Studying the Usage of Text-To-Text Transfer Transformer to Support Code-Related Tasks

Antonio Mastropaolo*, Simone Scalabrino†, Nathan Cooper‡, David Nader Palacio‡, Denys Poshyvanyk‡,
Rocco Oliveto†, Gabriele Bavota*
*SEART @ Software Institute, Università della Svizzera italiana (USI), Switzerland
†University of Molise, Italy
‡SEMERU @ Computer Science Department, William and Mary, USA

***Abstract*—Deep learning (DL) techniques are gaining more and more attention in the software engineering community. They have been used to support several code-related tasks, such as automatic bug fixing and code comments generation. Recent studies in the Natural Language Processing (NLP) field have shown that the Text-To-Text Transfer Transformer (T5) architecture can achieve state-of-the-art performance for a variety of NLP tasks. The basic idea behind T5 is to first pre-train a model on a large and generic dataset using a self-supervised task (*e.g.,* filling masked words in sentences). Once the model is pre-trained, it is fine-tuned on smaller and specialized datasets, each one related to a specific task (*e.g.,* language translation, sentence classification). In this paper, we empirically investigate how the T5 model performs when pre-trained and fine-tuned to support code-related tasks. We pre-train a T5 model on a dataset composed of natural language English text and source code. Then, we fine-tune such a model by reusing datasets used in four previous works that used DL techniques to: (i) fix bugs, (ii) inject code mutants, (iii) generate assert statements, and (iv) generate code comments. We compared the performance of this single model with the results reported in the four original papers proposing DL-based solutions for those four tasks. We show that our T5 model, exploiting additional data for the self-supervised pre-training phase, can achieve performance improvements over the four baselines.**

***Index Terms*—Empirical software engineering, Deep Learning**

I. Introduction

Deep Learning (DL) has been used to support a vast variety of code-related tasks. Some examples include automatic bug fixing [1]–[4], learning generic code changes [5], code migration [6], [7], code summarization [8]–[11], pseudo-code generation [12], code deobfuscation [13], [14], injection of code mutants [15], automatic generation of assert statements [16], and code completion [17]–[21]. These works customize DL models proposed in the Natural Language Processing (NLP) field to support the previously listed tasks. For instance, Tufano *et al.* [1] used an RNN Encoder-Decoder architecture, commonly adopted in Neural Machine Translation (NMT) [22]–[24], to learn how to automatically fix bugs in Java methods. The model learned bug-fixing patterns by being trained on pairs of buggy and fixed methods mined from software repositories. This work, as the vast majority of the ones previously mentioned (*e.g.,* [5], [9], [11], [15], [16]), 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 example, in the work by Watson *et al.* [16] the input is a string representing a test method without an assert statement, and the output is an appropriate assert statement for the given test. In the approach by Haque *et al.* [11], the input is composed of strings representing a subroutine to document, while the output is a natural language summary documenting the subroutine.

Recent years have seen the raise 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.* [25] 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.

The goal of this work is to empirically investigate the potential of a T5 model when pre-trained and fine-tuned to support many of the previously listed code-related tasks also characterized by text-to-text transformations. 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.,* methods). Then, we fine-tune the model using four datasets from previous work with the goal of supporting four code-related tasks:

Automatic bug-fixing. We use the dataset by Tufano *et al.* [1], 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.* [15], 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 use the dataset by Watson *et al.* [16], 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 use the dataset by Haque *et al.* [11] where input strings are some representations of a Java method to summarize, & an output string is a textual summary.

Once the T5 model has been fine-tuned on all these tasks, we run it on the same test sets used in the four referenced works [1], [11], [15], [16] comparing the achieved results to those reported in the original work. Our results show that the T5 model is able to improve the performance of the original models in all four tasks.

Worth noticing is that, besides the different architecture of the T5 model, the latter can take advantage of a pre-training phase in which additional training data is provided as input as compared to the four baselines. This could explain, at least partially, the boost of performance that we observed. Also, as previously said, the additional pre-training is done in a self-supervised way (*i.e.,* by simply masking random tokens in the code/text used for pre-training), making this step relatively cheap to perform and scalable to large code bases that can be easily collected from sources such as GitHub. In contrast, the four baselines exploit a completely supervised training (*e.g.,* in the case of automatic bug-fixing, the baseline needs pairs of buggy and fixed methods to be trained). Building such a dataset for supervised training has a cost, and there are limitations in terms of the amount of data one can mine.

Besides the good performance ensured by the T5, having a single model able to support different tasks can benefit technological transfer since it simplifies the implementation and the maintenance of a tool supporting several tasks. The code and data used in this work are publicly available [26].

II. Related Work

DL techniques have been used to support many software engineering tasks. Due to space limitations, we discuss only the approaches related to the four tasks we subject to our study, with particular attention on those used as baselines. We also introduce notions needed to understand our experimental design.

