An Empirical Study on the Usage of Transformer Models for Code Completion

Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota

Abstract—Code completion aims at speeding up code writing by predicting the next code token(s) the developer is likely to write. Works in this field focused on improving the accuracy of the generated predictions, with substantial leaps forward made possible by deep learning (DL) models. However, code completion techniques are mostly evaluated in the scenario of predicting the next token to type, with few exceptions pushing the boundaries to the prediction of an entire code statement. Thus, little is known about the performance of state-of-the-art code completion approaches in more challenging scenarios in which, for example, an entire code block must be generated. We present a large-scale study exploring the capabilities of state-of-the-art Transformer-based models in supporting code completion at different granularity levels, including single tokens, one or multiple entire statements, up to entire code blocks (*e.g.,* the iterated block of a *for* loop). We experimented with several variants of two recently proposed Transformer-based models, namely RoBERTa and the Text-To-Text Transfer Transformer (T5), for the task of code completion. The achieved results show that Transformer-based models, and in particular the T5, represent a viable solution for code completion, with perfect predictions ranging from ∼29%, obtained when asking the model to guess entire blocks, up to ∼69%, reached in the simpler scenario of few tokens masked from the same code statement.

Index Terms—Code Completion, Deep Learning, Empirical Software Engineering

1 Introduction

Code completion is considered as one of the "killer" features of modern Integrated Development Environments (IDEs) [18], [49], [72]: It can provide developers with predictions about the next code token (*e.g.,* a method call) to write given the code already written in the IDE, thus speeding up software development and preventing potential mistakes [33], [35].

Several works in this field have been proposed. Most of them aim at advancing the performance of code completion tools, especially in terms of prediction accuracy. Such research has allowed moving from simple alphabetically ranked lists of recommendations for completing what a developer is typing (*e.g.,* a list of possible method calls matching what has been typed by the developer) to "intelligent" completions considering the context surrounding the code [18], [72], the history of code changes [72], and/or coding patterns mined from software repositories [10], [36], [39], [62], [63], [65], [77]. Last, but not least, Deep Learning (DL) models have been applied to code completion [8], [22], [47], [49], [73], [84], setting new standards in terms of prediction performance. Although the performance of code completion techniques has substantially improved over time, the type of support they provide to developers has not evolved at the same pace. Indeed, besides a few works focusing on predicting multiple code tokens (*e.g.,* [8], [73]) or even recommending entire statements (*e.g.,* [11], [83]), most of the approaches presented in the literature have only been experimented in the specific scenario in which the next token the developer is likely to type must be predicted. This leaves the following question partially unanswered: *how far can we go with DL-based token prediction (even beyond the source code line boundary)*?

We present a large-scale empirical study exploring the limits and capabilities of state-of-the-art DL models to support code completion. Besides generating the next token(s) the developer is likely to write, we apply DL models to generate entire statements and code blocks (*e.g.,* the body of an `if` statement). Among the many DL models proposed in the literature, we focus on models using the Transformer architecture [81]. In particular, in our recent work published at MSR 2021 [22] we evaluated the performance of a RoBERTa model [55] in the code completion tasks described above. RoBERTa is a BERT (Bidirectional Encoder Representations from Transformers) model [24] using a pre-training task in which random words in the input sentences are masked out using a special `<MASK>` token, with the model in charge of predicting the masked words. While experimenting with RoBERTa for the task of code completion, we faced an

- *M. Ciniselli is with SEART @ Software Institute, Università della Svizzera italiana, Switzerland. E-mail: matteo.ciniselli@usi.ch*
- *N. Cooper is with SEMERU @ William & Mary, USA. E-mail: nacooper01@email.wm.edu*
- *L. Pascarella is with SEART @ Software Institute, Università della Svizzera italiana, Switzerland. E-mail: luca.pascarella@usi.ch*
- *A. Mastropaolo is with SEART @ Software Institute, Università della Svizzera italiana, Switzerland. E-mail: antonio.mastropaolo@usi.ch*
- *E. Aghajani is with SEART @ Software Institute, Università della Svizzera italiana, Switzerland. E-mail: emad.aghajani@usi.ch*
- *D. Poshyvanyk is with SEMERU @ William & Mary, USA. E-mail: denys@cs.wm.edu*
- *M. Di Penta is with University of Sannio, Italy. E-mail: dipenta@unisannio.it*
- *G. Bavota is with SEART @ Software Institute, Università della Svizzera italiana, Switzerland. E-mail: gabriele.bavota@usi.ch*

important limitation that did not make it suitable for the study we wanted to perform (*i.e.,* the prediction of multiple masked tokens): In the RoBERTa pre-training task n `<MASK>` tokens must be used to mask n code tokens, thus implicitly suggesting to the model how many code tokens must be generated to autocomplete the masked statement. This would not be realistic in a real usage scenario, in which the code completion engine must *guess* the tokens to generate, without the developer suggesting how many tokens must be generated. To overcome this limitation, we had to adapt the RoBERTa pre-training objective to be able to guess, from a single `<MASK>` token masking one or more code tokens in the given statements, which and how many code tokens must be generated [22]. The adaptation of the RoBERTa pre-training objective was inspired by the recently proposed Text-To-Text Transfer Transformer (T5) architecture [68], suggesting this as a good fit for the task of code completion.

In this work, we extend our MSR 2021 paper [22] by showing that the T5 substantially overcomes the performance of the RoBERTa model, being able to correctly predict even entire code blocks, something that we found to be not achievable with RoBERTa. As in [22], we focus on three code prediction scenarios: (i) *token-level* predictions, namely classic code completion in which the model is used to guess the last n tokens in a statement the developer started writing; (ii) *construct-level* predictions, in which the model is used to predict specific code constructs (*e.g.,* the condition of an `if` statement) that can be particularly useful to developers while writing code; and (iii) *block-level* predictions, with the masked code spanning one or more entire statements composing a code block (*e.g.,* the iterated block of a `for` loop).

We compare the performance of several models. First, we use the RoBERTa model as presented in [22] as representative of BERT-like models. Second, we use T5 model for the task of code completion for the first time in this paper. The T5 has been recently shown to outperform many state-of-the-art techniques in code-related tasks [59]. In particular, Mastropaolo *et al.* [59] showed the possibility to train a single T5 model dealing with four code-related tasks, namely bug fixing, injection of code mutant, assert statements generation, and code summarization. Third, we experiment with an n-gram model as a baseline for DL-based models, also showing the impact on its performance of using a caching mechanism as proposed by Hellendoorn and Devanbu [36].

Both RoBERTa and T5 models are 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).

Several tasks can be used in the fine-tuning, to possibly take advantage of *transfer learning* (*i.e.,* the knowledge acquired on one task can be reused by the model for another task). For example, a single model trained on multiple translation tasks (*e.g.,* from English to German, English to French, and French to German) could be more effective than three different models each trained on a specific translation task (*e.g.,* English to German).

In our work, we want to investigate the performance of the two transformer-based models by also looking at the role played on the models' performance by the pre-training task and the transfer learning across different tasks. However, since this requires the training of many different variants of the experimented models, we adopt the following strategy. First, we compare RoBERTa and T5 by training three different models for the three code completion scenarios (*i.e.,* token-level, construct-level, and block-level) we experiment with. This implies creating three different RoBERTa and T5 models (six models overall). Then, we take the best performing one (T5) and we show that using pre-training increases its performance, even though the impact is limited. Finally, we show that fine-tuning a single T5 model to support all three prediction tasks boosts performance confirming transfer learning across the three very similar tasks (*i.e.,* knowledge acquired in one task can be used to perform another task).

The achieved results show that, for a typical code completion task (*i.e., token-level*), T5 correctly guesses all masked tokens in 66% to 69% of cases (depending on the used dataset), while RoBERTa achieving 39% to 52% and the n-gram model 42% to 44%. In the most challenging prediction scenario, in which we mask entire blocks, RoBERTa and the n-gram model show their limitations, being able to only correctly reconstruct the masked block in less than 12% of the cases, while the T5 achieves 30% of correct predictions.

It is worth noting that the goal of our study is not to show that the T5 model is the best option for neural-based code completion. Our work focuses on empirically exploring the capabilities of learning-based code completion techniques, and T5, RoBERTa, and the n-gram model have been chosen as representatives of the state-of-the-art.

In summary, as compared to our MSR 2021 paper [22], the contributions of this work are as the following: (i) we perform a comprehensive empirical study with an additional state-of-the-art approach, namely the T5 model, showing its very promising performance for the code completion task; (ii) differently from [22] in which three different RoBERTa models have been fine-tuned on the three code completion scenarios (*i.e.,* token-level, construct-level, and block-level) without pre-training and without testing the impact of transfer learning, we pre-train and fine-tune several versions of the best performing model (*i.e.,* the T5), to investigate these aspects; (iii) for the best performing model, we also explore the possibility of exploiting the confidence of the predictions as a measure of the prediction quality, showing the reliability of such an indicator.

The source code and data used in our work are publicly available in a comprehensive replication package [21].

2 Research Questions and Context

The study *goal* is to assess the effectiveness of Transformer-based DL models in predicting masked code tokens at different granularity levels. We address the following research questions (RQs):

RQ$_1$: *To what extent are transformer models a viable approach to learn how to autocomplete code?* This RQ investigates the extent to which T5 and RoBERTa can be used for predicting missing code tokens. We assess the quality of the generated predictions from both a quantitative (*i.e.,* BLEU score [25], Levenshtein distance [51]) and a qualitative (*i.e.,* perfect

predictions, potential usefulness of wrong predictions) point of view. RQ_1 is further detailed in the following two sub-RQs:

$RQ_{1.1}$: *To what extent does the number of masked tokens impact the prediction quality?* We train and test the approaches we experiment with on datasets in which masked code tokens span from few contiguous tokens in a given statement to multiple missing statements composing a code block. $RQ_{1.1}$ explores the limits of Transformer models when considering simple and more challenging code completion scenarios.

$RQ_{1.2}$: *To what extent are the performance of the models influenced by the specificity of the dataset employed for training and testing it?* While it is reasonable to expect that larger training datasets tend to help deep learning models, we are interested in answering $RQ_{1.2}$ from a different perspective. To address this RQ, we compare the autocompletion performance on two different datasets: a first, more general one, composed of Java methods; and a second, more specific one, composed of methods from Android apps. While the programming language is the same, the granularity of the two datasets is the same (*i.e.,* method-level granularity), methods in the second dataset make heavy use of Android APIs, and the same APIs are likely to be used for similar purposes, *e.g.,* app features dealing with GPS positioning share common API usages. We expect this to create "regularities" in the Android dataset to help model learning.

RQ_2: *What is the role of pre-training and transfer learning in the performance of Transformer-based models?* As explained in Section 1, both RoBERTa and T5 can be pre-trained and then fine-tuned on several tasks. RQ_2 investigates the boost in performance (if any) brought by (i) pre-training of the models, and (ii) fine-tuning a single model on several tasks to take advantage of transfer learning. Such an additional analysis has been performed only for the best-performing model (*i.e.,* the T5).

RQ_3: *How do transformer models compare to a state-of-the-art n-gram model?* An alternative to DL models is represented by statistical language models based on n-grams. In this research question, we compare the DL models to (i) a classical n-gram model and, (ii) in a smaller study, to the state-of-the-art n-gram cached model [36].

2.1 Context Selection: Datasets

Our study involves two datasets. The first one comes from our MSR'21 paper [22] and is used to fine-tune the RoBERTa and T5 models and to train the n-gram model. We refer to this dataset as *fine-tuning dataset* and it includes both a Java and an Android dataset to allow answering $RQ_{1.2}$. The fine-tuning dataset has been built starting from the CodeSearchNet dataset [41], which features Java methods mined from open source projects. The second dataset has been built specifically to answer RQ_2, *i.e.,* to have a different dataset that can be used to pre-train the best performing model among RoBERTa and T5 (*i.e., pre-training dataset*). The following section describes how the datasets have been built.

