Università della Svizzera italiana

Software Institute

STUDYING STRENGTHS AND WEAKNESSES OF CODE RECOMMENDERS

Matteo Ciniselli

Research Advisor

Prof. Dr. Gabriele Bavota

SEART

Dissertation Committee

Prof. Michele Lanza	Università della Svizzera italiana, Switzerland
Prof. Laura Pozzi	Università della Svizzera italiana, Switzerland
Prof. Martin Pinzger	Alpen-Adria-Universität Klagenfurt, Austria
Dr. Michele Tufano	Microsoft, USA

Dissertation accepted on 19 December 2023

Research Advisor
Prof. Gabriele Bavota

Ph.D. Program Co-Director
Prof. Walter Binder

Ph.D. Program Co-Director
Prof. Stefan Wolf

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Matteo Ciniselli
Lugano, 19 December 2023

Abstract

Given the high costs of software development and maintenance, tools and techniques have been proposed both in industry and academia to speed up programming activities. Some of these techniques focus on automating the generation of source code, by predicting the code tokens the developer would write starting from the ones already typed. We use the term *code recommender* to refer to these approaches.

In this thesis, we study the strengths and the weaknesses of code recommenders from several perspectives. We started by investigating the accuracy of recently proposed Deep Learning (DL) models in generating non-trivial and possibly multiple code statements. Indeed, previous work usually tested these techniques in the relatively easy task of predicting the single next token the developer is likely to write. We pushed the boundaries of the predictions to entire blocks of code, showing that DL models can achieve excellent results when predicting a few code tokens but their accuracy steadily decreases when asked to predict entire code statements. Still, even in such a challenging scenario, DL models are able to produce correct suggestions in $\sim$29% of cases. Given such a finding and the stunning capabilities of recently released products such as GitHub Copilot, we decided to investigate the extent to which DL models tend to copy code from the training set. Surprisingly, we found that DL models rarely copy verbatim from the training data, mostly generating original code. Finally, we studied the extent to which DL models generalize across different versions of the same programming language, showing that a model trained on a language version v_i exhibits a strong drop in performance when asked to predict completions on a version $v_j \neq v_i$.

Then, since most of the tools proposed in the literature are based on researchers' intuitions, we ran a survey with 80 practitioners to create a taxonomy of characteristics of code recommenders they consider important (*e.g.,* adapt the recommendations to the developer's programming style). Such a taxonomy can be used to inform future research in the field aimed at targeting the weaknesses of code recommenders as perceived by practitioners. To show that, we tackle one of the identified weaknesses (*i.e.,* limited knowledge of the coding context) and show how the performance of code recommenders can be boosted by enriching the coding context they are aware of before triggering a recommendation.

Acknowledgments

I arrived in Lugano nearly four years ago, yet it feels as though I've just started my Ph.D. This path has been filled with numerous challenges, and I'm sincerely grateful to everyone who has supported me throughout this journey.

First and foremost, I would like to express my gratitude to my advisor, Prof. Gabriele Bavota. His exceptional expertise and insightful feedback have been my compass, and his dedication and passion for the research have been a wellspring of inspiration. Over these years, I've learned a lot from him, both technical knowledge and soft skills, all of which will be tremendously useful for my upcoming challenges.

I also want to express my appreciation for all those who shared this path with me, leaving an indelible mark in my memory. My colleagues, Antonio and Rosalia, who started their Ph.D. almost at my same time, and have shared beautiful moments with me at conferences and in our daily life. The postdocs, Emad and Luca, who were a guiding light at the beginning of this adventure, when I was still refining my skills. A sincere thanks also to Ozren, Alejandro, Alessandro, and Alberto who are the past and the future of the SEART group. My gratitude extends to all the members of REVEAL and the Software Institute itself. I had the privilege, during my visiting experience, of immersing myself in a new reality in Molise, meeting wonderful people like Rocco, Emanuela, Giovanni, and the entire Stake Lab.

I would also like to express my sincere gratitude to the members of my dissertation committee: Prof. Michele Lanza, Prof. Laura Pozzi, Prof. Martin Pinzger, and Dr. Michele Tufano. Their constructive feedback and thoughtful suggestions have significantly enhanced the quality of this thesis. Thank you for taking time to review my thesis, and attend my defense.

A heartfelt thanks are also extended to my friends. Andrea, who moved to Switzerland over a decade ago and unwittingly prompted me to do the same, alleviating my thoughts and anxieties with his insightful counsel, and Matteo, who adds brilliance to Milan's evenings with his profound knowledge. Special appreciation goes out to the friends of my hometown, Bresso, with whom I've shared over two decades of my life, experiencing pleasant moments and forging lasting memories.

I reserve my deepest gratitude for my family, my mother, Maddalena, my brother, Don Marco, who have always been a steady presence whenever I needed support, and also my father, Mario, who is watching over me from above. Although after my relocation we see only occasionally, they will forever be an integral part of my life. And finally, an immense thanks to my wife, Chiara, and her extraordinary family. She has stood by my side every day, encouraging me to leave a secure job in Milan to embrace an amazing adventure in Lugano. We recently celebrated our marriage, and that was the best day of my entire life. I am truly blessed to have her in my life, and during our 8 years together, she has accelerated my personal growth and contributed to my evolution into a better version of myself.

Contents

1

Introduction

Recommender systems for software developers have been defined by Robillard *et al.*[RMWZ14] as "software tools that can assist developers with a wide range of activities", ranging from bug fixing [DAP+17, LNNN15, LHL+17] to documentation writing and retrieval [HLX+18, MBP+15, MBP+17] and code review activities [TPT+21, TMM+22]. Among the mostly automated tasks there is code completion, which is considered one of the "killer" features of modern Integrated Development Environments (IDEs) [BMM09, KZTC21, RL10]. It can provide developers with predictions about the next token(s) to write given the code already written in the IDE, thus speeding up the code writing and preventing potential mistakes [HWM09, HWM11].

The quality of the recommendation, at the beginning merely based on alphabetical order, drastically improved thanks to the development of data-driven techniques. The enhanced code recommenders started to suggest "intelligent" completions that leverage the context surrounding the code [BMM09, RL10], the history of code changes [RL10], and/or coding patterns mined from software repositories [HBS+12, NNN+12, TSD14, ARSH14, NNN16, NKZ17, HD17]. Recently, also Deep Learning (DL) models [KS19, KZTC21, ASLY20, SDFS20, WVVP15] have led to state-of-the-art performance for code completion. The rise of DL was facilitated by the unprecedented amount of code-related data publicly available on platforms such as GitHub that, at time of writing, counts over 120 million public repositories and over 100 million users. The support provided to developers, initially limited to the prediction of the single next token the developer is likely to write, improved over time, being able to recommend multiple contiguous tokens [ASLY20, SDFS20, CTJ+21].

Building on top of this research, OpenAI and GitHub presented GitHub Copilot [copb], a code assistant that achieved state-of-the-art performance in code recommendation. Thanks to the massive training on a huge corpus of more than 150Gb of data from over 50M public GitHub repositories, Copilot is able to even recommend entire methods just starting from their signature or the skeleton of a class given the very first typed tokens. Furthermore, in February 2022, DeepMind introduced AlphaCode [alp], a system for code generation that achieved human performance in competitive programming tasks. Unlike programming, that usually requires solving simple tasks, competitive programming problems require deeper reasoning, showing the latent potential of DL models.

In November 2022, OpenAI introduced ChatGPT, a conversational language model powered by a DL model. ChatGPT is capable of generating human-like responses in natural language, including those related to software-related tasks, such as implementing a given program.

While anecdotal evidence suggests that these tools are production-ready, there is little empirical evidence about their actual capabilities. Indeed, in the literature there is a lack of empirical studies investigating the behavior of these tools, like the inclination of the model to copy already seen code [zie] or the capability of the model to generalize to different datasets [HPGB19, AKL21]. Moreover, the tools proposed are based on researchers' intuitions and just a few studies investigated what developers want from code recommenders and how they interact with them [MCB15, JLJ19].

1.1 Thesis Statement

We formulate our thesis as follows:

> *The novel AI-empowered code recommenders are changing the way in which developers write code. Although these tools achieved astounding performance, solid empirical knowledge about their strengths and weaknesses can push the boundaries even further.*

1.2 Research Contributions

The research contributions of this thesis can be grouped into two high-level categories: (i) studies aimed at objectively investigating strengths and weaknesses of the state-of-the-art via technology-based experiments (*i.e.,* experiments not involving humans) which assess the performance of DL-based code recommenders from a specific perspective; and (ii) research investigating the practitioners' perspective on code recommenders and exploiting it to improve the performance and usefulness of code recommenders.

1.2.1 Objectively Investigating Strengths and Weaknesses of Code Recommenders

Despite the significant improvement over time in the performance of code recommenders, most of the approaches in the literature came with an important limitation in the support provided to developers: They mainly focused on the prediction of the single next code token the developer is likely to write. Only recently a few techniques started focusing on predicting multiple contiguous tokens [ASLY20, SDFS20, copb]. We studied the accuracy of data-driven techniques when employed in three code completion scenarios having different levels of difficulty, including the automated generation of complete code blocks composed by multiple statements. All code completion scenarios involved the automated generation of code tokens to complete partially implemented Java methods.

We started by testing in these scenarios two models: (i) an *n*-gram model, a probabilistic language model that showed promising results when used to model source code

[TSD14, HD17], and (ii) the recently presented RoBERTa model [LOG+19], a DL model that leverages the Transformers architecture. We compared their performance in the different code completion scenarios previously mentioned, showing how the more recent DL model (RoBERTa) substantially outperforms the *n*-gram model. We collected our findings in the following work [CCP+21b]:

> **An Empirical Study on the Usage of BERT Models for Code Completion**
> Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Denys Poshyvanyk, Massimiliano Di Penta, Gabriele Bavota. In *18th International Conference on Mining Software Repository (MSR 2021)*, 108–119

While being superior to the *n*-gram model, in our MSR'21 paper we observed that RoBERTa substantially suffers when generating complex predictions spanning over multiple statements (∼10% of correct predictions as compared to the ∼52% achieved in the simplest code completion scenario). To overcome the limitations of the RoBERTa model, we experimented with the novel Text-To-Text Transfer Transformer (T5) model [RSR+20], another Transformer-based architecture that achieved state-of-the-art performance in several NLP tasks. We showed the superiority of the T5 model in all scenarios, with correct predictions going up to ∼69%, with a ∼29% achieved in the most complex scenario. The paper presenting this work has been published in IEEE Transactions on Software Engineering [CCP+21a]:

> **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, Gabriele Bavota. In *IEEE Transactions on Software Engineering (TSE 2021)*, 4818–4837

In such a work we observed some very surprising recommendations generated by the experimented models. With "surprising" we refer to quite complex and challenging statements correctly generated by the models. This, in addition to the remarkable performance achieved by recently presented code recommenders such as GitHub Copilot and AlphaCode [copb, alp], made us wonder whether these models tend to copy code from their training data when generating predictions. Indeed, while in our experiments we guaranteed no duplicates between the Java methods present in the training and test set, it is still possible that specific code statements to generate for a prediction in the test set are available in the training set. Answering such a research question is crucial, since the vast majority of training data used for these models comes from open source repositories and hence its usage is regulated by specific Free and Open Source Software (FOSS) licenses.

These licenses regulate how the licensed source code can be distributed or modified, generating derivative work, but it is not clear if the code can be used for training DL-based code recommenders or if the generated source must be considered new code "written from scratch" rather than derivative work. The problem would be more relevant if these techniques tend to perform a verbatim copy from training data that spans different statements. We checked the percentage of Type-1 clones (*i.e.,* exact copies) and Type-2 clones (*i.e.,* copied code with changes to identifiers and types) between the training data and the T5 predictions. From our findings, there is no evidence that T5 tends to copy a significant amount of code from the training set. We reported the results of our investigation in the following paper [CPB22]:

> **To What Extent do Deep Learning-based Code Recommenders Generate Predictions by Cloning Code from the Training Set?**
> Matteo Ciniselli, Luca Pascarella, Gabriele Bavota. In *19th International Conference on Mining Software Repository (MSR 2022)*, 1167–178

Finally, in this research track we studied the extent to which DL models generalize across different versions of a given programming language. This means training the model in a specific version v_i and testing it on that same version as well as on other versions $v_j \neq v_i$. This allows studying the difference in performance (if any) observed across the test sets. We observed a strong decrease in performance when moving towards versions being far in time from the one used for training, thus suggesting the need for continuously re-training DL-based code recommenders being deployed. This work is currently under review:

> **On the Generalizability of Deep Learning-based Code Completion Across Programming Language Versions**
> Matteo Ciniselli, Alberto Martin-Lopez, Gabriele Bavota. *International Conference on Program Comprehension (ICPC 2024)*

1.2.2 The Practitioners' Perspective

The design of code recommenders in the literature is usually guided by the researchers' intuition, and rarely developers are involved in this process. To collect developers' opinion about code recommender systems, we ran a survey with 80 professional developers recruited through Amazon Mechanical Turk [amt] and via personal contacts asking their *desiderata* when it comes to code recommenders and, more in general, aspects they feel should be improved. The outcome of the survey is a taxonomy of characteristics of code recommenders desired by developers. Our findings indicate, for example, that practitioners would like to have code recommendations personalized to their development style. They also desire a tool able to support them in different code completion scenarios, ranging from identifier recommendations to the automated completion of entire lines of code. We reported the results of our investigation in the following paper [CPA+23]:

> **Source Code Recommender Systems: The Practitioners' Perspective**
> Matteo Ciniselli, Luca Pascarella, Emad Aghajani, Simone Scalabrino, Rocco Oliveto, Gabriele Bavota. In *45th International Conference on Software Engineering (ICSE 2023)*, 2161–2172

The output of this work is a taxonomy of 70 characteristics of code recommenders that are relevant for practitioners. Such a taxonomy can be used to derive a future research agenda. To show that, we tackled one of the weaknesses identified by practitioners: They consider particularly important that the code recommender is aware of the coding context in order to generate more relevant recommendations. For this reason, we performed an empirical study investigating the role of the provided context (*i.e.,* the information provided as input to the code recommender) on the performance of the DL-based recommender. Our finding showed the benefits of additional contextual information provided to the model, with a relative increase of 22% of Exact Match predictions (*i.e.,* the cases in which the recommendation of the model is the exact code we are expecting). The chapter is based on the following submission:

> **Deep Learning-based Code Completion: On the Impact on Performance of Additional Contextual Information**
> Matteo Ciniselli, Gabriele Bavota. *Submitted to Empirical Software Engineering (EMSE 2023)*

1.3 Outline

This dissertation is structured in the following chapters:

Chapter 2 presents an overview of the state of the art for source code recommender systems, including the main models proposed in the literature and empirical studies looking at code recommenders from different perspectives.

Chapter 3 presents an empirical study that compares the performance of different state-of-the-art models in three different code completion scenarios, ranging from suggesting a few tokens up to an entire block of code, trying to emulate different coding situations in which developers may need assistance. We evaluated the models' performance using several metrics extensively used in the literature.

Chapter 4 presents a study exploring to what extent DL models tend to verbatim copy snippets of code seen during the training. Our findings showed that the model seems to memorize simple and ubiquitous constructs, like *return* statements, while it rarely copies long snippets of code.

Chapter 5 investigates the capabilities of DL models to generalize to different versions of the same programming language. In detail, we trained the CodeT5 model on the Java 8 version, and we evaluated its prediction performance on 9 different versions, spanning over 20 years of language evolution.

Chapter 6 presents an empirical study involving 80 developers aiming at investigating important characteristics of code recommenders systems. Our results show that, despite the state-of-the-art solutions proposed in the literature achieving stunning performance in terms of accuracy, there are a lot of features that can support developers, for example providing the rationale behind the suggestions or enhancing the readability of the proposed code.

Chapter 7 presents an empirical study investigating the role of the contextual information on the performance of DL-based code recommenders. Our findings show the benefits of the additional contextual information.

Chapter 8 concludes the thesis and outlines directions for future work.

2

State of the Art

Code completion is a useful feature integrated into the IDEs that assists programmers by automatically recommending code snippets, reducing the effort of manually typing code and also preventing syntax errors. During the last years, several code completion approaches have been proposed by researchers and industry. The advent of novel and enhanced techniques improved the granularity of the suggestions, passing from recommending just a method call up to recommending even several lines of code, thus dramatically increasing the developers' productivity. Recently, tools like Copilot Chat [copa] further improved the support provided to developers, allowing them to interact with Github Copilot to solve coding-related questions directly inside the IDE.

This chapter contains the most relevant publications related to code recommenders, and in particular to their application in the context of code completion. Works related to recommending code examples [HM05, MBP+15, ARSH17, ASBN21] or retrieving relevant Stack Overflow discussions [PBL13, PBP+14, TR16, YHL16] are not presented since not central to this thesis. Section 2.1 discusses the code recommender systems proposed in the literature, while Section 2.2 reports on empirical studies investigating code recommenders from different perspectives. Section 2.3 discusses how the contributions of this thesis stem and are different from previous works in the literature.

2.1 Source Code Recommender Systems

First attempts in recommending code. At the beginning, the code recommenders integrated in the IDEs were simply suggesting the code to complete in alphabetical order. The Prospector tool by Mandelin *et al.* [MXBK05] was one of the first techniques aimed at supporting smart code completion by suggesting within the IDE variables or method calls from the user's code base. Developers can interact with the tool using a query that expresses their intent. Based on the input query, Prospector recommends the best suggestion from the code base, leveraging information contained in signatures and code examples. Prospector was then followed by improvements such as the InSynth tool by Gvero *et al.* [GKKP13] which, given a type expected at a given point in the source code, searches for type-compatible ex-

pressions.

Hill and Rideout [HR04] 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. A similar technique was proposed a few years later by Wen *et al.* [WAN+21]: In this case, the technique is specialized in recommending developers with the next method to implement, by learning from the change history which methods are usually implemented "together" by developers.

Bruch *et al.* [BMM09] 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. Such a system is implemented using three different variants based on (i) method usage frequency, (ii) association rules discovery, and (iii) an implementation of K-nearest-neighbor clustering named Best-Matching Neighbor (BMN). Their results showed the capability of their technique 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.* [PLM15] who added further contextual information and proposed 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.

Han *et al.* [HWM09] 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 [RL10] used information extracted from the change history of software systems to support the code completion of method calls and class names. In particular, they implemented different strategies that recommend, for example, the methods defined in the same class or the methods that have been recently changed by the developer. 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.

Hou and Pletcher [HP11] evaluated three mechanisms to enhance code completion techniques, namely sorting, filtering, and grouping that they implemented in their Better Code Completion (BCC) tool. This tool allows the user to customize the three mechanisms described before in order to improve the recommendation of API method calls. The outcome of their study is an assessment of the effectiveness of fourteen different configurations of the three mechanisms, measured using the capability of the tool to recommend the correct API in the first positions.

Asaduzzaman *et al.* [ARSH14] proposed a technique named CSCC (Context Sensitive Code Completion). They collected code examples from software repositories and, for each method call, represented its context as a set of methods, keywords, class, and interface names appearing within four lines of code. This contextual information was then used to filter out

method call recommendations. The assumption was that similar contexts implied similar method calls. CSCC outperformed previous approaches, achieving 86% precision and 99% recall.

The rise of statistical language models. Hindle *et al.* [HBS+12] pioneered the work on statistical language models applied to software. They conceived the idea of "naturalness of source code", conjecturing that, since the code has been written by humans, it tends to be repetitive and predictable, similarly to what happens with the English language. For this reason, they used *n*-gram models, a probabilistic language model previously used for predicting English words, to create a language-agnostic algorithm that is able to predict the next token in a given statement. The *n*-gram model is able to exploit these regularities in the code, leveraging the information contained in the previous tokens to predict the next one. The trained model's average entropy is between three and four bits, indicating a high degree of naturalness.

Raychev *et al.* [RVY14] approached the code completion problem through statistical language models, testing the capabilities of *n*-gram models, Recurrent Neural Networks, and their combination. They extracted sequences of method calls from a large code base, and used this dataset to train a language model able to predict API calls. Their best model, which combined both the approaches, achieved a 90% accuracy in the top-3 recommendations.

Nguyen *et al.* [NNN+12] proposed GraPacc, a context-sensitive code completion model trained on a database of API usage patterns. They extracted different patterns from the code, *i.e.,* a set of API elements, like classes and method calls, and control structures that defined the correct way in which developers may combine different API elements to perform a programming task. 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.* [NKZ17] for API completion in Android: Given an API method as a query, their approach recommends a set of relevant API usage patterns. They reported an 18% improvement of F-Measure when compared to pattern extraction using frequent-sequence mining.

Nguyen *et al.* [NN15] 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.

Tu *et al.* [TSD14] introduced a cache component in the *n*-gram model to exploit the "localness of code" (*i.e.,* the fact that some regularities of the code are specific to some files or projects and locally repetitive). Results show that since the code is locally repetitive, localized information can be used to improve performance. The enhanced model outperformed standard *n*-gram models by up to 45% in accuracy. In a related work, Franks *et al.* [FTDH15] implemented *CACHECA*, an Eclipse auto-completion plugin exploiting the aforementioned cache language model [TSD14]. In comparison to Eclipse built-in suggestions, their tool improves the accuracy of top 1 and top 10 suggestions by 26% and 34%, respectively.

Hellendoorn and Devanbu [HD17] proposed further improvements to the cached models proposed by Tu *et al.* [TSD14], aimed at considering specific characteristics of code, like

unlimited, nested, and scoped vocabulary, that can help the model to dynamically adapt to different projects. 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.

Deep learning-based solutions. Differently from Hellendoorn and Devanbu [HD17], Karampatsis *et al.* [KS19], a few years later, suggested 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* [Gag94]. In addition, the proposed neural network is able to dynamically adapt to different projects. Their best model outperformed *n*-gram models, achieving an entropy of 1.03 bits.

Kim *et al.* [KZTC21] leveraged the Transformers neural network architecture for code completion. This novel architecture can process the input code and capture the contextual dependencies across each input token, improving the efficiency and the performance in tasks involving sequential data, like the code generation. 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 experimented with, the best one reached a MRR up to 74.1% in predicting the next token.

Alon *et al.* [ASLY20] addressed the problem of code completion with a language-agnostic approach named Structural Language Model. It leveraged 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.* [SDFS20] introduced IntelliCode Compose, a general-purpose multilingual code completion tool capable of predicting code sequences of arbitrary token types. They did not leverage high-level structural representation, such as AST, and used subtokens to overcome the *out-of-vocabulary problem*. Their model can recommend an entire statement, and achieved a perplexity of 1.82 for the Python programming language.

Liu *et al.* [LLZJ20] presented Code Understanding and Generation pre-trained Language Model (CugLM), a Transformer-based neural architecture pre-trained with the goal of incorporating both code understanding and generation tasks. The prediction of the next token to write has been handled in two separate steps: They first predict the type of the token, while the second exploits this information to predict the actual token. They evaluated their approach on Java and TypeScript datasets, showing state-of-the-art performance.

A problem related to code completion has also been tackled by Watson *et al.* [WTM+20]: The authors exploited a sequence-to-sequence model to recommend assert statements for a given Java test case. They integrated the standard attention mechanism, allowing the model to pay particular attention to specific tokens, with the copy mechanism, used by the model to copy some tokens from the input sequence. This technique is able to generate assert statements with a top-1 accuracy of 31%.

Kanade *et al.* [KMBS20] showed how code embeddings can support code-related tasks, deriving a contextual embedding of source code by training Code Understanding BERT (CuBERT), based on the BERT model. They leveraged the enhanced embedding for different

tasks, ranging from exception type prediction to variable misuse and repair, with an improvement up to 15% of accuracy when compared to the BiLSTM baseline.

Svyatkovskiy *et al.* [SLH+21] 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.

Bhoopchand *et al.* [BRBR16] proposed a neural language model for code suggestion in Python, aiming at capturing long-range relationships among identifiers. To do that, they exploited a sparse pointer network, achieving a lower perplexity and a greater accuracy as compared to the *n*-gram models and LSTM.

Aye and Kaiser [AK20] proposed 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. Their model leverages static analysis and character-level input representation to handle out-of-vocabulary tokens.

Chen *et al.* [CPX+22] 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 model significantly outperformed the existing graph-based statistical approaches and the tree-based deep learning approaches for API recommendation.

Chen *et al.* introduced Copilot [CTJ+21], a new GPT model trained on more than 150Gb of data from GitHub. They evaluated the performance of their model by checking whether the proposed solution can pass all the test cases defined for that method, showing that standard match-based metrics like the BLEU score are not well suited for measuring the accuracy of the model. Their trained model achieved state-of-the-art performance in the demanding task of predicting the whole method when providing as input to the model just a natural language description of what the developer wants to implement.

Feng *et al.* [FGT+20] proposed CodeBERT, a bimodal Transformer trained on code and English text, able to capture semantic connections between natural and programming language. The authors pre-trained the model using the ubiquitous masked language modeling objective, together with the novel replaced token detection, in which the model has to detect the token that has been replaced with a plausible alternative. Despite the authors focus on the problems of code search and code documentation, CodeBERT is suitable for code completion, having been trained using a "masking" pre-training objective.

Wang *et al.* [WWJH21] presented CodeT5, in which the T5 model has been pre-trained with a novel identifier-aware task, where the model has to first tag the identifiers and then predict the masked ones. The authors performed an ablation study showing the contribution of each objective. The semantic information introduced by the novel pre-training objectives allowed CodeT5 to achieve state-of-the-art performance on the CodeXGLUE benchmark. Wang *et al.* [WLG+23] subsequently extended their previous work by presenting CodeT5+, a family of encoder-decoder models where the modules can be easily combined to handle specific tasks, thanks to a plethora of pre-training objectives used in the training.

Finally, Izadi *et al.* [IGG22] presented CodeFill, a Transformer-based approach trained for single- and multi-token code completion by predicting both the type and value of the

masked tokens. Thanks to the capability of exploiting the token type information when predicting token values, CodeFill outperformed the previous techniques in terms of ROUGE and METEOR scores.

2.2 Empirical Studies on Source Code Recommender Systems

Most of the empirical studies in the literature focus on assessing the effectiveness of code recommenders and how they are used by developers. Proksch *et al.* [PANM16] evaluated a method-call recommender system on a real-world dataset featuring interactions captured in the IDE. They observed that commonly used evaluations based on synthetic datasets extracted *a-posteriori* from released code do not take into account context change: This has a major effect on the prediction quality.

In a similar research thread, Hellendoorn *et al.* [HPGB19] 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. They found that the experimented tools are less accurate on the real-world dataset, showing that synthetic benchmarks are not representative enough. Moreover, they found that such tools are less accurate in challenging scenarios, when developers would need them the most.

Mărășoiu *et al.* [MCB15] studied how developers use code completion in practice. They observed that many recommendations are not accepted by the users. The main reason for this low acceptance rate is due to the developers' limited familiarity with some of the recommended APIs. Similar findings have been reported by Arrebola and Junior [AJ17], who advocate for more context-awareness. They showed that half of the interactions did not produce any benefit for developers, since they are dismissed or interrupted.

Jin and Servant [JS18] examined the *hidden costs* of code recommendations, investigating 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, since the code completion tool they evaluated (IntelliSense) sometimes provides the right recommendation far from the top of the list. They observed that longer lists discourage developers from selecting a recommendation.

Lin *et al.* [JLJ19] focused on the performance of a *code2vec* [AZLY19] model in the context of method name recommendation. The authors retrained the model on a different dataset and assessed it in a more realistic setting where the training dataset did not contain any record from evaluation projects. The results suggested that while the dataset change had little impact on the model's accuracy, the new *project-based* setting negatively impacted the model. Jiang *et al.* [JLJ19] also evaluated the usefulness of *code2vec* suggestions by asking developers to assess the quality of suggestions for non-trivial method names. The evaluation results showed that 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. To confirm such findings, the authors developed a simple

heuristics-based approach. The comparison revealed the superiority of a simple heuristics-based approach over a state-of-the-art machine learning based approach.

Liu *et al.* [LSZ+20] investigated 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 attributed this observation to the relatively small datasets on which such models are evaluated. This study highlighted the importance of a database both in terms of size and the quality of data sources.

Similarly, Aye *et al.* [AKL21] investigated 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 significantly improved the accuracy of the models, with an increase of more than 12%. 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, with users that are more prone to accept the new recommendations.

Xu *et al.* [XVN22] ran a controlled experiment with 31 developers who were asked to complete implementation tasks with and without the support of two code recommenders. They found no significant gain in developers' productivity when using the code recommenders. Further analysis revealed several points that can help improve the effectiveness of the recommenders, like reporting more contextual information in the output to help developers in understand the suggestion or providing a unified and intuitive query syntax, that can help developers to better describe their intent.

Ziegler *et al.* [ZKL+22] ran a survey with Copilot users to investigate which quantitative measure better captures their perceived productivity when using the tool. They found that the acceptance rate of shown suggestions is the best predictor of perceived productivity, since the suggestions, although not always correct, can also be a useful template to start from.

Different studies investigate the effectiveness of contextual information in improving recommender systems for developers. Gail Murphy suggested that contextual information "*could enable software tooling to take a substantial step forward*" [Mur19]. They proposed different kinds of contexts that may be beneficial, like historical information, that can highlight the part of the systems that might need a change, or dynamic execution information, that leverages debugging information. The authors also emphasize the major room of improvement, due to the limited usage of context in current IDEs.

Tian and Treude [TT22] presented a preliminary study in which they evaluated how additional contextual information provided as input to a DL may improve its performance for clone detection and code classification. They experimented with the context from the method's call hierarchy, adding the caller methods and the callee methods, showing an improvement of up to 8%.

Relevant to this thesis is also the blog post by Albert Ziegler [zie], who investigated the

extent to which GitHub Copilot suggestions are copied from the training set. He concluded that Copilot rarely recommends pieces of code that are verbatim copies of instances from the training set, and when this happens these are coding solutions recurring across several systems, that also humans are likely to reuse.

2.3 Summary and Conclusions

We discussed the papers mostly related to the research presented in this thesis. It is worth mentioning that several of these works, especially those presenting DL-based solutions for code completion, have been published during the four years of my PhD (*e.g.,* GitHub Copilot [CTJ+21] was presented in 2021), and some of these works stemmed from the research presented in the following chapters.

