https://antoniomastropaolo.com.

Nathan Cooper received a B.S. degree in Software Engineering from the University of West Florida in 2018. He is currently a Ph.D. candidate in Computer Science at William & Mary under the advisement of Dr. Denys Poshyvanyk and is a member of the Semeru Research group. He has research interests in Software Engineering, Machine / Deep Learning applications for Software Engineering, information retrieval, and question & answering applications for Software Engineering. He has published in the top peer-reviewed Software Engineering venues ICSE and MSR. He has also received the ACM SIGSOFT Distinguished paper award at ICSE'20. More information is available at https://nathancooper.io/#/.

David N. Palacio is a Ph.D. Candidate in Computer Science at The College of William & Mary, where he is a member of the SEMERU Research Group supervised by Dr. Denys Poshyvanyk. He received his MSc. in Computer Engineering at Universidad Nacional de Colombia (UNAL), Colombia, 2017. His research is concentrated on interpretable methods for deep learning code generators, specifically, towards using causal inference to explain deep software models. His fields of interest lie in complexity science, neuroevolution, causal inference, and interpretable machine learning for the study and automation of software engineer processes. More information available at https://danaderp.github.io/danaderp/.

Simone Scalabrino is a Research Fellow at the University of Molise, Italy. He has received his MS degree from the University of Salerno, and his PhD degree from the University of Molise, defending a thesis on automatically assessing and improving source code readability and understandability. His main research interests include code quality, software testing, and empirical software engineering. He has received three ACM SIGSOFT Distinguished Paper Awards at ICPC 2016, ASE 2017, and MSR 2019. He is co-founder and CSO of datasound, a spin-off of the University of Molise. More information available at: https://dibt.unimol.it/sscalabrino/.



Denys Poshyvanyk is a Professor of Computer Science at William and Mary. He received the MS and MA degrees in Computer Science from the National University of Kyiv-Mohyla Academy, Ukraine, and Wayne State University in 2003 and 2006, respectively. He received the PhD degree in Computer Science from Wayne State University in 2008. He served as a program co-chair for ASE'21, MobileSoft'19, ICSME'16, ICPC'13, WCRE'12 and WCRE'11. He currently serves on the editorial board of IEEE Transactions on Software Engineering (TSE), ACM Transactions on Software Engineering and Methodology (TOSEM), Empirical Software Engineering Journal (EMSE, Springer), Journal of Software: Evolution and Process (JSEP, Wiley) and Science of Computer Programming. His research interests include software engineering, software maintenance and evolution, program comprehension, reverse engineering and software repository mining. His research papers received several Best Paper Awards at ICPC'06, ICPC'07, ICSM'10, SCAM'10, ICSM'13, CODAPSY'19 and ACM SIGSOFT Distinguished Paper Awards at ASE'13, ICSE'15, ESEC/FSE'15, ICPC'16, ASE'17, ESEC/FSE'19 and ICSE'20. He also received the Most Influential Paper Awards at ICSME'16, ICPC'17, ICPC'20 and ICSME'21. He is a recipient of the NSF CAREER award (2013). He is a member of the IEEE and ACM. More information is available at: http://www.cs.wm.edu/~denys/.

Rocco Oliveto is a Professor in the Department of Bioscience and Territory at University of Molise (Italy). He is the Chair of the Computer Science program and the Director of the Laboratory of Computer Science and Scientific Computation of the University of Molise. He received the PhD in Computer Science from University of Salerno (Italy) in 2008. His research interests include traceability management, information retrieval, software maintenance and evolution, search-based software engineering, and empirical software engineering. He is author of about 150 papers appeared in international journals, conferences and workshops. He serves and has served as organizing and program committee member of international conferences in the field of software engineering. He is a member of IEEE Computer Society and ACM.

Gabriele Bavota is an associate professor at the Faculty of Informatics of the Università della Svizzera italiana (USI), Switzerland, where he is part of the Software Institute and he leads the SEART research group. He received the PhD in Computer Science from the University of Salerno, Italy, in 2013. His research interests include software maintenance and evolution, code quality, mining software repositories, and empirical software engineering. On these topics, he authored over 140 papers appeared in international journals and conferences and has received four ACM Sigsoft Distinguished Paper awards at the three top software engineering conferences: ASE 2013 and 2017, ESEC-FSE 2015, and ICSE 2015. He also received the best/distinguished paper award at SCAM 2012, ICSME 2018, MSR 2019, and ICPC 2020. He is the recipient of the 2018 ACM Sigsoft Early Career Researcher Award for outstanding contributions in the area of software engineering as an early career investigator and the principal investigator of the DEVINTA ERC project. More information is available at: https://www.inf.usi.ch/faculty/bavota/.