2.1.1 Fine-tuning dataset

To create the *Java dataset*, we started from the CodeSearchNet Java Dataset provided by Husain *et al.* [41]. We decided to start from CodeSearchNet rather than from other datasets proposed in the literature (see *e.g.,* [58], [67]) since CodeSearchNet has been already subject to cleaning steps making it suitable for applications of machine learning on code. Also, CodeSearchNet is already organized at method-level granularity (*i.e.,* one instance is a method), while other datasets, such as the 50k [58], collect whole repositories. In particular, CodeSearchNet contains over 1.5M Java methods collected from open-source, non-fork, GitHub repositories. For details on how the dataset has been built, see the report by Husain *et al.* [41]. For our work, the most important criteria used in the dataset construction are: (i) excluding methods of fewer than three lines; (ii) removing near-duplicate methods using the deduplication algorithm from CodeSearchNet; this is done to not inflate the performance of the models as a result of overlapping instances between training and test sets [7] and (iii) removing methods with the name containing the "test" substring in an attempt to remove test methods; methods named "toString" are removed as well. The latter are often automatically generated by the IDEs with a very similar structure (*e.g.,* mostly concatenating class attributes). Thus, they rarely represent a challenging code completion scenario and can result in inflating the prediction accuracy.

To build the *Android dataset* we adopted a similar procedure. We cloned the set of 8,431 open-source Android apps from GitHub available in the AndroidTimeMachine dataset [28]. Then, we extracted from each project's latest snapshot the list of methods. This resulted in a total of ∼2.2M methods. Then, we applied the same filtering heuristics defined for the Java dataset, ending up with 654,224 methods. Since one of the goals of our study is also to compare the performance of the models when applied on a more generic (Java) and a more specific (Android) dataset, we randomly selected (using the `random` Python function) 654,224 methods from the Java dataset, to match the size of the Android dataset.

In our MSR paper [22], we also experimented with code abstraction as used in the previous studies [79], [80] to avoid the open vocabulary problem. However, new DL-based models do not suffer from this limitation anymore thanks to the usage of tokenizers exploiting techniques such as Byte Pair Encoding (BPE) [27]. For this reason, while in [22] we built two versions of the fine-tuning dataset (with and without abstraction), in this work we only focus on the datasets using raw source code since this is the real scenario in which code completion techniques are used. Such clarification is needed since, when building the *fine-tuning dataset*, methods for which parsing errors occurred during the abstraction process were excluded [22], leaving the Java dataset with 634,799 methods, and the Android one with 532,096.

Then, the three versions of each dataset (Java and Android) summarized in Table 1 were created using the following masking processes (note that Table 1 reports the number of instances in each dataset after the filtering steps described below):

Token masking. For each code line l in each method having more than one token we mask its last x tokens, where x is a random number between $1 \dots n-1$, where n is the number of tokens composing l. Given a method m having k lines with more than one token, we generate k versions of m, each of them having one line with the last x tokens masked and the remaining $k-1$ reported without any change (*i.e.,* no masked tokens, just the original raw source code). We set the maximum number of masked tokens to 10 (*i.e.,* if

$x > 10$ then $x = 10$). This scenario simulates the classic code completion task in which a developer is writing a statement and the code completion tool is in charge of autocompleting it.

Construct masking. We selected a number of code constructs for which it could be particularly useful to be supported with automated code completion. Given a method m, we use the scrML [4] toolkit to identify all m's tokens used to: (i) define the complete condition of an `if` statement or of a `while/for` loop (*e.g.,* in a statement having `for(int i=0; i<data.size(); i++)` we identify all tokens between parenthesis as those used to define the `for` loop); (ii) define the parameters in a method call (*e.g.,* in `copyFile(source, target)` the tokens "`source`", "`,`", and "`target`" are identified); and (iii) define the exception caught in a `catch` statement (*e.g.,* in `catch(IOException io)` we identify `IOException io` as the involved tokens). For m this results in a set S={T_1, T_2, ..., T_n}, where T_i represents a set of relevant tokens for one of the previously mentioned constructs (*e.g.,* T_i is the set of tokens used to define the `for` loop condition).

Given m, we generate $|S|$ versions of it, each one having one of the subject constructs masked. Also, in this case we set the maximum number of masked tokens to 10. This means that if a construct requires more than 10 tokens to be masked (this happened for 9.38% of the constructs in our dataset), it is not masked in our dataset.

The code completion tasks simulated by the construct masking resemble cases in which the developer uses the technique to get recommendations about non-trivial code tokens, representing decision points in the program flow (*e.g.,* condition of `if` statement) or error-handling cases (*e.g.,* exceptions to catch).

Block masking. We use srcML to identify in each method m its code blocks. We define a code block as the code enclosed between two curly brackets. For example, a block may be, besides the method body itself, the code executed in a `for/while` loop, when an `if/else/else if` condition is satisfied, etc. Then, given k the number of blocks identified in m, we create k versions of m each one having a specific code block masked. We set the maximum size of the masked block to two complete statements. This means that if a block is composed of more than two statements (which happened for 49.29% of the blocks in our dataset), it is not masked. This is the most challenging code completion scenario in which we test the experimented techniques. If successful in this task, code completion techniques could substantially speed up code implementation activities.

In summary, there are six fine-tuning datasets: For each of the two domains (Java or Android), there are three different masking levels (token, construct, block). These masking levels have been pick to simulate code completion tasks having different complexity (with *token masking* expected to be the simplest and *block-masking* the most complex).

Starting from the six datasets, we created the training, evaluation, and test sets in Table 1. As a first step, we filtered out specific instances from our datasets. First, when using generative deep learning models, the variability in length of the sentences (in our case, methods) provided as input can affect the training and performance of the model, even when techniques such as padding are employed. For this reason, we analyzed the distribution of methods length in our dataset, finding that two-thirds of them are composed of at most 100 tokens. For this reason, as done by Tufano *et al.* [80], we excluded from our datasets all the methods having more than 100 tokens. Second, RoBERTa cannot efficiently handle cases in which the *masked* tokens are more than the *non-masked* tokens. This often happens, for example, when masking the entire method body in the block-level masking approach. Thus, those instances are excluded as well.

TABLE 1
Study datasets. One instance corresponds to a method with masked token(s).

Domain	Masking Level	Dataset	#Instances	#Tokens
Java	Token	Training	750k	46.9M
		Evaluation	215k	13.4M
		Test	219k	13.6M
	Construct	Training	750k	48.2M
		Evaluation	104k	6.7M
		Test	106k	6.7M
	Block	Training	298k	19.1M
		Evaluation	39k	2.5M
		Test	40k	2.5M
Android	Token	Training	750k	47.4M
		Evaluation	198k	12.5M
		Test	201k	12.6M
	Construct	Training	750k	48.9M
		Evaluation	99k	6.4M
		Test	101k	6.5M
	Block	Training	205k	13.4M
		Evaluation	27k	1.7M
		Test	27k	1.8M

After the filtering steps, we split each of the six datasets into training (80%), evaluation (10%), and test (10%) sets. While the methods in the dataset are randomly ordered, the splitting we performed was not random to avoid biasing the learning. To explain this point, let us consider the case of the *block masking* dataset. Given a method m having k blocks in it, we add in the dataset k versions of m, each having one and only one block masked. Suppose that m contains two blocks b_1 and b_2, thus leading to two versions of m: One in which b_1 is masked (m_{b_1}) and b_2 is not and *vice versa* (m_{b_2}). With a random splitting, it could happen that m_{b_1} is assigned to the training set and m_{b_2} to the test set. However, in m_{b_1} the b_2 is not masked. Thus, when the model has to guess the tokens masked in m_{b_2} it would have the solution in the training set, resulting in boosted prediction performance. For this reason, we randomly select 80% of the methods in each dataset and assign all of their masked versions to the training set. Then, we proceed in the same way with evaluation and test sets.

Using this procedure, we obtained the datasets in Table 1. Important to note is that, given the original size of the datasets using token-level and construct-level masking, we decided to cap the training set to 750k instances (no changes were done in the evaluation and test sets). This was necessary given the computationally expensive process of training several DL models (as it will be clear later, our study required the training of 19 different DL-based models). Also, the size of the evaluation and test sets is slightly different since, as explained before, we split the dataset based on the methods (not on their masked versions) and each method can result

in a different number of its generated masked versions.

2.1.2 Pre-training dataset

To build the pre-training dataset, we used the GitHub Search platform [23] to identify all Java repositories having at least 100 commits, 10 contributors, and 10 stars. These filtering criteria only aim at reducing the likelihood of selecting toy and personal projects for the building of this dataset. We sorted the projects by their number of stars, cloning the top 6,000 and extracting from each of them the methods in the latest snapshot tagged as a release, to only rely on methods likely to be syntactically correct. Repositories for which no snapshot was tagged as a release were excluded, leaving 3,175 repositories. Finally, since we wanted to avoid extremely large projects to influence the dataset too much (*i.e.,* to contribute too many methods to the dataset), we cap the maximum number of methods to extract from each repository to 1,500. This was also due to limit the number of the pre-training instances to a manageable size according to our available hardware resources. In addition to the filters used while building the fine-tuning dataset (see Section 2.1.1), we also removed test methods identified as all those using the *@test* annotation or containing the word "test" in the method name after camel case splitting (*i.e.,* we do not exclude *updateStatus*). Also, since the goal of the pre-training dataset is to provide instances in which random tokens are masked to make the model "familiar" with a specific context (*i.e.,* the Java language in our case), we excluded very short methods (< 15 tokens) not having enough elements to mask and, for the same reasons explained for the fine-tuning dataset, long methods (in this case, > 200 tokens).

We then removed all the exact duplicates within the pre-training dataset, keeping in the dataset only the first occurrence of each duplicate. After having removed the duplicates, the dataset contained 1,874,338 different methods. Finally, we ensured that the pre-training dataset does not contain any methods belonging to the fine-tuning dataset (neither in the training, evaluation, or test sets). We found a total of 23,977 duplicates between the pre-training and the fine-tuning datasets, leading to a final number of 1,850,361 instances in the *pre-training* dataset.

2.2 Context Selection: Techniques

In this section we overview three experimented techniques, *i.e.,*RoBERTa [55], T5 [68], and n-gram [36]. We refer to the original papers presenting them for additional details.

2.2.1 RoBERTa

The first Transformer-based model leverages the off-the-shelf RoBERTa model, which is an Encoder-Transformer architecture. Details about the RoBERTa model are provided in a report by Liu *et al.* [55], while here, we mainly focus on explaining why it represents a suitable choice for code completion. BERT-based models, such as RoBERTa, use a special pre-training where random words in the input sentence are masked out with a special `<MASK>` token. This pre-training task is very well-suited to simulate a code completion task, in which the input is an incomplete code snippet the developer is writing and the masked tokens represent the code needed to autocomplete the snippet. However, one limitation of such a pre-training is that when attempting to predict multiple tokens, *e.g.,* an entire masked *if* condition, it requires the number of tokens to generate to be known, due to the fixed sequence length of Transformers [81]. To overcome this issue, we modify such an objective by masking spans of tokens using a single `<MASK>` token.

As previously explained, BERT models (such as RoBERTa) can be pre-trained and fine-tuned on several tasks [24]. The result will be a single model able to support different tasks and, possibly, taking advantage of what it learned for a specific task to also improve its performance in a different task. In our study, we start by comparing the RoBERTa and the T5 models in a scenario in which no pre-training is performed and a single model is built for each of the three code completion tasks previously described (*i.e.,* token, construct, and block masking) by using the *fine-tuning dataset*. Then, for the best performing model among the two (*i.e.,* T5), we also experiment with pre-training and multi-task fine-tuning. We trained six RoBERTa models, one for each dataset in Table 1.

As for the implementation of the RoBERTa model, we used the one provided in the Python *transformers* library [86]. We also train a tokenizer for each model to overcome the *out-of-vocabulary problem*. The *out-of-vocabulary problem* happens when a machine learning model deals with terms that were not part of the training set but appear in the test set. We trained a Byte Pair Encoding (BPE) [27] model using the HuggingFace's *tokenizers* Python library [2]. BPE uses bytes as vocabulary, allowing it to tokenize every text without requiring the unknown token often used in applications of DL to NLP, thus overcoming the *out-of-vocabulary problem*. When used on source code [46], BPE has been shown to address the *out-of-vocabulary problem*.