When we started investigating the topic, the usefulness of code completion techniques was mostly tested in predicting single (or very few) code tokens. The advent of DL-based solutions made these evaluations obsolete, since DL models can achieve excellent performance in this prediction scenario. To the best of our knowledge, we were the first to investigate the usage of large pre-trained models to predict non-trivial code components, like entire code blocks (see Chapter 3). The experimental design we adopted inspired several subsequent works in the field (see *e.g.,* [LRA23, CFLB22, NYN22]).

We also presented the first work assessing the extent to which DL-based solutions for code completion tend to perform verbatim copies of the training code when generating predictions, thus leading to potential licensing issues (Chapter 4). This was the first thorough empirical investigation about the phenomenon, whose only evidence was available in blog posts [zie].

Similarly, although there was preliminary evidence of the important role played by contextual information (*i.e.,* the input information provided to the model for generating the prediction) in improving the accuracy of code recommendations [TT22], there were no extensive studies focusing on this aspect before the one we present in Chapter 7.

Finally, this thesis provides the research community with several future research directions in the field that have been distilled by surveying 80 practitioners asking them which characteristics of code recommender systems they consider important (Chapter 6).

3

An empirical study on the usage of transformer models for code completion

3.1 Motivation

In this study, 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 [VSP+17], namely the RoBERTa model [LOG+19] and the Text-To-Text Transfer Transformer (T5) architecture [RSR+20]. RoBERTa is a BERT (Bidirectional Encoder Representations from Transformers) model [DCLT19] 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. The T5 model [RSR+20] leverages *transfer learning*, *i.e.,* the capability to reuse the knowledge acquired by the model in one task 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). 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).

To compare the performance of the two Transformer models presented before, we selected the *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 [HD17].

We focus on three different code completion scenarios, aiming at emulating the way in which developers complete their code: (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).

The main finding of our work is the substantial superiority of the T5 model, being able to correctly predict even entire code blocks, something that we found to be not achievable with RoBERTa. 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 achieves 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.

In our work, we also 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 source code and data used in our work are publicly available in a comprehensive replication package [repa].

3.2 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 [DM12], Levenshtein distance [Lev66]) and a qualitative (*i.e.,* correct 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 a 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, and 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 3.1, 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 [HD17].

3.2.1 Context Selection: Datasets

Our study involves two datasets. The first one 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 [HWG+19], 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.

Fine-tuning dataset

To create the *Java dataset*, we started from the CodeSearchNet Java Dataset provided by Husain *et al.* [HWG+19]. We decided to start from CodeSearchNet rather than from other datasets proposed in the literature (see *e.g.,* [MAL18, RvDJ13]) 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 [MAL18], 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.* [HWG+19]. 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 [ded] 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 [GMP+18]. 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 634,799 Java methods and 532,096 Android methods.

Then, the three versions of each dataset (Java and Android) summarized in Table 3.1 were created using the following masking processes (note that Table 3.1 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 $1 \ldots n-1$, 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 [src] 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 $S=\{T_1, T_2, \ldots, T_n\}$, 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.

Table 3.1. 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

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 picked 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 3.1. As a first step, we filtered out specific instances from our datasets. First, when using generative deep learning models, the variability in the 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.* [TWB+19], 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.

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 3.1. 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 made 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.

Pre-training dataset

To build the pre-training dataset, we used the GitHub Search platform [DAB21] 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 3.2.1), 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.

3.2.2 Context Selection: Techniques

In this section, we overview three experimented techniques, *i.e.,*RoBERTa [LOG+19], T5 [RSR+20], and *n*-gram [HD17]. We refer to the original papers presenting them for additional details.

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.* [LOG+19], 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 [VSP+17]. 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 [DCLT19]. The result will be a single model able to support different tasks and, possibly, take 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 3.1.

As for the implementation of the RoBERTa model, we used the one provided in the Python *transformers* library [WDS+19]. 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) [Gag94] model using the HuggingFace's *tokenizers* Python library [hug]. 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 [KBR+20], BPE has been shown to address the *out-of-vocabulary problem*.

T5

Raffel *et al.* [RSR+20] 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 [RSR+20]: 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.* [RSR+20] 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 [RSR+20].

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 [HD17].

3.3 Data Collection and Analysis

In this section, we detail the data collection and analysis procedure adopted to answer the research questions described in Section 3.2.

3.3.1 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 3.2. 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

RoBERTa

We performed hyperparameter tuning using the Weights & Biases's [wan] Python library on a Linux server with an Nvidia RTX Titan GPU. Table 3.2 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 "correct predictions". We define as "correct" 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 "correct". 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 ranges 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 computed the number of correct predictions. If we observe that, during

the training, the performance of the model is worsening in terms of correct 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 correct predictions on the evaluation set is lower than the number of correct 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 correct predictions on the evaluation test) obtained up to that moment. None of the models were trained for the whole 50 epochs.

Table 3.3. 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)	$LR_{starting} = 0.01$
	$LR_{end} = 1e{-}06$
	$Power = 0.5$

T5

We rely on the same configurations used by Mastropaolo *et al.* [MSC+21]. 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 3.3, and identify the best-performing configuration in terms of correct 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 3.1), the best performing configuration was the one using the Slanted Triangular learning rate, confirming the findings in [MSC+21]. Also, all T5 models we built use a *SentencePiece* [KR18] tokenizer trained on the pre-training dataset and are composed of 32k word pieces [MSC+21].

The best configuration we identified has been used to train six different T5 models (*i.e.,* one for each dataset in Table 3.1) 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 used 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 [RSR+20]. 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 3.1). 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.

Table 3.4. 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 correct predictions	To what extent is the approach able to generate predictions that need human intervention

3.3.2 Analysis of Results

To answer RQ_1 we compute the metrics summarized in Table 3.4 by running each trained model on the test sets in Table 3.1.

The first metric, *Bilingual Evaluation Understudy (BLEU)-n score*, assesses the quality of automatically translated text [DM12]. 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 [GZZK16, JAM17, WTM+20]. 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 [Lev66]. 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 correct 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 [Dod08] (p-values<0.001). Whenever an analysis requires running multiple test instances, we adjust p-values using the Benjamini-Hochberg procedure [YY95].

To (pairwise) compare the correct predictions of RoBERTa and T5, we use the McNemar's test [McN47], 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 correct prediction, (ii) only T5 provides a correct prediction, (iii) only RoBERTa provides a correct prediction, and (iv) neither T5 nor RoBERTa provides a correct 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 different masking levels, the number of correct and incorrect predictions achieved on Java and Android.

To compare the results of T5 and RoBERTa in terms of BLEU-n score and Levenshtein distance, we use the Wilcoxon signed-rank test [Wil45] and the paired Cliff's delta [GK05] effect size. Similarly, the comparison between datasets in terms of BLEU-n score and Levenshtein distance, being unpaired, is performed using the Wilcoxon rank-sum test [Wil45] and the unpaired Cliff's delta effect size.

For the T5, we also statistically compare the performance achieved (i) with/without pre-training, and (ii) with/without transfer learning. Also in this case, McNemar's test is used to compare correct 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.,* the first interval includes all predictions having a confidence c with $0 \leq c < 0.1$, the last interval has $0.9 \leq c$). Then, we report for each interval the percentage of correct predictions.

To corroborate our results with a statistical analysis, we report the OR obtained by build-

ing a logistic regression model relating the confidence (independent variable) with the extent to which the prediction achieved a correct 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 [Ken38], 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 [HD17] using the implementation made available by the authors [slp]. 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 correct predictions generated on the test sets. A statistical comparison is performed using the McNemar's test [McN47] and ORs.

3.4 Results Discussion

We start by contrasting the performances of T5 and RoBERTa (Section 3.4.1). Then, we show how the *n*-gram model compares with the DL-based ones (Section 3.4.2). Finally, Section 3.4.3 presents qualitative examples of correct predictions made by the models and discusses the semantic equivalence of non-correct predictions.

Note that, upon interpreting the achieved results, and especially those concerning the 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.

3.4.1 DL-based models performance comparison (RQ_1)

Fig. 3.1 depicts the results achieved by DL-based models in terms of correct predictions for different masking approaches, namely (from left to right) *token-masking*, *construct-masking*, and *block-masking*. The plots show the percentage of correct predictions (y axis) by the

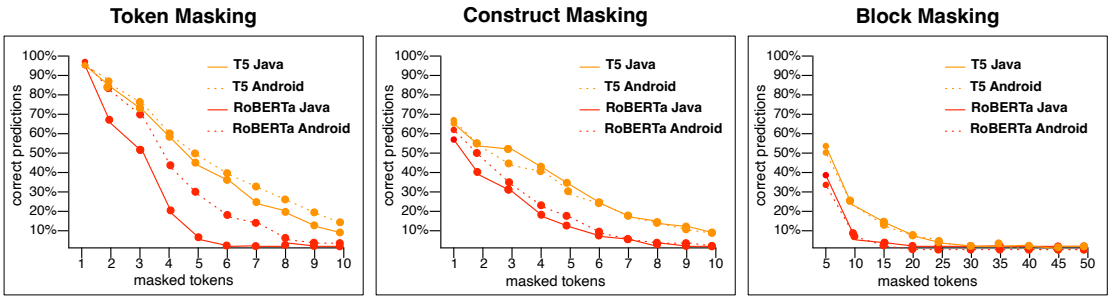

Figure 3.1. Percentage of correct predictions achieved by T5 and RoBERTa

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. 3.1 shows the percentage of correct 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 3.5 reports the average BLEU score in the four considered variants and the average normalized Levenshtein 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. 3.1 and Table 3.5 are achieved by the DL-based models in the simplest scenario, *i.e.,* single-task without pre-training. 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 3.6 as the percentage of correct predictions for different variants of the T5 model, *i.e.,* with/without pre-training and using single- and multi-task fine-tuning. Table 3.6 also reports the results achieved with the RoBERTa model in the simplest scenario to simplify the discussion of the results.

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. 3.1: (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. 3.1) 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 3.10); (iii) the T5 model (orange lines in Fig. 3.1) substantially outperforms RoBERTa (see statistics in Table 3.7 and Table 3.8). Also, the performance of RoBERTa drops more steadily as compared to that of T5 when the number

Table 3.5. BLEU score and Levenshtein distance for T5 and RoBERTa.

Token masking				
	Java		**Android**	
	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				
	Java		**Android**	
	T5	**RoBERTa**	**T5**	**RoBERTa**
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				
	Java		**Android**	
	T5	**RoBERTa**	**T5**	**RoBERTa**
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

of masked tokens increases.

Table 3.7 reports results of the McNemar's test and ORs for the comparison between T5 and RoBERTa in terms of their ability to perform correct 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 in providing a correct prediction than RoBERTa.

Concerning the comparison of BLEU scores or Levenshtein distances (whose average values are reported in Table 3.5) between T5 and RoBERTa, statistical results (Wilcoxon signed-rank test adjusted p-values and Cliff's d) are in Table 3.8 and Table 3.9. 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. 3.1 shows that, as expected, the lower the number of masked tokens the higher the correct 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 correct predictions for RoBERTa on the Java dataset drops to less than 10%, a major gap with the T5 model that keeps a percentage of correct 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,

Table 3.6. Correct predictions of T5 models with different fine-tuning strategies, and RoBERTa model

Dataset and Masking Level		T5			RoBERTa
		With Pre-training		No Pre-training	No Pre-training
		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 3.7. Correct 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

thus, more subject to regularities in the source code. However, the gap in terms of correct predictions between the Java and the Android dataset is much more marked for the RoBERTa model (*e.g.,* ∼20% at $x = 5$ against a ∼6% for the T5).

Looking at Table 3.5, the BLEU scores and the Levenshtein distance confirm what was observed for correct 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 statistical results are in the online appendix).

Construct masking. In this scenario (see central sub-graph in Fig. 3.1), T5 and RoBERTa achieve respectively above 65% and 55% of correct 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 3.10, 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

Table 3.8. BLEU score 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	
		***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)
	Construct	<0.001	0.22 (S)	<0.001	0.30 (S)	<0.001	0.32 (S)	<0.001	0.35 (M)
	Block	<0.001	0.39 (M)	<0.001	0.43 (M)	<0.001	0.47 (M)	<0.001	0.49 (L)
Android	Token	<0.001	0.17 (S)	<0.001	0.21 (S)	<0.001	0.27 (S)	<0.001	0.34 (M)
	Construct	<0.001	0.14 (N)	<0.001	0.20 (S)	<0.001	0.22 (S)	<0.001	0.27 (S)
	Block	<0.001	0.33 (M)	<0.001	0.39 (M)	<0.001	0.42 (M)	<0.001	0.44 (M)

Table 3.9. Levensthein distance comparison between T5 and RoBERTa: Wilcoxon signed-rank and Cliff's delta (N: negligible, S: small, M: medium, L: large)

Dataset	Masking	Levenshtein	
		***p*-value**	***d***
Java	Token	<0.001	-0.32 (S)
	Construct	<0.001	-0.21 (S)
	Block	<0.001	-0.38 (M)
Android	Token	<0.001	-0.17 (S)
	Construct	<0.001	-0.14 (N)
	Block	<0.001	-0.34 (M)

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 (the maximum boundary we set). The performance of T5 and RoBERTa in terms of correct 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. 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 correct prediction for the T5 model is composed of 36 and 39 tokens for Android (see Fig. 3.2) and Java datasets respectively, compared to the 13 and 15 tokens achieved with the RoBERTa model.

At this level (see Table 3.10), 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 3.5), and the devel-

```
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); } }
```

Figure 3.2. Correct prediction of 36 tokens generated by T5 in the Android dataset

Table 3.10. Comparison between different datasets for correct 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

oper 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 pre-training 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.*

Impact of pre-training and transfer learning (RQ_2)

As explained in Section 3.3.1, we trained seven additional T5 models to assess the impact of pre-training and transfer learning on its performance. First, we added to the six models for which we previously discussed the T5 performance (*i.e.,* no pre-training, single-task) the pre-training 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 3.6 shows the achieved results also reporting the performance of the previously discussed T5 and RoBERTa models (*i.e.,* no pre-training, single-task in Table 3.6). Results of a statistical comparison made using McNemar's test are reported in Table 3.11. As it is shown, the pre-training 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 pre-training. Looking at Table 3.6, the pre-training had a positive impact on the accuracy of T5, boosting the percentage of correct predictions from 1% to 4.7%, depending on the test dataset. The benefit of pre-training is more evident in the most challenging block-level scenario ($\sim$5%). Overall, when considering all test datasets as a whole, the percentage of correct predictions increases from 54.1% to 56.2% (+2.1%).

By training a single model on the six training datasets, the percentage of correct 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 $\sim$5%.

The performance improvement is also confirmed by the results achieved in terms of BLEU score and the Levenshtein distance that we report in our replication package [repa].

Answer to RQ_2: *We found both pre-training and multi-task fine-tuning to have a positive*

Table 3.11. Effect of different pre-training levels for T5: McNemar's test results. None indicates the T5 model with no pre-training and single-task fine-tuning. Single and Multi indicate 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

impact on the T5 performance. Overall, such an improvement accounts for +5.2% in terms of correct predictions (36,009 additional instances correctly predicted).

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. 3.3 reports the relationship between the percentage of correct predictions and the confidence of the model. The orange line shows the percentage of correct 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 correct predictions that are due to predictions in that confidence interval out of the total (*e.g.,* 78% of all correct predictions have a confidence higher than 0.9).

Fig. 3.3 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.

From a statistical perspective, a logistic regression model correlating the confidence level and the correct prediction outcome indicates a statistically significant (p-value <0.001) cor-

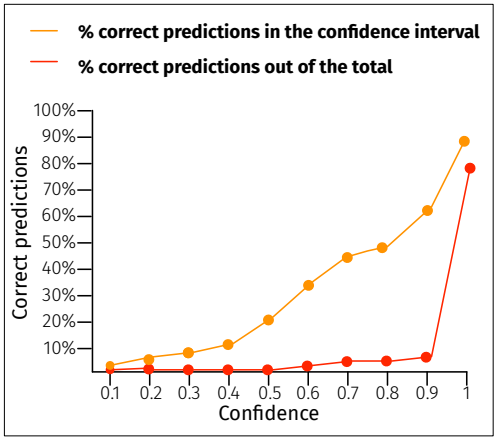

Figure 3.3. Correct predictions by the confidence of the model

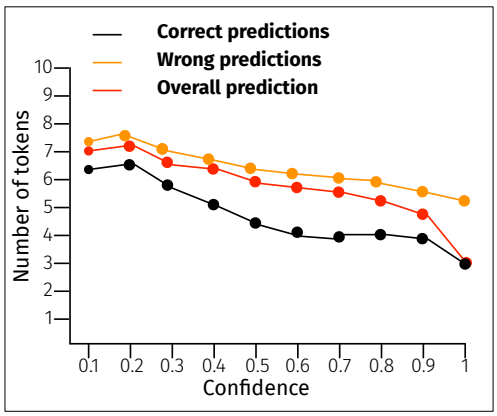

Figure 3.4. Average length (in tokens) of the predictions by confidence

relation, and an estimate of 6.58, which means 720 higher odds of a correct prediction for each unit increase of the confidence, *i.e.,* 72 higher odds of a correct prediction for a 0.1 increase of the confidence, *i.e.,* a tick on the x-axis of Fig. 3.3.

Fig. 3.4 analyzes the average length, in tokens, of the correct 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 correct predictions for the highest confidence interval (*i.e.,* 3 tokens) is much lower than the average number of tokens in correct 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 3.12. Correct predictions of the three models

Dataset and 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%

Table 3.13. 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

3.4.2 Comparison with an n-gram Model

We answer RQ_3 by comparing the DL-based models without pre-training 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 3.12 reports the comparison in terms of correct predictions between T5, RoBERTa, and the *n*-gram model in different evaluation scenarios, as well as the overall results. For example, T5 produced 61% correct predictions on the Java dataset when using token masking. The results of statistical tests (McNemar's test) are in Table 3.13.

One important clarification is needed to properly interpret the results of Table 3.12. 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 3.12 and Fig. 3.1.

Table 3.13 reports the 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 3.12) 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 correct predictions, with the T5 performing better achieving more than twice the number of correct 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 [HD17], which can leverage information about other code components coming from the same project (*e.g.,* same file or package [HD17]) of the method in which the prediction is performed. This is one of the advantages of the cache model [HD17] 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 3.14. Correct 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. [HD17]). If the method changes, 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 compared the performance of the *n*-gram model when such additional information is provided (and not) on these instances.

Table 3.14 reports the achieved results. As expected, the performance of the *n*-gram model increased 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.*

3.4.3 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. 3.5 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 [repa].

Given the achieved results showing the superiority of the T5 model, we had a better look at a sample of the 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 correct predictions of the T5 spanned across dozens of tokens, being almost unrealistic, we checked whether the 21 correct 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 transposition 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