A. Automatic Bug-Fixing

Many techniques have been proposed for the automatic fixing of software bugs. Several of them [27]–[35] 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 [36], [37]. In this section we focus on techniques exploiting DL for bug-fixing.

Mesbah *et al.* [3] 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.* [2] present SequenceR, a sequence-to-sequence approach trained on over 35k single-line bug-fixes. SequenceR takes as input the buggy line together with its "abstract buggy context", meaning the relevant code lines from the buggy class.

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.* [4] and Tufano *et al.* [1]. The latter is the one we compared our approach with and, thus, we describe it in more details.

Tufano *et al.* [1] 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.

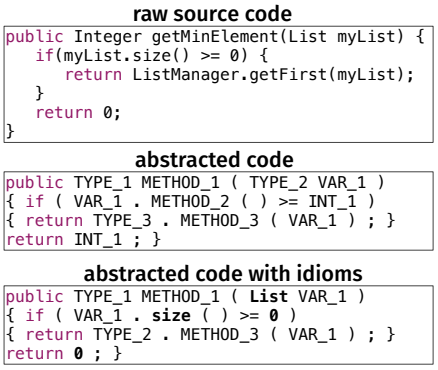

Fig. 1: Abstraction process [1]

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 [2], while compilation failures are addressed in [3]).

Thus, we picked it as representative DL technique for automatic bug-fixing and we use the two datasets by Tufano *et al.* [1] to fine-tune the T5 model for the "automatic bug-fixing" problem, comparing the achieved performance with the one reported in the original paper.

B. Injection of Code Mutants

Brown *et al.* [38] 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.* [15] built on this idea by presenting an approach using NMT to inject mutants representative of real bugs. The idea is similar to the previously described "bug-fixing" paper [1] with, however, the learning happening in the opposite direction. Indeed, given a bug-fixing commit, the input to the model is in this case the "fixed method" (*i.e.,* the method obtained after the bug-fixing activity) while the target is the buggy method (before the bug-fix). This allows the model to learn how to inject in a working code a mutant representative of real bugs. The applied methodology is the same described for the bug-fixing work [15], 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.* [15] 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).

C. Generation of Assert Statements in Test Methods

Watson *et al.* [16] start from the work by Shamshiri *et al.* [39], who observed that tools for the automatic generation of test cases such as Evosuite [40], Randoop [41] and Agitar [42] exhibit insufficiencies in the automatically generated assert statements.

[1]A subset of this dataset named $GM_{ident-lit}$ has also been used in the original paper [15] 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.

Thus, they propose ATLAS, an approach for generating syntactically and semantically correct unit test assert 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.* [1]. 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).

This is the only DL-based technique proposed in the literature to generate assert statements. We use the datasets by Watson *et al.* [16] 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.

D. 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 [43]–[46] and abstractive [9], [11], [47]–[49]. The former create a summary of a code component which includes information extracted from the component being summarized, while the latter may include in the generated summaries information that is not present in the code component to document. DL techniques have been used to support the generation of abstractive summaries.

Hu *et al.* [49] 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 [50], showing the superiority of their approach with respect to the competitive technique presented in [51].

Allamanis *et al.* [52] 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.* [8] 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.* [49] and Iyer *et al.* [51].

The approach by Haque *et al.* [11] 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.* [8].

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.* [11].

III. Multitask Learning for Code-related Tasks

The T5 model was introduced by Raffel *et al.* [25] to support multitask learning in the domain of NLP. This approach is based on two phases: *pre-training*, which allows defining a shared knowledge-base useful for a large class of sequence-to-sequence tasks, and *fine-tuning*, which specializes the model to specific tasks of interest. In this section, we first provide basic information about the T5 model (refer to [25] for a detailed explanation of the architecture). Then, we explain how we adapted it to the software engineering domain, with the goal of supporting the four tasks previously described: *automatic bug-fixing*, *generation of assert statements in test methods*, *code summarization*, and *injection of code mutants*. Such a process is depicted in Fig. 2. Finally, we describe the hyperparameter tuning of the model and the adopted decoding strategy.

A. T5 in a Nutshell

The T5 model is based on the transformer model architecture [53] that allows to handle a variable-sized input using stacks of self-attention layers [54] instead of RNNs or CNNs. When an input sequence is provided, it is mapped to a sequence of embeddings that is passed into the encoder.

The *encoders* are all identical in structure and each one is comprised of two subcomponents: a *self-attention layer* followed by a small *feed-forward network*. Layer normalization [55] is applied to the input of each subcomponent while a residual skip connection [56] adds each input of the subcomponent to its output. Dropout [57] is applied within the feed-forward network, on the skip connection, on the attention weights, and at the input and output of the entire stack. The *decoders* work similarly to the encoders: Each self-attention layer is followed by an additional attention mechanism that attends to the output of the encoder.