2.2.2 T5

Raffel *et al.* [68] presented the T5 model that leverages multi-task learning to implement *transfer learning* in the NLP domain. The T5 has been presented in five pre-defined variants [68]: small, base, large, 3 Billion, and 11 Billion that differ in complexity, size, and, as a consequence, training time. $T5_{small}$, the smaller variant, has 60 million parameters while $T5_{11B}$, the largest, has 11 billion parameters. Despite Raffel *et al.* [68] report that highlights the largest model offers the best accuracy, its training time is sometimes too high to justify its use. Given our computational resources, we opted for the $T5_{small}$ model; therefore, we expect that our results represent a lower bound for the performance of a T5-based model.

T5 offers two advantages as compared to other DL models: (i) it is usually more efficient than RNNs since it allows to compute the output layers in parallel, and (ii) it can detect hidden and long-ranged dependencies among tokens, without assuming that nearest tokens are more related than distant ones. The latter is particularly relevant in code-related tasks. For example, a local variable could be declared at the beginning of a method (first statement), used in the body inside an `if` statement, and finally returned in the last method's statement. Capturing the dependency existing between these three statements, that might even be quite far from each other (*e.g.,* variable declaration and return statement), can help in better modeling the source code with

a consequent boost of performance for supporting code-related tasks.

For additional details about the T5 architecture, we refer the reader to the original work presenting this model [68].

2.2.3 n-gram

As a baseline for comparison, we used the widely studied statistical language models based on n-gram. An n-gram model can predict a single token following the $n-1$ tokens preceding it. Even though the n-gram model is meant to predict a single token given the $n-1$ preceding tokens, we designed a fair comparison for the scenario in which we mask more than one token. In particular, we use the n-gram model in the following way: Let us assume that we are predicting, using a 3-gram model, how to complete a statement having five tokens T, of which the last two are masked (M): <T_1, T_2, T_3, M_4, M_5>, with M_4 and M_5 masking T_4 and T_5, respectively. We provide as input to the model T_2 and T_3 to predict M_4, obtaining the model prediction P_4. Then, we use T_3 and T_4 to predict M_5, thus obtaining the predicted sentence <T_1, T_2, T_3, P_4, P_5>. Basically, all predictions are joined to predict multiple contiguous tokens.

The n-gram models are trained on the same training sets used for the fine-tuning of the DL models without, however, masked tokens. We experiment with both the standard n-gram model (*i.e.*, the one discussed above) as well as, in a smaller study, with the n-gram cached model proposed by Hellendoorn and Devanbu [36].

3 Data Collection and Analysis

In this section we detail the data collection and analysis procedure adopted to answer the research questions described in Section 2.

3.1 Training of Models

We detail the process used for the training and hyperparameters tuning of the two deep learning models that we experimented with.

TABLE 2
Hyperparameters Tuned for the RoBERTa Models.

Hyperparameter	Experimented Values	Best
Learning rate	{$5e^{-5}$, $3e^{-5}$, $2e^{-5}$}	$5e^{-5}$
Batch size	{16, 32, 64}	64
# Hidden Layers	{6, 12, 16}	12
# Attention Heads	{6, 12, 16}	16
Hidden Layer Size	{256, 512, 768, 1024}	768
Intermediete Size	{3072, 4096}	4,096

3.1.1 RoBERTa

We performed hyperparameter tuning using the Weights & Biases's [5] Python library on a Linux server with an Nvidia RTX Titan GPU. Table 2 reports the hyperparameters we tuned, the range of values we tested for them, and the value in the best configuration we found. Besides those parameters, we used an attention dropout probability of 0.1, and a hidden layer dropout probability of 0.3. For the tokenizer, the vocabulary size was set to 50k. The hyperparameter search was performed using the training and the evaluation sets of the Android dataset with token masking. We picked as the best configuration the one that, when applied to the evaluation set, was able to obtain the highest number of "perfect predictions". We define as "perfect" a prediction that exactly matches the code written by the developers. Thus, the model correctly guesses *all* masked tokens. If one of the masked tokens is different we do not consider the prediction as "perfect". While, in principle, a different hyperparameter tuning would be necessary for each dataset, such a process is extremely expensive, and preliminary investigations we performed on a subset of the other datasets showed minor differences in the achieved best configuration.

The training was performed across servers using their GPUs. The first was equipped with an Nvidia Tesla V100S, the second with an Nvidia RTX Titan, and the third with 3 Nvidia GTX 1080Ti. The training time strongly depends on the size of the dataset and the used server but ranged between 28 and 114 hours per model. Note that, once trained, each model can be used to perform predictions in the split of a second (on average, 0.12 seconds on a laptop CPU), thus making them a viable solution for "real-time" code completion.

We train each model for a maximum of 50 epochs. However, we adopted the following stopping condition. At the end of each training epoch, we executed the model on the evaluation set and we compute the number of perfect predictions. If we observe that, during the training, the performance of the model is worsening in terms of perfect predictions on the evaluation set (*i.e.*, the model is likely overfitting to the training set), we stop the training. In particular, given a model trained for n^{th} epoch, we stop the training if the number of perfect predictions on the evaluation set is lower than the number of perfect predictions achieved after the $n-4$ epoch. This ensures that the models can have some fluctuations in performance for up to three epochs. Then, if it is still not improving, we stop its training and take the best model (in terms of perfect predictions on the evaluation test) obtained up to that moment. None of the models were trained for the whole 50 epochs.

TABLE 3
Hyperparameters Tuned for the T5 Models.

Learning Rate Type	Parameters
Constant (C-LR)	$LR = 0.001$
Slanted Triangular (ST-LR)	$LR_{starting} = 0.001$ $LR_{max} = 0.01$ $Ratio = 32$ $Cut = 0.1$
Inverse Square Root (ISQ-LR)	$LR_{starting} = 0.01$ $Warmup = 10,000$
Polynomial Decay (PD-LR)	$LR_{starting} = 0.01$ $LR_{end} = 1e{-}06$ $Power = 0.5$

3.1.2 T5

We rely on the same configurations used by Mastropaolo *et al.* [59]. In particular, concerning the pre-training, we do not tune the hyperparameters of the T5 model because the pre-training step is task-agnostic, and this would provide limited benefits. Instead, we experiment with four different

learning rate schedules for the fine-tuning phase, using the configurations reported in Table 3, and identify the best-performing configuration in terms of perfect predictions on the evaluation sets. Each of the four experimented configurations has been trained for 100k steps (∼7 epochs) before assessing its performance on the evaluation sets. Across all six evaluation datasets (Table 1), the best performing configuration was the one using the Slanted Triangular learning rate, confirming the findings in [59]. Also, all T5 models we built use a *SentencePiece* [50] tokenizer trained on the pre-training dataset and are composed of 32k word pieces [59].

The best configuration we identified has been used to train six different T5 models (*i.e.,* one for each dataset in Table 1) and assess their performance on the corresponding test set. These results can be used to compare directly the T5 and the RoBERTa model when fine-tuned without pre-training and in a single-task setting (*i.e.,* no transfer learning). Since we found the T5 to perform better than RoBERTa, we also use this model to answer RQ_2. Thus, in addition to these six models, we also built additional seven models: six of them leverage pre-training plus single-task fine-tuning. In other words, they are the equivalent of the first six models we built, with the addition of a pre-training phase.

For pre-training the T5 model, we randomly mask 15% of the tokens in each instance (method) of the *pre-training dataset*. The pre-training has been performed for 200k steps (∼28 epochs), since we did not observe any improvement going further. We used a 2x2 TPU topology (8 cores) from Google Colab to train the model with a batch size of 256, with a sequence length (for both inputs and targets) of 256 tokens. As a learning rate, we use the *Inverse Square Root* with the canonical configuration [68]. The training requires around 26 seconds for 100 steps.

Finally, we created a T5 model exploiting both pre-training and multi-task fine-tuning (*i.e.,* a single model was first pre-trained, and then fine-tuned on all six datasets in Table 1). This was done to check the impact of transfer learning on the model performance. Overall, we trained 13 T5 models: six with no pre-training and single-task fine-tuning, six with pre-training and single-task fine-tuning, and one with pre-training and multi-task fine-tuning.

3.2 Analysis of Results

To answer RQ_1 we compute the metrics summarized in Table 4 by running each trained model on the test sets in Table 1.

The first metric, *Bilingual Evaluation Understudy (BLEU)-n score*, assesses the quality of automatically translated text [25]. The BLEU score computes the weighted percentage (*i.e.,* considering the number of occurrences) of words appearing in translated text and the reference text. We use four variants of BLEU, namely BLEU-1, BLEU-2, BLEU-3, and BLEU-4. A BLEU-n variant computes the BLEU score by considering the n-grams in the generated text. Most of the previous work in the SE literature adopts the BLEU-4 score [31], [43], [82]. However, such a variant cannot be computed when the target prediction (in our case, the number of masked tokens) is lower than four. For this reason, we compute four different versions from BLEU-1 to BLEU-4. BLEU-1 can be computed for all predictions, while BLEU-n with n>1 only for predictions having a length (*i.e.,* number of tokens) higher or equal than n. The BLEU score ranges between 0% and 100%, with 100% indicating, in our case, that the code generated for the masked tokens is identical to the reference one.

The Levenshtein distance [51]. To provide a proxy measure of the effort needed by developers to convert a prediction generated by the model into the reference (correct) code, we compute the Levenshtein distance at token-level: This can be defined as the minimum number of token edits (insertions, deletions or substitutions) needed to transform the predicted code into the reference one. Since such a measure is not normalized, it is difficult to interpret it in our context. Indeed, saying that five tokens must be changed to obtain the reference code tells little without knowing the number of tokens in the reference code. For this reason, we normalize such a value by dividing it by the number of tokens in the longest sequence among the predicted and the reference code.

The percentage of perfect predictions tells us about the cases in which the experimented model can recommend the very same sequence of tokens which were masked in the target code.

We statistically compare the results achieved by RoBERTa and T5 using different statistical analyses. We assume a significance level of 95%. As explained below, we use both tests on proportions and non-parametric tests for numerical variables; parametric tests cannot be used because all our results in terms of BLEU score or Levenshtein distance deviate from normality, according to the Anderson-Darling test [6] (p-values<0.001). Whenever an analysis requires running multiple test instances, we adjust p-values using the Benjamini-Hochberg procedure [87].

To (pairwise) compare the perfect predictions of RoBERTa and T5, we use the McNemar's test [61], which is a proportion test suitable to pairwise compare dichotomous results of two different treatments. To compute the test results, we create a confusion matrix counting the number of cases in which (i) both T5 and RoBERTa provide a perfect prediction, (ii) only T5 provides a perfect prediction, (iii) only RoBERTa provides a perfect prediction, and (iv) neither T5 nor RoBERTa provides a perfect prediction. Finally, we complement the McNemar's test with the Odds Ratio (OR) effect size.

The comparison between different datasets, aimed at addressing $RQ_{1.2}$, is performed, again, through a proportion test, but this time, being the analysis unpaired (*i.e.,* we are comparing results over two different datasets), we use Fischer's exact test (and related OR) on a matrix containing, for different approaches and for different masking levels, the number of correct and incorrect predictions achieved on Java and Android.

To compare results of T5 and RoBERTa in terms of BLEU-n score and Levenshtein distance, we use the Wilcoxon signed-rank test [85] and the paired Cliff's delta [30] effect size. Similarly, the comparison between datasets in terms of BLUE-n score and Levenshtein distance, being unpaired, is performed using the Wilcoxon rank-sum test [85] and the unpaired Cliff's delta effect size.

For the T5, we also statistically compare the performance

TABLE 4
Summary of the evaluation metrics used in our study

Metric	Purpose
BLEU score	Overall prediction quality for different prediction lengths
Levenshtein distance	Proxy of the effort needed to adapt the prediction
% of perfect predictions	To what extent is the approach able to generate predictions that need human intervention

achieved (i) with/without pre-training, and (ii) with/without transfer learning. Also in this case, McNemar's test is used to compare perfect predictions.