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++; }
```

Figure 3.5. Examples of correct predictions generated by T5

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.

3.5 Threats to Validity

Threats to *construct validity* concern the relationship between theory and observation. One threat, also discussed by Hellendoorn *et al.* [HPGB19], 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 3.2.1, 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 correct prediction. However, the BLEU score may be sufficient to assess the performance of code-related tasks [RGL+20] and, in general, it is difficult to evaluate the usefulness of semantic equivalent predictions or incorrect 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 3.3.2. 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 3.3.2 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 [HD17]. 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.* [HWG+19]. 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.

3.6 Conclusion

We investigated the ability of Transformer-based DL-models in dealing with code completion tasks having 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 [RSR+20], RoBERTa [DCLT19], and the cached *n*-gram model [HD17], the T5 resulted in being 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 [copb]) and different training strategies could help in achieving better results (*e.g.,* Tufano *et al.* [TDS+20] 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 correct 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.

4

To what extent do deep learning-based code recommenders generate predictions by cloning code from the training set?

4.1 Motivation

The rise of DL to support code-related tasks was possible thanks to the unprecedented amount of development data being publicly available (*e.g.,* from GitHub) and that can be used for training the DL models. The vast majority of this data comes from open-source repositories and hence its usage is regulated by specific Free and Open Source Software (FOSS) licenses (*e.g.,* Apache 2.0[1], GPL v3.0[2]). These licenses regulate how the licensed source code can be redistributed or modified to create *derivative work*. However, they were not meant to define whether (i) the licensed code can be used for training DL-based code recommender, nor (ii) the code recommended by the trained models should be considered as *derivative work* or, instead, as new code "written from scratch". Such a discussion is part of a non-trivial debate that will likely result in the update of FOSS licenses in the near future.

From the studies previously presented in this thesis, it is clear that DL models achieved astounding results, being able to correctly predict even two Java statements. Moreover, the significant dependence of the model's performance on the Java version used for the training may be due to the capabilities of the model to "memorize" some version-dependent constructs without being so good at generalizing to similar ones. If DL-based code recommenders tend to clone code snippets from the training set when generating predictions, it would be difficult to consider the recommendations as new and original code (rather than as derivative work). Although it is reasonable that the generated recommendations are influenced by the training dataset (*e.g.,* by often reusing frequent code identifiers), the problem would be more relevant if these techniques perform a verbatim copy of the training code, possibly spanning several statements.

In this thesis, we are interested in investigating the following research question (RQ):

[1] `https://www.apache.org/licenses/LICENSE-2.0`

[2] `https://www.gnu.org/licenses/gpl-3.0.html`

To what extent do DL-based code recommender systems generate predictions by cloning code snippets from the training set?

To answer our research question, we need: (i) a DL-based code completion technique able to recommend several tokens/statements; (ii) access to the dataset(s) used to train the DL model; and (iii) access to the predictions generated by the technique on a test set. Unfortunately, such an analysis is not possible using GitHub Copilot since the training dataset is not publicly available. For this reason, we adopt the Text-To-Text-Transfer-Transformer (T5), a DL model that has been used to support several code-related tasks [MSC+21], including code completion [CCP+21a].

We start from the pre-trained T5 made available by Ciniselli *et al.* [CCP+21a], who also made publicly available the pre-training dataset. The model has been pre-trained on ∼2M Java methods. Then, we fine-tuned this model on a new dataset we built featuring 841,318 Java methods to train the model in predicting non-trivial code snippets composed of several statements. Indeed, predictions only spanning a few code tokens are not interesting for our goal, since they cannot really be defined as "clones" even if copied from the training dataset. We built four versions of the fine-tuning dataset, each one applying a different strategy to remove near duplicates [All19], including more and less strict strategies. Indeed, one of the assumptions we want to test is that a higher presence of near-duplicates in the training set pushes the model to copy more, since it observes over and over the same pattern.

Once trained the four models on the different datasets, we used them to generate predictions on a test set of ∼16k instances (*i.e.,* ∼16k Java methods with missing code statements that must be guessed by the model). Then, we used the SIMIAN clone detector [sima] to identify Type-1 (*i.e.,* exact copies) and Type-2 (*i.e.,* copied code with changes to identifiers and types) clones between the generated predictions and both the pre-training and fine-tuning training datasets. Also, we checked whether the model tends to "copy" more when (i) dealing with predictions having different lengths, since we expect smaller predictions to more likely result in clones; and (ii) generating correct or wrong predictions.

The achieved results show that ∼10% of the generated predictions represent a Type-1 clone of code in the training data, with this percentage going up to ∼80% when considering Type-2 clones. However, the percentage of clones steadily drops when the model is asked to predict several statements, with basically no clones generated when the prediction comprises at least four lines of code.

We also found that correct predictions are more likely to contain clones of code present in the training datasets. Such a result stresses once more the importance of a proper dataset cleaning to avoid overlaps between the training and the test datasets.

4.2 Study Design

The *goal* of this study is to understand the extent to which DL-based code recommenders are prone to suggest code snippets being *clones* of instances present in the training set. The *context* is represented by the four training datasets described in Section 4.2.2 and by the

snippets of code generated by a state-of-the-art code recommender built on top of the Text-To-Text-Transfer-Transformer (T5) [CCP+21a].

Our study aims at answering the following research question:

> **RQ.** *To what extent do DL-based code recommender systems generate predictions by cloning code snippets from the training set?*

Our RQ sheds some light on the implications of using recommendations generated by DL-based code recommenders in terms of possible licensing issues occurring when using the generated recommendations in the development of commercial software. We answer our RQ by training a DL-based code recommender to then use it in thousands of code completion tasks (*i.e.,* an incomplete code is provided as input to the model, that is required to generate the missing code). The generated recommendations are then compared to the instances in the training set looking for clones.

4.2.1 Study Context: DL-based Code Recommender

The first step in the selection of the study context is the code recommender to use. The latest and likely more effective code recommender available, namely GitHub Copilot [CTJ+21], was not an option for our study. Indeed, while we have access to Copilot, the dataset used to train it is not publicly available, making it impossible to check whether the code it recommends is a clone of the instances in its training set. For this reason, we focused on another DL-based code recommender recently proposed by Ciniselli *et al.* [CCP+21a].

The authors trained a T5 model supporting code completion in Java at method-level granularity. This means that once trained the model can take as input an *incomplete* Java method (*i.e.,* a method missing one or more contiguous code tokens) and it can predict the missing tokens. The T5 has been trained in two phases. The first, named pre-training, aims at providing the model with a general knowledge about the language of interest. Ciniselli *et al.* [CCP+21a] used a classic denoising pre-training task in which the model has been fed with ∼2M Java methods having 15% of their tokens (even non-contiguous ones) randomly masked, with the model asked to predict them. Once pre-trained, the model has been fine-tuned to support code completion at different levels: (i) *token-level*, in which the last x tokens (with $1 \leq x \leq 10$) of a given statement have been masked and must be predicted; (ii) *construct-level*, in which the masking has been focused on specific code constructs, such as the conditions of `if` statement or of a `while/for` loop, the parameters of method calls, etc. (see [CCP+21a] for details); and (iii) *block-level*, in which entire code blocks[3] up to two statements are masked.

The fine-tuning has been performed on different datasets, for a maximum of 750k instances (methods) involved.

In the reported evaluation, the T5 achieved correct predictions ranging from ∼29%, obtained when asking the model to guess entire blocks, up to ∼69%, reached in the simpler scenario of a few tokens masked from the same code statement [CCP+21a].

[3] A code block is defined as the code enclosed between two curly brackets.

Table 4.1. Threshold (Th) configurations and number of methods after the filtering.

Name	Set Th.	Multiset Th.	# of Methods
Very Weak	0.8	0.7	838k
Weak	0.6	0.5	711k
Strong	0.4	0.3	483k
Very Strong	0.3	0.2	273k

We start from the pre-trained model made available by Ciniselli *et al.* in their replication package [repa], but we fine-tune it from scratch on a new dataset described in Section 4.2.2. Indeed, the fine-tuned model used by Ciniselli *et al.* [CCP+21a] was not appropriate for our study, since they trained the T5 to predict at most two code statements. In our work, we are interested in identifying code clones in the generated predictions and, for this reason, we want to train the T5 to generate also longer predictions. Indeed, clones represented by a few code tokens or a single statement are unlikely to be relevant, but mostly due to the syntactic sugar of programming languages.

4.2.2 Study Context: Datasets Construction

To create our fine-tuning dataset we started from the CODESEARCHNET dataset provided by Husain *et al.* [HWG+19]. CODESEARCHNET features over ∼1.5M Java methods collected from open source non-forked GitHub repositories. On top of them, we added the ∼2.2M Java methods made available by Ciniselli *et al.* [CCP+21a] and collected from GitHub Android repositories. From the overall set, we removed methods (i) having less than three lines of code, since they do not allow training the model on "code completion scenarios" that are interesting in the context of code clones (*i.e.,* when asking the model to generate several missing code statements); (ii) having the name containing the *test* substring in an attempt to remove test methods; and (iii) containing non-ASCII characters (*e.g.,* Chinese text) to avoid confusing the model during training. We then excluded all methods having a duplicate in the pre-training dataset used by Ciniselli *et al.*, obtaining the final set of 841,318 instances.

As a final step, we applied the approach defined by Allamanis [All19] and Lopes *et al.* [LMM+17] and used in the construction of the CODESEARCHNET database [HWG+19] to identify near-duplicates in our fine-tuning dataset. The idea behind this approach is to look at the percentage of overlapped tokens between two instances (*i.e.,* methods) and identify them as near-duplicates if such an overlap exceeds a specific threshold. The code implementing such an approach is publicly available [ded], and allows setting two thresholds making the identification of near-duplicates weaker (*i.e.,* only very similar methods are considered clones) or stronger. Both thresholds define the percentage of shared tokens needed to consider two methods as duplicates. However, the first threshold (*Multiset Threshold*) considers the entire list of tokens in a method (including duplicates), while the second (*Set Threshold*) is computed on the unique set of tokens in the compared methods.

We experiment with the four different combinations of thresholds reported in Table 4.1, that resulted in four different fine-tuning datasets composed by the *# of Methods* instances not considered near-duplicates (out of the starting ∼850k). In the following, we refer to the four combinations as *very weak* (*i.e.,* the combination of thresholds rarely reports methods

as near-duplicates), *weak*, *strong*, and *very strong* (*i.e.,* the combination of thresholds frequently reports methods as near-duplicates) and to the resulting datasets as $dataset_{very\_weak}$, $dataset_{weak}$, $dataset_{strong}$, and $dataset_{very\_strong}$. The choice of experimenting with four different combinations was dictated by the following assumption: It is possible that a training dataset featuring several near-duplicates (such as $dataset_{very\_weak}$) pushes the model to "copy" more from the training set as compared to a dataset featuring almost no near-duplicates ($dataset_{very\_strong}$). Indeed, seeing a code snippet over and over across several training samples may push the model to use that coding pattern more when generating the predictions. This is just an assumption we validate in our study.

As described in Section 4.2.3, the four datasets have been used to fine-tune four T5 models starting from the pre-trained model by Ciniselli *et al.* [CCP+21a].

4.2.3 Model Training

We do not discuss the architectural details of T5, which have been widely documented by Raffel *et al.* [RSR+20] to better focus on implementation choices that are relevant for this study. The specific architecture we use is the $T5_{small}$ presented in [RSR+20].

Tokenization. T5 uses a tokenizer to pre-process the input and generate the stream of output tokens. As it will be clear later, in our study the model will be fed with a Java method in which specific code statements have been masked (input) and will be asked to generate the masked statements (output). The adopted tokenization strategy impacts the vocabulary that the model can exploit to generate the output. Previous work studied the pros and cons of different tokenization strategies, spacing from word- [DGMH+23] to character-level [MM04] tokenization. The former is prone to the out-of-vocabulary problem [KBR+20] while the latter struggles with long inputs. To mitigate these limitations, Sennrich *et al.* [SHB16] introduced the Byte Pair Encoding (BPE) tokenization that splits the input into a sequence of subtokens (*e.g.,* it can split the word "*string*" into two parts "*str*" and "*ing*", where "*str*" is the root and "*ing*" is the specialization). Through this sub-splitting, BPE has the advantage of reducing the vocabulary size (as well as the out-of-vocabulary problem) and allows the model to generate composed words. For example, when applied to source code the model can generate unseen identifiers whose name is a composition of the sub-tokens present in the vocabulary. The T5 exploits a *SentencePiece* [KR18] tokenizer which uses the same idea behind BPE. Ciniselli *et al.* [CCP+21a] trained a 32k-word SentencePiece tokenizer on the pre-training dataset. Since we reused their pre-training dataset, we also reused their trained *SentencePiece* tokenizer.

Datasets masking and splitting. To train T5 with the goal of generating non-trivial snippets of code composed by multiple statements, we adapted one of the *block-level* masking scenarios proposed by Ciniselli *et al.* [CCP+21a].

In particular, we apply the following processing to each instance (*i.e.,* Java method) in each of the four datasets (*i.e.,* $dataset_{very\_weak}$, $dataset_{weak}$, $dataset_{strong}$, and $dataset_{very\_strong}$). In each method, we identify all blocks (*i.e.,* code snippets enclosed between two curly brackets) composed by at least two and at most six statements. Given a method including n of such blocks, we generate n versions of it, each having one of the n blocks masked with

Table 4.2. Study datasets. We reported the number of instances for the training and the number of distinct methods.

Filtering Level	Dataset	#Instances	#Distinct Methods
Very Weak	Training	1.2M	797k
Weak		1.02M	671k
Strong		703k	446k
Very Strong		390k	240k
All levels	Evaluation	15,783	8,560
	Test	15,742	8,506

the special token <EXTRA_ID_0>. During training and testing the model is then fed with methods including a masked block and it is required to generate it. The lower and upper bound for the block size (*i.e.,* two and six lines) are defined, based on our experience, to obtain code generation tasks that are non-trivial (at least two statements), relevant in the context of cloning, and addressable given the employed DL model, training dataset, and hardware resources (at most six statements). To account for trivial statements, we count only statements composed by at least two characters to exclude, for example, statements including only a closing parenthesis. Thus, a block composed by one 10-character statement and one 1-character statement is not considered in our study, since it does not match the "two statements" lower bound.

Starting from these instances, we created the training, evaluation, and test datasets in Table 4.2. First, it is important to clarify that the splitting across training, evaluation, and test sets was not random to avoid biasing the learning. To explain this point, let us go back to the way in which we created the instances (*i.e.,* Java methods with a masked code block) for our dataset. Given a method m having n blocks composed by two to six statements, we added in the dataset n 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, the splitting was done at "method-level" rather than "instance-level". This means that all instances related to a method m can belong to only one of the three sets (training, evaluation, testing). The number of unique methods in each set is reported in Table 4.2 in the *# Distinct Methods* column, while the number of instances these methods generated is represented in the *# Instances* column.

Second, as previously said, in our study we want to investigate whether the prevalence of near-duplicates in the training dataset (*i.e.,* the different filtering levels we applied, from *very weak* to *very strong*) plays a role in the generation of code clones. For this reason, while the training sets for the four datasets can be different, with $dataset_{very\_weak}$ being the largest one (*i.e.,* containing more near-duplicates), we wanted to have the same evaluation and test sets for all datasets, to allow for a fair comparison. For this reason, we built the

Table 4.3. 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)	$LR_{starting} = 0.01$ $LR_{end} = 1e{-}06$ $Power = 0.5$

Table 4.4. Number of fine-tuning steps and best checkpoint found.

Filtering Level	# Training Steps	Best Checkpoint
Very Strong	152k	20k
Strong	275k	40k
Weak	400k	225k
Very Weak	470k	450k

evaluation and the test dataset with ∼15k instances each. Those instances meet the following requirements: (i) they have been extracted from the methods that were present in all four datasets, independently from the near-duplicates filtering level that has been applied; and (ii) they are balanced in terms of prediction difficulty, featuring ∼3k instances in which a two-statement block is masked, ∼3k in which a three-statement block is masked, etc., up to six-statement blocks masking.

Hyperparameters tuning. We started from the pre-trained T5 model by Ciniselli *et al.* [CCP+21a]. Then, to find the best T5 configuration for the fine-tuning, we followed the hyperparameters tuning procedure previously used by Mastropaolo *et al.* [MSC+21]. In particular, we trained four different configurations of the T5 that differ for the type of learning rate applied during the training (see Table 4.3 for the four configurations). Once trained, each model has been run on the evaluation set to assess its performance in terms of percentage of correct predictions, namely the percentage of instances in the evaluation set for which the model managed to exactly predict the masked code. This process has been performed on the largest dataset (*i.e.,* $dataset_{very\_weak}$), assuming that the best configuration identified on it represents the best choice also for the other three datasets. Each model has been fine-tuned for 50k (corresponding to ∼10 epochs given the used batch size of 256). The *Slanted Triangular* learning rate obtained the best results and, thus, has been used in the study to train our models.

Fine tuning. We fine-tuned T5 on each of the four datasets for ∼100 epochs, by varying the number of fine-tuning steps based on the dataset size (*i.e.,* larger datasets require more steps to reach 100 epochs, see *# Training Steps* in Table 4.4). To avoid overfitting, we saved the trained models every 5k steps and identified the best one (see *Best Checkpoint* in Table 4.4) on the evaluation set in terms of percentage of correct predictions.

These are the models we will use to generate the predictions on the test set and analyze

the overlap in terms of code clones between the generated predictions and the code in the training sets.

4.2.4 Data Collection And Analysis

Once trained the four models (one on each training dataset), we run them on the test set that, as previously explained, is the same for all four models. This results in the models generating the code blocks predicted as needed to fill the masked code. These blocks of generated code are the ones we want to contrast against the training set used for each model to identify code clones.

To identify the clone detector to use, we started from the literature review by Rattan *et al.* [RC07], looking at the list of tools documented by the authors. Several of them are no longer available (*e.g.,* [HK09], [GKKP13]), while others do not work on syntactically incorrect code that may be generated by the T5 models (*e.g.,* [RC08], [cpd]). Given the study's constraints, we used the SIMIAN clone detector [sima]. Besides being very robust and scalable, SIMIAN can identify Type-1 and Type-2 clones [RC07]. SIMIAN works at line-level granularity, looking for duplicated lines among the code dataset provided as input. Therefore, having two snippets of code that are identical but written on a different number of lines, may fool the clone detection algorithm. For this reason, we formatted all methods in our datasets using the IntelliJ formatter [int], to ensure they all adopt the same coding style (*e.g.,* maximum number of characters per line).

We run SIMIAN to identify both types of clones between the code generated by the four models and their respective training set. Then, we compute the percentage of code clones generated by each model by looking at different characteristics of the clones and of the predictions. In particular, we consider in our analysis:

- *The amount of near-duplicates in the training set*. As previously explained, this has been achieved by training four T5 models on the four datasets we built. We report the percentage of clones generated by each of these models.
- *The length of the generated code*. We split the generated code blocks in different buckets based on the number of statements they feature (from ≥ 2 to ≥ 5 at steps of 1). Then, we compute the percentage of clones in each bucket. We expect smaller predictions to contain a higher percentage of clones.
- *The clone type*. When reporting the percentage of clones, we distinguish between Type-1 and Type-2 clones. Type-2 clones will be, by construction, more prevalent, since they are a superset of Type-1 clones, being exact copies but for the different use of identifiers and types.
- *The training dataset from which the prediction has been "cloned"*. Each T5 model exploits two training datasets, the pre-training and fine-tuning. We analyze from where the model is more likely to "copy".
- *The correctness of the prediction*. We look at whether the models are more likely to "copy" when generating correct or wrong predictions.

We complete our study with a correlation analysis in which we assess whether the model is more prone to clone code in specific circumstances. In particular, we use the Spearman test [Zar72] to correlate the presence of clones with (i) the *Cyclomatic Complexity* of the test method (ii) the number of lines of the method, and (iii) the *confidence* of the prediction. The first two metrics are computed using the Python Lizard library [liz]. The last one is a *score* we computed for each prediction made by T5. Such a score x ranges from minus infinity to 0 and it is the log-likelihood of the prediction itself and can be normalized by computing the $exp(ln(x))$. This brings the confidence score between 0.0 and 1.0, with 1.0 indicating the scenario in which the model is confident about the generated prediction.

Finally, it is worth mentioning that, despite the use of the *SentencePiece* tokenizer, the T5 may generate predictions including *unknown tokens*. We performed all the above described analysis both when considering all the instances of the test set as well as when removing the ones that contain at least one *unknown token*. The latter are less likely to result in code clones, since the training sets do not contain unknown tokens. We report in this thesis the results achieved when excluding from the test set the 6,986 instances for which at least one of the four models generated an unknown token; the results on the whole test set are available in our replication package [repe], but we anticipate that they are inline with those discussed in this thesis.

4.2.5 Replication Package

The datasets, models, training/testing code, and the achieved raw data are publicly available in our replication package [repe].

4.3 Results Discussion

To answer our research question, we analyze the percentage of code predictions being *clones* of instances present in each training dataset (*i.e.,* pre-training and fine-tuning).

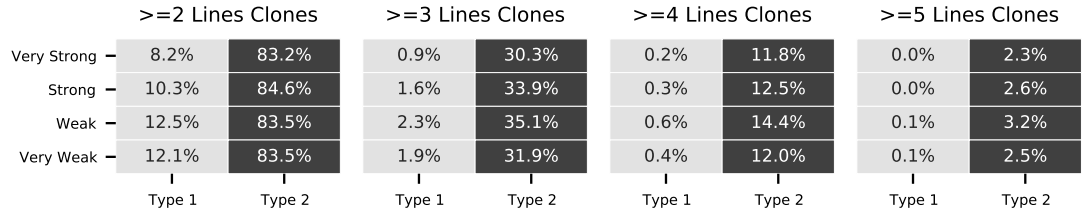

Figure 4.1. Percentage of Type-1 and Type-2 clones in predictions having different lengths.

Impact of the amount of near-duplicates in the training set. Fig. 4.1 shows the percentage of Type-1 and Type-2 clones found in the predictions of the four models fine-tuned on the different datasets (*i.e.,* $dataset_{very\_strong}$, $dataset_{strong}$, etc.). We also organize the predictions in different buckets based on the length of the predicted code (from ≥ 2 up to ≥ 5). As a reminder, $dataset_{very\_strong}$ is the smallest (273k instances) and has been built using

very strict criteria for the exclusion of near duplicates; $dataset_{very\_weak}$ is instead the largest (838k instances), with a higher presence of near duplicates. Note that the gap in dataset size is substantial when moving from the *Very Strong* to the *Strong* configuration (+77% of instances) and from *Strong* to *Weak* (+47%), but limited when going from *Weak* to *Very Weak* (+18%).

Before commenting on the achieved results is also worthwhile to remind that the number of generated predictions is 15,742-6,986 for all datasets, since in this results discussion we removed all cases for which at least one of the four models generated an unknown token. Thus, a 10% of clones indicates ∼900 generated clones.

With the increase of near duplicates in the training set, the percentage of Type-1 clones in the generated predictions increases as well. This trend is visible across the four datasets with the exception of $dataset_{very\_weak}$ for which the percentage of generated Type-1 clones is inline with those of $dataset_{weak}$. As previously said, this is likely to be due to the small differences in size (*i.e.,* in the presence of near-clones) between the two datasets.

Another trend is also clear when looking at Fig. 4.1: longer predictions are less likely to be the result of cloned snippets from the training set. If we focus, for example, on the dataset resulting overall in the highest percentage of generated Type-1 clones ($dataset_{weak}$), such a percentage moves from 12.5% for predictions featuring at least two statements, to 2.3% for those having at least three statements, down to 0.1% when only considering the generation of non-trivial code snippets composed by at least five statements. Such a result is quite expected since longer predictions are statistically less likely to be equal to instances present in the training set as compared to shorter predictions.

The analysis of Type-2 clones follows similar trends with a couple of notable differences. First, as expected, the percentage of Type-2 clones is much higher as compared to that of Type-1 clones (from 10 to 100 times higher). This can be explained by the weaker requirements adopted to consider two code snippets as Type-2 clones. Indeed, while the code structure should be similar, two instances are considered as Type-2 clones even if they adopt completely different identifiers and types. Still, confirming what observed for Type-1 clones, the percentage of clones in the predictions steadily drops with the increase in size of the predictions, moving from ∼80% to ∼2.5% when the prediction's size increases from at least 2 to at least 5 lines. Finally, differently from what observed for Type-1 clones, there is no substantial difference in the percentage of Type-2 clones across the different datasets. This may be due to the fact that, differently from Type-1 clones that are verbatim copies of code from the training set, in the case of Type-2 clones the amount of near-duplicates (*i.e.,* frequent instances in the training set) is less likely to play a role.

Take away 1. Around 10% of the generated predictions represent a Type-1 clone of code present in the training data, with this percentage going up to ∼80% when considering Type-2 clones. However, the percentage of clones steadily drops when the model is asked to predict several statements, with basically no clones generated when the prediction comprises at least four lines of code. This indicates that advanced tools such as GitHub Copilot that can generate entire functions are very unlikely to "copy code" from the training data.

Pre-training and fine-tuning datasets. We also analyzed the provenance of the clones, meaning whether the cloned predictions tend to come from the pre-training or from the fine-tuning dataset. Fig. 4.2 summarizes the achieved results. For both types of clones (*i.e.,* Type-1 and Type-2) Fig. 4.2 reports the the percentage of clones coming (i) only from the pre-training (*i.e.,* the prediction has a clone only in the pre-training dataset); (ii) only from the fine-tuning; and (iii) from both training datasets (*i.e.,* both training datasets have at least one instance representing a clone of the prediction). In interpreting the reported percentages it is important to consider that the chances of cloning from the pre-training are expected to be higher, since it contains more "material" from which the model can copy. For example, in the *Very Strong* configuration, we have 1.85M methods in the pre-training dataset and 273k methods in the fine-tuning dataset. Therefore, 87% of the training methods belong to the pre-training dataset and only 13% to the fine-tuning dataset. The *Very Weak* configuration is the most balanced, still having, however, 69% of training instances coming from the pre-training dataset against the 31% of the fine-tuning dataset.

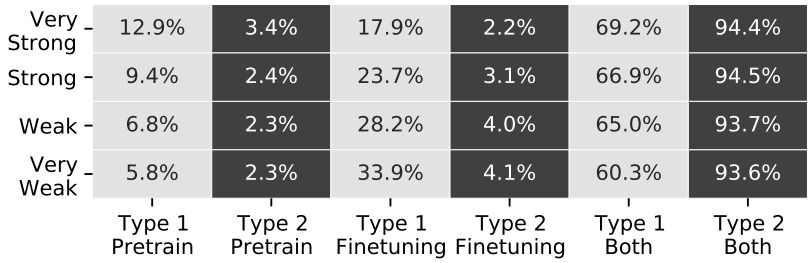

Figure 4.2. Percentage of Type-1 and Type-2 clones coming from the pre-training and fine-tuning dataset

For both types of clones, most of the cloned predictions have a clone both in the pre-training and in the fine-tuning datasets.

As expected, this is especially true for Type-2 clones, for which almost the totality of cloned predictions follow this pattern. More interesting are the results for Type-1 clones, namely exact copies. While also here there is a substantial percentage of predictions having clones both in the pre-training and in the fine-tuning datasets (∼60%), most of the predictions only cloned from one of the two training datasets come from the fine-tuning. This result is not really expected considering that the pre-training dataset provides most of the "training material". However, possible explanations for this result lie in the order in which the two training phases are applied to the DL model and in their objective. The pre-training changes the weights of the neural network from a random distribution to a distribution able to deal with the pre-training denoising task (*i.e.,* guessing masked tokens in the input instance). The tokens masked in the pre-training are randomly selected and may not be contiguous. This means that in very few cases the model sees instances in which it is required to generate complete code statements. The fine-tuning, instead, starts from the pre-trained model or, in other words, from the distribution of weights obtained after pre-training. These weights are then modified to support the fine-tuning task that, in our case, explicitly requires the model

to generate complete code blocks. Thus, when the model is asked during testing to generate code blocks, the weights learned during the fine-tuning may come more "handy", pushing the model to reuse more knowledge acquired during the fine-tuning rather than during the pre-training.

Take away 2. Most of the cloned predictions have clones in both the pre-training and the fine-tuning. For example, in the case of Type-1 clones, ∼65% come from both training datasets. This indicates that code instances seen repeatedly during both training phases are, as expected, more likely to influence the generated predictions. The remaining ones (*i.e.,* clones that only come from one of the two training sets), are more likely to come from the fine-tuning. This goes against what we expected since the pre-training is substantially larger than the fine-tuning.

Correct and wrong predictions. Fig. 4.3 shows the percentage of Type-1 and Type-2 clones created by the model when generating correct (CP) and wrong (WP) predictions. Also in this case the percentages must be read by keeping in mind that, across the test instances on which the four models have been tested, all models achieved ∼3% of correct predictions (∼500 cases), implying that we should expect ∼3% of clones being related to correct predictions. Before commenting the results, a few notes on the low percentage of correct predictions achieved by the models (more in our Validity Evaluation section — Section 4.4): (i) such a percentage is inline with what has been reported in the literature for applications of DL models to automate other code-related tasks, such as bug-fixing [TWB+19] or code review automation [TPT+21]; (ii) the code generation tasks on which we tested T5 included the generation of code blocks up to six lines of code, thus being non-trivial; (iii) we consider as correct only predictions that are identical to the masked code, meaning that even predictions that are slightly different but semantically equivalent are considered wrong; (iv) finally, as we will discuss in Section 4.4, our inspection of wrong predictions confirmed that the model learned how to generate syntactically correct code, confirming the validity of the performed training.

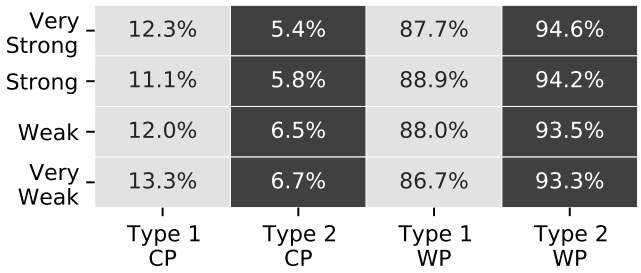

Figure 4.3. Percentage of Type-1 and Type-2 clones in correct (CP) and wrong (WP) predictions.

The cases in which the predictions represent verbatim copies of code in the training data (Type-1 clones) occur ∼4× more than expected in correct predictions (see Fig. 4.3), with

∼12% of clones belonging to correct predictions, only representing ∼3% of all predictions. This is likely due to instances (methods) in the training set that, while being different from all methods in the test set, share with them the part of code that we masked. Such a result also raises a warning for the evaluation of DL models in the context of code generation: It might be not enough to just remove duplicates and/or near-duplicates across instances, since the parts of code the model is required to generate in the test set may still be present as verbatim copy in the training set, even when very strict criteria are used to remove duplicates (as in the case of our *Very Strong* configuration).

Similar observations can be made for Type-2 clones, even though here the percentage of clones in correct predictions is only ∼2× higher than what we expected.

Take away 3. Correct predictions are more likely to contain clones of code present in the training datasets. Such a result stresses once more the importance of a proper dataset cleaning to avoid overlaps between the training and the test datasets.

Correlation analysis. We now discuss the results related to the correlation analysis aimed at understanding if specific characteristics of (i) the code snippet to predict, (ii) the instance containing it, or (iii) the confidence of the model in generating the prediction, may affect the presence/absence of code clones.

More in detail, as explained in Section 4.2, we consider (i) the Cyclomatic Complexity (CC) of the entire instance (method), (ii) the CC of the model's input (*i.e.,* the entire method without the block to predict), (iii) the number of lines of the method, (iv) the number of lines of the block to predict, and (v) the confidence of the T5 model.

We found no significant correlation between all these metrics and the presence of Type-1 clones in the generated predictions. While we identified some significant correlations for Type-2 clones, they are all lower than |0.15|, thus indicating very low correlations (complete results in our replication package [repe]).

Qualitative Examples. Finally, we present in this section some qualitative examples of the clones generated by the T5 model. Remember that the clones refer only to the part of code predicted by the model, ignoring the surrounding context. In other words, two different methods containing the same block of code predicted by T5 are considered as clones.

```
/ cloned code /
errorFragment = new ErrorFragment();
Bundle msg = new Bundle();
msg.putString("msg", ex.getMessage());
errorFragment.setArguments(msg);
getSupportFragmentManager().beginTransaction()
  .replace(R.id.message, errorFragment).commit();

/ expected prediction /
{
errorFragment = new ErrorFragment();
Bundle msg = new Bundle();
msg.putString("msg", "No␣or␣poor␣internet␣connection.");
errorFragment.setArguments(msg);
getSupportFragmentManager().beginTransaction()
```

```
  .replace(R.id.message, errorFragment).commit();
}
```

Listing 4.1. Type-1 Clone Generated in Wrong Prediction

The top part of Listing 4.1 shows an example of Type-1 clone generated by the T5, while the bottom part reports the correct prediction the model was expected to generate (*i.e.,* the code we masked). As it can be seen, the prediction task we asked the model to perform was far from trivial and the T5 managed to generate a prediction really close to the expected one. Such a prediction was the result of cloning four of the five statements to predict from the training dataset. This is one of those cases in which, despite the model did not generate a correct prediction, it went quite close to the target, with the only difference being one of the arguments used in the `putString` invocation.

Also Listing 4.2 shows a Type-1 clone generated in a wrong prediction. In this case, we want to put the focus on the fact that the model cloned a snippet of code that represents a quite popular template in Java when converting a `String` to an `Integer`. The model understood the need for implementing a check on the value of an `Integer` but failed in implementing the check that was actually needed (the one shown in the expected prediction).

```
/ cloned code /
  if (s != null)
  {
    try {
      return Integer.parseInt(s);
    } catch (NumberFormatException e) {
  }
}
return null;

/ expected prediction /
{
  Integer interval = OptionAdvanceParser.DEFAULT_INTERVAL;
  if (optionSet.hasArgument("interval")) {
    interval = (Integer) (optionSet.valueOf("interval"));
    if (interval < 1) {
      throw new IllegalArgumentException("Interval␣cannot␣be␣set␣below␣1.0");
    }
  }
  return interval;
}
```

Listing 4.2. Type-1 Clone Generated in Wrong Prediction

```
/ cloned code /
do {
  CategorySource categorySource = CategorySource.categoryFromCursor(cursor);
  catsSrcs.add(categorySource);
} while (cursor.moveToNext());

/ expected prediction /
{
  do {
  CategorySource categorySource = CategorySource.categoryFromCursor(cursor);
  catsSrcs.add(categorySource);
  } while (cursor.moveToNext());
}