The output of the final decoder block is fed into a dense layer with a softmax output, to produce the output probabilities over the vocabulary. Differently from the generic transformer model, the T5 model [25] uses a simplified form of position embeddings, where each embedding is a scalar that is added to the corresponding logit used for computing the attention weights. As pointed out by the authors, for efficiency they also share the position embedding parameters across all layers.

The T5, in particular, and a transformer model, 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 [25]: *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 [25], even if the accuracy of the most complex variants are higher than the less complex models, the training complexity increases with the number of parameters. Considering the available computational resources, in our work we decided to use the simplest $T5_{small}$ model. We expect the results achieved in our study to be a lower bound for the performance of a T5-based model. Nevertheless—as reported in Section V—the $T5_{small}$ model is still able to outperform state-of-the-art approaches.

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.

B. Pre-training of T5

In the *pre-training* phase we use a self-supervised task similar to the one used by Raffel *et al.* [25], consisting of masking tokens in natural language sentences and asking the model to guess the masked tokens. Differently, we did not perform the pre-training by only using natural language sentences, since all the tasks we target involve source code. Thus, 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 [58], 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. Then, since for three of the four tasks we support (*i.e., automatic bug-fixing* [1], *generation of assert statements* [16], and *injection of code mutants* [15]) the authors of the original papers used an abstracted version of source code (see Section II), we used the `src2abs` tool by Tufano [1] to create an abstracted version of each mined Java method.

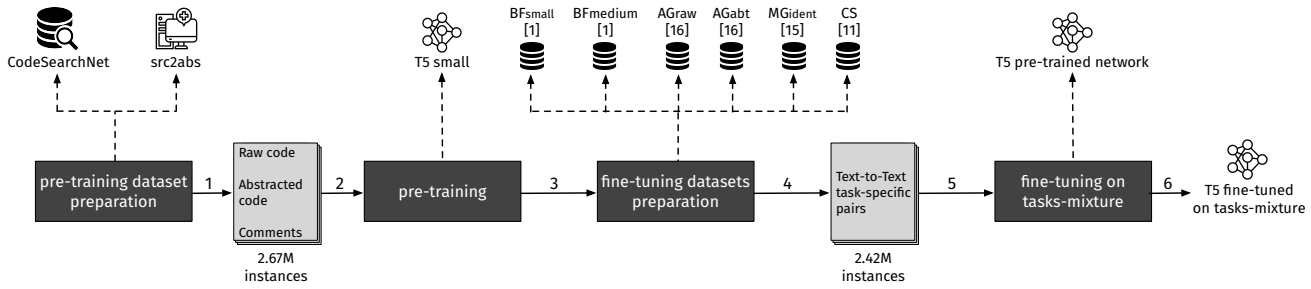

Fig. 2: Overview of the approach used to pre-train and fine-tune the T5 model.

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. Finally, 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 2,672,450 instances reported in Table I. This is the dataset we use for pre-training the T5 model, using the BERT-style objective function Raffel *et al.* used in their final experiments and consisting of randomly masking 15% of tokens (*i.e.,* words in comments and code tokens in the raw and abstracted code).

Data sources	Instances
Source code	1,569,773
Abstracted source code	766,129
Technical natural language	336,548
Total	2,672,450

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

Finally, since we pre-train the model on a software-specific dataset, we needed to create a new vocabulary to accommodate the tokens in our dataset. For this reason, we created a new *SentencePiece* model [59] (*i.e.,* a tokenizer for neural text processing) by using the entire pre-training dataset.

C. Fine-tuning of T5

We use a slightly modified version of the multi-task learning approach used by Raffel *et al.* [25]: we fine-tune the model on a mixture of tasks instead of performing fine-tuning for each single task.

Task	Dataset	Evaluation-set	Training-set	Test-set
Bug Fixing	BF_{small} [1]	5,835	46,680	5,835
	BF_{medium} [1]	6,546	52,364	6,545
Mutant Generation	MG_{ident} [15]	11,560	92,476	11,559
Assert Generation	AG_{abs} [16]	15,809	126,477	15,810
	AG_{raw} [16]	18,816	150,523	18,815
Code Summarization	CS [11]	104,272	1,953,940	90,908
Total		162,838	2,422,460	149,472

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

We do this because of the relatively small size of the specialized datasets available. Table II reports summary characteristics of the datasets we use for each task. Also, for each task we have to provide a consistent framing of the input that allows the model to recognize the tasks that should be performed given an input sequence of tokens. We use a special token sequence indicating the task at hand (*e.g.,* "generate small patch" for BF_{small}, followed by the token ":" and by the input required by the task.