Finally, we take the best performing model (*i.e.,* T5 with pre-training and multi-task fine-tuning) and we check whether the *confidence* of the predictions can be used as a reliable proxy for the "quality" of the predictions. If this is the case, this means that in a recommender system built around the trained model, the developer could decide to receive recommendations only when their confidence is higher than a specific threshold. T5 returns a score for each prediction, ranging from minus infinity to 0. This score is the log-likelihood (ln) of the prediction. Thus, if it is 0, it means that the likelihood of the prediction is 1 (*i.e.,* the maximum confidence, since $ln(1) = 0$), while when it goes towards minus infinity, the confidence tends to 0.

We split the predictions performed by the model into ten intervals, based on their confidence c going from 0.0 to 1.0 at steps of 0.1 (*i.e.,* first interval includes all predictions having a confidence c with $0 \leq c < 0.1$, last interval has $0.9 \leq c$). Then, we report for each interval the percentage of perfect predictions.

To corroborate our results with a statistical analysis, we report the OR obtained by building a logistic regression model relating the confidence (independent variable) with the extent to which the prediction achieved a perfect prediction (dependent variable). Given the independent variable estimate β_i in the logistic regression model, the OR is given by e^{β_i}, and it indicates the odds increase corresponding to a unit increase of the independent variable. We also determine the extent to which the confidence reported by the model correlates with the number of masked tokens. To this extent, we use the Kendall's correlation [48], which does not suffer from the presence of ties (occurring in our dataset) as other non-parametric correlations.

To address RQ_3, for all the datasets, we compare the performance of the DL-based models with that of an n-gram model. In particular, we perform a first large-scale comparison using a standard n-gram language model and, on a smaller dataset, we also compare the experimented techniques with the state-of-the-art cached n-gram model [36] using the implementation made available by the authors [3]. We detail later why the cached n-gram model was too expensive to run on the entire dataset.

We tried to design a fair comparison, although the n-gram model is designed to predict a single token given the n tokens preceding it. Thus, in a scenario in which we mask more than one token, we use the n-gram model in the following way: We run it to predict each masked token in isolation. Then, we join all predictions to generate the final string (*i.e.,* the set of previously masked tokens). The n-gram models are trained on the same training sets used for the fine-tuning of the DL-based models without, however, masked tokens. We compare the three approaches in terms of perfect predictions generated on the test sets. A statistical comparison is performed using the McNemar's test [61] and ORs.

4 Results Discussion

We start by contrasting the performances of T5 and RoBERTa (Section 4.1). Then, we show how the n-gram model compares with the DL-based ones (Section 4.2). Finally, Section 4.3 presents qualitative examples of correct predictions made by the models and discusses the semantic equivalence of non-perfect predictions.

Note that, upon interpreting the achieved results, and especially those concerning the perfect (correct) predictions, a direct comparison with the results achieved in previous works on code completion is not possible. This is because most of the studies in the literature experiment with code completion models when predicting a single next token the developer is likely to write. As we will show, in such a specific scenario the models we experiment with can achieve extremely high accuracy ($> 95\%$ of correct predictions). However, their performance strongly decreases when predicting longer sequences composed of multiple tokens or even multiple statements.

4.1 DL-based models performance comparison (RQ_1)

Fig. 1 depicts the results achieved by DL-based models in terms of perfect predictions for different masking approaches, namely (from left to right) *token-masking*, *construct-masking*, and *block-masking*. The plots show the percentage of perfect predictions (y axis) by the number of masked tokens (x axis). For example, in the *token masking* scenario we randomly mask, for each source code line l having more than one token, its last x tokens, where x is a random number between $1 \ldots n-1$, with n being the number of tokens of l, and x is capped to a maximum of 10. The results achieved by the T5 are reported in orange while those for RoBERTa in red; continuous lines represent the results achieved on the Java dataset, while the dashed lines are used for the Android dataset.

The left-side graph in Fig. 1 shows the percentage of perfect predictions when we only mask the last token (*i.e.,* one masked token), the last two tokens, etc.. The scale on the x axis is different when dealing with the block masking scenario since here we mask entire blocks thus having, in some cases, dozens of masked tokens. Each point indicates that between $x-5$ and x tokens were masked, *e.g.,* for the first data point at most 5 tokens were masked, for the second between 5 and 10, etc..

Table 5 reports the average BLEU score in the four considered variants and the average normalized Levenshtein

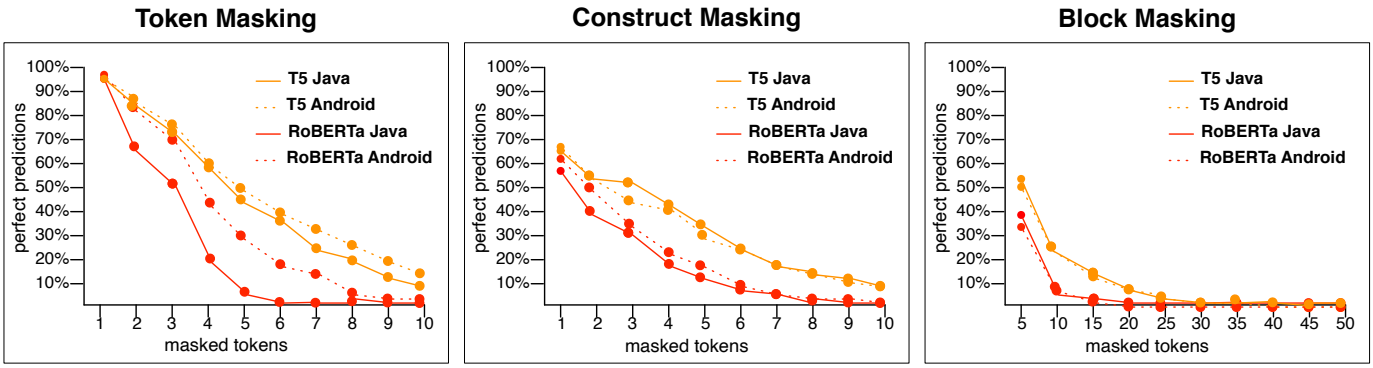

Fig. 1. Percentage of perfect predictions achieved by T5 and RoBERTa

TABLE 5
BLEU score and Levenshtein distance for T5 and RoBERTa.

	Java		Android	
Token masking	**T5**	**RoBERTa**	**T5**	**RoBERTa**
BLEU-1	0.83	0.60	0.85	0.73
BLEU-2	0.73	0.43	0.76	0.61
BLEU-3	0.60	0.23	0.64	0.44
BLEU-4	0.47	0.10	0.51	0.28
Levenshtein	0.16	0.35	0.14	0.24
Construct masking				
BLEU-1	0.68	0.51	0.68	0.57
BLEU-2	0.55	0.34	0.57	0.43
BLEU-3	0.48	0.24	0.49	0.33
BLEU-4	0.37	0.14	0.43	0.26
Levenshtein	0.32	0.48	0.32	0.41
Block masking				
BLEU-1	0.65	0.44	0.62	0.44
BLEU-2	0.57	0.32	0.54	0.31
BLEU-3	0.49	0.21	0.46	0.21
BLEU-4	0.41	0.13	0.38	0.13
Levenshtein	0.35	0.54	0.37	0.55

distance achieved by T5 and RoBERTa. Also in this case the results are grouped based on the masking level and dataset.

The results in Fig. 1 and Table 5 are achieved by the DL-based models in the simplest scenario, *i.e.*, single-task without pretraining. To answer $RQ_{1.3}$ we run additional experiments for the best model (*i.e.*, T5). The results of such experiments are provided in Table 6 as the percentage of perfect predictions for different variants of the T5 model, *i.e.*, with/without pretraining and using single- and multi-task fine-tuning. Table 6 also reports the results achieved with the RoBERTa model in the simplest scenario to simplify the discussion of the results.

4.1.1 Impact of number of masked tokens ($RQ_{1.1}$) and specificity of the dataset ($RQ_{1.2}$)

Three findings immediately emerge from the analysis of Fig. 1: (i) as expected, the higher the number of masked tokens, the lower the performance of the models; (ii) the results achieved on the more specific dataset (*i.e.*, Android, dashed lines in Fig. 1) are substantially better as compared to the ones achieved for Java only in the token-masking scenario with the RoBERTa model (see statistics in Table 9); (iii) the T5 model (orange lines in Fig. 1) substantially outperforms RoBERTa (see statistics in Table 7 and Table 8). Also, the performance of RoBERTa drops more steadily as compared to that of T5 when the number of masked tokens increases.

Table 7 reports results of the McNemar's test and ORs for the comparison between T5 and RoBERTa in terms of their ability to perform perfect predictions. As it can be seen, the (adjusted) p-values always indicate a statistically significant difference, and the ORs indicate that T5 has between 2.94 and 8.87 higher odds to provide a perfect prediction than RoBERTa.

Concerning the comparison of BLEU scores or Levenshtein distances (whose average values are reported in Table 5) between T5 and RoBERTa, statistical results (Wilcoxon signed-rank test adjusted p-values and Cliff's d) are in Table 8. Also in this case, differences are always statistically significant, with varying effect sizes (generally larger for greater levels of BLEU score, and for Java than Android) in favor of T5 (for the Levenshtein distance a negative d is in favor of T5, as it is a distance).

Token masking. The left part of Fig. 1 shows that, as expected, the lower the number of masked tokens the higher the perfect predictions. Not surprisingly, the models are very effective when we only mask the last token in a statement. Indeed, in most cases, this will be a semicolon, a parenthesis, or a curly bracket. Thus, it is easy for the model to guess the last token. When moving to more challenging scenarios like the last five tokens masked in a statement, the percentage of perfect predictions for RoBERTa on the Java dataset drops to less than 10%, a major gap with the T5 model that keeps a percentage of perfect predictions higher than 40%. As for the dataset, both models achieve significantly better performance on the Android dataset (Fisher's test p-value$<$0.001 and OR$<$ 1), which is more specific and, thus, more subject to regularities in the source code. However, the gap in terms of perfect predictions between the Java and the Android dataset is much more marked for the RoBERTa model (*e.g.*, $\sim$20% at $x = 5$ against a $\sim$6% for the T5).

Looking at Table 5, the BLEU scores and the Levenshtein distance confirm what was observed for perfect predictions: performances for the Android dataset are better than for the Java one. According to Wilcoxon rank-sum test, all differences, except for RoBERTa at Block level, are statistically significant, yet with a negligible/small Cliff's d (detailed

TABLE 6
Perfect predictions of T5 models with different fine-tuning strategies, and RoBERTa model

Dataset and Masking Level		T5			RoBERTa
		With Pretraining		No Pretraining	No Pretraining
		Single-task	Multi-task	Single-task	Single-task
Java	Token	62.9%	66.3%	61.0%	38.9%
	Construct	51.2%	53.0%	48.4%	33.4%
	Block	27.2%	28.8%	22.9%	8.7%
Android	Token	64.8%	69.3%	63.8%	51.8%
	Construct	49.3%	50.8%	46.8%	37.4%
	Block	27.5%	29.7%	22.8%	9.4%
Overall		56.2%	59.3%	54.1%	38.7%

TABLE 7
Perfect prediction: Mcnamar's test comparison between T5 and RoBERTa

Dataset	Masking	*p*-value	OR
Java	Token	<0.001	8.87
	Construct	<0.001	4.69
	Block	<0.001	8.14
Android	Token	<0.001	4.47
	Construct	<0.001	2.94
	Block	<0.001	7.61

statistical results are in the online appendix).

Construct masking. In this scenario (see central subgraph in Fig. 1), T5 and RoBERTa achieve respectively above 65% and 55% of perfect predictions when a single token is masked for both datasets. Note that, in this scenario, also a single-token prediction is not trivial since we are in a context in which such a single token represents (i) the complete condition of an `if` statement or a `while/for` loop, or (ii) the parameters in a method call, or (iii) the exception caught in a `catch` statement. When the prediction is represented by a single token, it is usually related to a Boolean used in an `if` condition (*e.g.*,`if(true)`, `if(valid)`, etc.) or the single parameter needed for a method invocation.