/ code from the training set /
do {
  CategorySource category = CategorySource.categoryFromCursor(cursor);
  categories.add(category);
} while (cursor.moveToNext());
```

Listing 4.3. Type-2 Clone Generated in Wrong Prediction

Finally, Listing 4.3 shows an example of Type-2 clones, reporting the prediction on top, the expected code in the middle, and the cloned instance from the training in the bottom. In this case, the model reused the code structure just changing the identifier name from `category` to `categorySource`. The model generated something very close to the expected solution, fully relying, however, on code already seen during the training that led it to a wrong prediction.

4.4 Validity Discussion

Threats to *construct validity* concern the relationship between the theory and what we observe. In our analysis, similarly with what done in the literature (*e.g.,* [HPGB19, CCP+21a]), we simulate the usage of the code recommender by masking blocks of code. We are aware that the masked blocks may not completely reflect the way in which developers would use such recommenders in practice.

Another threat is related to the limited prediction performance achieved by our models (∼3% of correct predictions, detailed results in [repe]). As already discussed, this level of performance is not unusual for the automation of challenging code-related tasks [TWB+19, TPT+21], such as the code generation task subject of our study. However, a reader may wonder whether the low percentage of clones we observed in the predictions may be due to the fact that, in most of cases, the model just generates garbage, thus not being "able" to generate clones of training instances. This is not unusual to observe for DL models trained, for example, on very little data and that tend to generate long meaningless sequences of frequent tokens observed in the training set (*e.g.,* sequences of parentheses when dealing with code-related datasets). While we trained our models on millions of Java methods, we inspected a sample of 200 wrong predictions (50 for each of the four models) to manually check whether the generated predictions, while wrong, followed a correct Java syntax. Such

an analysis has been performed by the first author, and resulted in only 7 (3.5%) instances classified "meaningless predictions" in terms of Java syntax.

Threats to *internal validity* concern factors, internal to our study, that could affect our results. The hyperparameters tuning phase described in detail in Section 4.2 can have a strong impact on the performance of DL models and, consequently, on their likelihood of generating clones. Due to feasibility reasons, we calibrated the hyperparameters only for the $dataset_{very\_weak}$, assuming that the chosen configuration would also work well for the other datasets. Hence, it is possible that evaluating a plethora of different configurations may improve the performance of the model and/or impact our findings about the generated clones.

Note also that, as explained, we observed that for some of the instances in the test set the models generated *unknown tokens* that are less likely to result in clones from the training set. For this reason, we reported in the thesis the results achieved when ignoring these instances, while the results on the whole dataset are available in our replication package [repe]. As previously said, the main observed findings still hold when also considering the predictions featuring unknown tokens.

Finally, it is worth mentioning that the accuracy of the SIMIAN clone detector may influence the achieved results. To partially address this threat, the first author inspected 100 randomly selected predictions from our dataset for which Simian identified Type-1 (50 predictions) and Type-2 (50) clones. Since for a single prediction there might be dozens of clones, we inspected 3 clones per each prediction. Concerning Type-1 clones, without surprise, the accuracy was 100%, since those are predictions being exact copies of code in the training sets. As for Type-2 clones, judging the identified instances was not always straightforward due to the limited size of the identified clones (between 2 and 6 lines). These clones are exact copies at the AST-level and, looking at them from this perspective, we confirm the correctness of all inspected cases. However, especially when looking at very small clones (*e.g.,* 2 lines), we acknowledge that some pieces of code, while sharing the same AST structure, may look different to a human, since using different identifiers and/or types.

Threats to *external validity* are related to the possibility to generalize our results. We chose the T5 model, that showed a strong ability to adapt to different code-related tasks [MSC+21] achieving remarkable performance. The datasets used in our experiments are all Java-related. We do not claim any generalizability of our findings in terms of adopted DL model and subject programming language.

We relied on the SIMIAN clone detector [simb] that, as explained in Section 4.2, is robust in the processing of syntactically incorrect code. Results may change by using other clone detectors. Moreover, in our study we only considered code clones composed by at least two non-trivial lines. Considering shorter clones, while being an option, is likely to artificially inflate the percentage of clones with instances that are not really relevant in terms of possible licensing issues.

4.5 Conclusion

Code recommenders are becoming more and more popular. GitHub Copilot [CTJ+21] substantially pushed the adoption of these tools by developers, making central questions related to possible licensing issues that may come from their adoption in a commercial setting. Indeed, these tools are usually trained on open-source code, the usage of which is regulated by FOSS licenses.

While the posed licensing questions are part of a non-trivial debate among the open source community, researchers, and tool builders, we started investigating a concrete and related research question: *To what extent do DL-based code recommender systems generate predictions by cloning code snippets from the training set?*

We trained a Text-To-Text-Transfer-Transformer (T5) model on over ∼2M Java methods with the task of recommending code blocks aimed at finalizing the methods' implementation. Then, we used a clone detector to check whether the predictions it generated on the test set represent Type-1 or Type-2 clones of instances in the training datasets. Our findings show that, while for short predictions the trained deep learning model is likely to generate a non-trivial percentage of clones (∼10% Type-1 and ∼80% Type-2), such a percentage quickly approaches 0% when the model generates more complex predictions composed by at least four code statements.

In summary, our findings suggest that the likelihood of obtaining clones generated by DL-based code recommenders that are possibly "harmful" in terms of licensing issues is extremely low.

Our future work will mainly focus on replicating our work on larger, more performant, DL-based code recommenders and different clone detectors, with the goal of confirming or confuting our findings.

The material used in this study is publicly available [repe].

5

On the Generalizability of Deep Learning-based Code Completion Across Programming Language Versions

5.1 Motivation

Just like code completion tools and approaches rapidly become more mature, programming languages themselves evolve at a fast pace too. In the last five years, languages such as Python [pyt] and JavaScript [javd] have released major versions every year, and Java [javb] has released a new version every six months. These new versions introduce new features and syntax such as new keywords, operators, data types, APIs, and constructs. For instance, Java 8 introduced the `Stream` API, which allows developers to perform functional-style operations on collections of objects, and Java 11 introduced the `var` keyword, which allows developers to declare local variables without specifying their type. These new features may impact the performance of DL-based code completion models, which may be more prone to overfitting the specific language version used for their training, or may simply be unaware of new syntax constructs introduced in new language versions, thus being unable to predict them. This problem is usually referred to as *concept drift* [GZB+14, LLD+19], where the data upon which a machine learning model has been trained on evolves over time, eventually leading to a performance degradation of the model or even invalidating it.

In this thesis, we study the impact of language evolution on the performance of DL-based code completion models. More specifically, we investigate to what extent code completion models can generalize across different language versions, including both older and newer versions as compared to the one used for training.

We focus on Java as a good representative of a mature language with a long history of releases over the last 30 years. The selected DL model for the evaluation is CodeT5 [WWJH21], a state-of-the-art code model based on the Text-To-Text Transfer Transformer (T5) [RSR+20] architecture. This model has been shown to perform well in a variety of code-related tasks such as code summarization, code generation, and defect detection, among others [WWJH21]. For our experiments, we pre-train and fine-tune CodeT5 exclusively on Java 8 code, and then

assess its code completion capabilities on nine different Java versions (including Java 8 itself) ranging from Java 2 (released in 1998) to Java 17 (released in 2021). To make our study more comprehensive, we evaluate the model performance at three different code completion granularity levels, namely, token-level (*i.e.,* predicting the last *n* tokens in a statement), construct-level (*i.e.,* predicting whole constructs such as `if` conditions), and block-level (*i.e.,* predicting all statements within a code block, such as the body of a `for` loop).

The results of our study show significant performance differences across different language versions, with gradual decreases as we move away from the target version used for training—Java 8—and the worst performance being obtained in the most far apart versions—Java 2 and 17. This finding highlights the potential benefits of using version-specific DL models for code completion and, even more significantly, the importance of retraining the model on newer language versions as more training data is progressively available. As a matter of fact, we found that a small fine-tuning on the target language version can significantly improve the model performance, thus suggesting that the model can be adapted to new language versions with relatively little effort.

Our work raises awareness on the importance of retraining DL models on new language versions, which can hopefully pave the way towards more effective code completion tools that developers can use to boost their productivity. Moreover, we believe that our results can be used to inform the design of online incremental training techniques for DL-based code completion models, which can be trained on new language versions as soon as they are released, thus keeping up with the fast-paced evolution of programming languages.

All code and data used in our study are publicly available [repc].

5.2 Study Design

The *goal* of this study is to assess the generalizability of a state-of-the-art DL-based code completion technique across different versions of the same programming language. In particular, we aim at answering the following research question (RQ):

> *To what extent do DL-based code completion techniques generalize across different language versions?*

Our empirical study is focused on the specific *context* represented by: (i) CodeT5 [WWJH21] as a representative DL model which has been used in the literature for the automation of code-related tasks [CAD+22, BWH22, TC22, ZJW+23, LWG+22, ADS22, WWW+22]; and (ii) code from 784 Java repositories for which we managed to reliably identify the used Java version.

CodeT5 is a T5 model [RSR+20] pre-trained on code and natural language (*i.e.,* code comments). Wang *et al.* [WWJH21] exploited the CodeSearchNet dataset [HWG+19] for pre-training CodeT5. This dataset includes functions written in six programming languages (Go, Java, JavaScript, PHP, Python, and Ruby). Some of these functions also include a summary comment (*e.g.,* the Javadoc comment for Java). On top of CodeSearchNet, Wang *et al.*

collected additional functions from C/C# repositories hosted on GitHub. Overall, their pre-training dataset featured 8,347,634 functions, 3,158,313 of which paired with their documentation. The employed pre-training objectives were the classic masked language model (*i.e.,* self-supervised pre-training by randomly masking 15% of the input asking the model to predict it) as well as a novel identifier-aware pre-training objective devised by the authors.

In our study, we fine-tune CodeT5 for the code completion task *only* with source code belonging to a specific Java version v_i, and we test it on multiple test sets, each featuring a different Java version (including v_i). The performance obtained on the v_i test set provides a reference point for evaluating the performance on all the other test sets ($v_j \neq v_i$). To factor out the impact of pre-training on the CodeT5 model (whose pre-training dataset does include code from multiple Java versions [HWG+19]), we perform the experiments not only with the pre-trained model (*i.e.,* the publicly available CodeT5 checkpoint [cod]), but also with a non pre-trained one.

5.2.1 Data Collection and Datasets Creation

In this section, we describe the process for collecting the data used to create the training and test sets. The training dataset is used to fine-tune CodeT5 for the code completion task on a single Java version, while the test sets are used to assess the generalizability of the model across different Java versions.

We used the tool by Dabic *et al.* [DAB21] to select from GitHub all the non-forked Java repositories having more than 100 commits, 10 contributors, 50 issues, and 10 stars. We applied these filters in an attempt to remove toy/personal projects. This query resulted in 5,632 repositories. One of the requirements for our study is the ability to identify the Java version used in the project. For this reason, we only selected from the 5,632 repositories those using Maven. Indeed, Maven projects feature a `POM` (Project Object Model) file where developers can specify configuration details useful for building the project, including the used Java version. To increase the probability of collecting compilable code from these projects (*i.e.,* to avoid training the DL model on syntactically wrong code), we excluded all projects which cannot be compiled using Maven. In particular, we attempted the compilation for the last two GitHub releases of each repository, keeping only the ones for which compilation succeeded: we first check the latest release and, only if its compilation fails, we move to the second-last. Overall, we collected 784 repositories which can be successfully compiled and which explicitly report the adopted Java version in the `POM` file.

From each repository, we randomly selected up to 1,000 Java files, ignoring the ones having the word "test" in the filename, aiming to exclude test files and create a more cohesive dataset made of production code only. The choice of capping the maximum number of files per project to 1,000 aims at avoiding that very large projects may contribute too much code to the final dataset.

We used the set of collected files to build a method-level dataset of code completion tasks by slightly adapting the masking procedure proposed by Ciniselli *et al.* [CCP+21a]. In particular, Ciniselli *et al.* proposed three method-level completion tasks having different levels of difficulty:

- **Token masking.** For each line of code, we masked the last *n* tokens in a statement, with *n* randomly ranging from three to ten, and then we asked the model to predict them. We chose to mask at least three tokens to avoid trivial completions (*e.g.,* only predicting the semicolon ending a Java statement). Moreover, we masked at most three random statements for each method to avoid generating too many instances from the same method. Indeed, each method can generate multiple training/testing instances, each with a specific statement being partially masked. The *token masking* scenario emulates the code completion task in which the DL model is trying to complete the statement the developer is writing.

- **Construct masking.** Ciniselli *et al.* suggests that code completion performed on specific types of code constructs (*e.g.,* the condition of `if` statements) can be particularly useful to developers. We follow their *construct masking* scenario, by masking all tokens used to implement: (i) the condition of an `if` statement or of a `while/for` loop, *e.g.,* "`for(int i=0; i<dict.size(); i++)`" is masked as "`for(<MASK>)`", with the model in charge of predicting the masked tokens; and (ii) the exception caught in a `catch` statement. Also in this case, a single method can contribute multiple instances to the dataset (*e.g.,* if it has three `if` statements, three instances are created, each of which having the condition of one `if` statement masked). We used srcML [src] to identify the statements of interest and perform the masking.

- **Block masking.** This is the most challenging completion task. All code statements enclosed between curly brackets are considered a block (*e.g.,* the corpus of an `if` statement [CCP+21a]). As in the previous case, we used srcML [src] to identify all blocks in each method and then create multiple instances of it, each having one entire block masked. As Ciniselli *et al.*, we only mask blocks featuring at most three statements to cap the task complexity.

We applied these masking procedures to all methods in the collected Java files, generating three different datasets. Exceptionally, we excluded methods that: (i) contained non-ASCII characters, which caused issues during training; and (ii) contained less than three or more than 50 statements (including signature), since they are either too short to prepare any meaningful completion scenario or too long to be provided as input to the DL model, whose maximum input length is 512 tokens. We removed duplicate methods to avoid leaking information between the training and test sets. In the end, we collected 1,052,141 different methods, from which we derived a total of 2,846,746 *token-masking* instances, 783,546 *construct-masking* instances, and 1,303,444 *block-masking* instances. Note that the *construct-masking* instances are less than the overall number of methods since not all methods feature the construct types we mask (*i.e.,* `if`, `while`, `for`, and `catch`).

The methods and corresponding instances are spread across 10 different Java versions as reported in Table 5.1.

Table 5.1. Number of methods and instances for each Java version.

Java Version	# Methods	# Token Instances	# Construct Instances	# Block Instances
2	4,530	12,167	1,389	2,802
5	8,549	21,178	6,510	11,103
6	68,379	181,877	53,991	85,517
7	99,405	266,162	72,108	122,593
8	809,704	2,194,600	601,891	1,007,040
9	4,809	13,879	3,393	5,401
11	38,521	105,700	29,975	44,632
14	6,160	17,203	5,842	8,644
16	7,169	20,593	4,946	9,805
17	4,915	13,387	3,501	5,907
ALL	**1,052,141**	**2,846,746**	**783,546**	**1,303,444**

Creating the Test Sets

Our goal is to create 30 different test sets, each one representing one of the 10 Java versions featured in our dataset and one of the three masking scenarios we adopted. For example, one test set will feature Java 2 methods in which entire blocks have been masked (*i.e., block masking*). In creating the test sets, we must make sure that they all feature instances having a similar level of complexity, so that any observed performance differences in the model predictions can be attributed to the Java version used in the test set, and not to other factors. To address this issue, we defined a number of metrics to assess the *complexity* of each instance in our dataset, where an instance is a Java method having some of its tokens masked (based on the three masking scenarios previously described):

1. *Number of lines in the method*. Longer methods may provide more contextual information to the model and allow for a simpler completion (*e.g.,* completing 10 out of 100 tokens may be easier than completing 10 out of 15 tokens).

2. *Average number of characters per line*. Very long statements in a method may suggest a higher complexity of its instructions. The average number of characters per line is computed as the total number of characters in a method divided by the number of lines it features.

3. *Number of masked characters*. The higher the number of masked characters, the higher the complexity of the code completion task (*i.e.,* guessing 20 characters is likely more complex than guessing five characters).

4. *Number of lines masked (only for block masking)*. Similarly to the number of masked characters, masked blocks featuring a higher number of lines are likely more challenging to predict.

We computed these metrics for all collected methods. Table 5.2 reports their mean values on the whole dataset. These represent our reference values that we use to build the test sets

Table 5.2. Complexity metrics computed on the dataset.

Metric	Value
Mean # of lines in method	7.7
Mean # of characters per line	28.1
Mean # of masked characters (*token masking*)	18.2
Mean # of masked characters (*construct masking*)	27.9
Mean # of masked characters (*block masking*)	37.5
Mean # of masked lines (*block masking*)	1.3

Algorithm 1 Algorithm used for generating the test sets.