1) Datasets Used for Fine-Tuning: In the following, we describe the details of the datasets we use for *fine-tuning* the model for the four targeted tasks.

Automatic Bug Fixing (BF). We use the dataset by Tufano *et al.* [1] 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 method [1].

Injection of Code Mutants (MG). For the MG task we exploited one of the two datasets provided by Tufano *et al.* [5]: 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.* [16] 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 [60], 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.* [11] 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 [11]. 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.

2) Data Balancing: The datasets we use for fine-tuning have different sizes, with the one for code summarization dominating the others. 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.* [61], 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, while fine-tuning, we decided not to perform any particular adaptation of our training set, following the true data distribution when creating each batch: 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.

D. Decoding Strategy

Given the values of the output layer, 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 sequence (*e.g.,* the buggy method).

Beam-search is an alternative decoding strategy previously used in many DL applications [62]–[65]. 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).

E. Hyperparameter Tuning

For the *pre-training* phase, we use the default parameters defined for the T5 model [25]. 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 (we use $LR = 0.001$, *i.e.,* the value used in the original paper [25]); (ii) *Inverse Square Root Learning Rate* (ISR-LR): the learning rate decays as the inverse square root of the training step (the same used for pre-training by Raffel *et al.*); (iii) *Slanted Triangular Learning Rate [66]* (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 III reports the specific parameters we use for each scheduling strategy: the values are the default ones reported in the papers that introduced them.

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	$LR_{starting} = 0.01$ $LR_{end} = 1e{-}06$ $Power = 0.5$

TABLE III: Learning-rates tested for hyperparameter tuning.

Dataset	Metric	C-LR	ST-LR	ISQ-LR	PD-LR
BF_{small} [1]	Accuracy@1	6.9%	**13.2%**	11.0%	0.27%
BF_{medium} [1]	Accuracy@1	2.9%	**5.5%**	3.3%	0.0%
MG_{ident} [15]	BLEU-A	75.6%	**78.2%**	77.7%	12.0%
AG_{abs} [16]	Accuracy@1	33.7%	39.7%	**39.8%**	2.0%
AG_{raw} [16]	Accuracy@1	48.9%	**57.6%**	56.7%	2.1%
CS [11]	BLEU-A	23.3%	23.6%	**24.3%**	3.4%
# Best Results		0	**4**	2	0

TABLE IV: Hyperparameter tuning results.

We pre-train the model for a total of 100k steps in the four configurations on the whole pre-training set and we test it on the evaluation sets of the datasets provided by the original papers we compare with.

We compute the following metrics: for BF and AG, we compute the percentage of perfect predictions achieved with the greedy decoding strategy (Accuracy@1); for MG, we compute the BLEU score [50]; for CS, we compute BLEU-A, the geometric average of the BLEU-{1,2,3,4} scores [50]. Basically, for each task we adopt one of the evaluation metrics used in the original paper (details about these metrics are provided in Section IV-A). We report in Table IV the achieved results (in bold the learning rate obtaining the best performance for each metric/dataset). As it can be noticed, the *Slanted Triangular Learning Rate (ST-LR)* allows to achieve the best performance in most of the cases. For this reason, we decided to use this particular learning rate in our model.

Several other hyperparameters could have been tuned. Given the high computational cost to train the model (∼343 hours on a colab [67] instance with 8 tpu cores and 35.5GB of RAM), we did not manage to perform a comprehensive hyperparameter tuning. The high dimensionality of the model, indeed, makes hyperparameter tuning not very cost-effective: we preferred to use the computational power available to increase the number of steps for training the model.

IV. Study Design

The *goal* of our study is to understand whether *multi-task learning*, in general, and a T5-based model, in particular, is suitable for automating code-related tasks. The *context* is represented by the datasets introduced in Section II, *i.e.,* the ones by Tufano *et al.* for bug fixing [1] and injection of mutants [15], by Watson *et al.* for assert statement generation [16], and by Haque *et al.* for code summarization [11].

Our study is steered by the following research question: *Is the T5 model suitable for code-related tasks such as automatic bug fixing, injection of mutants, assert statement generation and code summarization?*

A. Experimental Procedure

We use the model we trained and tuned as we specified in Section III and we run it on the test sets provided in the previously described datasets. Our baselines are the state-of-the-art models described in Section II. For each task and dataset, we compare the results achieved by our model with the results reported in the original papers.

Task	Baseline	Accuracy@K	BLEU-n	ROUGE LCS
BF	[1]	$\{1, 5, 10, 25, 50\}$	-	-
MG	[15]	$\{1\}$	$\{A\}$	-
AG	[16]	$\{1, 5, 10, 25, 50\}$	-	-
CS	[11]	-	$\{1, 2, 3, 4, A\}$	$\{P, R, F\}$

TABLE V: Baselines and evaluation metrics for the tasks.