Also in this case, a higher number of masked tokens implies lower performance, and again the T5 outperforms RoBERTa significantly for both datasets although the gap is smaller. Finally, as shown in Table 9, while with RoBERTa results for Android are better, for T5 we achieve an OR$\simeq$ 1.

In terms of BLEU score and Levenshtein distance, the achieved values are worse as compared to the token-level masking, confirming the more challenging prediction scenario represented by the construct-level masking. On average, the developer may need to modify ∼40% and ∼30% of the predicted tokens to obtain the reference code (small variations are observed between Java and Android) when using RoBERTa and T5, respectively.

Block masking. This represents the most challenging prediction scenario: The masked part can involve an entire statement or even span over two statements (maximum boundary we set). The performance of T5 and RoBERTa in terms of perfect predictions are respectively above 50% and 35% when dealing with small masked blocks, up to five tokens. These blocks are mostly related to `return` statements representing a code block (*e.g.*, the value to return when an `if` condition is satisfied), such as `{ return false; }`, `{ return null; }`, etc.

For longer blocks, the performance substantially drops.

```
protected void fireScriptEnded(String plugin, Hook hook, Script script)
{ Object[] listeners = _listeners.getListenerList();
for (int i = listeners.length-2; i>=0; i-=2) <MASK> }

{ if (listeners[i]==ScriptListener.class)
{ ((ScriptListener)listeners[i+1]).scriptEnded(plugin, hook, script); } }
```

Fig. 2. Perfect prediction of 36 tokens generated by T5 in the Android dataset

When considering blocks having between six and ten masked tokens, RoBERTa is able to generate a correct prediction in ∼5% of cases, as compared to the ∼25% achieved by the T5. The largest masked block reporting a perfect prediction for the T5 model is composed of 36 and 39 tokens for Android (see Fig. 2) and Java datasets respectively, compared to the 13 and 15 tokens achieved with the RoBERTa model.

At this level (see Table 9), the difference in terms of performance between Java and Android is not so evident, and even insignificant for T5.

As expected, the BLEU scores are the lowest in this scenario (Table 5), and the developer may need to revise, on average, $\sim$ 50% and $\sim$ 35% of the predicted tokens, independently from the dataset of interest, when using RoBERTa and T5, respectively.

Answer to RQ$_{1.1}$: *As the number of masked tokens increases, the DL-based models have a harder time generating correct predictions. Still, the performance achieved by the T5 model looks promising and, as we will discuss later, can be further pushed through proper pretraining and multi-task fine-tuning.*

Answer to RQ$_{1.2}$: *When looking at the best model* (i.e., *the T5), its performance on the two datasets is quite similar, with no major differences observed. A strong difference in performance is only observed in the token-masking scenario with the RoBERTa model.*

4.1.2 Impact of pre-training and transfer learning (RQ$_2$)

As explained in Section 3.1.2, we trained seven additional T5 models to assess the impact of pretraining and transfer learning on its performance. First, we added to the six models for which we previously discussed the T5 performance (*i.e.*, no pretraining, single-task) the pretraining phase (obtaining a pre-trained model in the single-task scenario, *i.e.*, no transfer learning). Then, we take the pre-trained model, and fine-tuned it in a multi-task setting, investigating the impact of transfer learning.

Table 6 shows the achieved results also reporting the performance of the previously discussed T5 and RoBERTa models (*i.e.*, no pretraining, single-task in Table 6). Results

TABLE 8
BLEU score and Levensthein distance comparison between T5 and RoBERTa: Wilcoxon signed-rank and Cliff's delta (N: negligible, S: small, M: medium, L: large)

Dataset	Masking	BLEU 1		BLEU 2		BLEU 3		BLEU 4		Levenshtein	
		p-value	*d*	*p*-value	*d*	*p*-value	*d*	*p*-value	*d*	*p*value	*d*
Java	Token	<0.001	0.33 (S)	<0.001	0.41 (M)	<0.001	0.51 (L)	<0.001	0.62 (L)	<0.001	-0.32 (S)
	Construct	<0.001	0.22 (S)	<0.001	0.30 (S)	<0.001	0.32 (S)	<0.001	0.35 (M)	<0.001	-0.21 (S)
	Block	<0.001	0.39 (M)	<0.001	0.43 (M)	<0.001	0.47 (M)	<0.001	0.49 (L)	<0.001	-0.38 (M)
Android	Token	<0.001	0.17 (S)	<0.001	0.21 (S)	<0.001	0.27 (S)	<0.001	0.34 (M)	<0.001	-0.17 (S)
	Construct	<0.001	0.14 (N)	<0.001	0.20 (S)	<0.001	0.22 (S)	<0.001	0.27 (S)	<0.001	-0.14 (N)
	Block	<0.001	0.33 (M)	<0.001	0.39 (M)	<0.001	0.42 (M)	<0.001	0.44 (M)	<0.001	-0.34 (M)

TABLE 9
Comparison between different datasets for perfect predictions - results of Fisher's exact test (OR<1 indicate better performances for Android)

Masking	Method	*p*-value	OR
Token	T5	<0.001	0.89
	RoBERTa	<0.001	0.59
Construct	T5	<0.001	1.07
	RoBERTa	<0.001	0.84
Block	T5	0.67	1.01
	RoBERTa	0.01	0.93

TABLE 10
Effect of different pretraining levels for T5: McNemar's test results. None indicates the T5 model with no pre-training and single-task finetuning. Single and Multi indicates the pre-trained model with single- and multi-task fine-tuning, respectively.

Dataset	Masking	Comparison	*p*-value	OR
Java	Token	single vs. none	<0.001	1.44
		multi vs. single	<0.001	1.81
		multi vs none	<0.001	2.33
	Construct	single vs. none	<0.001	1.61
		multi vs. single	<0.001	1.34
		multi vs none	<0.001	1.92
	Block	single vs. none	<0.001	2.19
		multi vs. single	<0.001	1.32
		multi vs none	<0.001	2.32
Android	Token	single vs. none	<0.001	1.23
		multi vs. single	<0.001	2.27
		multi vs none	<0.001	2.61
	Construct	single vs. none	<0.001	1.58
		multi vs. single	<0.001	1.28
		multi vs none	<0.001	1.81
	Block	single vs. none	<0.001	2.14
		multi vs. single	<0.001	1.39
		multi vs none	<0.001	2.39

of a statistical comparison made using McNemar's test are reported in Table 10. As it is shown, the pretraining has a positive (OR> 1) and statistically significant effect in all cases, and the fine-tuning in a multi-task setting outperforms the single-task pretraining. Looking at Table 6, the pretraining had a positive impact on the accuracy of T5, boosting the percentage of perfect predictions from 1% to 4.7%, depending on the test dataset. The benefit of pretraining is more evident in the most challenging block-level scenario (∼5%). Overall, when considering all test datasets as a whole, the percentage of perfect predictions increases from 54.1% to 56.2% (+2.1%).

By training a single model on the six training datasets, the percentage of perfect predictions further increases, going up to an overall 59.3%. Note that improvements can be observed on all test datasets and, for the token-masking scenario, they can reach ∼5%.

The performance improvement is also confirmed by the

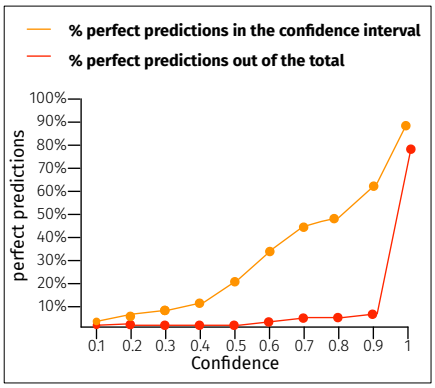

Fig. 3. Perfect predictions by the confidence of the model

results achieved in terms of BLEU score and the Levenshtein distance that, for the sake of brevity, we report in our replication package [21].

Answer to RQ$_2$: *We found both pretraining and multi-task fine-tuning to have a positive impact on the T5 performance. Overall, such an improvement accounts for +5.2% in terms of perfect predictions (36,009 additional instances correctly predicted).*

4.1.3 T5 Confidence Level

The T5 returns a *score* for each prediction, ranging from minus infinity to 0. This score is the log-likelihood of the prediction itself. If the score is -2 then it means that the log-likelihood of the prediction is -2. Hence, the likelihood is 0.14 ($ln(x) = -2 \implies x = 0.14$) and this implies that the model has a confidence of 14% for the prediction to be correct. If the score is 0, repeating the same computation as above, the model has the confidence of 100% about the prediction itself.

Fig. 3 reports the relationship between the percentage of perfect predictions and the confidence of the model. The orange line shows the percentage of perfect predictions within each confidence interval (*e.g.*, 90% of predictions having a confidence higher than 0.9 are correct), while the red line reports the percentage of perfect predictions that are due to predictions in that confidence interval out of the total (*e.g.*, 78% of all perfect predictions have a confidence higher than 0.9).

Fig. 3 shows a strong relationship between the confidence of the model and the correctness of the prediction. While this result might look minor, it has an important implication: It would be possible to build a reliable code completion tool around the T5 model. Indeed, the tool could be configured to only trigger recommendations when the confidence of the prediction is higher than a given threshold (*e.g.*, 0.9). This would result in an extremely high precision.

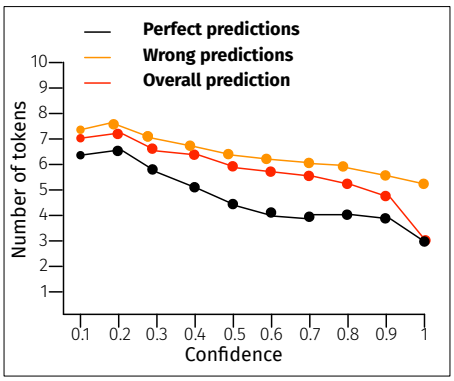

Fig. 4. Average length (in tokens) of the predictions by confidence

From a statistical perspective, a logistic regression model correlating the confidence level and the perfect prediction outcome indicates a statistically significant (p-value $<$0.001) correlation, and an estimate of 6.58, which means 720 higher odds of a perfect prediction for each unit increase of the confidence, *i.e.*, 72 higher odds of a perfect prediction for a 0.1 increase of the confidence, *i.e.*, a tick on the x-axis of Fig. 3.

Fig. 4 analyzes the average length, in tokens, of the perfect predictions (yellow line), wrong predictions (orange line), and for all the predictions (red line) among all confidence intervals. It is clear that the length of the prediction is related to the confidence, since the model has higher confidence for shorter predictions. Indeed, the average number of tokens in perfect predictions for the highest confidence interval (*i.e.*, 3 tokens) is much lower than the average number of tokens in perfect predictions for the lowest confidence interval (*i.e.*, 6 tokens). This confirms previous findings showing that the model is more likely to correctly predict shorter statements.

From a statistical perspective, this is confirmed by a significant (p-value $<$0.001), negative, and moderate Kendall's correlation (τ=-0.36).

TABLE 11
Perfect predictions of the three models

Dataset	Masking Level	T5	RoBERTa	n-gram
Java	Token	61.0%	38.9%	30.4%
	Construct	48.8%	33.9%	12.5%
	Block	22.9%	8.7%	4.6%
Android	Token	63.8%	51.9%	35.4%
	Construct	47.1%	37.8%	17.6%
	Block	22.8%	9.4%	6.6%
Overall		54.3%	38.8	24.9%

4.2 Comparison with an n-gram Model

We answer RQ_3 by comparing the DL-based models without pretraining and in the single-task setting to the n-gram model. We opted for this comparison for the sake of fairness, since in this way the n-gram model has been trained on exactly the same dataset as the two DL-based models.

Table 11 reports the comparison in terms of perfect predictions between T5, RoBERTa and the n-gram model in different evaluation scenarios, as well as the overall results.