```
n ← 5000                                   ▷ Number of selected methods for that specific Java version
δ ← 0.05                                   ▷ Difference allowed between the reference metrics and the test set metrics
function BUILDTESTSETS(n, δ)
    S ← RANDOMLYSELECTSAMPLE(n)
    for steps ← 1 to 5000 do
        constraintMet ← True
        for i ← 1 to NumberOfMetrics do
            R_Ci ← REFERENCEVALUEFORMETRIC(i)
            S_Ci ← CALCULATESAMPLEMETRIC(S, i)
            if S_Ci − R_Ci > δ then
                constraintMet ← False
                m_j ← FINDMETHOD(S, i, R_Ci)        ▷ random method having C_i > R_Ci
                m_k ← FINDMETHOD(S, i, R_Ci)        ▷ random method having C_i < R_Ci
                S ← REPLACEMETHOD(S, m_j, m_k)
            else if S_Ci − R_Ci < δ then
                constraintMet ← False
                m_j ← FINDMETHOD(S, i, R_Ci)        ▷ random method having C_i < R_Ci
                m_k ← FINDMETHOD(S, i, R_Ci)        ▷ random method having C_i > R_Ci
                S ← REPLACEMETHOD(S, m_j, m_k)
            end if
        end for
        if constraintMet = True then
            return S
        end if
    end for
    n ← n − 500
    return BUILDTESTSETS(n, δ)
end function
```

for the different Java versions. In particular, we build the test sets by adopting the algorithm reported in Algorithm 1, explained in the following.

We target the building of test sets featuring n=5,000 instances each (*e.g.,* 5k instances in the Java 8 test set using *token masking*). We set as constraint that each test set should feature instances being close, in terms of complexity, to the reference metric values in Table 5.2, with a margin δ of ±5% for each metric. This means, for example, that a test set can have a mean number of lines per method being 7.7 ± 0.38. The same holds for the other metrics in Table 5.2. The algorithm starts by randomly selecting a sample S of 5,000 Java methods from the dataset, checking for each metric C_i how far this sample is from the reference value. If the constraint for C_i is satisfied (*i.e.,* S_{C_i} is within δ from the reference value R_{C_i}), no changes to S are performed. Otherwise, if S_{C_i} - $R_{C_i} > \delta$ and it is a positive number (*i.e.,* methods in S have a higher value for the metric C_i as compared to the reference value R_{C_i}), we remove

from S a randomly selected method m_j having $C_i > R_{C_i}$ and we add a randomly selected method m_k having $C_i < R_{C_i}$. If, instead, methods in S have a lower value for the metric C_i as compared to R_{C_i}, we remove from S a randomly selected method m_j having $C_i < R_{C_i}$ and we add a randomly selected method m_k having $C_i > R_{C_i}$. Essentially, we seek convergence of the considered metrics. We perform this procedure for a maximum of 5,000 steps. If after 5,000 steps the sample S does not meet the constraint for all metrics, we reduce the number of methods to collect by 500 (*i.e.,* $n = n - 500$) and repeat the process.

Table 5.3. Number of methods and instances for each set for each Java version.

Dataset	Java Version	# Methods	# Token Instances	# Construct Instances	# Block Instances
train	8	724,130	1,962,673	538,427	900,927
eval	8	80,574	218,377	59,729	99,886
test	2	3,000	7,692	879	2,052
	5	5,000	12,626	4,670	6,866
	6	5,000	13,272	4,032	6,394
	7	5,000	13,440	3,547	6,069
	8	5,000	13,550	3,735	6,227
	11	5,000	13,547	3,494	5,650
	14	5,000	13,855	4,178	6,750
	16	5,000	14,356	3,568	6,849
	17	4,000	10,894	2,925	4,834

Following this procedure, we managed to collect 5,000 methods for all test sets with few exceptions: for Java 2, which featured 4,530 overall methods in the whole dataset, we managed to collect 3,000 methods. For Java 17, we collected 4,000. Lastly, for Java 9, we did not reach convergence at any test set size, hence its exclusion from our study.

Creating the Training (Fine-Tuning) and Evaluation Set

Given the distribution of methods across the Java versions in our dataset (see Table 5.1), we decided to use Java 8 as the "training version", since it features ∼80% of the overall methods we mined. This means that Java 8 is the version on which we train CodeT5 to then test its performance on the test sets belonging to the different versions (including Java 8 itself). To create the training and evaluation sets, we took the 804,704 Java 8 methods which were not included in the Java 8 test set and split them into training (90%) and evaluation (10%) sets, the latter being used for hyperparameters tuning (as described in the next section). Table 5.3 reports the number of methods and instances for the training, evaluation, and test sets.

5.2.2 Hyperparameters Tuning and Training

We adopt the default parameters used in the paper presenting CodeT5 [WWJH21], only experimenting with different learning rates. More specifically, we evaluated three different values (*i.e.,* $1e^{-5}$, $2e^{-5}$, $5e^{-5}$), using the AdamW optimizer [LH19] to update the weights.

We trained the model for 10K steps using a batch size of 12 and we evaluated the performance of each of the three configurations on the evaluation set in terms of the percentage of exact match predictions (*i.e.,* the predicted code tokens are identical to the masked ones). We performed this tuning for both: (i) a CodeT5 model fine-tuned based on the publicly available pre-trained checkpoint [cod]; and (ii) a CodeT5 fine-tuned from scratch, not pre-trained at all. Indeed, as previously explained, CodeT5 has been pre-trained on source code, including Java code. Thus, it has seen Java code written in versions different from the Java 8 we want to use as "training version". Experimenting also with a non pre-trained model in our study allows us to factor out this further confounding factor.

For both models (*i.e.,* pre-trained and non pre-trained) we found the best configuration to be the one in Table 5.4. Using this configuration, we trained the two models for 15 epochs, for a total of 4,250,000 training steps with a batch size of 12. The trainings took 26 days using an NVIDIA GPU GeForce RTX 3090 with 24GB of RAM.

During the training, we saved a checkpoint every 50K steps, evaluating the performance on the evaluation set. We aimed to use early stopping, however, in both trainings, the best checkpoint was the last one for the pre-trained model and the second-last for the non pre-trained one. We are aware that this indicates potential margin for improvements for both models. However, the increment in performance among the latest checkpoints was very minor as it can be seen in Fig. 5.1. Thus, we decided to not invest additional time in further training the models.

Table 5.4. Configuration used for the CodeT5 training.

Hyperparameter	**Value**
Learning rate	$5e^{-5}$
Batch size	12
# Encoder block	12
# Decider block	12
# Attention Heads	12
Hidden Layer Size	768

5.2.3 Evaluation and Analysis

We evaluated both trained models (*i.e.,* with and without pre-training) on all the 27 test sets (*i.e.,* 9 Java versions × 3 masking scenarios), collecting their predictions. For each test set, we compute the percentage of Exact Match (EM) predictions. We focus our discussion on the gap of performance (if any) existing between the EMs observed on the Java 8 test set (featuring instances written using the same language version used for the training and hyperparameters tuning) and those obtained on other Java versions, both those preceding and following version 8.

We statistically compare the results achieved on the Java 8 test set and on all other test sets assuming a significance level of 95%. We compute the Fisher's exact test (and related OR) on a matrix containing, for the different test sets and for different masking levels, the number of correct (EM) and incorrect predictions. To account for multiple test instances—

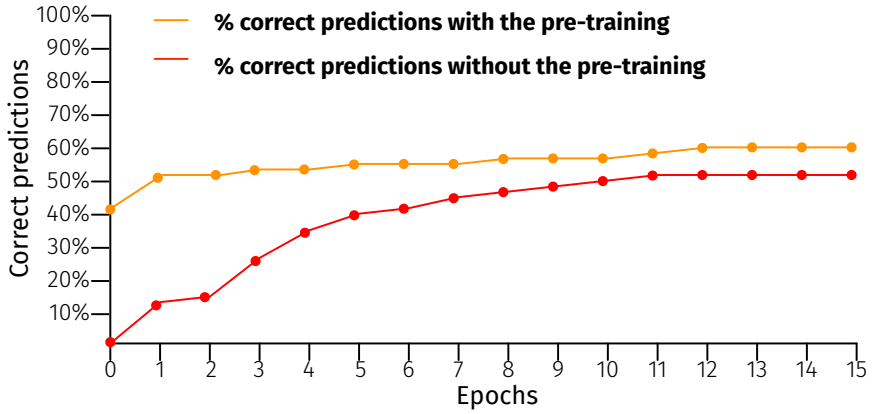

Figure 5.1. Percentage of Exact Match predictions for both models at different training epochs.

we contrast the performance on Java 8 with all other versions—we adjust p-values using the Benjamini-Hochberg procedure [YY95].

5.3 Results Discussion

We discuss the achieved results by presenting: (i) performance differences across language versions; (ii) possible reasons behind the observed differences in performance; and (iii) the impact of version-specific fine-tuning on the model performance. The combination of these analyses allows us to provide a comprehensive answer to our RQ.

5.3.1 Performance Differences

We computed the percentage of EM predictions for the test set of each Java version, both for the pre-trained and non pre-trained model. Fig. 5.2 reports the results obtained. The x-axis shows the Java version, with the distance between data points being proportional to the time passed between Java version releases. The y-axis shows the percentage of correct predictions made by the pre-trained (dashed lines) and non pre-trained (continue lines) models in the three masking scenarios, *i.e.,* token-level (orange line), construct-level (red), and block-level (dark red).

As illustrated, the trend in performance is similar for different masking scenarios and models. As expected, the best results are obtained for Java 8 (*i.e.,* when testing the model on the same Java version seen during training), where the pre-trained model is able to correctly predict up to 70% of instances for the token-level masking scenario. The percentage of EM predictions gradually decreases as we move away from Java 8, both back and forward in time. Despite the similar trend overall, there are notable differences across language versions and masking scenarios. As expected, block-level masking is the most challenging scenario, since the model has to guess up to several entire statements, while token-level masking is the easiest one, with just a few tokens to predict. Construct-level masking is in the middle:

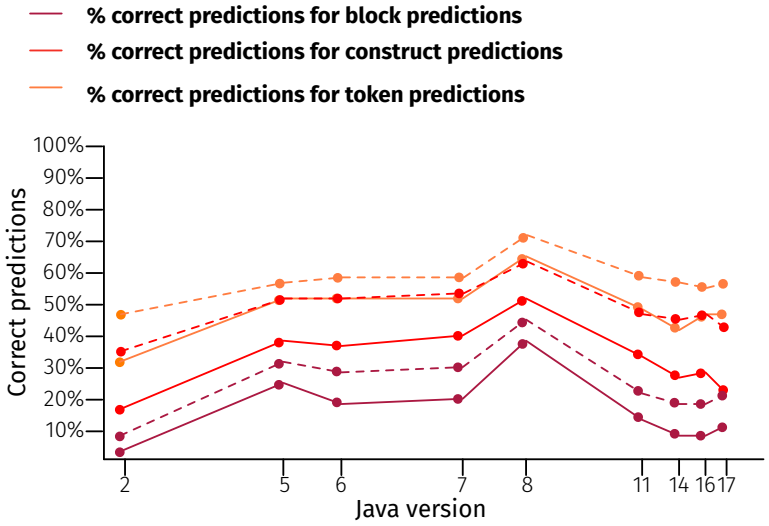

Figure 5.2. Percentage of Exact Match predictions with and without pre-training on different Java versions.

constructs define the application logic and hence are difficult to predict, while still limited in size.

The impact of pre-training is relevant, with an average improvement across all Java versions and masking scenarios of +9%.

In detail, looking at the performance of the model for each specific completion scenario, the improvement goes from 5% to 16% for token-level masking, and from 12% to 19% for construct-level masking. For block-level masking the improvement is less evident, ranging from 3% to 8% block, likely due to the difficulty of the task. Thus, pre-training the model can be very valuable, especially when the predictions are quite challenging, involving for example logic constructs.

The pre-training dataset is a mixture of all different Java versions [HWG+19] and this can help the model in predicting code that, despite being different from the one used for training (fine-tuning), has been seen during the pre-training phase.

In terms of language version, the worst results are obtained for Java 2, probably because this is an archetypal version of the language (Java 2 was released in 1998 while Java 8 in 2014). It is worth noticing that the performance on the last three versions (*i.e.,* Java 14, 16 and 17) is very similar, likely due to very small differences between these versions, all released between March 2020 and September 2021.

We also reported in Table 5.5 the results of the Fisher's Exact test (and related OR) when comparing the performance on the Java 8 test set and all other test sets (in terms of EM predictions) for the non pre-trained model (results for the pre-trained one are available in our replication package [repc]).

The p-values, after adjustment, are always very close to 0, indicating a statistically significant difference in the performance observed on the different test sets. The OR goes from

Table 5.5. Comparing EM predictions on the Java 8 test set *vs* the test sets of the other versions: adjusted *p*-value and OR.

Java version	Masking scenario	Fisher p-value	Fisher OR
2	token	$\ll 0.001$	3.74
	construct	$\ll 0.001$	4.73
	block	$\ll 0.001$	13.73
5	token	$\ll 0.001$	1.73
	construct	$\ll 0.001$	1.84
	block	$\ll 0.001$	1.72
6	token	$\ll 0.001$	1.69
	construct	$\ll 0.001$	1.92
	block	$\ll 0.001$	2.28
7	token	$\ll 0.001$	1.59
	construct	$\ll 0.001$	1.63
	block	$\ll 0.001$	2.16
11	token	$\ll 0.001$	1.70
	construct	$\ll 0.001$	1.97
	block	$\ll 0.001$	2.65
14	token	$\ll 0.001$	2.06
	construct	$\ll 0.001$	2.51
	block	$\ll 0.001$	3.92
16	token	$\ll 0.001$	1.83
	construct	$\ll 0.001$	2.27
	block	$\ll 0.001$	3.66
17	token	$\ll 0.001$	1.78
	construct	$\ll 0.001$	3.05
	block	$\ll 0.001$	2.64

1.59 to 13.73 indicating much higher odds of observing a correct prediction on the Java 8 test set rather than on the others. For example, in the comparison between Java 8 and Java 14 in the token-level scenario, the OR=2.06 indicates that the odds of an EM prediction for Java 8 are $\sim$2 times higher than for Java 14.

Key insight: Regardless of the masking scenario and the use of pre-training, there are significant performance differences across language versions, ranging from 11% (w.r.t. Java 5, block masking, non pre-trained model) to 37% (w.r.t. Java 2, block masking, pre-trained model).

5.3.2 Reasons Behind Performance Differences

The observed drop in performance provides strong evidence of the *concept drift* [GZB+14, LLD+19] issue caused by programming language evolution. In this section, we try to better link the observed drop in performance to the changes implemented across the Java versions. In particular, we looked for the official Java documentation reporting the new APIs introduced in each Java version different from Java 8. Unfortunately, we only found this

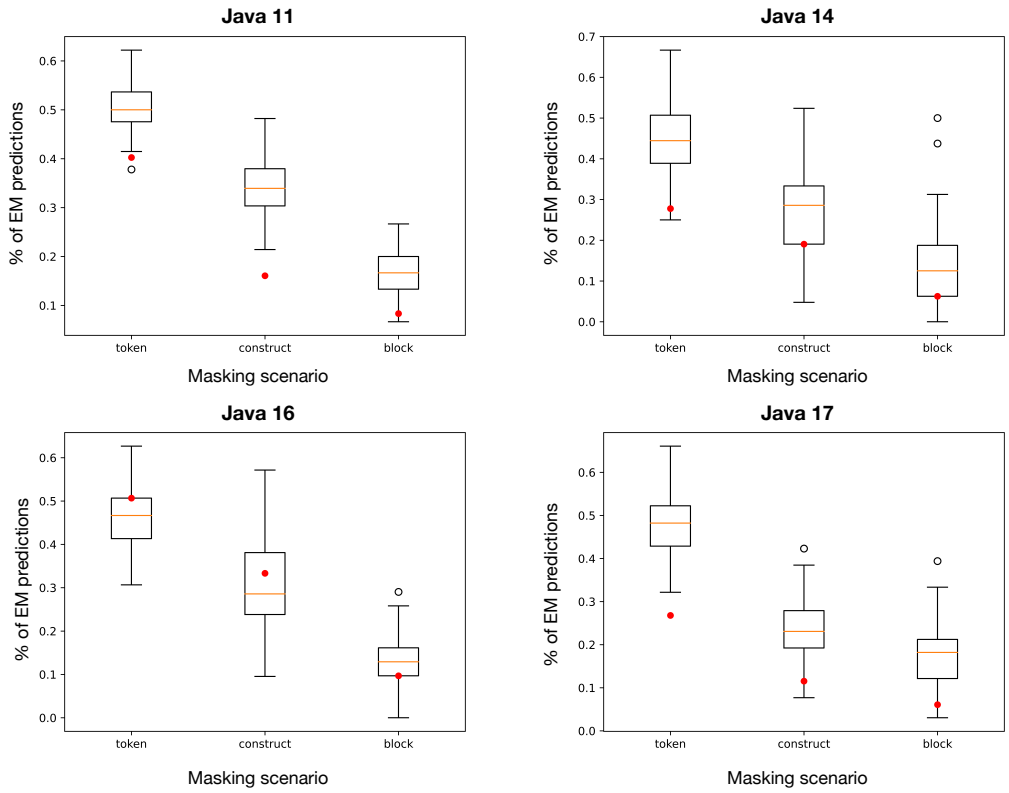

Figure 5.3. Distribution of the percentage of EM predictions for 100 subsets of instances not featuring new APIs (box plots) *vs* the percentage of EM predictions in the set featuring new APIs (red dots). On average, each set features 43 instances.

information for the newer versions (*i.e.,* those following Java 8), and only for the versions listed in Table 5.6 we identified new APIs.

Based on this information, we conducted an analysis to compare the percentage of EM predictions in the instances from the test sets containing new APIs against the percentage of EM predictions in the instances not featuring any new APIs released after Java 8. The goal of this analysis is to understand whether the presence of new APIs (not seen in the Java 8 training set) in the code to predict has an impact on the model performance.

It is worth mentioning that the code containing new APIs represents less than 1% of the instances of the test sets. Thus, the observed drop in performance is certainly not *only* due to the new APIs introduced in the new Java versions.

For example, Java 9 also introduced an improved `try-with-resource` statement and the diamond operator extensions which partially changed the language syntax.

Still, we compare the performance of the code completion model when dealing with instances featuring and not featuring new APIs to get an idea of what the impact of new "code tokens" unseen in the training set can be. To allow for a fair and robust comparison, given n the number of instances featuring new APIs in the test set of Java version j (with

Table 5.6. New APIs introduced in specific Java versions.

Java version	Features
9	*List.of, Set.of, Map.of, takeWhile, iterate, dropWhile, ifPresentOrElse*
10	*List.copyOf, orElseThrow*
11	*isBlank, lines, strip, stripLeading, stripTailing, repeat, Files.writeString, Files.readString, HttpClient, HttpRequest, HttpResponse*
12	*indent, transform, Files.mismatch, Collector.teeing, NumberFormat.getCompactNumberInstance*
13	*newFileSystem, stripIndent, translateEscapes, formatted*

$j \in \{11, 14, 16, 17\}$, namely one of the versions following Java 8 and considered in our study) we randomly select from the same test set 100 subsets of n instances each, all not featuring new APIs. Then, we compare the percentage of EM predictions obtained on the set featuring instances with new APIs with the distribution of EM predictions obtained for the 100 subsets of equal size featuring instances not exploiting new APIs.

Our findings are presented in Fig. 5.3. For each Java version and masking scenario, we report a boxplot illustrating the distribution of the percentage of EM predictions across the 100 randomly sampled subsets, *i.e.,* those not featuring new APIs. We also report a red dot representing the percentage of EM predictions in the instances containing new APIs. As illustrated, for the Java 11, 14, and 17 test sets, the performance observed on the instances featuring the new APIs usually falls below (or inline) with the first quartile, indicating that in $\geq$75% of cases, we observed better performance in the instances not featuring new APIs. This holds for all code completion scenarios (*i.e.,* token, construct, block). The only exception to this trend is Java 16, in particular when dealing with token- and construct-level completions.

Key insight: The introduction of new APIs in the language has, in most cases, an impact on the model performance, with a noticeable drop of EM predictions for instances featuring new APIs. Still, no strong claims can be made on this finding given the lack of statistical analysis, which we do not perform given the low number of instances in the test sets exhibiting new APIs (43, on average, in each of the 12 test sets).

5.3.3 Impact of Version-Specific Fine-Tuning

Given the major drop in performance observed when moving away from Java 8, possible strategies to address such a performance decrease are worth being investigated. For this reason, we studied the extent to which a small additional fine-tuning performed on each of the eight Java versions $v_j \neq 8$ may increase performance on the v_j test set. In particular, we created eight additional fine-tuning datasets (one per each Java version different from Java 8) by using methods that have been excluded while building the test sets (*i.e.,* excluded by

Table 5.7. Number of methods and instances of the version-specific fine-tuning datasets for each Java version.

Java Version	# Methods	# Token Instances	# Construct Instances	# Block Instances
2	1,530	4,475	510	750
5	3,549	8,552	1,840	4,237
6	63,379	168,605	49,959	79,123
7	94,405	252,722	68,561	22,124
11	92,153	105,700	33,111	38,982
14	1,160	3,348	1,664	1,894
16	2,169	6,237	1,378	2,956
17	915	2,493	576	1,073
ALL	**259,260**	**552,132**	**157,599**	**151,139**

Algorithm 1) and thus not used in any step of our study. This resulted in the datasets listed in Table 5.7. The datasets have different sizes, also allowing us to observe whether particularly small datasets (*e.g.,* Java 17) are anyway sufficient to observe any practical improvement. Each of these datasets has been split into 90% for the additional fine-tuning and 10% for evaluation. The fine-tuning has been performed for only five epochs on top of the model fine-tuned for Java 8, in an attempt to simulate its adaptation to a different Java version.

In total, 16 new models have been trained (*i.e.,* 8 Java versions with/without pre-training). We assessed the EM predictions of the models after each training epoch on the corresponding evaluation set, selecting the best performing one to be run on the test set (*i.e.,* the one adopting the same version as the version-specific fine-tuning dataset).

Fig. 5.4 reports the achieved results for the non pre-trained (a) and for the pre-trained (b) models. Each subfigure features eight pairs of bars, one pair for each of the eight Java versions for which we further fine-tuned the model (*i.e.,* all but Java 8). The red bars represent the absolute improvement in EM predictions observed on the test set adopting the same version used for further fine-tuning the model, while the orange bars report changes in performance observed on the Java 8 test set. We computed the performance on both test sets in an attempt to understand whether the adaptation of the model to a specific Java version (different from Java 8) may have a negative impact on the version that the model was originally trained on (Java 8). We statistically analyzed the results using the Fisher's exact test and related OR, comparing the performance of the model with and without the version-specific fine-tuning for each of the test sets previously described. For the statistically significant comparisons (*i.e.,* p-value < 0.05), we reported the OR value on the top of each bar. Since the trend is similar for both pre-trained and non pre-trained model, we focus our discussion on the non pre-trained one, *i.e.,* Fig. 5.4 a).

The second round of fine-tuning significantly increased the accuracy of the predictions. The most notable improvement was obtained for Java 2—the most negatively affected version by the concept drift problem—where the percentage of correct block-level predictions jumped from 3% to 33%, with an overall improvement of 31% across all masking scenarios. All improvements resulted in statistically significant differences with large ORs (*i.e.,* 2.5 in the token-level, 5.9 in the construct-level, and 11.1 in the block-level).

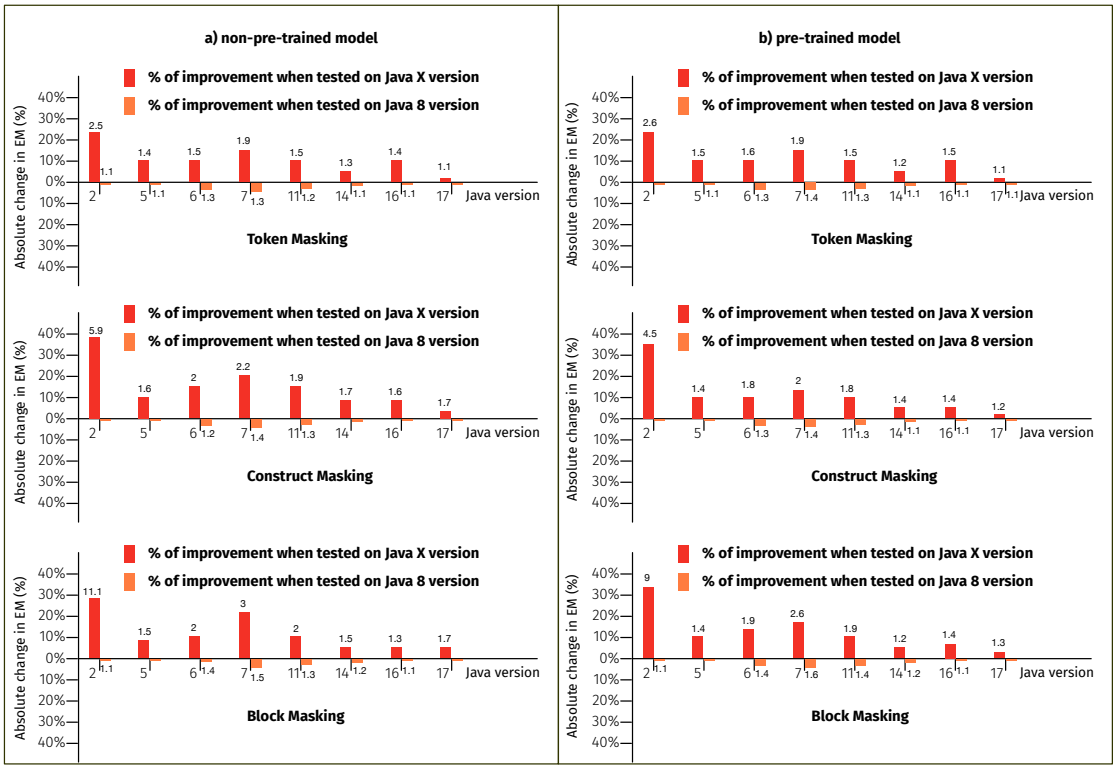

Figure 5.4. Difference in the percentage of EM predictions with and without the version-specific fine-tuning, evaluated on the same Java version test set and on the Java 8 test set.

For the other Java versions, the improvement ranges from 3% to 15% for token-level predictions, from 7% to 39% for construct-level predictions, and from 4% to 29% for block-level predictions. Overall, the average improvement is 11% across all masking scenarios. Interestingly, even a rather small fine-tuning dataset such as the one used for Java 17 still allowed to achieve statistically significant better performance in all masking scenarios, although with ORs limited to a maximum of 1.7.

Such a major improvement has however a small price to pay. Indeed, re-adapting the model to a specific version did have an impact on the performance of the model for the version it was originally trained on (Java 8). However, as shown in Figure 5.4 (orange bars), the performance drop was very small in most cases (4% on average). These findings show that a short additional fine-tuning on a specific Java version can significantly improve the performance of the model on that version, while incurring a negligible performance degradation for the original version the model was trained on.

Key insight: A limited fine-tuning—with few training instances and epochs—on a specific language version can lead to significant performance improvements in the model predictions (up to 40%) at the cost of a negligible performance drop in the original version (4% on average). The most noticeable improvements are achieved for the most challenging tasks, *i.e.,* construct-level, and block-level predictions.

5.4 Threats to Validity

Conclusion validity. We applied appropriate statistical analysis, using specific statistical tests and effect size measures, following common guidelines in the literature [AB14, YY95].

Construct validity. These revolve mainly around the way in which we evaluate the task of code completion, *i.e.,* by masking code elements. While we acknowledge that this might not be completely representative of how developers write code, we evaluate three different completion tasks (token-level, construct-level and block-level) that allow us to evaluate the model performance in different scenarios and with different amounts of masked code. Another threat in this regard is *how* we evaluate the model performance, *i.e.,* by measuring the percentage of *Exact Matches*. While it is possible that the model predicts different but semantically equivalent code, we believe that this metric is still a good proxy for the model performance, as previous work shows that distinct code suggestions tend to be semantically different [CCP+21b, ARSH14, ASLY20, HD17].

Internal validity. On the one hand, it is possible that the observed performance differences across language versions may be due to the specific test sets used for the experiments. To partially address this threat, we implemented Algorithm 1, whose goal is to balance the complexity of the predictions in the tests. On the other hand, the impact of version-specific fine-tuning (Section 5.3.3) might depend on the size of the fine-tuning datasets. In acknowledging this, we reported the sizes of all version-specific datasets used (Table 5.7).

External validity. Our study is characterized by the selected DL model, programming language, and reference language version. CodeT5 was selected as a representative code model that has demonstrated strong performance across a range of tasks [WWJH21]. The choice of Java was motivated by its popularity and widespread use, and because it is possible to reliably identify a project's Java version based on its `POM` file. Finally, we selected Java 8 as the reference version due to its prevalence in the dataset, as shown in Table 5.1.

5.5 Conclusion and Future Work

Programming languages evolve rapidly. To support developers effectively, coding tools must keep up with this fast pace. DL-based code completion approaches and, more broadly, intelligent coding assistants such as the recently released GitHub Copilot Chat [git] are inevitably affected by these changes. In this thesis, we investigated the impact of language evolution on the performance of code models for the task of code completion. Our findings

demonstrate that DL-based code completion models are susceptible to the concept drift problem, with significant performance differences across different language versions, particularly when deviating from the target version used for training. In aiming to mitigate this problem, we evaluated the impact of fine-tuning the model with a limited version-specific dataset. The results are encouraging, with noticeable gains in performance even with datasets of just a few hundred samples. We believe that our research can inform the design and deployment of more robust code completion models, which can be continuously refined as new training data (code from new language versions) becomes available.

Our future research will target two main goals. First, we plan to make our study more comprehensive and extend it to other DL models and programming languages. Second, we will delve deeper into the impact of the small version-specific fine-tuning on the model performance, as well as the perceived improvements by developers in real-world scenarios. These efforts will contribute to the development of adaptable code completion tools, ensuring they remain effective amid the dynamic nature of programming languages.

6

Source Code Recommender Systems: The Practitioners' Perspective

6.1 Motivation

Recommender systems are becoming more and more popular in software engineering. These tools can support developers in several tasks [RMWZ14], such as documentation writing and retrieval [XP06, MBP+15, MBP+17, HLX+18], code refactoring [BLMO14, TCC18], bug fixing [GDFW12, TWB+19, LWN20], bug triaging [TNAN11, XLD+17], code review [TPT+21, TMM+22] etc. Among these, *source code recommender systems* support developers in writing code.

While research on code recommender systems is extremely active, with new tools frequently released, able to achieve new state-of-the-art performance, no previous work investigated what the *desiderata* of software developers are. In other words, the techniques proposed in the literature are mostly based on assumptions made by researchers. For example, a recent work by Wen *et al.* [WAN+21] targets the automatic implementation of whole code functions. However, it is unclear whether developers are actually looking for such a type of support or if, instead, they prefer the classic code completion implemented in IDEs. Also, none of the existing tools and techniques customize their code recommendations based on the developer's coding style and/or to their expertise and it is unclear whether such a functionality would be important for practitioners. To answer these questions, we present a study involving 80 practitioners which is aimed at investigating the characteristics of code recommenders that they consider important. In particular, after collecting demographic information, we asked participants their opinion about the code recommenders they use (*e.g.,* copilot, default code completion in IDE, etc.) from three perspectives: (i) *coverage* (*i.e.,* in how many coding scenarios the tool can provide recommendations), (ii) *accuracy* (*i.e.,* to what extent recommendations are close to what practitioners need), and (iii) *usability* (*i.e.,* how friendly the user interface is). For each of these three aspects participants were asked to describe improvements (if any) they would like to see in the recommender they mostly use. Then, we ask them in an open-ended question what the characteristics of code recommenders they consider important are.

Each answer we received has been independently analyzed by five authors through an open-coding inspired approach with the goal of assigning a set of tags to it. The extracted tags represent requirements expressed by practitioners for code recommenders and, after conflict resolution, have been organized in a hierarchical taxonomy. Such a process has been performed iteratively four times, with 20 practitioners taking part in each iteration. We stopped once the output taxonomy converged (*i.e.,* no additional requirements were added to the taxonomy from the answers we received in the last iteration).

The output of our study is a taxonomy of 70 "requirements" that should be considered when designing code recommender systems (Fig. 6.2). For example, our taxonomy highlights that practitioners are interested in *adaptive* recommendations, meaning that the recommended code should be automatically adapted to the code under development (*e.g.,* reusing identifiers when possible) and to the developer's coding style.

Significance of research contribution. The taxonomy of 70 "requirements" for code recommenders, the output of our study, provides a rich research roadmap in the field of code recommender systems. Indeed, as our empirical evidence shows, the *desiderata* of practitioners in this context are not always aligned with what offered by state-of-the-art techniques.

Based on our taxonomy, researchers can have a clear understanding of what the priorities are when designing code recommenders.

Data availability. We release the survey used in the study, the collected answers with the results of the manual analysis we performed on them, plus additional material in our replication package: `https://code-recommenders.github.io`.

6.2 Study Design

The *goal* of the study is to investigate what the *desiderata* of software practitioners are when it comes to code recommender systems. The *context* consists of *objects*, *i.e.,* a survey designed to investigate the study goal, and *subjects* (referred to as "participants"), *i.e.,* 80 practitioners recruited through Amazon Mechanical Turk (MTurk) [amt] and personal contacts.

We aim at answering the following research question:

> *What are the characteristics of code recommender systems that are considered important by practitioners?* Despite the many code recommender systems proposed in the literature, no study investigated what practitioners actually need in terms of automatic support during coding activities. Answering our RQ can guide the development of better code recommender systems.

6.2.1 Context Selection — Participants

We recruited participants through two channels. First, we used MTurk [amt], a crowdsourcing website to hire people for on-demand tasks. We enrolled participants that (i) have successfully completed in the past at least 50 tasks on MTurk; (ii) have an approval rate for their past tasks greater than 90% (*i.e.,* more than 90% of the tasks they performed in the past have

been approved by the requester); (iii) hold the MTurk Master qualification assigned to "top workers". We only involved practitioners in our survey, excluding students (at any level) and researchers.

Participants who completed our survey were paid 10$ upon a manual verification in which we made sure that the provided open answers (described in the following) were meaningful and written in correct English. We collected 31 complete surveys from MTurk. In addition, we invited practitioners in the authors' contact network. This resulted in an additional 49 answers, leading to a total of 80 participants. As explained later, developers were not invited altogether, but in four rounds of 20 participants each. Demographics about participants are presented in Section 6.3.

6.2.2 Context Selection — Survey

Our survey has been implemented in Qualtrics [qua] and is available in our replication package [repd].

Participants were initially presented with a welcome page, which explained the goal of the study, reported its expected duration (∼15 minutes), and set the context by explaining what source code recommender systems are:

> *With source code recommender systems we refer to approaches that can be used to automatically suggest code to developers while they are writing code. The classic example in this context is the code completion feature implemented in IDEs.*
> *However, some tools go beyond the classic code completion task and recommend longer pieces of code to developers to autocomplete a task they perform (see,* e.g.*,* `https://copilot.github.com/`*).*

By agreeing to participate, they started our survey composed of three steps.

Step ①: Demographic Information. We asked participants to indicate (i) their job position (*e.g.,* developer, tester), (ii) the programming language and the IDE they mostly use, and (iii) the number of years of programming experience.

Step ②: Experience with Code Recommenders. The second step included questions about the participants' experience with code recommender systems. We asked what tool(s) they use as code recommender, with the possibility of selecting "*The default one in the IDE*" and/or specifying the tool(s) in an open text box. If participants indicated that they did not use any code recommender, the survey stopped. We also collected the frequency with which participants check code recommendations: Occasionally (*i.e.,* less than 50% of times a recommendation is available), Most of times (*i.e.,* more than 50%, but not always), and Always.

Then, we asked to rate the code recommendation capabilities of the tool they mostly use from three perspectives: (i) *coverage* (*i.e.,* in how many coding scenarios the tool can provide recommendations), (ii) *accuracy* (*i.e.,* to what extent recommendations are close to what practitioners need), and (iii) *usability* (*i.e.,* how friendly the user interface is). For each of these three aspects, participants could indicate their answer on a three-point scale (Low, Medium, High), or select a "Not sure" option. We provided participants with detailed

Table 6.1. Characteristics of code recommendations collected from literature

Attribute	Description	References
Concise Code	The recommended code must be as short and simple as possible.	[KLHK09, NSMB12, HPGB19]
Correct Code	The recommended code must be bug-free.	[KLHK09]
Familiar	If multiple recommendations are possible, the one using code that is more familiar to the developer must be used (*e.g.,* the code using APIs already used in the past by the developer receiving the recommendation).	[RTX09]
High readability	The recommended code must be readable (*e.g.,* avoid very long statements, adopt indentation).	[MBP+15]
High reusability	The recommended code must be easy to reuse (*e.g.,* a code using object types not available in the language but defined in other projects from which the recommended code has been learned is difficult to reuse).	[MBP+15]
Inline Documentation	The recommended code features comments explaining the code step-by-step.	[NSMB12]
Meets coding layout	The recommended code must be adapted to the context of the recommendation by adopting the same coding layout (*e.g.,* same indentation, spaces between code tokens).	[KATF18]
Meets naming conventions	The recommended code must be adapted to the context of the recommendation by adopting the same naming conventions (*e.g.,* if variables are named with camelCase, the same convention must be adopted in the recommended code).	[KATF18]
Precise Typing Information	A recommended code using AVerySpecificType should be preferred over a recommended code using Object.	[PGBG12]
Responsive	The responsiveness of the code recommendation system in terms of time needed to generate a recommendation.	[SDFS20]
Same name	The recommended code must be adapted to the context of the recommendation by using the same variable names of the code it completes when possible.	[PGBG12]
Syntactical Correctness	The recommended must not introduce syntax errors.	[dSAEWV16, WSLJ20, SDFS20]
Step-by-step Solution	In case the recommended code spans across many statements, the code is divided into multiple chunks (by using a blank line), each one responsible for a sub-task.	[NSMB12]
Vulnerability-free	The recommended code must be vulnerability-free.	[SSTS21]

explanations about the three perspectives [repd]. For each of them, we also asked the improvements participants would like to see in the code recommender(s) they commonly use (*e.g.,* what they would like to have in terms of "coverage" that is not currently supported). Finally, we ask a specific question related to the accuracy of the recommendations: *Assume that a code recommendation tool provides a list of recommendations sorted by likelihood of being relevant (*i.e., *the most relevant on top). How many of these recommendations would you be willing to read to find the right one?* Answers to this question can inform the evaluation of code recommenders by researchers. Indeed, when evaluating approaches for automatically generating code, researchers often assess their performance for the top-*k* recommendations (*e.g.,* top-50 in [TWB+19]). Knowing how many recommendations developers are willing to inspect can help in setting the number of generated solutions to realistic values.

Step ③: Characteristics of Code Recommenders. The third and last step is the core of our survey, in which we asked participants about the characteristics of code recommendations they consider important. In the first question, participants could describe in an open text box the characteristics of code recommendations they perceive as most important, accompanying each one with a short explanation/rationale. We clarified that they could include both functional and non-functional characteristics.

In the second question, the survey showed a list of 14 characteristics we preliminarily defined, asking participants to select the ones they consider important (if any). To determine such 14 categories, we manually analyzed the state of the art related to code recommender systems. Specifically, we focused on papers that presented: (i) techniques for code generation/completion (see *e.g.,* [SDFS20]), (ii) empirical studies about code generation/completion techniques (*e.g.,* [SSTS21]), (iii) techniques to generate code examples (*e.g.,* [MBP+15]), and (iv) empirical studies about code examples used by developers (*e.g.,* [NSMB12]). For example, we extracted the *high reusability* characteristic from the paper by Moreno *et al.* [MBP+15].