We use different metrics for the different tasks, depending on the metrics reported in the papers that introduced our baselines. Table V reports the baselines and metrics used to evaluate the results for each task, that we define below.

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 [1] (BF) and [16] (AG). Tufano *et al.* [5] do not report results for $K > 1$ for the MG task. Thus, we only compare the results with $K = 1$.

BLEU score (Bilingual Evaluation Understudy) [50] 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. 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 [68]. 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 [11]. 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.

Besides such effectiveness metrics, we also perform an additional analysis: we compute the *inference time*, *i.e.,* the time needed to run the model 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. 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 on all the instances of each task. This allows understanding the *efficiency* of the model and to what extent it can be used in practice.

Finally, for each task, we also compute the complementarity between T5 and the baseline approach. For each dataset d we consider and the related baseline approach BL_d, we first define the sets of perfect predictions obtained by the two approaches PP_{T5_d} and PP_{BL_d} with a fixed beam size $K = 1$.

Then, we compute three metrics:

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

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

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

V. Results Discussion

We report a summary of the results achieved by T5 (in red) and by the respective baselines (in orange) for the four tasks we consider (*i.e.,* BF, MG, AG, and CS) in Fig. 3. We also show the inference times for all the tasks and the overlap metrics between T5 and the experimented baselines in Table VI and Table VII, respectively. We discuss the results task by task below.

TABLE VI: Inference time with different beam size values.

K	BF_{small}	BF_{medium}	MG_{ident}	AG_{abs}	AG_{raw}	CS
1	0.41	1.84	0.31	0.35	0.36	0.12
5	0.62	1.13	0.54	0.79	0.66	0.17
10	0.72	1.55	0.62	1.17	1.20	0.24
25	1.30	3.35	1.13	2.45	2.66	0.40
50	2.16	5.31	2.04	4.82	4.96	0.74

A. Automatic Bug Fixing (BF)

When using T5 for automatically fixing bugs, the accuracy achieved using a greedy decoding strategy ($K = 1$) is very similar to the one achieved by the baseline on both the datasets we consider, *i.e.,* BF_{small} and BF_{medium}. While on the first one there is a 1% improvement, on the other the results are exactly the same. However, when increasing the beam size, the difference becomes larger: on BF_{small} the improvement ranges between 8-10%, while on BF_{medium} it is lower, and it ranges between 4-9%. In general, it can be noticed that the improvement margin is constant.

The time needed to generate a fix depends on the dataset, *i.e.,* on the number of tokens of the input. If we use the BF_{small} dataset, the average inference time ranges between 0.41s ($K = 1$) and 2.16s ($K = 50$), while it is larger on the BF_{medium} dataset (1.84s for $K = 1$ and 5.31s for $K = 50$).

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

Dataset (d)	$Shared_d$	$OnlyT5_d$	$OnlyBL_d$
BF_{small}	37.67%	36.52%	25.81%
BF_{medium}	28.78%	36.06%	35.16%
MG_{ident}	41.03%	46.65%	12.32%
AG_{abs}	39.78%	36.19%	24.03%
AG_{raw}	11.68%	84.30%	4.02%
CS	4.97%	93.46%	1.57%

There is a considerable overlap between the perfect predictions done by the two approaches (see Table VII): ∼38% of perfect predictions on BF_{small} and ∼29% on BF_{medium} are shared by the two techniques.

The remainder are perfect predictions only with T5 (∼36% on BF_{small} and ∼36% on BF_{medium}) or only with the baseline (∼26% on BF_{small} and ∼35% on BF_{medium}). This indicates that the two approaches are complementary for the BF task suggesting that, even if T5 was not able to fix some bugs, it is still possible to automatically fix (a subset of) such bugs with a specialized ML-based approach. This recalls the need to further enrich the architecture of a transfer learning method with the goal of further improving its ability to exploit the knowledge acquired on specific tasks.

B. Injection of Code Mutants (MG)

Looking at Fig. 3, we can observe that using T5 to generate mutants allows to obtain much more accurate results than the baseline, with the Accuracy@1 improving by 11%, with 1,240 additional perfect predictions (+62% as compared to the baseline). The average BLEU score improves by ∼0.01 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 [69].

As for the inference time (Table VI), we observed similar results compared to the BF task on the BF_{small} dataset: with $K = 1$, the average inference time is 0.31s, while for $K = 50$ it is 2.04s. We do not report perfect predictions at $K = 50$ since those were not reported in the original paper [15].

Similarly to BF, also for MG the percentage of shared perfect predictions (Table VII) is quite high (∼41%) with, however, T5 being the only one generating ∼46% of perfect predictions as compared to the ∼12% of the baseline approach.