TABLE 12
Comparison with the n-grams model: results of McNemar's test

Dataset	Masking	Comparison	p-value	OR
Java	Token	T5 vs. RoBERTa	<0.001	8.93
		RoBERTa vs. n-grams	<0.001	2.21
		T5 vs. n-grams	<0.001	10.31
	Construct	T5 vs. RoBERTa	<0.001	4.65
		RoBERTa vs. n-grams	<0.001	5.29
		T5 vs. n-grams	<0.001	11.62
	Block	T5 vs. RoBERTa	<0.001	8.15
		RoBERTa vs. n-grams	<0.001	2.85
		T5 vs. n-grams	<0.001	14.38
Android	Token	T5 vs. RoBERTa	<0.001	4.47
		RoBERTa vs. n-grams	<0.001	4.26
		T5 vs. n-grams	<0.001	10.14
	Construct	T5 vs. RoBERTa	<0.001	2.91
		RoBERTa vs. n-grams	<0.001	5.30
		T5 vs. n-grams	<0.001	9.04
	Block	T5 vs. RoBERTa	<0.001	7.62
		RoBERTa vs. n-grams	<0.001	1.90
		T5 vs. n-grams	<0.001	10.00

For example, T5 produced 61% perfect predictions on the Java dataset when using token masking. Results of statistical tests (McNemar's test) are in Table 12.

One important clarification is needed to properly interpret the results of Table 11. Since the n-gram model uses a different script to tokenize the code, we excluded from the test sets cases in which the tokens to predict (*i.e.*, the masked ones) are tokenized differently between the DL-based approaches and n-gram one (*e.g.*, one identifies 4 tokens and the other one 5). This resulted in the exclusions of a few hundred instances from each test set and explains the slightly different performances reported for T5 and RoBERTa between Table 11 and Fig. 1.

Table 12 reports results of the statistical comparison among the three models, using McNemar's test. DL-based models achieve better performance in all experimented datasets, and McNemar's tests always indicate statistically significant differences, with ORs ranging between 1.90 (RoBERTa vs n-grams, block masking for Android) and 14.38 (block masking, T5 vs n-grams for Java).

In the token masking scenario, the performance of the n-gram model is very competitive when compared with RoBERTa, while the T5 performs substantially better. When masking specific constructs, the gap in performance becomes stronger (see Table 11) with a substantial gap, especially between T5 and n-gram. Finally, in the block masking experiment, RoBERTa and n-gram techniques struggle to obtain a high percentage of perfect predictions, with the T5 performing better achieving more than twice the number of perfect predictions as compared to the competitive techniques.

While the DL-based models showed superior performance, there are two important aspects to consider. First, the n-gram model allows for faster training. We estimate four to five times less training time needed for the n-gram model as compared to the DL-based models. We do not report precise data since such a study would require executing the training many times on the same machine, and such an analysis is out of the scope of this work. Once trained all models can generate predictions in fractions of a second. Second, the comparison presented as of now concerns the standard

n-gram model. However, we also experimented with the cached n-gram model [36], which can leverage information about other code components coming from the same project (*e.g.,* same file or package [36]) of the method in which the prediction is performed. This is one of the advantages of the cache model [36] and, in a real scenario, it should be possible to use this information assuming that the method on which the prediction is performed is not the first one written in the whole system. However, such experimentation is quite expensive to perform since it requires the cloning of the whole repositories hosting every test method. This is why it has only been performed on a small sample of our dataset.

TABLE 13
Perfect predictions of n-gram model when providing the cloned repository (WC) vs. when not providing (NC). In comparison to DL-based models (200 methods)

Dataset and Masking Level		T5	RoBERTa	n-gram	
				NC	WC
Java	Token	65.5%	42.2%	32.5%	43.9%
	Construct	56.0%	38.0%	14.5%	20.5%
	Block	25.8%	8.5%	5.2%	8.5%
Android	Token	69.9%	50.9%	35%	42.2%
	Construct	52.8%	37.8%	13.9%	22.0%
	Block	33.6%	13.0%	9%	11.9%
Overall		57.7%	38.2%	23.9%	31.5%

For a given method m_t in the test set, we clone its repository and check if the source code of m_t in the latest system snapshot is exactly the same as in the test set. If this is the case, we run the prediction on m_t providing the cloned repository as a test folder, in such a way that it is leveraged by the cache model (this is done through the implementation of Hellendoorn *et al.* [36]). If the method changed, we discard it and move to the next one. Since such a process is very expensive, we collected 200 methods from each test set, and we compare the performance of the n-gram model when such additional information is provided (and not) on these instances.

Table 13 reports the achieved results. As expected, the performance of the n-gram model increase thanks to the use of the information in the test project. On these same instances, the performance of T5 and RoBERTa models are always superior but in the case of Java token and block masking for RoBERTa.

Answer to RQ$_3$: *The n-gram model is a competitive alternative to RoBERTa, while the T5 confirms its superior performance. It is worth highlighting the much cheaper cost of training (and possibly re-training several times) an n-gram model as compared to a DL-based approach.*

4.3 Qualitative Results

To give a better idea to the reader about the capabilities of the experimented models in supporting code completion, we report in Fig. 5 examples of correct predictions for the T5 model in different scenarios/datasets. Examples of predictions for the RoBERTa and n-gram model are available in the replication package [21].

Given the achieved results showing the superiority of the T5 model, we had a better look at a sample of the

Android Token

```
public SongViewHolder(View itemView) { super(itemView); albumSongNameTextView =
(TextView) itemView <MASK> }
.findViewById(R.id.albumSongNameTextView);
```

Java Token

```
public Sheet getSheet() { if (sheet != null) { return sheet; } UIComponent
parent = getParent(); while (parent != <MASK> parent = parent.getParent(); }
return (Sheet) parent; }
null && !(parent instanceof Sheet)) {
```

Android Construct

```
public static int i(String tag, String msg) { _log( <MASK>); return
Log.i(tag, msg); }
_prioToLevel(Log.INFO), tag, msg
```

Java Construct

```
public DoubleToLongFunction mask(ThrowingDoubleToLongFunction<? extends X>
function) { Objects.requireNonNull(function); return d ->
maskException( <MASK>); }
() -> function.applyAsLong(d)
```

Android Block

```
public void setAnchor(float anchorU, float anchorV) { if (marker != null) <MASK> else
{ markerOptions.anchor(anchorU, anchorV); } }
{ marker.setAnchor(anchorU, anchorV); }
```

Java Block

```
private void calculateMean() { double sum = 0; Integer count = 0; for(int i=0;
i<data.length; i++) { if (calibrationFlag[i]) <MASK> } mean = sum /
count.doubleValue(); }
{ sum += data[i]; count++; }
```

Fig. 5. Examples of perfect predictions generated by T5

wrong predictions it generates, to see whether some of them are semantically correct (*e.g., return 0x0;* is equivalent to *return 0;*) despite being different from the reference code written by the developers. The first author looked at 200 wrong predictions generated within the highest confidence interval, finding that only in three cases the prediction was semantically equivalent, with the reference code including extra (unnecessary) brackets not generated by the T5 model (*e.g.,* T5 predicts *entry;* instead of *(entry);*). Overall, it appeared that several of the generated predictions, while wrong, might still speed up the implementation process, for example when $n-1$ out of the n parameters needed for a method invocation are correctly predicted. Clearly, only a user study with developers can help in assessing the actual usefulness of these predictions during real coding activities.

Since we found cases in which the perfect predictions of the T5 spanned across dozens of tokens, being almost unrealistic, we checked whether the 21 perfect predictions having more than 30 tokens were already present in the training set. Indeed, while we ensure that there are no duplicated methods between training and test, it is possible that two different methods m_1 and m_2 have the same masked part (*i.e.,* the two methods are different in the non-masked part but they have the same set of masked tokens). Only one out of the 21 inspected cases was already present in the training set and related to the transpose of a matrix. The model was able to correctly predict very complex masked parts such as *"{ if (defaultProviders != null && index < defaultProviders.length) { return defaultProviders[index].getRebuild(defaultProviders, index + 1); } } "*.

Finally, it is worth commenting on the possible reasons behind the superior performance we observed for the T5 as compared to RoBERTa and for the DL-based models as compared to the n-gram model. RoBERTa predicts all of the masked tokens at the same time, whereas T5 predicts them one by one. This means that RoBERTa cannot use the

previously generated tokens to predict the next one, while the T5 exploits this additional information. Concerning the superior performance of the DL-based model as compared to the n-grams, this most likely comes down to the context window it is able to see. Indeed, the n-gram model can only see (and leverage) a few tokens when predicting the next one, while both T5 and RoBERTa have a better view of the coding context, seeing all the tokens surrounding the masked ones (which could be hundreds). A solution could be to scale up the n-gram model which, however, would become too demanding in terms of computational cost.

5 Threats to Validity

Threats to *construct validity* concern the relationship between theory and observation. One threat, also discussed by Hellendoorn *et al.* [37], is related to how we simulate the extent to which code completion intervenes during development, *i.e.*, by masking source code elements. As explained in Section 2.1, we consider different masking levels, not only to evaluate the amount of code completion that can be predicted but also to simulate different ways a developer writes source code, especially because we cannot assume this is done sequentially. However, we are aware that the considered masking levels cover a limited number of cases that may not completely reflect how developers write code.

Another threat is related to how we assess the code completion performances. On the one hand, 100% BLEU score clearly reflects a perfect prediction. However, the BLEU score may be sufficient to assess the performance of code-related tasks [71] and, in general, it is difficult to evaluate the usefulness of semantic equivalent predictions or imperfect yet useful. To mitigate this threat, we report some qualitative examples, indicating how partially-complete recommendations could still be useful.

Threats to *internal validity* concern factors, internal to our study, that could influence its results. To this extent, an important factor that influences DL performance is the calibration of hyperparameters, which has been performed as detailed in Section 3.2. We are aware that due to feasibility reasons we only performed a limited calibration of the hyperparameters. Hence, it is possible that a more detailed calibration would produce better performances. Also, note that we did not experiment with a pre-trained version of RoBERTa. Indeed, to simplify our experimental design and reduce the training cost we decided to only pre-train the best-performing model (*i.e.,* T5).

When building the pre-training dataset we capped to 1,500 the maximum number of instances that a single project can contribute to our dataset. This has been done to avoid a handful of projects strongly influencing the training of the model. We acknowledge that different (and maybe better) results could be obtained by considering the whole code base of each project for pre-training.

Threats to *conclusion validity* concern the relationship between evaluation and outcome. As explained in Section 3.2 we used appropriate statistical procedures, also adopting p-value adjustment when multiple tests were used within the same analysis.

Threats to *external validity* are related to the generalizability of our findings. On the one hand, we have evaluated the performances of the models on two large datasets. At the same time, we do not know whether the obtained results generalize to different domains than Android, and other programming languages than Java. A further threat is that our study is limited to the RoBERTa and T5 models for DL and, as a baseline for n-gram models, the one by Hellendoorn and Devanbu [36]. While we claim such models are well-representative of the current state-of-the-art, it would be desirable to investigate how alternative approaches would work for the different evaluation scenarios. Also, when building our fine-tuning dataset, we started from the CodeSearchNet Java Dataset provided by Husain *et al.* [41]. In this dataset, short methods (those having less than three lines), as well as methods containing *test* in their name have been excluded. This means that the results of our study do not generalize, for example, to very short methods implementing critical tasks in less than three lines of code.

6 Related Work

We start by detailing the literature related to code completion techniques and, more specifically, we highlight the approaches aimed at (partially) automating code writing. Then, we present studies investigating the effectiveness of code completion techniques. For the sake of brevity, we do not discuss recently proposed techniques for automating bug-fixing [14], [20], [80], modeling activities [56], learning code changes [17], [79], as well as source code search engines that can be used to identify pieces of code for reuse [15], [29], [60], [70], [75], [76].

6.1 Code Completion Approaches

The Prospector tool by Mandelin *et al.* [57] is one of the first techniques aimed at supporting code completion by suggesting within the IDE variables or method calls from the user's code base. Prospector was then followed by improvements such as the InSynth tool by Gvero *et al.* [32] which, given a type expected at a given point in the source code, searches for type-compatible expressions. Other approaches focus on specific elements of API usage completion. The work from Zhang *et al.* [88] aims at recommending parameter usages, achieving 64% of useful recommendations and 53% of perfect ones.