The full list of characteristics with the corresponding references they were extracted from is available in Table 6.1. Note that, to define such a list, we did not perform a systematic literature review to identify **all** papers in the surveyed areas: We relied on the experience of the six authors to identify a set of 41 peer-reviewed papers published in international conferences and journals and applied backward snowballing on their references to identify additional relevant works.

At the end, we inspected 53 papers. Each paper was assigned to one author in charge of adding to a spreadsheet the list of "characteristics" described in the paper (if any). In some cases, the papers from which we extracted a characteristic did not explicitly point to the need for considering such a characteristic when building code recommender systems.

However, this could be inferred from the text of the paper. One example is the work by Schuster *et al.* [SSTS21]. The authors show that "*neural code autocompleters are vulnerable to poisoning attack*" [SSTS21]. Thus, we infer that *Vulnerability-free* is one of the characteristics to assess for code recommenders. Similarly, Nasehi *et al.* [NSMB12] studied what makes a good code example on Stack Overflow. Again, this is something not directly linked to a code recommender. However, we assume that characteristics of good code examples could be relevant for recommended code as well.

Table 6.2. Rounds of data collection (20 participants each)

Round	New L1 Charac.	New L2 Charac.	New L3 Charac.	New L4 Charac.	Conflicts
I	5	25	13	0	24%
II	0	9	6	6	21%
III	0	0	6	0	24%
IV	0	0	0	0	24%

6.2.3 Data Collection and Analysis

Given the goal of our study, we decided to run it in multiple rounds until we reached saturation in the collected taxonomy of characteristics. First, we invited 20 practitioners to complete our survey (first round). Then, to answer our RQ, we analyzed the data collected in steps ② and ③ in the open answers by following an open-coding inspired approach: Five of the authors independently assigned a set of tags to each of the open answers provided by the 20 participants that described through written text the improvements they would like to see in terms of coverage, accuracy, and usability of the code recommenders they use and the characteristics of code recommenders they perceive as most important. Each tag was meant to encode a specific characteristic (*e.g., Early prediction* was derived from the answer "*if the completions are not timely, it is just easier to type the whole thing myself sometimes*").

Conflicts (*i.e.,* different tags assigned by the five authors to the same answer) were solved through online meetings involving all authors. The set of "characteristics" derived through this analysis was then complemented with the ones selected by developers as important from the list extracted from the literature. The output of this analysis is a hierarchical taxonomy of characteristics of code recommenders (Fig. 6.2), with level-1 nodes indicating root categories, level-2 nodes indicating sub-categories of a specific root category, and so on.

This first round was followed by three additional rounds, each of which added 20 participants. We stopped with this process when the execution of a new round did not result in the addition of any new characteristic in our taxonomy, indicating that a good level of saturation was reached. We are aware that additional rounds may further strengthen the generalizability of our taxonomy. However, we had to balance the comprehensiveness of the taxonomy with the feasibility of the study given our limited access to professional developers.

Table 6.2 reports for each round (i) the number of new characteristics added to the different levels of the taxonomy and (ii) the percentage of conflicts arisen during open coding. The number of characteristics reported in Table 6.2 for each level does not match the final number in Fig. 6.2: This happens because we reorganized the taxonomy after each round for readability reasons (*e.g.,* some level-2 characteristics were moved to level-3 as child of a new level-2 category). However, the overall number of characteristics (70) matches the one in our final taxonomy (Fig. 6.2).

Concerning conflicts, since we had five authors inspecting each answer in an open coding setting in which the codes (*i.e.,* characteristics of code recommenders) to extract were not pre-defined but had to emerge from the data, we had some form of conflict very frequently, *e.g.,* one of the five authors not extracting a characteristic indicated by the remaining four

authors. These cases were usually trivial to solve in online meetings. Other types of conflict required instead longer discussions, and these are the ones we document in Table 6.2.

In particular, we report the percentage of cases in which there was no majority in extracting a "characteristic" from an open answer (*i.e.,* less than three authors reported it). As it can be seen, this happened in ∼20% of cases in each round.

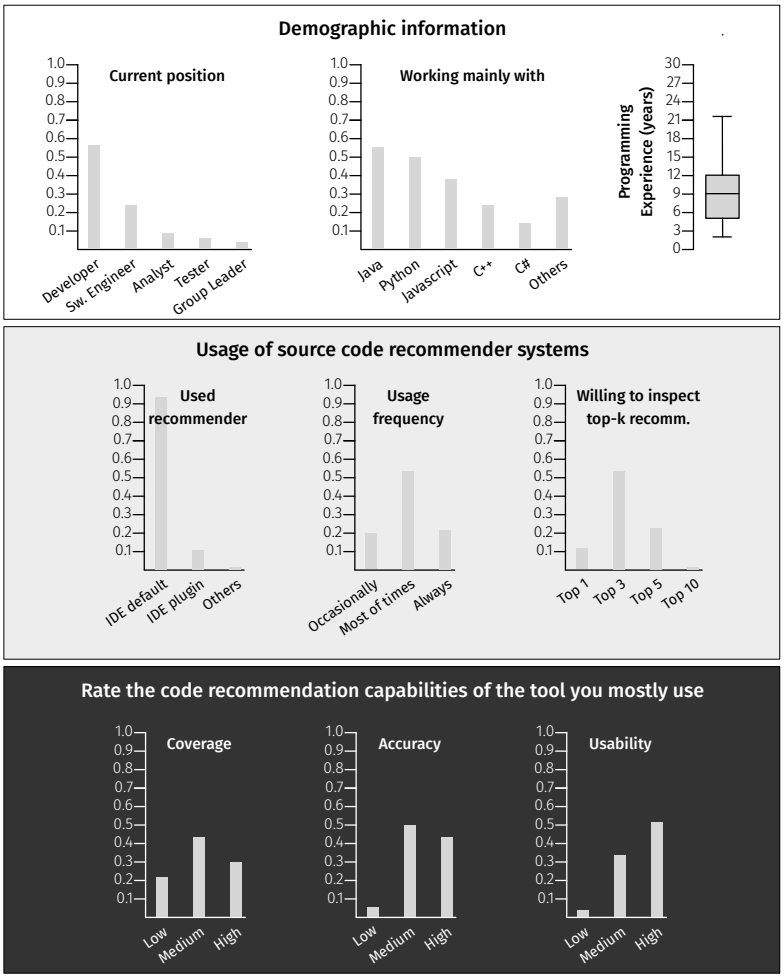

Figure 6.1. Demographic information (top), usage of source code recommenders (middle), and assessment of code recommender systems (bottom).

6.3 Results Discussion

We start by summarizing demographic information about the study participants and their experience with code recommender systems. Then, we discuss the taxonomy of characteristics of code recommender systems, which is the main outcome of this study.

6.3.1 Demographics and Experience with Code Recommenders

Fig. 6.1 presents a visualization of demographic data for the 80 practitioners involved in our study. Most of the participants are developers (47) or software engineers (20), with others classifying themselves as analysts, testers, and group leaders. Participants mostly work with Java, Python and/or Javascript.

In terms of experience, 86% of participants has more than five years of programming experience, with an average of 9.7 years, and a median of 9. The vast majority (85%) of participants use the IDE's built-in code completion feature as their *only* code recommender system.

Among the used IDE plugins, GitHub Copilot is the most represented one, with 5 mentions all coming from the latest round we performed. Indeed, when we ran the first three rounds Copilot was not yet publicly available.

Less than 19% of the participants claimed to "occasionally inspect the code recommendations provided", with the remaining ones looking at them *most of times* or *always* when they are available.

Participants indicated the willingness to inspect at most the top-5 code recommendations provided (95%), with the majority (56%) focusing only on the top-3. This suggests researchers to limit the assessment of code recommenders to the top-5 recommended solutions, since others are very unlikely to be considered by developers.

When asked about their assessment of the code recommender they use in terms of coverage, accuracy, and usability, developers reported that they are mostly happy with the tools they use as for these aspects. Specifically, ∼44% of them considered the accuracy high, and ∼50% medium; ∼58% evaluated the usability as high, and ∼38% as medium; and ∼31% judged the coverage high and ∼46% medium. The coverage of the provided support (*i.e.,* the variety of scenarios in which the tool is able to recommend code) is the aspect achieving the lowest rates, with ∼23% of participants reporting a low coverage.

6.3.2 Taxonomy Discussion

Fig. 6.2 reports the taxonomy of characteristics participants indicated as relevant for code recommender systems. The number on the top-right of the level-1 and level-2 categories indicates the number of participants who mentioned such a characteristic as important (out of 80). While some characteristics have only been mentioned by few developers (*e.g., Awareness* → *Developer's task*) we decided to include all of them for completeness.

On top of that, Fig. 6.3 reports the definition of all categories up to level-3, without including the ones for level-4 categories. However, these usually represent specifications of level-3 categories that are quite intuitive and we include in our replication package [repd] complete definitions for all categories, with a search interface that helps in quickly identifying the category of interest.

We discuss our taxonomy, going through each of the five root categories depicted in Fig. 6.2. We use the 💡 icon to highlight lessons learned for future research in the field.

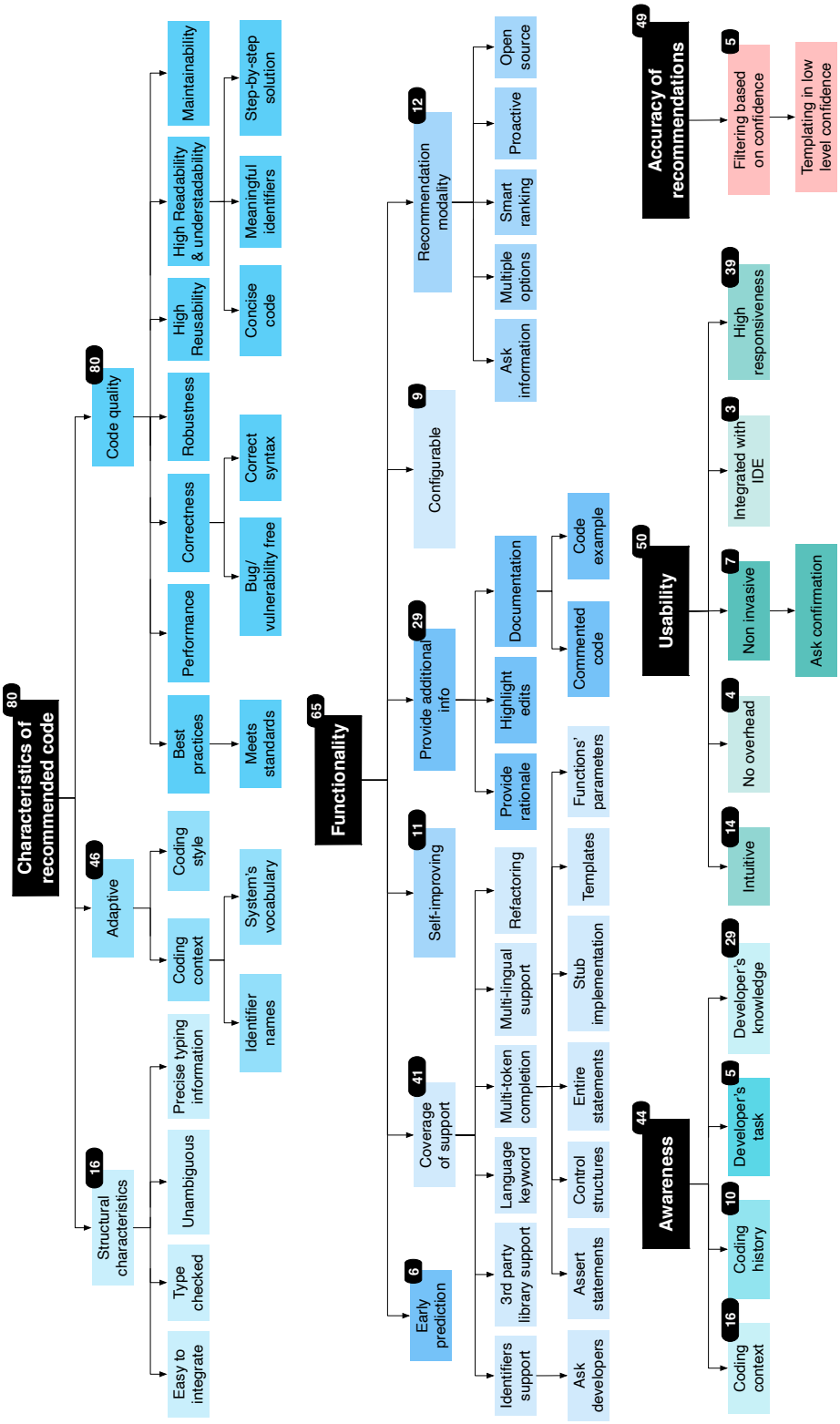

Figure 6.2. Taxonomy of characteristics of code recommender systems

Characteristics of recommended code. This root category groups characteristics of the recommended code that participants consider important. All participants mentioned at least one aspect related to the *quality* of the recommended code. Note that, at a first sight, one may think that code quality is not relevant for code recommenders, since they often recommend a few code tokens to complete a statement the developer is writing. This is, however, not the case for the new generation of code recommenders (see *e.g.,* GitHub Copilot [copb] or recently proposed works in the literature [SDFS20, WAN+21, CCP+21a]) that can recommend complete code statements or even entire functions.

Receiving recommendations having a high readability and understandability is a priority for developers (66 mentions). 39% of the participants indicated their preference for concise recommendations, *e.g., "I do not use completion that suggests entire snippets. The reason is that if I get an entire snippet I'll need to understand it to make sure is what I need and, based on my experience, this is not faster than writing the code myself."* This goes somehow in contrast with recent work targeting the automatic implementation of whole functions [WAN+21, copb]. However, as we will see later when discussing the *Coverage of Support*, developers are willing to use recommendations for complex scenarios (such as entire functions) if they have a high confidence in the received recommendations. Also, in case of recommendations composed by several statements, 14 participants indicated the importance of organizing these recommendations as *Step-by-step solutions*, meaning that the code is divided into multiple chunks (using a blank line), each one responsible for a sub-task. Such a feature requires the ability of the recommender to identify sub-tasks within the suggested code.

More in general, readability and understandability are two important aspects of code recommendations, *e.g., "If I don't understand what the suggestion is about at a first glance, I ignore it and I continue programming"*. However, to the best of our knowledge, no code recommender explicitly focuses on these aspects when deciding which recommendation to trigger. While this could be possible exploiting the readability metrics previously defined in the literature [BW10, PHD11, Dor12, SLOP18, MKX+18], it is still unclear to what extent such metrics work on artificial code.

Other participants pointed out the importance of the recommended code to meet *best practices* and *coding standards*. This includes the possibility to customize the notion of coding standard (*e.g., "meets coding standards of the company in terms of code quality"*, *"pushing good coding standards, either general ones or customized by the user"*). This is another aspect currently unsupported in the state of the art.

Several other aspects of code quality have been mentioned by participants (*e.g.,* ensure good *performance*, *robustness*, and *reusability* of the recommended code). For example, in the case of performance one practitioner wrote: *"Now that more complex recommendations are possible thanks to tools like copilot, aspects related to code quality should be taken into account more, for example by picking among two possible recommendations the one ensuring better performance"*.

A crucial quality aspect mentioned by 69 participants is, as expected, the *correctness* of the recommended code. This means that the recommendation must not break the syntax and/or introduce bugs/vulnerabilities in the code under development (*e.g., "the IDE must never recommend a solution that leads to a malfunction"*). While such a finding might look

obvious, it triggers a few considerations about state-of-the-art techniques. For example, DL models have been proposed to support code completion [CCP+21a, LLZJ20, WSLJ20].

However, recent work by Ciniselli *et al.* [CCP+21a] showed that, in the best case scenario, these models can recommend a correct code completion in less than 70% of cases, with much worse results (<30%) achieved for challenging completions (*e.g.,* recommend entire statements). While the study by Ciniselli *et al.* assessed the accuracy of the recommendations (*i.e.,* the extent to which the tool completes the code as expected), it is likely that a tool not providing an accurate recommendation breaks the code syntax or even introduces bugs. 💡 Our survey suggests that developers are unlikely to use tools that could potentially "break" their code in ∼35% of cases. For the reasons discussed, the notion of correctness is strongly related to the **accuracy of recommendations**, that is another root category in our taxonomy (see Fig. 6.2).

The majority of surveyed developers (61%) reported the accuracy of recommendations as an important characteristic of code recommenders (*e.g.,* "*inaccurate recommendations would create more work than manually writing the code*", "*the recommender system should have a minimum amount of false positive hits*", "*accuracy below 80% is not acceptable*"). 💡 Some developers even proposed solutions to overcome issues related to the low accuracy in specific scenarios (these led to the two child categories of *Accuracy of recommendations* in Fig. 6.2). For example, a participant mentioned "*it must be possible to configure the suggestions, to get less but more likely to be correct*".

Basically, the possibility to filter out recommendations in which the recommender system has a low confidence could help in addressing limited accuracy in challenging scenarios. Some developers also indicated their willingness to consider recommendations that are not 100% accurate as long as they can be easily modified to obtain the needed code: "*I often accept the recommendation even if it's not 100% accurate if I think that adjusting it takes less time than writing code from scratch*".

Going back to the *characteristics of recommended code* tree, developers are also looking for *adaptive* recommender systems: Developers mentioned that the tool should adapt the recommendations to their coding style (*e.g.,* "*once it learns my coding style it should go with that*") and to the coding context (*e.g.,* "*context-sensitivity is also important; I would like practical and stylistic compatibility with the existing code*").

💡 Automatically inferring the developer's coding style is far from trivial. One possibility would be to investigate the integration between approaches to learn coding style/conventions [Rei07, ABBS14] and code recommender systems. More in general, our survey seems to suggest the need for "modeling software developers and their coding practices" with all difficulties and privacy concerns this may lead to. Having such information may substantially boost the usefulness of code recommenders, that could better adapt their recommendation to the user receiving them.

Finally, the *structural characteristics* subtree groups characteristics of the recommended code that concern its structure. These have been considered relevant by a lower number of developers (16). An interesting point to highlight here is the need for code recommendations that are *easy to integrate* into the code under development. This aspect is related to the adaptability of the recommendations: if the recommended code adapts to the coding context,

for example, reusing identifiers when needed, it can make the developer's life easier. This is in contrast with what has been done in retrieval-based approaches that recommend relevant pieces of code from a code base (*e.g.,* Stack Overflow) leaving the integration effort on the developer's shoulders. 💡 An approach able to automate, at least in part, the integration process could substantially increase the usefulness of code recommenders.

Functionality. This category groups features developers would like to see in code recommenders. Most of the answers point to specific wishes in terms of *coverage of support*. This means that developers would like to have a wider support in terms of code recommendations, something that goes beyond the tools they currently use. Such a subtree supports several of the directions that the software engineering research community is currently investigating.

Approaches for *multi-token completion* (*i.e.,* code completion that goes beyond recommending the next token the developer is likely to write, for example, an entire statement) are a hot research topic [KZTC21, KS19, LLZJ20, CCP+21a, copb] as well as the automated generation of *assert statements* [WTM+20, TDSS22] (see Fig. 6.2). 💡 Our results show the need for techniques able to support developers in complex code completion scenarios (*e.g.,* "*code completion provides good support in finding the right API in a class and a few other things, but rarely suggests something challenging such as conditional statements*", "*cool would be suggesting which asserts are needed to test the code under development*"). While no developer asked for the automated implementation of whole functions from scratch [WAN+21], some mentioned the possibility to automatically implement *stub functions* starting from their signature as recently done in Copilot [copb]: 💡 "*When writing new functions I often create stubs of the other functions I need to invoke; the IDE could propose implementations for those stubs*".

💡 An interesting research challenge also comes from the suggestion for *multi-lingual support*: "*autocomplete when mixing languages such as inline SQL code when writing database statements in C#*". Such a support is currently missing in code recommenders. Finally, for what concerns the *coverage of support*, it is also interesting the possibility to integrate the capabilities to autocomplete *refactoring* operations started by the developer. This is something that has been accomplished by Foster *et al.* [FGL12] with their WitchDoctor tool.

Another representative category in the functionality tree is "*provide additional info*", which relates the will of developers to receive additional information accompanying the code recommendation.

This includes the possibility to *provide a rationale* justifying non-trivial code recommendations (*e.g.,* "*it should give a quick reason that I can understand why it's suggesting I do it*"), or documenting the recommended code (*e.g.,* "*assuming the recommended code is not trivial, a documentation/explanation for the suggestion is needed*").

While code recommenders mostly focus on generating meaningful code recommendations, retrieval-based techniques (*i.e.,* those retrieving relevant code from a code base) and properly trained DL-based techniques can provide support for the automated documentation of the recommended code [HHC+20, copb]. 💡 More challenging is the generation of a rationale explaining to the developer why a given recommendation is relevant for what they are doing.

Finally, the other child categories under *functionality* have been mentioned by a few developers. They concern: (i) the ability of the tool to improve over time based on accept-

ed/rejected recommendations (*self-improving*); (ii) the possibility to configure aspects of the tool, such as defining shortcuts (*configurable*); (iii) the way in which the recommendations are presented or generated (*recommendation modality*), and (iv) the need for having the recommendation quickly triggered to avoid manually writing most of the code thus reducing the usefulness of the recommendation (*early prediction*). These categories are described in Fig. 6.3.

Usability. This root category concerns aspects that influence the usability of the code recommender. Among all, *high responsiveness* was the most important requirement highlighted by developers (39 mentions) (*e.g., "these recommendation systems are often slow and lag the entire UI, especially on large projects"*). 💡 This confirms the relevance of works investigating efficiency aspects in code completion tools [SLH+21].

Along this line, participants expect tools which are *intuitive* (*e.g.,* allowing a gradual learning curve for all developers) and *well integrated with IDE*, as one developer underlined: *"clean interface making it easy to use is pretty important"*.

Finally, an interesting aspect highlighted by some developers is the need for *non invasive* code recommenders, always leaving the final word to the developer: "*It shouldn't change the code without my confirmation, even though the code I'm writing is not logically right; a warning would be nice*".

Awareness. The last root category in our taxonomy is *awareness*: the code recommender must be "aware" of different aspects to improve its recommendations. Most importantly from the participants' point of view (29 mentions) is the awareness of developer's knowledge. This implies that the same recommender system triggered on the same code by two different developers could produce different recommendations. 💡 Code recommendations using APIs familiar to the developer (*e.g.,* that the developer used in the past) can be favored as well as recommendations using code constructs that are familiar to the developer. As previously observed, this requires the ability of the recommender to "model" the developer's knowledge, exploiting it in generating the recommendations with the goal of minimizing the comprehension effort for the developer.

A second important aspect is the awareness of *coding context*, *i.e.,* the code recommender should consider the current code context (*e.g.,* the code the developer is writing in the IDE) when providing suggestions. It is important to explain the difference between *awareness of coding context* and *adapted to coding context* previously described: in the former case, the recommended code is triggered by what has been written in the IDE (*e.g.,* it completes the implementation of a method under development), while in the latter the suggested code is adapted (*i.e.,* changed) to match up with the current code context (*e.g.,* to reuse identifiers present in the code). These two aspects should be combined together to obtain useful recommendations.

Other interesting requirements developers expressed are the awareness of *coding history* and of the *developer's tasks*. 💡 Exploiting the change history of the system on which recommendations are generated [RL08, RL10] can "*help with repetitive tasks*". Instead, being aware of the tasks assigned to the developer to which recommendations are proposed can be exploited for a better customization of the recommendations. For example, the recommender can identify the issues assigned to the developer by mining the issue tracker, inferring the

one they are working on in the IDE and targeting recommendations aimed at completing the issue being addressed.

6.4 Validity Discussion

Threats to **construct validity** concern the relationship between theory and observation. The characteristics output of our study are personal opinions of the surveyed developers. To provide information about the "support" each characteristic had, we included the number of participants who mentioned the characteristics (at least for top-level categories, complete data available in [repd]).

Threats to **internal validity** concern factors internal to our study that could have influenced the results. The participants to our survey are likely more interested in code recommenders than others, thus providing a "biased view" of the investigated phenomenon. However, they still provided quite different views on what it is important in code recommenders.

When presenting the results of our study, we often report "*quotes*" from the answers provided by participants. These quotes are not always verbatim, with changes introduced to fix typos or to shorten them without, however, changing their meaning.

To limit subjectivity bias during the open coding procedure, five authors independently inspected each answer we received, with a following discussion aimed at solving conflicts when needed. On top of that, the answers we collected together with the codes (*i.e.,* categories of our taxonomy) we assigned them are publicly available in our replication package [repd].

Finally, contrary to our expectation, we found difficulties in recruiting software practitioners through MTurk. Indeed, when we ran our survey, we did not set the strict selection criteria for participants described in Section 6.2.1. This resulted in the collection of mostly low-quality answers, often clearly copied/pasted from online sources that we had to exclude.

Also, we had to forbid the access to our survey from specific countries due to bots providing random answers. In the end, we preferred to collect fewer answers but of high quality. Indeed, whenever we had a doubt about the quality of answers collected through our survey, we discarded the corresponding response.

Such a "quality issue" is less relevant when it comes to answers we collected through our contact network, which represent the majority of completed surveys in our study (49 out of 80). Indeed, those are all professional developers (*i.e.,* working in companies) who voluntarily agreed to participate in our survey. To check the extent to which our taxonomy generalizes to these two different groups of participants we involved (*i.e.,* practitioners from MTurk *vs* practitioners in our contact network) we checked the number of categories in our final taxonomy that have been indicated as relevant (i) by both groups, (ii) by MTurk participants only, and (iii) by practitioners in our contact network only. Out of the 70 categories in our taxonomy 51 (73%) have been indicated by participants belonging to both groups, 3 (4%) by MTurk's participants only, and 16 (23%) by practitioners in our contact network only. These results indicate an overall "agreement" among the two groups for two-thirds of our taxonomy. Also, the fact that only three out of 70 categories have been contributed

exclusively by MTurk's participants support the validity of the answers collected through this platform, since most of the "requirements" we extracted from them have been confirmed by practitioners in our contact network. The three categories being MTurk-only are: *Functionality → Coverage of support → Language keywords*, *Functionality → Recommendation modality → Open source*, *Functionality → Recommendation modality → Proactive: perceives when developer needs help*.

Threats to **external validity** concern the generalization of our findings. The obvious threat is the limited number of participants involved in the survey (80). Such numbers are in line with previously published survey studies (*e.g.,* [FL02, dSAdO05, BDO+13]) but, of course, replications can help in corroborating our findings and complementing them.

6.5 Conclusion and Future Work

Our study partially fills the lack of empirical investigations aimed at collecting practitioners' *desiderata* when it comes to code recommenders. We ran a survey involving a total of 80 practitioners to investigate characteristics of code recommenders they perceive as important. As output of our study, we defined a taxonomy of 70 characteristics (Fig. 6.2) that can drive future research in the field. We make all (anonymized) answers we collected, the tags we assigned to them, and study material available in our replication package hosted at `https://code-recommenders.github.io`.

Our future works stem from the findings of our survey, and will focus on improving characteristics of code recommenders that our study highlighted as relevant. We detail three research directions we plan to pursue:

- *Improving the awareness of code recommenders*. One aspect several of the participants in our study stressed as important is what we defined as the "awareness of the code recommenders". In other words, what the information available to the recommender is when it synthesizes suggestions. Being aware of the developers' knowledge (*i.e.,* their expertise, past implementation tasks, etc.) could result in more relevant recommendations that might be particularly suitable and easy to understand and reuse for the developer receiving them. Integrating such a "knowledge" in the recommender is far from trivial and requires the development of techniques allowing to automatically infer (i) the programming style of software developers, and (ii) their expertise, namely the specific programming languages, libraries, notions (*e.g.,* design patterns), etc. they are at ease with. This could be done by mining the past developers' activities from software repositories (*e.g.,* versioning system, issue tracker).

- *Making code quality a first-class citizen in code recommendations*. Given the increasing complexity of the recommendations supported by tools such as GitHub Copilot, code quality must become a priority for the recommended code. This is a clear outcome of our survey. Important aspects to focus on are the absence of bugs/vulnerabilities and the promotion of readability and understandability. All these quality aspects can benefit from tailored design decisions made (i) when training the code recommender,

by curating the quality of the code snippets composing the training set; and (ii) at post-processing stage, with checks done before triggering the recommendation to the user. Also, alternative recommendations could be ranked based on their quality.

- *Augmenting the coverage of support*. Despite the gigantic steps ahead made in the last few years, code recommenders still struggle in specific coding scenarios for which they do not offer (or offer limited) support. In this context, two interesting research directions stemming from our taxonomy are (i) better support for multi-lingual code, and (ii) generation of templates rather than raw source code when the recommender is not confident. Concerning the first point, we plan to assess the effectiveness of recent transformer models trained on several programming languages (*e.g.,* CodeBERT [FGT+20]) when dealing with multi-lingual code. Depending on the observed performance, the proposal of alternative strategies to deal with this problem will be investigated. As for the template generation, we plan to work on a model specifically trained for generating abstract code templates rather than raw source code. Such a model could be triggered when the standard code recommender is not confident in suggesting the raw code.

Characteristics of recommended code

This category groups together specific code characteristics (detailed in the leaf nodes) that might be desirable for the recommended code.

- **Structural characteristics:** The recommended code meets specific structural characteristics of the code (detailed in the leaf nodes) that might be desirable.
  - **Easy to integrate:** The recommended code does not require major changes to be integrated in the code under development (e.g., a code using object types not available in the language but defined in other projects from which the recommended code has been learned is difficult to reuse).
  - **Type checked:** The recommender checks types when completing statements, ensuring values assigned to a variable are compatible with its type.
  - **Unambiguous:** The recommended code is unambiguous (e.g., it doesn't suggest multiple imports with the same name, making difficult to identify the one actually needed).
  - **Precise typing information:** The recommended code relies on specific types (e.g., java.time.LocalDate) rather than the generic one (e.g., Object).
- **Adaptive:** The recommended code is customized based on specific factors (detailed in the leaf nodes).
  - **Coding context:** The recommended code is adapted to the code the developer is writing (see leaf nodes).
  - **Coding style:** The recommended code is adapted to the developer's coding style (e.g., using extra parenthesis to better format code if this practice is used by the developer).
- **Code Quality:** The recommended code meets specific quality criteria (see leaf nodes).
  - **Best Practices:** The recommended code meets best coding practices. The latter depend on the language/paradigm in use.
  - **Performance:** The recommended code is optimized in terms of performance (e.g., it does not introduce unneeded operations).
  - **Correctness:** The recommended code must be bug-free.
  - **Robustness:** The recommended code is robust when dealing with possible erroneous situations (e.g., a null check is implemented if needed).
  - **High Reusability:** The recommended code is easy to reuse. This is particularly relevant for recommender systems suggesting code components at higher granularity (e.g., methods).
  - **High Readability & Understandability:** The recommended code must be readable (e.g., avoid very long statements, adopt indentation) and easy to understand.
  - **Maintainability:** The recommended code is easy to maintain.

Functionality

This category groups together the features developers would like to see in source code recommenders (see leaf nodes).

- **Early Prediction:** The recommender is able to trigger recommendations early in the coding process (e.g., the developer just wrote a few code tokens).
- **Coverage of Support:** This subcategory details the desiderata of developers when it comes to the coverage of the recommender tool (i.e., the different coding scenarios it can support).
  - **Identifiers support:** The recommender provides support for identifiers, suggesting them when needed.
  - **3rd party library support:** The recommender is able to propose code recommendations when dealing with code using third-party libraries (e.g., invocation of an API in a library).
  - **Language keywords:** The recommender is able to suggest the language keywords when needed.
  - **Multi-token completion:** The recommender can generate code recommendations that are not limited to a single token, but span several tokens.
  - **Multi-lingual support:** The recommender can generate code recommendations even when multi-lingual code statements (e.g., C# and SQL) are written by the developer.
  - **Refactoring:** The recommender is able to autocomplete a refactoring operation started by the developer.
- **Self-improving:** The recommender improves the suggestions based on the feedback received by the developer for past recommendations.
- **Provide additional info:** The recommender provides additional information together with the suggestion
  - **Provide rationale:** The tool provides a rationale to justify the given recommendation (e.g., you are seeing this code because ...).
  - **Highlight edits:** If the code that the developer is writing is very similar to code that can be recommended, the tool highlights the differences that might be implementation errors.
  - **Documentation:** The recommended code includes documentation.
- **Configurable:** The recommender is customizable based on developers' preferences (e.g., see leaf node).
- **Recommendation modality:** The modality used by the recommender when suggesting a recommendation.
  - **Ask information:** If the recommender cannot infer what the developer is doing it is able to ask for additional information to better contextualize the recommendations.
  - **Multiple options:** The recommender is able to generate multiple recommendations for a given scenario.
  - **Smart ranking:** When multiple options are available, the tool ranks the recommendations using a smart criterion rather than alphabetical order.
  - **Proactive:** The recommender perceives when the developer needs help (e.g., they are not writing for long time) and starts triggering recommendations. It does not need to be explicitly invoked by the developer.
  - **Open source:** If the recommended code is retrieved from a dataset, such a dataset must feature open source code which is less likely to result in licensing issues.

Awareness

This category groups together information items the recommender should be aware of to improve its recommendations.

- **Coding context:** The recommended code is based on a specific coding context, usually representing the code written by the developer in the IDE.
- **Coding history:** The recommended code is generated/improved based on development activities performed in the past. The past development activities could be related to the developer receiving the recommendation or to other developers. In other words, "natural" coding solutions are favored.
- **Developer's task:** The code recommender is aware of the task(s) the developer is working on. Differently from the coding context that only captures the code under development in a specific moment, in this case a more complete view of the developer's tasks is available (e.g., the tool could mine from the issue tracker the issues assigned to the developer and use this information to generate/improve its recommendations).
- **Developer's knowledge:** If multiple recommendations are possible, the one using code that is more familiar to the developer must be used (e.g., the code using APIs already used in the past by the developer receiving the recommendation).

Usability

This category groups together usability aspects they would like to see in source code recommenders (see leaf nodes).

- **Intuitive:** The recommender is intuitive (e.g., shallow learning curve).
- **No overhead:** The recommender does not hinder coding (e.g., a button can be pressed to accept/decline the recommendations) and does not require context-switch.
- **Non invasive:** The recommender is never in control of the code writing, it only provides suggestions.
  - **Ask confirmation:** The recommender asks confirmation to the developer before implementing code.
- **Integrated with IDE:** The recommender is integrated within the IDE.
- **High responsiveness:** The recommender is responsive, not causing lagging while coding.

Accuracy of recommendations

The recommended code is accurate (i.e., the recommender is able to suggest a code semantically equivalent to the one the developer was going to write).

- **Filtering based on confidence:** The recommender does not trigger recommendations when its confidence is low.
  - **Templating in low level confidence:** If the recommender has a low confidence about a recommendation, it can suggest a template for the recommendation rather than the raw code.

Figure 6.3. Definitions for taxonomy's characteristics up to level-3

7