C. Generation of Assertions in Test Methods (AG)

T5 achieves very similar results compared to the baseline on the AG_{abs} (see Fig. 3): when abstracting the tokens, both approaches achieve very similar levels of accuracy, and such values are reasonably high with the increase of K (*e.g.,* they both achieve 65% accuracy with $K = 50$). However, when using the more challenging non-abstracted dataset AG_{raw}, T5 allows to achieve much better results: it achieves a 29% higher accuracy with $K = 1$, while for larger K values the gap in performance ranges between 35-38%. The most interesting result, however, is that T5 achieves similar results both with and without abstraction, with the the Accuracy@1 being higher when considering AG_{raw} then when considering AG_{abs}. The fact that T5 is capable of handling raw source code makes its usage more straightforward compared to the baseline: it does not need pre- and post-processing steps for such a task.

Assert generation is very fast for low values of K (0.36s for both the datasets with $K = 1$), while it gets much slower for higher values of K, at a higher rate compared to other tasks (4.82s for AG_{abs} and 4.96s for AG_{raw} with $K = 50$).

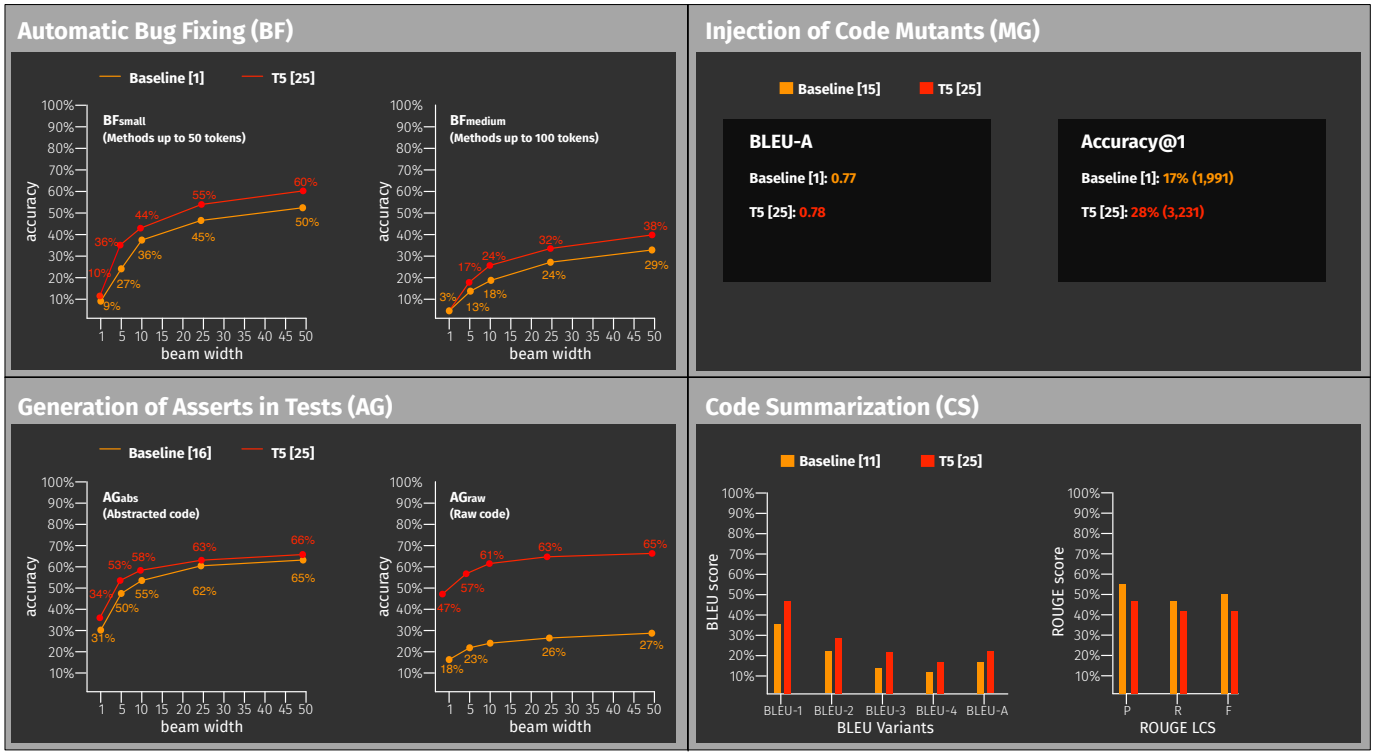

Fig. 3: Performance of the T5 model against the experimented baselines.

In terms of overlap, we found a trend similar to BF on AG_{abs}: we have ∼40% of perfect predictions shared between the two approaches, while the remainder instances are distributed between the ones only predicted by T5 (∼36%) and the ones only predicted by the baseline (∼24%).