Hill and Rideout [38] proposed a technique to automatically complete the body of a method. Their approach can support such a completion for what the authors define as "atomic clones" (*i.e.*, small units of implementation that are unavoidable in Java to implement specific requirements). The presented tool uses the K-Nearest Neighbour to identify a clone of a method under development. Such a clone is then used to recommend the completion of the method body.

Bruch *et al.* [18] introduced the intelligent code completion system, able to filter out from the list of candidate method calls recommended by the IDE those that are more relevant to the current working context. Their results show the capability to correctly predict up to 82% of method calls actually needed by developers, and up to 72% of those that are relevant to the current development context. The approach by Bruch *et al.* has been improved by Proksch *et al.* [66], by adding further contextual information and by proposing a Pattern-based

Bayesian Networks approach. As a result, Proksch *et al.* were able to substantially reduce the model size while keeping about the same level of prediction accuracy. Differently from the aforementioned approaches, we do not restrict code completion to method calls.

Han *et al.* [34] proposed a technique exploiting a Hidden Markov Model (HMM) to autocomplete multiple keywords starting from abbreviated inputs. This means that the user (*i.e.,* the developer) only writes a few characters of the keyword of interest that is then expanded by the HMM. The authors show that their model can save up to 41% of keystrokes.

Robbes and Lanza [72] used information extracted from the change history of software systems to support the code completion of method calls and class names. Their approach has been implemented in a tool named OCompletion, and the performed empirical evaluation demonstrated its ability to propose a correct match in the top-3 results in 75% of cases.

Asaduzzaman *et al.* [10] proposed a technique named CSCC (Context Sensitive Code Completion). They collect code examples from software repositories and, for each method call, represent its context as a set of methods, keywords, class, and interface names appearing within four lines of code. This contextual information is then used to filter out method call recommendations. The assumption is that similar contexts imply similar method calls. CSCC outperforms previous approaches, achieving 86% precision and 99% recall.

Hindle *et al.* [39] pioneered the work on statistical language models applied to software. They conceived the idea of "naturalness of source code" and used n-gram models to create a language-agnostic algorithm that is able to predict the next token in a given statement. The trained model's average entropy is between three and four bits, indicating a high degree of naturalness.

Raychev *et al.* [69] approach the code completion problem through statistical language models. They extract sequences of method calls from a large code base, and use this dataset to train a language model able to predict API calls. Their model achieves a 90% accuracy in the top-3 recommendations.

Nguyen *et al.* [63] proposed GraPacc, a context-sensitive code completion model trained on a database of API usage patterns. These patterns are then matched to a given code under development to support code completion. GraPacc achieves up to 95% precision and 92% recall. A similar approach was later on proposed by Niu *et al.* [65] for API completion in Android: Given an API method as a query, their approach recommends a set of relevant API usage patterns. They report an 18% improvement of F-Measure when comparing to pattern extraction using frequent-sequence mining.

Tu *et al.* [77] introduced a cache component to exploit the "localness of code" in the n-gram model. Results show that since the code is locally repetitive, localized information can be used to improve performance. The enhanced model outperforms standard n-gram models by up to 45% in accuracy. In a related work, Franks*et al.* [26] implemented *CACHECA*, an Eclipse auto-completion plugin exploiting the aforementioned cache language model [77]. In comparison to Eclipse built-in suggestions, their tool improves the accuracy of top 1 and top 10 suggestions by 26% and 34%, respectively.

Nguyen *et al.* [64] presented GraLan, a graph-based statistical language model that the authors instantiated to recommend the next API element needed in a given code, where an API element is a method call together with the control units (*e.g.,* `if` statements) needed for its usage. The reported empirical evaluation showed that GraLan can correctly recommend the correct API element in 75% of cases within the first five candidates.

Hou and Pletcher [40] evaluated three mechanisms to enhance code completion techniques, namely sorting, filtering, and grouping. Also this works focuses on code completion related to API methods and the outcome of their study is an assessment of the effectiveness of fourteen different configurations of the three mechanisms.

Asaduzzaman *et al.* [11] proposed a technique to recommend developers with examples of framework extensions. Given a class under development, the approach recommends code examples showing how to integrate frameworks in specific extension points. While the approach by Asaduzzaman *et al.* recommends relatively large code completion fragments, it is limited to a specific scenario, *i.e.,* framework extension points, whereas the approaches we experiment with are more general in that respect.

Hellendoorn and Devanbu [36] proposed further improvements to the cached models aimed at considering specific characteristics of code (*e.g.,* unlimited, nested, and scoped vocabulary). Then, they compare their model with DL-based models, showing its superiority. Also, they show that the two families of techniques can be combined together, leading to an unprecedented 1.25 bits of entropy per token. Karampatsis *et al.* [47], a few years later, suggested instead that neural networks are the best language-agnostic algorithm for code completion. They proposed to overcome the *out-of-vocabulary problem* by using *Byte Pair Encoding* [27]. In addition, the proposed neural network is able to dynamically adapt to different projects. Their best model outperforms n-gram models, achieving an entropy of 1.03 bits.

Kim *et al.* [49] leveraged the Transformers neural network architecture for code completion. They provide the syntactic structure of code to the network by using information from the Abstract Syntax Tree to fortify the self-attention mechanism. Among the several models they experiment with, the best one reached a MRR up to 74.1% in predicting the next token.

Alon *et al.* [8] addressed the problem of code completion with a language agnostic approach named Structural Language Model. It leverages the syntax to model the code snippet as a tree. The model, based on LSTMs and Transformers, receives an AST representing a partial expression (statement), with some missing consecutive tokens to complete. Their best model reached state-of-the-art performance with an exact match accuracy for the top prediction of 18.04%.

Svyatkovskiy *et al.* [73] introduced IntelliCode Compose, a general-purpose multilingual code completion tool capable of predicting code sequences of arbitrary token types. They do not leverage high-level structural representation, such as AST, and use subtokens to overcome the *out-of-vocabulary problem*. Their model can recommend an entire statement, and

achieves a perplexity of 1.82 for the Python programming language.

Liu *et al.* [53] presented a Transformer-based neural architecture pre-trained with the goal of incorporating both code understanding and generation tasks. Afterwards, the model was then fine-tuned on the classic code completion task (*i.e.,* predicting the next token to write).

A problem related to code completion has also been tackled by Watson *et al.* [82]: The authors exploit a sequence-to-sequence model to recommend assert statements for a given Java test case. This technique is able to generate a specific type of code statement, with a top-1 accuracy of 31%. Also, Kanade *et al.* [45] show how code embeddings can support code-related tasks, including *variable misuse and repair*, related to code completion when focusing on a single token.

Svyatkovskiy *et al.* [74] proposed a different perspective on neural code completion, shifting from a generative task to a learning-to-rank task. Their model is used to rerank the recommendations provided via static analysis, being cheaper in terms of memory footprint than generative models. To this aim, Avishkar *et al.* [16] proposed a neural language model for code suggestion in Python, aiming to capture long-range relationships among identifiers exploiting a sparse pointer network.

To address the out-of-vocabulary problem in standard neural language models, Jian *et al.* [52] proposed a pointer mixture deep learning model for Python benefiting from the pointer copy mechanism. Such architecture helps the model to generate an out-of-vocabulary word from local context through a pointer component when generating a within-vocabulary token is not possible.

A considerable step forward, has been taken recently by Aye and Kaiser [12] proposing a novel language model to predict the next top-k tokens while taking into consideration some real-world constraints such as (i) prediction latency, (ii) size of the model and its memory footprint, and (iii) validity of suggestions. Chen *et al.* [19] proposed a deep learning model for API recommendation combining structural and textual code information based on an API context graph and code token network. The evaluation model significantly outperforms the existing graph-based statistical approach and the tree-based deep learning approach for API recommendation.

To the best of our knowledge, our work is the first to present a comprehensive study on the effectiveness of Transformer models for code completion tasks, *pushing this problem forward by attempting the automatic generation of an entire code block* (*e.g.,* the body of a `for` statement).

6.2 Studies About the Effectiveness of Code Completion Approaches

Although code completion techniques are likely to be beneficial for developers, their limitations (*e.g.,* prediction latency, accuracy) can bound their practical usefulness. For this reason, several studies investigated the effectiveness of code completion techniques.

Jin and Servant [44] investigated the effect of different recommendation list lengths on the developers' productivity. They found that lengthy suggestion lists are not uncommon and reduce the developer's likelihood of selecting one of the recommendations.

Lin *et al.* [42] focus on the performance of a *code2vec* [9] model, in the context of method name recommendation. The authors retrain the model on a different dataset and assess it in a more realistic setting where the training dataset does not contain any record from evaluation projects. The results suggest that while the dataset change had little impact on the model's accuracy, the new *project-based* setting negatively impacted the model. Lin *et al.* [42] also evaluated the usefulness of *code2vec* suggestions by asking developers to assess the quality of suggestions for non-trivial method names. The evaluation results show the model rarely works when it is needed in practice. Further investigation also revealed that around half of successful recommendations (48%) occur for simpler scenarios, such as setter/getter methods or when the recommended name is copied from the method body source code.

Hellendoorn *et al.* [37] studied 15,000 real code completions from 66 developers founding that typically-used code completion benchmarks — *e.g.,* produced by artificially masking tokens — may misrepresent actual code completion tasks. The study by Hellendoorn *et al.* suggests that further research is needed to assess the actual applicability of DL-based code completion to the real-world. This is however out of scope for our work, because our aim is to assess the capability of DL models to predict non-trivial portions of code going beyond a single method call or parameter.

Liu *et al.* [54] investigate the performance of deep learning-based approaches for generating code from requirement texts. For that, they assessed five state-of-the-art approaches on a larger and more diverse dataset of pairs of software requirement texts and their validated implementation as compared to those used in the literature. The evaluation results suggest that the performance of such approaches, in terms of common metrics (*e.g.,* BLEU score), is significantly worse than what was reported in the literature. The authors attribute this observation to the relatively small datasets on which such models are evaluated.

Similarly, Aye *et al.* [13] investigate the impact of using real-world code completion examples (*i.e.,* code completion acceptance events in the past) for training models instead of artificial examples sampled from code repositories. The usage of such realistic data on n-gram and transformer models suggests a significant accuracy decrease. Later, an A/B test conducted with Facebook developers confirmed that the autocompletion usage increases by around 6% for models trained on real-world code completion examples.

Our work, differently from previous studies, aims at assessing the capability of state-of-the-art Transformer-based models in predicting non-trivial snippets of code. In contrast, it is out of this study scope to assess the developer's perception of the prediction models that would require an extensive study with developers.

7 Conclusion

We investigated the ability of Transformer-based DL-models in dealing with code completion tasks having a different level of difficulty, going from the prediction of a few tokens within the same code statement, up to the entire code blocks

we masked. Among the three models we experimented with, namely T5 [68], RoBERTa [24], and the cached *n*-gram model [36], the T5 resulted to be the most effective in supporting code completion.

Our study provided a series of highlights that will guide our future research. First, when the code to complete spans over multiple statements (two in the case of our experiments), these models, with the training we performed, are still far from being a valuable solution for software developers. Indeed, even the best-performing model (T5) struggles in guessing entire code blocks. However, the performance we reported should not be seen as an "upper bound" for these techniques, since larger models may be trained on more data can be adopted (*e.g.*, the recently proposed GitHub Copilot [1]) and different training strategies could help in achieving better results (*e.g.*, Tufano *et al.* [78] showed that pre-training on English text helps transformer models in improving performance even in code-related tasks). Besides working on these research directions we also plan to investigate alternative solutions mixing, for example, retrieval-based and DL-based solutions.

Second, the confidence of the predictions generated by the T5 turned out to be a very reliable proxy for the quality of its predictions. This is something fundamental for building tools around this model, as it can be used by developers to just ignore low-confidence recommendations. Future studies will investigate how the developers perceive the usefulness of recommendations having different characteristics, including length, confidence, and covered code constructs.