Deep Learning-based Code Completion: On the Impact on Performance of Additional Contextual Information

7.1 Motivation

One of the most noticeable results of the adoption of Deep Learning (DL) in software engineering (SE) is the recent release of DL-based programming assistants such as GitHub Copilot [copb]. These tools *redefined* the notion of code completion, moving it from techniques able to recommend the next few tokens the developer is likely to type to tools capable of automatically generating entire functions.

While Copilot likely is the most well-known representative of this generation of code completion tools, it is the natural follow-up of years of research done in this field [PGBG12, GKKP13, KATF18, WSLJ20, CCP+21a]. Most of these works proposed novel solutions with the main goal of improving the state-of-the-art performance of code completion tools. When talking about "performance", we refer to the accuracy of the technique in recommending the expected code (*e.g.,* the tokens needed to complete a code statement or an entire function starting from its signature).

Our study on the characteristic of the code recommenders desired by developers highlighted the importance of the *awareness* of external source of information, like coding history or developer's knowledge, that allow the model to generate familiar code that leverage, for example, the APIs frequently used by each developer. Another important factor that the models should be aware of, that could potentially bolster their performance, is the contextual information provided as input to the model for triggering the recommendation: This is the information available to the model to decide which code completion recommendation to generate. For example, in the recent work by Ciniselli *et al.* [CCP+21a] the DL model is provided as input a Java method with one or more missing statements to complete. In this case, the incomplete Java method is the only information the model can rely upon to predict the missing statements. Such a design choice ensures shorter inputs for the model and, as a consequence, shorter training time. However, this choice may limit the prediction capabil-

ity of the model which could benefit, for example, from knowing what the other methods implemented in the same class are. While previous work suggests the positive impact that additional contextual information may have when using DL-based solutions for code-related tasks [TT22], to the best of our knowledge there is no study deeply investigating the role of contextual information on the prediction accuracy of DL-based code completion techniques. We aim at filling this gap.

We start by defining three families of contextual information which can be provided to a DL model to improve its prediction capabilities. To provide a high-level explanation of each of them, let us focus on the same method-level code completion task defined by Ciniselli *et al.* and previously summarized (*i.e.,* provide as input an incomplete Java method and ask the model to generate the missing part). We use IM_i to refer to a generic incomplete method to finalize and assume that the developer D_j is the one working on it (*i.e.,* the person who will receive the completion recommendation).

The first family of contextual information we experiment with is the *coding context*: These are contexts augmenting the input provided to the model with code components having structural relationships with IM_i (*e.g.,* the methods invoking/invoked by IM_i). The assumption here is that knowing more about the code base can help the model in generating the correct prediction.

The second family is the *process context*, providing the model with information related to the development process carried out in the project IM_i belongs to (*e.g.,* what the open issues possibly related to IM_i are when a code completion is triggered on IM_i). The idea behind the process context is that knowing what the ongoing tasks are could help the model in predicting the missing code.

Finally, the third family is the *developer context*, augmenting the input with information characterizing the recent development activity of D_j (*i.e.,* the developer currently working on IM_i). One of the developer contexts we experiment with augments the model's input with method invocations recently and frequently used by D_j. The assumption is that the model might exploit information about the recent development activities of D_j to improve its predictions.

Overall, we defined and implemented 8 types of contexts belonging to the three above-described families and experimented with them (and their combinations) by training and testing 18 Text-To-Text-Transfer-Transformer (T5) [RSR$^+$20] models. Our results show that: (i) additional contextual information helps in boosting prediction performance, with the ones belonging to the *coding context* usually bringing the larger boost; (ii) by combining different types of contexts it is possible to achieve a substantial relative improvement over the baseline (*i.e.,* the model not exploiting additional contextual information) of up to +22%.

7.2 Types of Contextual Information

This section introduces the types of contextual information we experiment with. They are all depicted in Fig. 7.1 which does also include what we refer to as "baseline" (see red part in Fig. 7.1). The baseline represents the common code completion scenario experimented in the literature, in which the DL model is only fed with the piece of code to complete. We

adopt the method-level completion recently experimented by Ciniselli *et al.* [CCP+21a] in which one or more statements are masked in a Java method (see the <MISSING CODE> tag Fig. 7.1) with the model in charge of predicting them. The baseline will be used to assess the boost in performance (if any) provided by the additional contextual information provided to the model. The examples in Fig. 7.1 are extracted from a real instance present in the training dataset that will be described in Section 7.3.

In the following we use IM_i to refer to the incomplete method provided to the model (*i.e.,* `handleDataProcessException` in Fig. 7.1). All contexts we describe represent additional information that is provided to the model on top of the "baseline" representation.

The goal of this section is not to provide all technical details about *how* we create these contexts, but rather to present and justify them. Technicalities about how we built the different datasets needed to experiment with these contexts are presented in Section 7.3.

We experiment with three families of contexts: *coding* (green in Fig. 7.1), *process* (blue), and *developer* (yellow).

7.2.1 Coding Context

The basic idea here is to augment what the model knows about the code to complete with additional information extracted from the code base. We devise three types of coding contexts.

The first, **method calls**, provides the model with the complete signature of the methods invoked by or invoking the method to complete (IM_i). Note that, being IM_i an "artificial" incomplete method we created by replacing some of its statements with the <MISSING CODE> tag, methods that were invoked in the replaced statements are not included in the context. Indeed, in a real usage scenario, those statements would not exist (*i.e.,* the developer is writing them). In Fig. 7.1 it can be seen that we use the special tag <OUT> to mark methods invoked by IM_i and <IN> for methods invoking IM_i.

The rationale behind this context is that the model might benefit from knowing the code components structurally coupled (via method calls) to IM_i. It is important to discuss at this point why we only include the signatures of the coupled methods rather than their full implementation which, in theory, could provide even more information to the model. Such a choice is due to technical limitations of the DL models, which are able to deal with input sequences of limited length. For example, recent work in the SE literature capped the input instances to 512 tokens [GRL+21, YX19b, YX19a, CCP+21b, ABLY19, CCP+21a, AKL21]. In our work, we pushed this boundary to 1,024 tokens which still limits the contextual information size. Thus, in all contexts, we present such a tradeoff between the amount of additional information and the size of the input sequence that has been considered.

The second coding context we define is the **class signatures**, providing the model with the signature of all other methods contained in the class implementing IM_i (bottom-left corner of Fig. 7.1). The rationale behind this context is that, accordingly to the "high cohesion" principle [SMC74], classes are supposed to group together related methods (*e.g.,* methods implementing related responsibilities).

Finally, **most similar method** is the third coding context we defined. When experiment-

ing with DL models for code completion there must be no duplicates between the training and the test datasets. Otherwise, the model would be asked to complete methods it has already seen during training, thus artificially inflating its performance. However, it is reasonable to think that, given an incomplete method IM_i from the test set, a "similar" method may exist in the training set. A concrete example from our dataset is depicted in the bottom-right corner of Fig. 7.1: The method `connectionClose` from our training set is the most similar to `handleDataProcessException` (our IM_i). Indeed, it can be seen that they share some logic. Our assumption is that such an additional contextual information can help the model in predicting the missing statements. Note that we retrieve the most similar method *only from the training set*. This is important since, in a real scenario, the instances in the training set are the only ones known to the model. We detail in Section 7.3 how the most similar method is identified.

7.2.2 Process Context

The process contexts provide the model with information capturing "what is going on" in the project when a recommendation must be triggered to complete IM_i. We define two types of process contexts, both exploiting information from the issue tracker of the project IM_i belongs to. The assumption is that if a developer is implementing code aimed at addressing an open issue (*e.g.,* a bug to fix, a new feature to implement), information extracted from such an issue may help the model in recommending the needed code. An issue is usually composed by a *title*, which summarizes its content, and a *body*, which provides a more detailed description of the issue.

These two elements are the ones driving the definition of our two contexts, named **issue title** and **issue body** (see Fig. 7.1). For both of them we start by identifying for the given IM_i the most similar open issue at the time t the code completion on IM_i is invoked (*i.e.,* the issue most likely describing needed implementation tasks which may affect IM_i — details in Section 7.3). We use such an open issue to create the two contexts, one featuring the issue title and the other one the issue body. While a context featuring their combination would make sense, this is not presented since in our experiment we test with several combinations of contexts both from the same family (*i.e.,* the two process contexts together) and from different families (*i.e.,* a process context mixed up with a coding context).

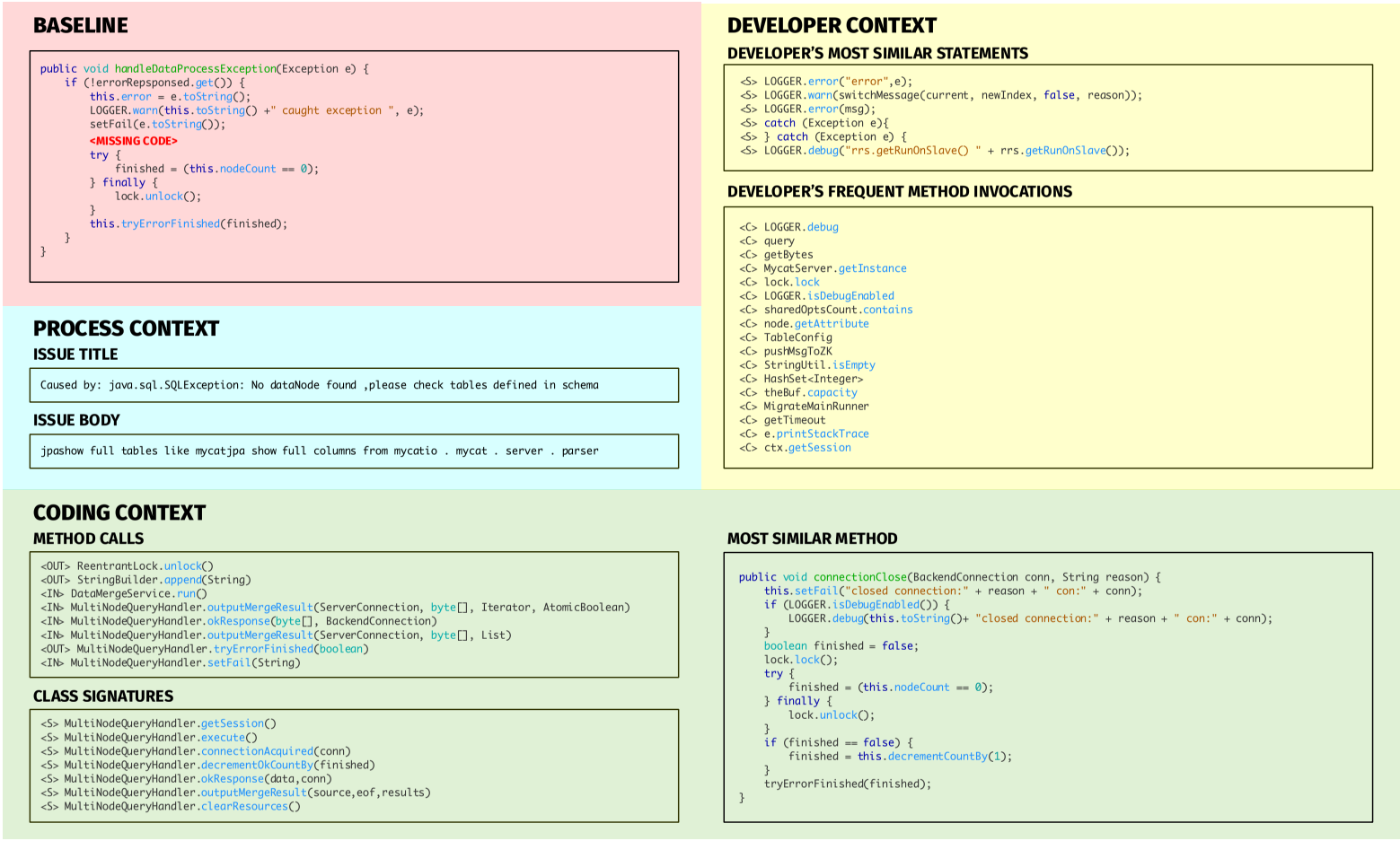

Figure 7.1. Experimented contexts: Examples from a real instance in our dataset.

7.2.3 Developer Context

The last family of contexts provides the DL model with information about the developer D_j currently working on IM_i (*i.e.,* the one who will receive the completion recommendation). The idea is that different developers may have different coding styles [CGP07, HDB21] (*e.g.,* may favor the usage of specific APIs they are familiar with).

The first context in this family is the **developer's most similar statements** (top-right corner in Fig. 7.1). We mine the recent commits performed by D_j and extract from each of them the source code statements they added, deleted, or edited. Each statement in this set is then compared with the method to complete IM_i to identify the statements being more similar to the code under development. The most similar statements are added as contextual information to the model input, separated by an <S> tag as shown in Fig. 7.1.

The second *developer context* is the **developer's frequent method invocations**, providing the model with information about method calls (both internal and external to the project) recently and frequently used by D_j, as depicted in Fig. 7.1. Technical details about the building of the two developer contexts are presented in Section 7.3.

7.3 Building the "context datasets"

Experimenting with the contexts described in Section 7.2 requires the building of several datasets aimed at training/testing the DL model to assess the impact of the different contextual information on its performance. The first step in this process is the selection of software repositories from which the information needed for the different contexts can be extracted. We used the tool by Dabic *et al.* [DAB21] to select all GitHub non-forked Java repositories having more than 100 commits, 10 contributors, 50 issues, and 10 stars. These filters ensure that the selected projects (i) have a substantial history, needed to extract information related to the *developer context*; (ii) are not personal/toy projects, since at least 10 contributors took part in their development; and (iii) use the issue tracker, a requirement for the extraction of the *process context*. The 10-star filter is mandatory in the search tool by Dabic *et al.* [DAB21], since projects with less than 10 stars are not indexed. The result of this query was a list of 5,632 candidate repositories. The extraction of the coding context (*e.g.,* the identification of the methods invoked by or invoking the method of interest) requires the source code of the selected projects to be compilable. For this reason, we excluded projects that do not explicitly tagging releases in GitHub and, for the remaining ones (*i.e.,* those tagging releases), we checked out their last two releases trying to compile them using Maven or Gradle.

The choice of focusing on the last two releases rather than only on the last one aimed at increasing the number of projects suitable for our study. Through this process, we successfully compiled at least one release for 1,072 repositories. For each of these repositories, we indicate with R_c the successfully compiled release.

We use these repositories to build eight training/testing datasets, one for the baseline approach emulating the work by Ciniselli *et al.* [CCP+21a] and one for each of the seven contexts introduced in Section 7.2. It is important to remember that all datasets will feature instances in the form $IM_i \rightarrow C_r$, where IM_i represents an incomplete method (*i.e.,* a method

from which specific statements have been masked) and C_r represents the expected completion (*i.e.,* the code the model should generate). C_r is identical across all eight datasets, while IM_i changes based on the contextual representation it features.

We start by checking out all 1,072 compilable releases from which we randomly selected up to 1,000 Java files per project. The cap at 1,000 files per repository has been defined to avoid very large repositories to influence too much the final dataset (*e.g.,* contributing 50% of the final instances). Also, when selecting those files, we ignored those containing the word "test" in their name or belonging to a package having "test" in their path. This was done in an attempt to exclude test files, thus building a more cohesive dataset only featuring production code instances. From each of the selected files, we extracted the implemented methods using JAVALANG [javc]. We then removed all methods exceeding 682 tokens. While this may look like a *magic number*, it represents two-thirds of the space available for the model's input. Indeed, as detailed in Section 7.4, our DL model accepts inputs up to 1,024 tokens in length. We decided to dedicate up to 682 tokens to the representation of IM_i and at least 342 tokens for representing the additional contextual information. The contextual information is appended to IM_i (*i.e.,* it comes after). Thus, in case the contextual information makes the input longer than 1,024 tokens, part of it will be cut and ignored by the model. For example, if IM_i requires 550 tokens and a specific type of contextual information requires 750 tokens, the last 276 tokens of the context will not be seen by the model.

The above-described process resulted in 42,182 collected methods. Ciniselli *et al.* [CCP+21a] masked randomly selected statements in methods to create the instances $IM_i \rightarrow C_r$. We decided to adopt a different approach for the masking, with the goal of better simulating a developer writing code and receiving completion recommendations. In particular, rather than masking randomly selected statements, we run `git blame` on each method in our dataset, retrieving the latest commit before R_c (*i.e.,* the compiled release) which changed at least one code statement in the method. Let us assume that the identified commit changes a single statement s_k: We create an instance $IM_i \rightarrow C_r$ in which IM_i has s_k masked and $C_r = s_k$. Such an instance simulates a realistic change which has been actually performed in the history of the project. It could also happen that the identified commit changes several statements in the method. To limit the complexity of the code completion problem, we decided to mask at most two complete statements when creating an IM_i. Thus, if five statements have been modified (s_1, s_2, s_3, s_4, and s_5), we create three $IM_i \rightarrow C_r$ instances from the corresponding method: The first has s_1 and s_2 masked, the second has s_3 and s_4 masked, the third has only s_5 masked.

The above-described process generates the "baseline" dataset, in which the IM_i in the instances is only composed by the incomplete method, with no additional contextual information. In the following we describe how we built the remaining seven datasets. We ensure no duplicates in our datasets, removing instances having identical IM_i. It is important to clarify since now that, in order to fairly compare the performance of the DL model when exploiting different contextual information, all eight datasets must feature exactly the same $IM_i \rightarrow C_r$ instances, with the only difference being the representation of IM_i. Since specific contextual information (*e.g.,* open issues) cannot be extracted for all instances, once built all datasets we computed their intersection, featuring 85,266 $IM_i \rightarrow C_r$ instances which are

present in all datasets.

7.3.1 Coding Context

To extract the **method calls** context, we run JAVA-CALLGRAPH [java] on the corresponding compiled release, thus identifying the IM_i's call graph. As previously explained, methods invoked in the masked part of IM_i have been ignored, since in a real scenario those are the statements that the developer is writing (*i.e.,* for which they expect a recommendation).

Concerning the **class signatures** context, we relied on the methods previously extracted using JAVALANG [javc], appending to IM_i those being implemented in its same class as shown in Fig. 7.1.

Finally, for the **most similar method** context, we defined a process to identify, among all methods in the training set, the most similar to IM_i (not considering in IM_i the masked statements). Given the size of the datasets usually adopted to train DL models, we need a scalable and accurate procedure to compute the similarity between a given input method IM_i and all methods in the training set. We start by computing the token-level Jaccard similarity [Han04] between IM_i and each instance in the training set. Such a metric is very efficient to compute and basically indicates the overlap in code tokens between two methods. Then, we select the top-k methods in the training set which, accordingly to the Jaccard similarity, are the most similar to IM_i. Finally, for each of these k methods we compute their CrystalBLEU [EP22] similarity with IM_i, re-ranking them based on this metric. The recently proposed CrystalBLEU has been shown to be the metric better correlating with human assessment when judging the similarity between code snippets. The drawback of this metric is its scalability, which makes it unsuitable to compute the similarity between all instances in the training set and IM_i. In summary, we use the Jaccard similarity as a preliminary filter to identify candidate similar methods. Then, we refine such a set using a more reliable metric with the goal of selecting the *most* similar method, being the one augmenting the contextual information. In our implementation, we set k=20 to achieve a good compromise between scalability and accuracy. While different values may lead to better performance, the goal of our study is not to find the best possible (combination of) contextual information for code completion, but rather to show that this information can play a substantial role in the model's performance.

7.3.2 Process Context

The creation of the two process contexts described in Section 7.2.2 requires the identification of the IM_i "most similar issue".

We trained a Transformers and Sequential Denoising Auto-Encoder (TSDAE) [WRG21] model for such a task. TSDAE is a denoising auto-encoder based on BERT that can be used to create embeddings (*i.e.,* high dimensional real-valued representations of text). By providing a textual instance to TSDAE, it returns an embedding representing that specific text. We leverage these embeddings to measure the similarity between IM_i and the set of open issues.

To train TSDAE for such a task we built a dataset to make the model learn when an issue is relevant for a given IM_i. We used the instances in the "baseline" training set as a starting

point. As explained, each $IM_i \rightarrow C_r$ instance has been built by looking for the latest commit (l_c) that changed the method from which IM_i derived. We indicate the date in which l_c has been performed with t. Given an instance, we identify all issues whose status was open at time t. Then, we checked if l_c can be "linked" to one of the open issues. A link between l_c and an open issue is established if l_c's commit message contains an explicit reference to the issue id (*e.g.,* "*fixed issue #134*") or to the issue url (*e.g.,* "*working on issues/134*"). We established such a link for 27,851 instances in our training set. Each of them was used to train TSDAE for the task of identifying the "open issue" relevant for a given IM_i. Indeed: (i) l_c is the commit that lastly modified the method from which IM_i is derived; (ii) l_c has been performed at time t and can be linked to a specific issue OI_n that was open at that time. As a consequence, we can create one training instance for TSDAE indicating that OI_n is relevant for IM_i. Both IM_i and the text composing OI_n are subject to standard pre-processing before they are provided to TSDAE: We exclude Java keywords, remove punctuation, and split camelCase identifiers.

To choose the best configuration for TSDAE, we performed hyperparameters tuning and experimented with the different configurations on a validation set we built starting from the "baseline" validation set using the same procedure described for the training set (3,434 instances). We experimented with six different configurations of TSDAE involving three different schedulers and two different learning rates. Each model has been trained for four epochs, since we did not observe any improvement increasing the training time. The best configuration has been identified as the one having the highest Mean Reciprocal Rank (MRR), indicating the ability of the model to correctly rank in the first positions the issue relevant for IM_i. Once identified the best configuration (complete data in our replication package [repb]), we assessed the performance of the trained TSDAE on a test set derived from the "baseline" test set (3,256 instances). We achieved a MRR of 0.34, which is substantially better as compared to that of a random ranker which, on our test dataset, would obtain a MRR of 0.14.

We create the **issue title** and **issue body** context datasets by exploiting the trained TSDAE to identify the most relevant open issue for each IM_i. Instances for which no open issues were found at time t are excluded from this dataset and, as a consequence, from all other datasets and our experiment for the reasons previously explained (*i.e.,* the need to compare the DL models when trained/tested on exactly the same instances).

7.3.3 Developer Context

The core idea is to provide the model with information characterizing the developer who will receive the completion recommendation.

In our dataset every $IM_i \rightarrow C_r$ instance has been derived from a l_c commit that impacted the method IM_i by changing the C_r statements. Being a commit, l_c has been authored by a developer D_j which, in our study design, is the one who would have received the completion recommendation while working on IM_i. Thus, we start by retrieving up to ten past commits performed by D_j before l_c and impacting at least one Java file. We store the diff of these commits as the set of lines of code they added, deleted, and modified.

Then, we create the **developer's most similar statements** context by identifying, in this set, the ten statements having the highest similarity with the method IM_i (see Fig. 7.1). As

usual, we do not consider the masked statements (*i.e.,* the ones to complete) when computing the similarity. The similarity is based on the percentage of overlapping tokens between each statement and IM_i excluding Java keywords and punctuation.

Concerning the **developer's frequent method invocations**, we use srcML [src] to parse the same set of lines recently added, deleted, or changed by D_j to extract all impacted method calls (both internal or external to the project). Then, we sort them by frequency, keeping up to 100 most frequent calls in the additional context (see Fig. 7.1 for an example).

The choice of keeping only the top-10 most similar statements as compared to the top-100 most frequent calls, is due to the fact that entire statements are usually longer than method calls. Thus, given the space available to represent contextual information, we can fit more method calls as compared to entire statements.

7.4 Study Design

The *goal* of this study is to assess the impact on the performance of a DL-based code completion technique of additional contextual information provided to it as input. The *context* is represented by the datasets described in Section 7.3 that provide eight different types of contexts including the "baseline" one.

We answer the following research question (RQ):

> **RQ.** *To what extent do different types of contextual information impact the performance of DL-based code completion models?*

We assess "performance" by looking at the number of correct predictions generated by the different variations of the DL model (*i.e.,* those trained using the different datasets presented in Section 7.3, each of which represents a different context). In addition to that, we test combinations of the contextual information presented before (*e.g., issue title + issue body*), for a total of 18 different models involved in our study. In the following we detail the DL model we use and how we trained it (Section 7.4.1), and the process we use to collect and analyze the data output of our study (Section 7.4.2).

7.4.1 DL Model and Training Procedure

As previously explained, we build on top of the work by Ciniselli *et al.* [CCP+21a] that we use as "baseline". Thus, we adopt their same DL model, namely the Text-To-Text-Transfer-Transformer (T5) [RSR+20]. T5 has been presented by Raffel *et al.* [RSR+20] in five variants characterized by different architectures and, consequently, by a different number of trainable parameters going from 60 Million for $T5_{small}$ up to 11 Billion for $T5_{11B}$.

A larger number of parameters implies better performance at the cost of longer training times [RSR+20]. While Ciniselli *et al.* [CCP+21a] opted for the smallest $T5_{small}$, we decided to adopt the $T5_{base}$ (220 Million parameters), being it more representative of large language models which may be deployed in practice. For architectural details about $T5_{base}$ we point the reader to the work by Raffel *et al.* [RSR+20].

Before being specialized for a task at hand (in our case, code completion), T5 can be pre-trained using a self-supervised task. The goal of the pre-training is to expose the model to the language of interest, making it learning its structure. A typical pre-training objective is the *masked language model*: Assuming the interest in teaching T5 the structure of the Java language, we can provide the model with Java snippets in which 15% of the tokens composing them have been masked, asking T5 to guess those tokens. Once pre-trained, the model can then be subject to the second training phase, named fine-tuning, in which it is exposed to the specific task of interest (*e.g.,* completing entire code statements).

Concerning the pre-training, we start from an already pre-trained T5 that has been trained for 1M steps on the C4 dataset [RSR+20], featuring 20TB of web-extracted English text. Indeed, previous work showed that starting from a model pre-trained on English is beneficial when dealing with code as compared to the randomly initialized weights of a non pre-trained model [TDS+20]. Starting from this checkpoint, we additionally pre-train the model for 500k steps using the previously described *masked language model* objective on a Java pre-training dataset we built. The dataset features 12,671,475 Java methods that have been extracted from GitHub projects also in this case identified using the search platform by Dabic *et al.* [DAB21]. Also in this case we targeted non-forked projects with a long change history (>500 commits), at least a small development team (>10 contributors), and at least 10 stars. Java methods containing non-ASCII characters, being longer than 512 tokens, or being already present in the fine-tuning datasets have been excluded.

Table 7.1. Fine-tuning Datasets.

Context	Instances Length		
	Mean	Median	St. Dev.
Baseline	243	205	166
Method Calls	380	326	253
Class Signatures	733	481	1,128
Most Similar Method	447	384	296
Issue Title	260	222	167
Issue Body	550	361	1,429
Frequent Invocations	467	418	252
Most Similar Statements	517	445	2,355
Most Similar Method + Class Signatures	941	693	1,169
Method Calls + Class Signatures	871	621	1,153
Most Similar Method + Method Calls	584	507	374
Most Similar Method + Method Calls + Class Signatures	1,079	829	1,200
Issue Title + Issue Body	567	378	1,430
Frequent Invocations + Most Similar Statements	741	647	2,365
Best Code + Best Process	601	525	376
Best Code + Best Developer	808	735	423
Best Developer + Best Process	484	435	254
Best Code+ Best Developer + Best Process	825	753	425

The fine-tuning datasets are the ones described in Section 7.3 plus their combinations as reported in Table 7.1. All datasets are composed by instances in the form $IM_i \rightarrow C_r$, in which IM_i is the method to complete possibly augmented with contextual information, and C_r the expected completion.

Table 7.1 also reports statistics about the length of the instances (in terms of number of tokens) in each dataset. All fine-tuning datasets feature 85,266 instances, split into 80% training (68,215), 10% evaluation (8,526), and 10% test set (8,525). While the datasets have been randomly split, all instances referring to the same method belong to the same set, to avoid biasing our results. Indeed, it is worth remembering that a method may generate multiple instances in our dataset, since we could mask different parts of it.

As shown in Table 7.1, we experiment with: (i) the baseline model; (ii) the seven types of context introduced in Section 7.3; (iii) all combinations of contexts within the same family (*e.g.,* combinations of the three *coding contexts*; and (iv) combinations of contexts across different families (*e.g.,* combining a *coding context* with a *process context*). For these cross-families combinations, we reduce the number of experiments to run by only considering the best combination within each family. This means that we combine the best *coding context* (which could be a single context or a combination of multiple *coding contexts*) with the best *process context*; then, we combine the best *coding context* with the best *developer context*; etc. We assess what the best context is within each family by looking at the number of correct predictions (*i.e.,* cases in which the recommended code is identical to the expected one) generated by the models.

Training procedure. We pre-trained and fine-tuned T5 using a Google Colab's 2x2 TPU topology (8 cores) [col]. We also trained a 32k word-pieces SentencePiece tokenizer [KR18] used by the model to represent the input/output. The tokenizer has been trained on 1M Java methods randomly extracted from the pre-training dataset and 712,634 English sentences from C4 [RSR$^+$20]. The maximum number of tokens for the input has been set to 1,024 and the batch size to 32.

We performed hyperparameters tuning assessing the performance of the four different T5 configurations experimented by Ciniselli *et al.* [CCP$^+$21a]. These configurations differ in the way they handle the learning rate. We fine-tuned each configuration for 30k steps and assessed its performance on the evaluation set in terms of its ability to generate correct predictions. To reduce the cost of such a procedure, we found the best configuration only on the "baseline" dataset (*i.e.,* no additional contextual information provided), and used it in all experiments. The best configuration found was the one using a constant learning rate equal to 0.001.

Such a configuration has been used to fine-tune the 18 different models (*i.e.,* different contexts and combinations of contexts). Each of these models has been fine-tuned for 160k steps, corresponding to ∼75 epochs on the 68,215 instances of the training dataset. To avoid overfitting, we saved a checkpoint for each model every 5k training steps. Then, we evaluated the performance of the different checkpoints on the evaluation set, picking the best one as the final model to use in our experiments on the test set.

7.4.2 Data Collection and Analysis

We run the 18 trained models on the test set assessing their performance in terms of correct predictions. To evaluate whether the difference in correct predictions generated by two models is statistically significant, we use the McNemar's test [McN47], useful to compare

dichotomous results of two different treatments, together with the Odds Ratio (OR) effect size. We account for multiple comparisons by adjusting *p*-values using the Holm's correction [Hol79].

We also assess the complementarity between the baseline and each contextual model by computing the percentage of correct predictions generated by (i) both models (*i.e.,* for a given code completion instance, both models correctly recommend the completion); (ii) the baseline only; (iii) the contextual model only. This analysis allows us to understand the potential of combining multiple models.

7.5 Results

Table 7.2 reports the percentage of correct predictions generated by T5 when trained using the *baseline* representation, the different types of contexts, and their combinations (within- and cross-family).

The baseline achieves 30.58% of correct predictions which is in line with what observed by Ciniselli *et al.* [CCP+21a] when investigating the ability of T5 to generate entire statements (in our case, up to two statements). When providing the model with additional contextual information of a specific type (Context Type = "Single" in Table 7.2), we observe an increase in performance ranging from a relative +0.6% (*issue body*) to a +7% (*most similar method*). Also providing the model with the *method calls* context (*i.e.,* the methods invoking and invoked by the method to complete) provides a substantial boost in correct predictions (+6%).

Table 7.2. Percentage of correct predictions.

Context Type	**Context**	**%Correct Prediction**
None	Baseline	30.58%
Single	Method Calls	32.46%
	Class Signatures	31.33%
	Most Similar Method	32.68%
	Issue Title	31.32%
	Issue Body	30.77%
	Frequent Invocations	31.80%
	Most Similar Statements	31.00%
Within-family	Most Similar Method + Class Signatures	32.72%
	Method Calls + Class Signatures	33.03%
	Most Similar Method + Method Calls	33.35%
	Most Similar Method + Method Calls + Class Signatures	33.88%
	Issue Title + Issue Body	31.20%
	Frequent Invocations + Most Similar Statements	31.17%
Cross-family	Best Code + Best Process	33.75%
	Best Code + Best Developer	33.58%
	Best Developer + Best Process	31.50%
	Best Code+ Best Developer + Best Process	33.54%

When statistically comparing the performance of the *baseline* with the models exploit-

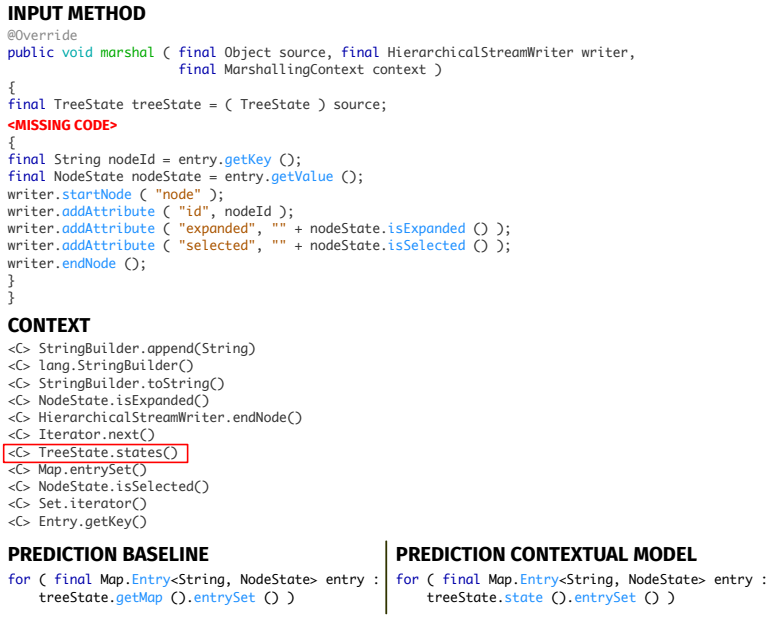

Figure 7.2. *Method calls* context helping the prediction.

ing individual types of contextual information, we found significant differences (adjusted p-value < 0.05) in all cases but for *issue body* and *most similar statements*. The OR for the significant differences ranges between 1.23 (*class signatures*) and 1.64 (*method calls*). An OR=1.64 indicates 64% higher odds of obtaining a correct prediction using the contextual model as compared to the *baseline*. The complete results of the statistical analysis are available in our replication package [repb].

Fig. 7.2 provides an example in which the *baseline* model was not able to generate the expected completion, wrongly recommending `treeState.getMap` in the `for` loop. Instead, when providing the model with *method calls* context, the model was able to exploit such information to identify `treeState.state` as the correct method call to feature in the `for`.

Fig. 7.3 provides instead an example in which the *baseline* model wrongly recommends `ability.accept(ability);` as completion. Instead, when providing the model with the *most similar method* as additional context (in this case, a Type-II clone), the model was able to exploit such a similarity to match `readyMap` in the method to complete to `activatedMap` in the contextual method, thus realizing the need for generating `readyMap.put(ability, false);`.

Worth mentioning is also the +4% ensured by the *frequent invocations* context (OR=1.44), featuring the method calls frequently used by the developer receiving the recommendation.

Among the *process contexts*, feeding the model with the title of the most relevant open issue seems to help (+2.5%), while the improvement is close to zero when the issue body is provided.

By combining the contexts belonging to the same family (Context Type = "Within-family" in Table 7.2), the improvement in performance can be pushed further when it comes to

INPUT METHOD