There is, instead, a small overlap on the AG_{raw} dataset: only ∼12% of the instances are perfectly predicted by both the approaches, with ∼84% of them correctly predicted only by T5.

D. 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%. 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 (∼6% 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 [11]. This means counting the comments generated by a technique that are exactly equal to the ones manually written by humans. T5 managed to generate 11.4% of perfect predictions (10,401 instances) against the 3.4% (3,048) of the baseline technique (over 3 × better).

Code summarization is the fastest task to complete for T5: it takes only 0.12s for $K = 1$ and 0.74s for $K = 50$.

As expected from previous results, the majority of the perfect predictions for this task can be done only using T5 (∼93%). A limited percentage of perfect predictions is shared (∼5%), and a minority of instances can be only predicted through the baseline (∼2%).

E. Qualitative Examples

We show in Fig. 4 four examples of perfect predictions by T5; for the sake of space limitations, for BF and MG, we only report the parts of code that T5 modified. In the first one (BF), the developers used the `!=` operator for comparing two objects instead of calling the `equals` method. T5 was able to fix the bug by (i) adding a call to `equals` and, less obvious, (ii) adding the `!` operator before the method call. In the second example (MG), T5 generates a mutant by replacing the correct string (`string_1`) with a different one (`string_2`). In the third example (AG), the test checks if a call to `setPosition` on the variable `instant1` does not change its value (*i.e.,* it should be equal to `result`). Such an assertion is not trivial to generate since `result` is used in the assertion even if `p` is closer to the assert placeholder. Finally, in the last example (CS), T5 detects that (i) the method applies a given procedure to the pairs, (ii) the pairs belong to a receiver, and (iii) this happens only if there is a receiver. This shows how T5 is able to generate a summary for a method that even a developer could struggle to understand.

Automatic Bug Fixing (BF)

```
public void method_1(int var_1, int var_2, Intent data) {
  super.method_1(var_1, var_2, data);
  if ((data != null) && ((data.method_2(var_3)) != string_1)) {
    var_4.setText(data.method_2(var_3));
  }
}

> if ((data != null) && (!(data.method_2(var_3)).equals(string_1))) {
```

Injection of Code Mutants (MG)

```
public void method_1(result) {
  if (result == null) {
    var_1.setEnabled(false);
    var_2.setEnabled(false);
    return;
  }
  var_3.setText((string_1 + result));
}

> var_3.setText((string_2 + result));
```

Generation of Assert Statements in Test Methods (AG)

```
//test method
void testSetPosition() {
  Position result = instant1.getPosition();
  Position p = org.geotools.temporal.Object.defaultPosition(new Date());
  ((org.geotools.temporal.Object.DefaultInstant)(instant1)).setPosition(p);
  <assertplaceholder>;
}
//focal method
public Position getPosition() { return this; }

> assertFalse(instant1.getPosition().equals(result))
```

Code Summarization (CS)

```
public boolean forEachPair(final IntObjectProcedure procedure) {
  for (int i = table.length; i-- > 0;) {
    if (state[i] == FULL)
      if (!procedure.apply(table[i], values[i]))
        return false;
  }
  return true;
}

> "applies a procedure to each key value pair of the receiver if any"
```

Fig. 4: Examples of perfect predictions done by T5.

F. Answer to our Research Question

Our study showcases the potential of T5 for code-related tasks. The T5 model achieved better performance as compared to all baselines we experimented with. However, it is important to highlight that there are many factors that may have contributed to such a result. Indeed, the high effectiveness we obtained on all the tasks we experimented with might not only be related to the T5 architecture (*e.g.,* the fact that the T5 supports transfer learning with knowledge acquired on a task that can be reused on other tasks) but to other differences between the study presented in this paper and the experiments performed in the original work. While the datasets used for testing the techniques are exactly the same, two aspects must be considered. First, the type of the model we use (*i.e.,* the transformer model): using such a model in a single-task setting may still allow to achieve an improvement over the respective baselines Second, as previously explained, the pre-training phase may provide "knowledge" to the model that is not available in the training sets used for the fine-tuning (and, thus, not used by the competitive techniques).

The results in terms of inference time show that T5 is able to complete all the tasks very quickly: it always takes less than 6 seconds even to generate 50 alternative solutions.

Note that the inference times we reported are based on the usage of a consumer-level device and by only using CPUs: when using GPUs (Nvidia Tesla P100 provided by Google Colab), the time needed for each task flattens to at most $\sim$0.5 seconds for $K = 50$, *i.e.,* the task variability previously reported disappears. Finally, the overlap analysis indicates that some instances of the considered tasks were not resolved correctly by T5 but were resolved by the baselines. This means that such instances can be still resolved automatically by a ML-based approach. Such a consideration suggests that there is still room for improving the accuracy of T5.