Finally, a user study is also needed to understand what is the level of accuracy (in terms of perfect predictions) needed to consider tools built around these models as effective for developers. In other words, it is important to understand the "percentage of wrong predictions" a developer can accept before considering the tool counterproductive. Such a study is also part of our research agenda.

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

REFERENCES

[1] "Github copilot https://copilot.github.com."
[2] *Hugging Face's Tokenizer Repositor*, https://github.com/huggingface/tokenizers.
[3] *N-gram Cached Model*, https://github.com/SLP-team/SLP-Core.
[4] *ScrML Website*, https://www.srcml.org/.
[5] *Weights and Biases Website*, https://www.wandb.com/.
[6] *Anderson–Darling Test*. New York, NY: Springer New York, 2008, pp. 12–14. [Online]. Available: https://doi.org/10.1007/978-0-387-32833-1_11
[7] M. Allamanis, *CodeSearchNet Deduplication Algorithm*, https://github.com/github/CodeSearchNet/blob/master/src/dataextraction/dedup_split.py.
[8] U. Alon, R. Sadaka, O. Levy, and E. Yahav, "Structural language models of code," in *International Conference on Machine Learning*. PMLR, 2020, pp. 245–256.
[9] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.
[10] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou, "Context-sensitive code completion tool for better api usability," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 621–624.
[11] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou, "Recommending framework extension examples," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 456–466.
[12] G. A. Aye and G. E. Kaiser, "Sequence model design for code completion in the modern ide," *arXiv preprint arXiv:2004.05249*, 2020.
[13] G. A. Aye, S. Kim, and H. Li, "Learning autocompletion from real-world datasets," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2021, pp. 131–139.
[14] 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.
[15] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes, "Sourcerer: A search engine for open source code supporting structure-based search," in *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06. ACM, 2006, p. 681–682.
[16] A. Bhoopchand, T. Rocktäschel, E. Barr, and S. Riedel, "Learning python code suggestion with a sparse pointer network," *arXiv preprint arXiv:1611.08307*, 2016.
[17] S. Brody, U. Alon, and E. Yahav, "Neural edit completion," *arXiv preprint arXiv:2005.13209*, 2020.
[18] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE 2009, 2009, pp. 213–222.
[19] C. Chen, X. Peng, Z. Xing, J. Sun, X. Wang, Y. Zhao, and W. Zhao, "Holistic combination of structural and textual code information for context based api recommendation," *IEEE Transactions on Software Engineering*, 2021.
[20] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, 2019.
[21] M. Ciniselli, "Replication package https://github.com/mciniselli/T5_Replication_Package.git."
[22] M. Ciniselli, N. Cooper, L. Pascarella, D. Poshyvanyk, M. Di Penta, and G. Bavota, "An empirical study on the usage of bert models for code completion," in *Proceedings of the 18th Working Conference on Mining Software Repositories*, ser. MSR '21, 2021, p. To Appear.
[23] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in github for MSR studies," in *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*. IEEE, 2021, pp. 560–564.
[24] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Jun. 2019, pp. 4171–4186.
[25] M. Dreyer and D. Marcu, "HyTER: Meaning-equivalent semantics for translation evaluation," in *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Montréal, Canada: Association for Computational Linguistics, Jun. 2012, pp. 162–171. [Online]. Available: https://www.aclweb.org/anthology/N12-1017
[26] C. Franks, Z. Tu, P. Devanbu, and V. Hellendoorn, "Cacheca: A cache language model based code suggestion tool," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 705–708.
[27] P. Gage, "A new algorithm for data compression," *C Users J.*, vol. 12, no. 2, p. 23?38, 1994.
[28] F.-X. Geiger, I. Malavolta, L. Pascarella, F. Palomba, D. D. Nucci, and A. Bacchelli, "A graph-based dataset of commit history of real-world android apps," in *Proceedings of the 15th International Conference on Mining Software Repositories, MSR*. ACM, May 2018. [Online]. Available: https://androidtimemachine.github.io

[29] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A search engine for finding highly relevant applications," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. ACM, 2010, p. 475–484.

[30] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*, 2nd ed. Lawrence Earlbaum Associates, 2005.

[31] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 631–642. [Online]. Available: http://doi.acm.org.proxy.wm.edu/10.1145/2950290.2950334

[32] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac, "Complete completion using types and weights," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, 2013, pp. 27–38.

[33] S. Han, D. R. Wallace, and R. C. Miller, "Code completion from abbreviated input," in *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 332–343.

[34] ——, "Code completion from abbreviated input," in *2009 IEEE/ACM International Conference on Automated Software Engineering*, 2009, pp. 332–343.

[35] ——, "Code completion of multiple keywords from abbreviated input," *Automated Software Engineering*, vol. 18, no. 3-4, pp. 363–398, 2011.

[36] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, 2017, p. 763?773.

[37] V. J. Hellendoorn, S. Proksch, H. C. Gall, and A. Bacchelli, "When code completion fails: a case study on real-world completions," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, 2019, pp. 960–970.

[38] R. Hill and J. Rideout, "Automatic method completion," in *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, 2004, pp. 228–235.

[39] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE 2012. IEEE Press, 2012, pp. 837–847.

[40] D. Hou and D. M. Pletcher, "An evaluation of the strategies of sorting, filtering, and grouping api methods for code completion," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2011, pp. 233–242.

[41] H. Husain, H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *CoRR*, vol. abs/1909.09436, 2019. [Online]. Available: http://arxiv.org/abs/1909.09436

[42] L. Jiang, H. Liu, and H. Jiang, "Machine learning based recommendation of method names: how far are we," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 602–614.

[43] 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:.

[44] X. Jin and F. Servant, "The hidden cost of code completion: Understanding the impact of the recommendation-list length on its efficiency," in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 70–73.

[45] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," 2020.

[46] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big code != big vocabulary: Open-vocabulary models for source code," in *Proceedings of the 42nd International Conference on Software Engineering, ICSE 2020*, 2020, p. To Appear.

[47] 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

[48] M. Kendall, "A new measure of rank correlation," *Biometrika*, 1938.

[49] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 150–162.

[50] 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.

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

[52] J. Li, Y. Wang, M. R. Lyu, and I. King, "Code completion with neural attention and pointer networks," *arXiv preprint arXiv:1711.09573*, 2017.

[53] F. Liu, G. Li, Y. Zhao, and Z. Jin, "Multi-task learning based pre-trained language model for code completion," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2020. Association for Computing Machinery, 2020.

[54] H. Liu, M. Shen, J. Zhu, N. Niu, G. Li, and L. Zhang, "Deep learning based program generation from requirements text: Are we there yet?" *IEEE Transactions on Software Engineering*, 2020.

[55] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized BERT pretraining approach," *CoRR*, vol. abs/1907.11692, 2019. [Online]. Available: http://arxiv.org/abs/1907.11692

[56] P. Mäder, T. Kuschke, and M. Janke, "Reactive auto-completion of modeling activities," *IEEE Transactions on Software Engineering*, 2019.

[57] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: helping to navigate the API jungle," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, 2005, pp. 48–61.

[58] P. Martins, R. Achar, and C. V. Lopes, "50k-c: A dataset of compilable, and compiled, java projects," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, 2018, pp. 1–5.

[59] A. Mastropaolo, S. Scalabrino, N. Cooper, D. N. Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Studying the usage of text-to-text transfer transformer to support code-related tasks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 336–347.

[60] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie, "Exemplar: A source code search engine for finding highly relevant applications," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1069–1087, 2012.

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

[62] A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A large-scale study on repetitiveness, containment, and composability of routines in open-source projects," in *Proceedings of the IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR 2016)*, 2016, pp. 362–373.

[63] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen, "Graph-based pattern-oriented, context-sensitive source code completion," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 69–79.

[64] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 858–868.

[65] H. Niu, I. Keivanloo, and Y. Zou, "Api usage pattern recommendation for software development," *Journal of Systems and Software*, vol. 129, pp. 127–139, 2017.

[66] S. Proksch, J. Lerch, and M. Mezini, "Intelligent code completion with bayesian networks," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 1, pp. 3:1–3:31, 2015.

[67] S. Raemaekers, A. van Deursen, and J. Visser, "The maven repository dataset of metrics, changes, and dependencies," in *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 221–224.

[68] 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.

[69] 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 2014, 2014, pp. 419–428.

[70] S. P. Reiss, "Semantics-based code search," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. IEEE Computer Society, 2009, p. 243–253.

[71] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, "Codebleu: a method for automatic evaluation of code synthesis," *CoRR*, vol. abs/2009.10297, 2020. [Online]. Available: https://arxiv.org/abs/2009.10297

[72] R. Robbes and M. Lanza, "Improving code completion with program history," *Automated Software Engineering*, vol. 17, no. 2, pp. 181–212, 2010.

[73] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1433–1443.

[74] A. Svyatkovskiy, S. Lee, A. Hadjitofi, M. Riechert, J. Franco, and M. Allamanis, "Fast and memory-efficient neural code completion," 2020.

[75] S. Thummalapenta and T. Xie, "Parseweb: A programmer assistant for reusing open source code on the web," in *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07. Association for Computing Machinery, 2007, p. 204–213.

[76] ——, "Spotweb: Detecting framework hotspots and coldspots via mining open source code on the web," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 327–336.

[77] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 269–280. [Online]. Available: https://doi.org/10.1145/2635868.2635875

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

[79] 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.

[80] M. Tufano, C. Watson, G. Bavota, M. Di 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.

[81] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 5998–6008. [Online]. Available: http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf

[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] F. Wen, E. Aghajani, C. Nagy, M. Lanza, and G. Bavota, "Siri, write the next method," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 138–149. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00025

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

[85] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[86] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, and J. Brew, "Huggingface's transformers: State-of-the-art natural language processing," *ArXiv*, vol. abs/1910.03771, 2019.

[87] B. Yoav and H. Yosef, "Controlling the false discovery rate: A practical and powerful approach to multiple testing," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 57, no. 1, pp. 289–300, 1995.

[88] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou, "Automatic parameter recommendation for practical API usage," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, 2012, pp. 826–836.

Matteo Ciniselli 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 Mathematical Engineering from Politecnico di Milano, Italy, in April 2015. His research interests include the study of deep-learning models to support code-related tasks. More information available at: https://www.inf.usi.ch/phd/cinism.

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/#/.

Luca Pascarella is a postdoctoral researcher at the Università della Svizzera italiana (USI), Switzerland, where he is part of the Software Institute. He received his Ph.D. in Computer Science from the Delft University of Technology (TU Delft), The Netherlands, in 2020. His broader mission aims to smooth engineering tasks through data-driven algorithms, which leverage the large amount of information recorded during modern engineering processes. His research interests include empirical software engineering, mining software repository, and code review. He received an ACM SIGSOFT Distinguished Paper Award at MSR 2017 and a Best Paper Award Honorable Mention at CSCW2018. More information available at: https://lucapascarella.com.

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.

Emad Aghajani is a postdoctoral researcher in the SEART research group, at Software Institute, the Università della Svizzera italiana (USI), Switzerland. He finished his Ph.D. studies in Computer Science in 2020 under the supervision of Prof. Michele Lanza and Prof. Gabriele Bavota at USI. He received his M.Sc. in Software Engineering from Sharif University of Technology, Iran, in 2016. His research interests include software evolution, software maintenance, mining software repositories, and empirical software engineering. More information is available at: https://emadpres.github.io/.

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 and ICPC'20. 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/.

Massimiliano Di Penta is a full professor at the University of Sannio, Italy. His research interests include software maintenance and evolution, mining software repositories, empirical software engineering, search-based software engineering, and service-centric software engineering. He is an author of over 300 papers appeared in international journals, conferences, and workshops. He serves and has served in the organizing and program committees of more than 100 conferences, including ICSE, FSE, ASE, ICSME. He is editor-in-chief of the Journal of Software: Evolution and Processes edited by Wiley. He is in the editorial board of ACM Transactions on Software Engineering and Methodology, and of the Empirical Software Engineering Journal. He has served the editorial board of the IEEE Transactions on Software Engineering.

Gabriele Bavota 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/.