VI. Threats to Validity

Construct validity. For both the pre-training and the fine-tuning of our model we re-used available datasets, just performing some additional cleaning (*e.g.,* removal of duplicates after abstraction in the dataset used for the pre-training). Even though we remove duplicates from the pre-training dataset and double-check all the datasets used for the fine-tuning, it is possible that instances in the pre-training dataset appear in some of the test datasets we reused. For example, a code comment included among the pre-training instances we used could have a duplicate, by chance, in the test set of the CS task, thus helping the T5 model in the prediction. While removing instances from the test sets was not an option since this would not allow a fair comparison between the T5 results and the ones reported in the original papers, we decided to investigate such overlap, to have an idea of the extent to which it could have influenced our findings. We found 0 duplicates between the pre-training dataset and the test sets of: BF_{small}, AG_{abs}, AG_{raw}; 1 in MG_{ident}; 2 in BF_{medium}; and 147 (out of 90,908) in the CS test set. Thus, the influence of duplicates on the reported results should be marginal.

Internal validity. An important factor that influences DL performance is hyperparameters tuning. For the pre-training phase, we used the default T5 parameters selected in the original paper [25] since we expect little margin of improvement for such a task-agnostic phase. For the fine-tuning, due to feasibility reasons, we did not change the model architecture (*e.g.,* number of layers) but we experiment with different learning rates. We are aware that a more extensive calibration would likely produce better results.

External validity. We experimented the T5 model on four tasks using six datasets. The main generalizability issue is related to the focus on Java code. However, excluding the abstraction component, our approach is language agnostic.

VII. Conclusion

We investigated the usage of a T5 model to support four code-related tasks: *automatic bug-fixing*, *generation of assert statements in test methods*, *code summarization*, and *injection of code mutants*. The achieved results show that the T5 model can be successfully used for these tasks, with performance superior to the four baselines. However, as explained in Section V-F, such a finding deserves additional investigations TO better understand what makes T5 performing better.

Also, Raffel *et al.* [25], who originally introduced T5, showed that larger T5 models are able to achieve much better results as compared to the *small* T5 model we used in this work. From this perspective, the results reported in this paper should be considered as a lower bound of the T5 capabilities.

Code and data used in this paper are publicly available [26].

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 was supported in part by the NSF CCF-1955853 and CCF-2007246 grants. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

REFERENCES

[1] 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.

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

[3] 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.

[4] 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

[5] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning 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.

[6] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "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.

[7] ——, "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.

[8] 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.

[9] 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:.

[10] 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.

[11] S. Haque, A. LeClair, L. Wu, and C. McMillan, "Improved automatic summarization of subroutines via attention to file context," 2020.

[12] 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.

[13] 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.

[14] 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.

[15] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "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.

[16] 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.

[17] 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

[18] U. Alon, R. Sadaka, O. Levy, and E. Yahav, "Structural language models of code," *arXiv*, pp. arXiv–1910, 2019.

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

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

[21] S. Brody, U. Alon, and E. Yahav, "Neural edit completion," *arXiv preprint arXiv:2005.13209*, 2020.

[22] 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.

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

[24] 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.

[25] 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.

[26] "Replication package https://github.com/antonio-mastropaolo/T5-learning-ICSE_2021"

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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

[39] 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.

[40] 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.

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

[42] "Utilizing fast testing to transform java development into an agile, quick release, low risk process." [Online]. Available: http://www.agitar.com/

[43] 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.

[44] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Generating parameter comments and integrating with method summaries," in *2011 IEEE 19th International Conference on Program Comprehension*, 2011, pp. 71–80.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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

[52] 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

[53] 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.

[54] J. Cheng, L. Dong, and M. Lapata, "Long short-term memory-networks for machine reading," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, 2016, pp. 551–561.

[55] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *arXiv preprint arXiv:1607.06450*, 2016.

[56] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[57] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[58] 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.

[59] 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

[60] 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.

[61] 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

[62] A. Graves, "Sequence transduction with recurrent neural networks," *CoRR*, vol. abs/1211.3711, 2012. [Online]. Available: http://arxiv.org/abs/1211.3711

[63] N. Boulanger-Lewandowski, Y. Bengio, and P. Vincent, "Audio chord recognition with recurrent neural networks." in *ISMIR*. Citeseer, 2013, pp. 335–340.

[64] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *CoRR*, vol. abs/1409.0473, 2014.

[65] 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

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

[67] "Google colaboraty," https://colab.research.google.com/.

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

[69] I. Caswell and B. Liang, "Recent advances in google translate," https://ai.googleblog.com/2020/06/recent-advances-in-google-translate.html, 2020.