```
public boolean isReady(UUID id, MAbility ability) {
    Map<MAbility, Boolean> readyMap = ready.get(id);
    if (readyMap == null) {
        ready.put(id, new HashMap<>());
        return false;
    }
    Boolean readyValue = readyMap.get(ability);
    if (readyValue != null) {
        return readyValue;
    } else {
        <MISSING CODE>
    return false;
    }
}
```

CONTEXT

```
public boolean isActivated(UUID id, MAbility ability) {
    Map<MAbility, Boolean> activatedMap = activated.get(id);
    if (activatedMap == null) {
        activated.put(id, new HashMap<>());
        return false;
    }
    Boolean activatedValue = activatedMap.get(ability);
    if (activatedValue != null) {
        return activatedValue;
    } else {
        activatedMap.put(ability, false);
    return false;
    }
}
```

PREDICTION BASELINE

```
ability.accept(ability);
```

PREDICTION CONTEXTUAL MODEL

```
readyMap.put(ability, false);
```

Figure 7.3. *Most similar method* context helping the prediction.

the *coding contexts*. The three *coding contexts* together result in a relative +11% in correct predictions as compared to the baseline (33.88% *vs* 30.58%) — p-value <0.0001, OR=1.9. Such an improvement is not obtained, instead, for the other two families of contexts (*i.e., process* and *developer*), for which the performance of the combinations is in line or slightly worse than the single context types taken in isolation.

The bottom part of Table 7.2 reports the results achieved by combining the contexts being the best performing of each of the three families. This includes the *issue title* as representative of the *process context*, and the *frequent method invocations* for the *developer context*. Concerning the *coding context* a longer discussion is needed: The best performing model is the one exploiting a combination of all information (*i.e., most similar method + method calls + class signatures*). However, such a context tends to saturate the 1,024 tokens available for the model's input. Thus, combining it with even additional contextual information would not make sense, since the input will be cut in most of cases. For this reason, we selected the second best-performing model among the *coding contexts*, namely *most similar method + method calls*. The latter, while ensuring performance similar to the best one (33.35% *vs* 33.88% of correct predictions) requires, on average, half of the tokens for its representation.

As it can be seen from Table 7.2, while improvements can be obtained in terms of correct predictions as compared to the *baseline* (p-value < 0.0001 in all comparisons, with ORs ranging between 1.31 and 1.9), none of the experimented cross-family combinations outperforms the best within-family combination featuring all coding contexts. Such a result may be due to the fact that we did not manage to experiment with cross-family combinations

involving the best within-family context (due to limitations in the input size).

Take-away. *Additional contextual information can have a substantial impact on the model's performance. Our experiments showed relative improvements up to +11% in terms of correct predictions.*

Complementarity Analysis and Confidence of the Predictions

Table 7.3 reports the results of the complementarity analysis concerning the correct predictions generated by the *baseline* and by the models using different combinations of contextual information. Given the *baseline* and a specific context C_i, this means computing the union of the correct predictions generated by both approaches, and then counting those (i) generated by both models (*i.e.,* both models generated a correct prediction for a given instance), (ii) generated by the *baseline* only, and (iii) generated by the model exploiting C_i only. For example, in the case of the *method calls* context, 78.12% of correct predictions are shared with the baseline, 8.29% are only generated by the *baseline*, and 13.59% are only generated by the model exploiting the *method calls* context.

Table 7.3. Complementarity analysis: between the correct predictions (CP) generated by the *baseline* and by the models exploiting different contextual information.

Context	%CP Shared	%CP Only Baseline	%CP Only Context
Method Calls	78.12%	8.29%	13.59%
Class Signatures	79.03%	9.4%	11.57%
Most Similar Method	73.8%	10.22%	15.98%
Issue Title	82.85%	7.48%	9.67%
Issue Body	82.74%	8.35%	8.91%
Frequent Invocations	80.45%	8.01%	11.54%
Most Similar Statements	79.79%	9.49%	10.72%
Most Similar Method + Class Signatures	72.62%	10.78%	16.6%
Method Calls + Class Signatures	75.73%	8.75%	15.52%
Most Similar Method + Method Calls	72.52%	10%	17.48%
Most Similar Method + Method Calls + Class Signatures	71.72%	9.75%	18.53%
Issue Title + Issue Body	81.68%	8.24%	10.08%
Frequent Invocations + Most Similar Statements	78.01%	10.15%	11.84%
Best Code + Best Process	72.72%	9.39%	17.89%
Best Code + Best Developer	73.27%	9.31%	17.42
Best Developer + Best Process	80.37%	8.48%	11.15%
Best Code+ Best Developer + Best Process	71.46%	10.32%	18.22%

The results in Table 7.3 provide one important message: There is a good complementarity between the *baseline* and the models exploiting additional contextual information. For example, while the best-performing model (*i.e., most similar method + method calls + class signatures*) generates 18.53% of correct predictions which are missed by the *baseline*, the latter is still able to generate 9.75% of correct predictions which are missed by the contextual model.

Such a result points to the possibility of combining multiple models (*i.e.,* models exploiting different input representations) to boost performance. One possibility is to trigger,

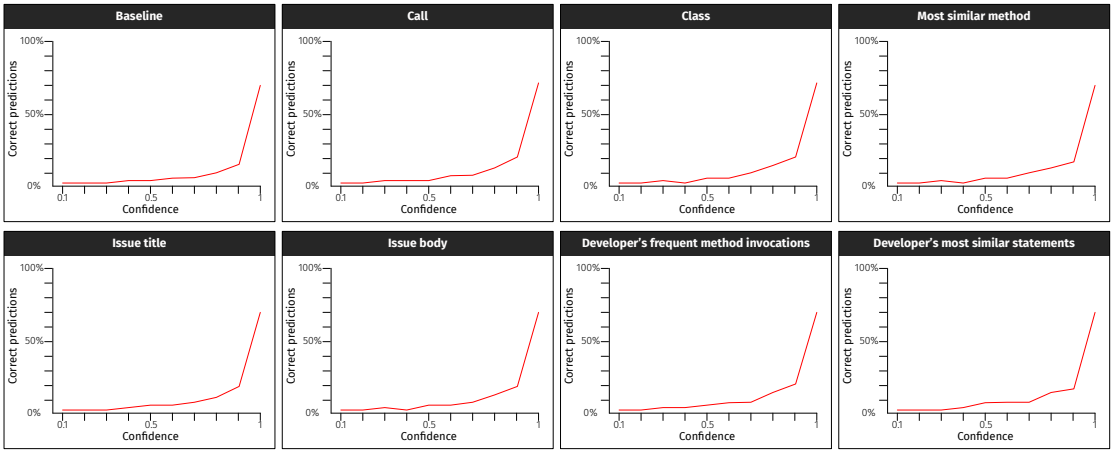

Figure 7.4. Percentage of correct predictions when varying the confidence of the model.

for a given code completion scenario, the model having the highest confidence in its prediction. Indeed, as most of DL models, T5 provides a *score* for each generated prediction. The score is a value lower than 0 representing the log-likelihood of the prediction. For example, having a log-likelihood of -1 means that the prediction has a likelihood of 0.37 ($ln(x) = -1 \implies x = 0.37$). The likelihood can be interpreted as the confidence of the model about the correctness of the prediction on a scale from 0.00 to 1.00 (the higher the better).

Fig. 7.4 shows the relationship between the percentage of correct predictions (Y-axis), and the T5 confidence (X-axis). We grouped the predictions into different buckets based on their confidence (*i.e.,* from 0.00 to 0.10, from 0.11 to 0.20, ..., from 0.91 to 1.00). The results are shown for the *baseline* and for the models exploiting each contextual information in isolation. There is a clear and strong trend indicating that the higher the confidence of the prediction, the higher the likelihood of obtaining a correct prediction. When the confidence is greater than 0.90, the models are usually able to recommend the correct completion in more than 70% of cases.

Given the complementarity observed for the different models and the reliability of the confidence as a "proxy" for the prediction quality, we experimented with a *confidence-based* model which, given a code completion scenario from our test set, recommends as completion the output of the model having the highest confidence among all those we experimented with (*i.e.,* the *baseline* and the ones exploiting different combinations of contextual information). Table 7.4 shows in the top part a performance comparison between the *baseline* and the *confidence-based* model in terms of correct predictions. As it can be seen, the *confidence-based* model is by far the best we experimented with, increasing the percentage of correct predictions generated by the *baseline* by a relative +22.4% (from 30.58% to 37.43%). This results in a statistically significant difference (p-value < 0.0001) with an OR=6.56.

Finally, the bottom part of Table 7.4 shows the complementarity analysis between the *baseline* and the *confidence-based* model. As it can be seen, there is a 20.9% of correct predictions only generated by the *confidence-based* model. Only a 3.19% of correct predictions is instead generated only by the *baseline* model.

Table 7.4. Baseline *vs* confidence-based model.

Measure	Baseline	Conf. model
PERFORMANCE COMPARISON		
Correct Predictions (#)	2,607	3,191
Correct Predictions (%)	30.58	37.43
COMPLEMENTARITY ANALYSIS		
Exact Match Prediction Shared	2,502/3,296 (75.91%)	
Exact Match Predictions only Baseline	105/3,296 (3.19%)	
Exact Match Predictions only Score model	689/3,296 (20.90%)	

Take-away. *Combining models using different contextual information by exploiting the confidence of their prediction can provide substantial benefits in terms of correct predictions for code completion.*

7.6 Validity Discussion

Construct validity. The usage of the correct predictions metric provides a limited view about the performance of the experimented models. For example, a model may generate a code completion prediction different but behaviorally equivalent to the expected one. Our experimental design simply considers such a prediction as wrong. However, given the goal of our study, we preferred to use a single and easy to interpret metric. For example, when comparing techniques it might be difficult to interpret what a +3% in terms of average CrystalBLEU [EP22] means in practice.

Internal validity. An important factor influencing DL performance is the calibration of hyperparameters which, due to feasibility reasons, we limited to the *baseline* model, assuming the others would benefit from the same configuration. However, a hyperparameters tuning also extended to the contextual models may only improve the performance of the latter, thus further reinforcing our finding: Additional contextual information can have a substantial impact on the model's performance.

Our experiment only focuses on "expanding" the contextual information provided to the model as compared to the *baseline*. One may wonder if good results can be obtained by, instead, shrinking the input representation. We experimented with this scenario, by creating two representations of the method to complete in which, given S the set of masked statements, we only provide the model with up to six or four statements surrounding it (rather than the whole method containing S as done with the *baseline*). To better understand, the representation using up to six surrounding statements inputs to the model S (the masked part to generate) with up to three statements above and below it. We say "up to", since not in all cases there will be at least three statements above/below S. We found that shrinking the contextual information provided to the model results in a strong drop of performance, with a relative loss in terms of correct predictions of -19.48% and -22.24% when providing only the six and the four surrounding statements, respectively. Complete results are available in our replication package [repb].

Conclusion validity. As explained in Section 7.4.2 we used appropriate statistical procedures, also adopting *p*-value adjustment when multiple tests were used within the same analysis.

The differences in performance among the different models might look minor at a first sight. For example, the *confidence-based* model achieves 37.43% of correct predictions versus the 30.58% of the *baseline*, resulting in a +22.4% relative improvement but *only* in a +6.85% absolute improvement.

Even the latter actually is a major improvement as compared, for example, to previous work in the literature proposing novel code completion techniques (*e.g.,* +0.8% in accuracy, Table 3 in [IGG22]).

External validity. We used T5 as representative of DL-based code completion techniques [CCP+21a]. Other DL models may lead to different results. Also, our study only targets Java and statement-level code completion (*i.e.,* completing up to two complete statements). Our findings may not generalize to other settings.

7.7 Conclusion and Future Work

We investigated how augmenting the contextual information provided to a DL model can benefit its performance in the context of code completion. We experimented with three families of contexts, namely *coding context*, *process context*, and *developer context*, showing that they can boost the correct predictions of the *baseline* up to a relative +11%. Also, the models exploiting different contextual information exhibit a good complementarity. For this reason, we combined them by exploiting the confidence of their predictions (*i.e.,* for a given code completion scenario the recommendation is triggered by the model having the highest confidence). This allowed us to achieve a relative improvement of +22% over the *baseline*.

Future work will mostly point to the generalizability of our findings to different languages and code-related tasks.

8

Conclusions and Future Work

In this thesis, we presented our research towards gaining a deeper understanding of the strengths and limitations of DL-based code recommenders. We started by investigating the behavior of DL models when facing complex code completions. Our findings, elaborated in Chapter 3, showed that DL models are still struggling when the code to recommend extends across multiple lines, while still producing some quite surprising (correct) predictions.

Based on these findings and the remarkable predictions offered by tools like GitHub Copilot, we started wondering whether DL models tend to copy the code encountered during the training phase at inference time. In Chapter 4 we found out that DL models did not exhibit a strong inclination to copy a significant amount of code from the training set. Then, in Chapter 5, we delved into the capability of DL models to generalize across different Java versions of the same programming language. Our findings revealed that DL models are strongly susceptible to the concept drift problem, resulting in a substantial degradation in performance when applied to a Java version not included in their training data.

We complemented the mostly quantitative investigations summarized above with a more qualitative study investigating which code recommender features are important from the developer's perspective. We conducted a survey involving 80 developers and the insights, discussed in Chapter 6, led to the identification of a taxonomy of 70 different characteristics of code recommenders that are crucial for developers. This taxonomy serves as a valuable guide for researchers in improving future recommender systems (see Chapter 6). As a concrete example of how our taxonomy can be used, we investigated the role of the additional contextual information on the performance of code recommender systems. Our results, presented in Chapter 7, showcased the capabilities of DL models to exploit the more complete contextual information, resulting in a substantial boost of performance.

8.1 Limitations and Future Work

8.1.1 Generalizability of our results to different programming languages and models

Our primary focus has been on the Java language, widely employed by developers all over the world. However, we believe that our findings could be generalized to different languages. Indeed, most of the languages share similar constructs and logical structures. Moreover, several of the models used in our experiments were at least pre-trained on CodeSearchNet dataset, that includes six different programming languages (Java, JavaScript, PHP, Go, Python, and Ruby). Despite this, a multi-language study assessing the generalizability of our results would be welcome.

Furthermore, we experimented with encoder-decoder models, like T5 and CodeT5. It would be interesting to validate whether the same findings hold also for decoder-only models, like the GPT-based models, that form the foundation of several state-of-the-art tools (*e.g.,* ChatGPT).

8.1.2 Focus on the developer's coding style

DL models, like GitHub Copilot, have demonstrated surprising capabilities to strongly tailor their suggestions to the input they receive. For instance, creating an ad-hoc context to augment the model's awareness, enhanced the quality of the code suggestions. This can be seen like an archetypal attempt to customize the prediction for a specific input. However, it poses the premise to developer-centric code recommenders, able to fit the suggestions to the user that is asking for support. However, integrating such "individual knowledge" into the recommender presents substantial challenges, since the model should be aware of a multitude of factors, including the libraries and design patterns that are familiar to each developer. Our future work will involve the extension of our context definitions, to create a novel context tailored to each developer, thus enabling the recommendation of personalized code.

8.1.3 Reduce the training time with small task-specific fine-tunings

Recently, the concern for the amount of energy wasted in nowadays society, has prompted interest in the research community to reduce the carbon footprint [PGL+21, SGM19] associated with the training of Large Language Models. During a MIT event, Sam Altman, CEO of OpenAI, a company that released several state-of-the-art models for text generation, revealed that training GPT-4 costed over 100 million USD. Moreover, training GPT-3, a model with 175 billion of parameters, required 14.8 days using 10,000 V100 GPUs [PGL+21]. These resource-intensive training processes can potentially have a disruptive environmental impact. To address these concerns and at the same time reduce the training cost, several researchers relied on already pre-trained models. Our analysis has confirmed the general sentiment regarding the generalizability of DL results on different java versions, showing the effectiveness of small fine-tunings to adapt a pre-trained model and give a significant

boost in the performance. In the future, we aim to delve deeper into the impact of small version-specific fine-tunings for increased efficiency.

8.1.4 Leverage suggestion from developers' community

Our survey with developers has shed some light on possible ways for improving code recommenders. We started focusing on enhancing the "awareness of code recommenders", leveraging the developers' knowledge to generate more contextually relevant recommendations. However, multiple suggestions from the developer community deserve exploration. For example, developers stressed the importance of the coverage of support (*i.e.,* the different situations in which the tool can assist them), with possible improvement for multi-lingual code, also highlighting the usability of the tool as a crucial aspect. In the future, we want to leverage these insights for improving the user experience with code recommenders.

8.2 Closing Words

In conclusion, our research has attempted to study the strengths and weaknesses of code recommender systems in order to build better tools, with an emphasis on identifying potential limitations and crafting developer-centric solutions. Our findings can contribute to enhance the functionality of code assistants to facilitate a harmonious collaboration between developers and AI models, also amplifying their productivity and effectiveness.

One emerging question is whether the advent of tools like GitHub Copilot and ChatGPT can impact the results described in this thesis, potentially rendering our results obsolete or even challenging their validity. Clearly, results highlighted in Chapter 3 can now be considered surpassed, since the task of generating a couple of statements can now be easily addressed by these models, even able to recommend entire functions. Investigating whether DL models tend to copy is still an interesting question, since it is possible that these models, trained on an unprecedented amount of lines of code, may tend to verbatim copy statements from the training data. We believe that even the generalizability across different versions of a programming language discussed in Chapter 5 is still a valid question, since the natural evolution of the languages should be always kept into account by researchers and practitioners, continuously updating their model when a new language version is released. The results presented in our taxonomy, exhaustively described in Chapter 6, could slightly be revised, since these models could potentially have sparked interest in different features which have been prompted to the practitioners' mind by the superior capabilities of these novel approaches. However, we asked the participants to envision characteristics for code recommenders, with no mention to the feasibility of their suggestions. For this reason, we believe that the changes to the taxonomy would be limited. The results we achieved by leveraging the contextual information are really promising, and they can be applied to any model. Hence, we believe that also GitHub Copilot and ChatGPT may benefit from this additional information and it would be really interesting to evaluate their performance when exploiting additional contextual information.

Bibliography

[AB14] Andrea Arcuri and Lionel C. Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Softw. Test. Verification Reliab.*, 24(3):219–250, 2014.

[ABBS14] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *22nd ACM/SIGSOFT International Symposium on Foundations of Software Engineering, FSE*, pages 281–293, 2014.

[ABLY19] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *7th International Conference on Learning Representations, ICLR*, 2019.

[ADS22] Toufique Ahmed, Premkumar T. Devanbu, and Anand Ashok Sawant. Learning to find usages of library functions in optimized binaries. *IEEE Trans. Software Eng.*, 48(10):3862–3876, 2022.

[AJ17] Fabio Villamarin Arrebola and Plinio Thomaz Aquino Junior. On source code completion assistants and the need of a context-aware approach. In *19th Human Interface and the Management of Information: Supporting Learning, Decision-Making and Collaboration, HCI*, pages 191–201, 2017.

[AK20] Gareth Ari Aye and Gail E. Kaiser. Sequence model design for code completion in the modern IDE. *CoRR*, abs/2004.05249, 2020.

[AKL21] Gareth Ari Aye, Seohyun Kim, and Hongyu Li. Learning autocompletion from real-world datasets. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP*, pages 131–139, 2021.

[All19] Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *ACM/SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, OOPSLA*, pages 143–153, 2019.

[alp] Alphacode. `https://storage.googleapis.com/deepmind-media/AlphaCode/competition_level_code_generation_with_alphacode.pdf`. Accessed: 2023-08-07.

[amt] Amazon mechanical turk. `https://www.mturk.com`. Accessed: 2023-05-31.

[ARSH14] Muhammad Asaduzzaman, Chanchal K. Roy, Kevin A. Schneider, and Daqing Hou. Context-sensitive code completion tool for better API usability. In *30th IEEE International Conference on Software Maintenance and Evolution ICSME*, pages 621–624, 2014.

[ARSH17] Muhammad Asaduzzaman, Chanchal K. Roy, Kevin A. Schneider, and Daqing Hou. Recommending framework extension examples. In *33th IEEE International Conference on Software Maintenance and Evolution ICSME*, pages 456–466, 2017.

[ASBN21] Shamsa Abid, Shafay Shamail, Hamid Abdul Basit, and Sarah Nadi. FACER: an API usage-based code-example recommender for opportunistic reuse. *Empir. Softw. Eng.*, 26(5):110, 2021.

[ASLY20] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models of code. In *International Conference on Machine Learning, ICML*, pages 245–256, 2020.

[AZLY19] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang., PACMPL*, 3(POPL):40:1–40:29, 2019.

[BDO+13] Gabriele Bavota, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. An empirical study on the developers' perception of software coupling. In *35th IEEE/ACM International Conference on Software Engineering, ICSE*, pages 692–701, 2013.

[BLMO14] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. Automating extract class refactoring: an improved method and its evaluation. *Springer Empirical Software Engineering, EMSE*, 19(6):1617–1664, 2014.

[BMM09] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *7th ACM Joint Meeting of the European Software Engineering Conference and the ACM/SIGSOFT International Symposium on Foundations of Software Engineering ESEC-FSE*, pages 213–222, 2009.

[BRBR16] Avishkar Bhoopchand, Tim Rocktäschel, Earl T. Barr, and Sebastian Riedel. Learning python code suggestion with a sparse pointer network. *CoRR*, abs/1611.08307, 2016.

[BW10] Raymond P. L. Buse and Westley Weimer. Learning a metric for code readability. *IEEE Trans. Software Eng.*, 36(4):546–558, 2010.

[BWH22] Nghi Bui, Yue Wang, and Steven C. H. Hoi. Detect-localize-repair: A unified framework for learning to debug with codet5. In *Findings of the Association for Computational Linguistics: EMNLP*, pages 812–823, 2022.

[CAD+22] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T. Devanbu, and Baishakhi Ray. Natgen: generative pre-training by "naturalizing" source code. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*, pages 18–30, 2022.

[CCP+21a] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. An empirical study on the usage of transformer models for code completion. *IEEE Transactions on Software Engineering, TSE*, 48(12):4818–4837, 2021.

[CCP+21b] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. An empirical study on the usage of BERT models for code completion. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*, pages 108–119, 2021.

[CFLB22] Fuxiang Chen, Fatemeh H. Fard, David Lo, and Timofey Bryksin. On the transferability of pre-trained language models for low-resource programming languages. In *30th IEEE/ACM International Conference on Program Comprehension, ICPC*, pages 401–412, 2022.

[CGP07] Filippo Corbo, Concettina Del Grosso, and Massimiliano Di Penta. Smart formatter: Learning coding style from existing source code. In *23rd IEEE International Conference on Software Maintenance ICSM*, pages 525–526, 2007.

[cod] Codet5. `https://huggingface.co/Salesforce/codet5-base`. Accessed: 2023-09-18.

[col] Google colab. `https://colab.research.google.com/`. Accessed: 2023-05-31.

[copa] Copilot chat. `https://docs.github.com/en/copilot/github-copilot-chat`. Accessed: 2023-12-19.

[copb] Copilot website. `https://copilot.github.com`. Accessed: 2022-11-10.

[CPA+23] Matteo Ciniselli, Luca Pascarella, Emad Aghajani, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. Source code recommender systems: The practitioners' perspective. In *45th IEEE/ACM International Conference on Software Engineering, ICSE*, pages 2161–2172, 2023.

[CPB22] Matteo Ciniselli, Luca Pascarella, and Gabriele Bavota. To what extent do deep learning-based code recommenders generate predictions by cloning code from the training set? In *19th IEEE/ACM International Conference on Mining Software Repositories, MSR*, pages 167–178, 2022.

[cpd] Cpd. `http://pmd.sourceforge.net`. Accessed: 2023-05-31.

[CPX+22] Chi Chen, Xin Peng, Zhenchang Xing, Jun Sun, Xin Wang, Yifan Zhao, and Wenyun Zhao. Holistic combination of structural and textual code information for context based API recommendation. *IEEE Trans. Software Eng.*, 48(8):2987–3009, 2022.

[CTJ+21] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[DAB21] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. Sampling projects in github for MSR studies. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR*, pages 560–564, 2021.

[DAP+17] Jayati Deshmukh, K. M. Annervaz, Sanjay Podder, Shubhashis Sengupta, and Neville Dubash. Towards accurate duplicate bug retrieval using deep learning techniques. In *IEEE International Conference on Software Maintenance and Evolution, ICSME*, pages 115–124, 2017.

[DCLT19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT*, pages 4171–4186, 2019.

[ded] Codesearchnet deduplication algorithm. `https://github.com/github/CodeSearchNet/blob/master/src/dataextraction/dedup_split.py`. Accessed: 2022-11-10.

[DGMH+23] Miguel Domingo, Mercedes García-Martínez, Alexandre Helle, Francisco Casacuberta, and Manuel Herranz. How much does tokenization affect neural machine translation? In *20th International Conference of Computational Linguistics and Intelligent Text Processing, CICLing*, pages 545–554, 2023.

[DM12] Markus Dreyer and Daniel Marcu. Hyter: Meaning-equivalent semantics for translation evaluation. In *Conference of the North American Chapter of the Asso-*

ciation for Computational Linguistics: Human Language Technologies, NAACL-HLT, pages 162–171, 2012.

[Dod08] Yadolah Dodge. *The concise encyclopedia of statistics*. Springer Science & Business Media, 2008.

[Dor12] Jonathan Dorn. A general software readability model. *MCS Thesis*, 5:11–14, 2012.

[dSAdO05] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia Marçal de Oliveira. A study of the documentation essential to software maintenance. In *23rd ACM Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information, SIGDOC*, pages 68–75, 2005.

[dSAEWV16] Luís Eduardo de Souza Amorim, Sebastian Erdweg, Guido Wachsmuth, and Eelco Visser. Principled syntactic code completion using placeholders. In *9th ACM/SIGPLAN International Conference on Software Language Engineering, SLE*, pages 163–175, 2016.

[EP22] Aryaz Eghbali and Michael Pradel. Crystalbleu: Precisely and efficiently measuring the similarity of code. In *44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE*, pages 341–342, 2022.

[FGL12] Stephen R. Foster, William G. Griswold, and Sorin Lerner. Witchdoctor: Ide support for real-time auto-completion of refactorings. In *34th IEEE/ACM International Conference on Software Engineering ICSE*, pages 222–232, 2012.

[FGT+20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. In *10th Conference on Empirical Methods in Natural Language Processing, EMNLP*, pages 1536–1547, 2020.

[FL02] Andrew Forward and Timothy Lethbridge. The relevance of software documentation, tools and technologies: A survey. In *2nd ACM Symp. on Doc. Eng, DocEng*, pages 26–33, 2002.

[FTDH15] Christine Franks, Zhaopeng Tu, Premkumar T. Devanbu, and Vincent J. Hellendoorn. CACHECA: A cache language model based code suggestion tool. In *37th IEEE/ACM International Conference on Software Engineering, ICSE*, pages 705–708, 2015.

[Gag94] Philip Gage. A new algorithm for data compression. *The C Users Journal*, 12(2):23–38, 1994.

[GDFW12] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for

$8 each. In *34th IEEE/ACM International Conference on Software Engineering, ICSE*, pages 3–13, 2012.

[git] Github copilot chat beta now available for all individuals. `https://github.blog/2023-09-20-github-copilot-chat-beta-now-available-for-all-individuals/` Accessed: 2023-10-08.

[GK05] Robert J Grissom and John J Kim. *Effect sizes for research: A broad practical approach*. Lawrence Erlbaum Associates Publishers, 2005.

[GKKP13] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In *35th ACM/SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, pages 27–38, 2013.

[GMP+18] Franz-Xaver Geiger, Ivano Malavolta, Luca Pascarella, Fabio Palomba, Dario Di Nucci, and Alberto Bacchelli. A graph-based dataset of commit history of real-world android apps. In *15th International Conference on Mining Software Repositories, MSR*, pages 30–33, 2018.

[GRL+21] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR*, 2021.

[GZB+14] João Gama, Indre Zliobaite, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM Comput. Surv.*, 46(4):44:1–44:37, 2014.

[GZZK16] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In *24th ACM/SIGSOFT International Symposium on Foundations of Software Engineering FSE*, pages 631–642, 2016.

[Han04] John M Hancock. Jaccard distance (jaccard index, jaccard similarity coefficient). *Dictionary of Bioinformatics and Computational Biology*, 2004.

[HBS+12] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. On the naturalness of software. In *34th IEEE/ACM International Conference on Software Engineering, ICSE*, pages 837–847, 2012.

[HD17] Vincent J. Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *11th ACM/SIGSOFT Joint Meeting on Foundations of Software Engineering ESEC-FSE*, page 763?773, 2017.

[HDB21] Heidi Hokka, Felix Dobslaw, and Jonathan Bengtsson. Linking developer experience to coding style in open-source repositories. In *28th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER*, pages 516–520, 2021.

[HHC+20] Yuan Huang, Shaohao Huang, Huanchao Chen, Xiangping Chen, Zibin Zheng, Xiapu Luo, Nan Jia, Xinyu Hu, and Xiaocong Zhou. Towards automatically generating block comments for code snippets. *Inf. Softw. Technol.*, 127:106373, 2020.

[HK09] Yoshiki Higo and Shinji Kusumoto. Enhancing quality of code clone detection with program dependency graph. In *16th Working Conference on Reverse Engineering, WCRE*, pages 315–316, 2009.

[HLX+18] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *26th IEEE/ACM International Conference on Program Comprehension, ICPC*, pages 200–210, 2018.

[HM05] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *27th International Conference on Software Engineering ICSE*, pages 117–125, 2005.

[Hol79] S. Holm. A simple sequentially rejective bonferroni test procedure. *Scandinavian Journal on Statistics*, 6(2):65–70, 1979.

[HP11] Daqing Hou and David M. Pletcher. An evaluation of the strategies of sorting, filtering, and grouping API methods for code completion. In *IEEE 27th International Conference on Software Maintenance, ICSM*, pages 233–242, 2011.

[HPGB19] Vincent J. Hellendoorn, Sebastian Proksch, Harald C. Gall, and Alberto Bacchelli. When code completion fails: a case study on real-world completions. In *41st IEEE/ACM International Conference on Software Engineering, ICSE*, pages 960–970, 2019.

[HR04] Rosco Hill and Joe Rideout. Automatic method completion. In *19th IEEE International Conference on Automated Software Engineering, ASE*, pages 228–235, 2004.

[hug] Hugging face's tokenizer repository. `https://github.com/huggingface/tokenizers`. Accessed: 2022-11-10.

[HWG+19] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436, 2019.

[HWM09] Sangmok Han, David R. Wallace, and Robert C. Miller. Code completion from abbreviated input. In *24th IEEE/ACM International Conference on Automated Software Engineering ASE*, pages 332–343, 2009.

[HWM11] Sangmok Han, David R. Wallace, and Robert C. Miller. Code completion of multiple keywords from abbreviated input. *Autom. Softw. Eng.*, 18(3-4):363–398, 2011.

[IGG22] Maliheh Izadi, Roberta Gismondi, and Georgios Gousios. Codefill: Multi-token code completion by jointly learning from structure and naming sequences. In *44th IEEE/ACM International Conference on Software Engineering, ICSE*, pages 401–412, 2022.

[int] Intellij. `https://www.jetbrains.com/idea/`. Accessed: 2023-05-31.

[JAM17] Siyuan Jiang, Ameer Armaly, and Collin McMillan. Automatically generating commit messages from diffs using neural machine translation. In *32nd IEEE/ACM International Conference on Automated Software Engineering, ASE*, pages 135–146, 2017.

[java] Java callgraph by georgios gousios and matthieu vergne and christoph laaber. `https://github.com/gousiosg/java-callgraph`. Accessed: 2023-05-31.

[javb] Java website. `https://www.java.com/`. Accessed: 2023-10-08.

[javc] Javalang. `https://github.com/c2nes/javalang/`. Accessed: 2022-11-28.

[javd] Javascript | mdn. `https://developer.mozilla.org/en-US/docs/Web/JavaScript`. Accessed: 2023-10-08.

[JLJ19] Lin Jiang, Hui Liu, and He Jiang. Machine learning based recommendation of method names: How far are we. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE*, pages 602–614, 2019.

[JS18] Xianhao Jin and Francisco Servant. The hidden cost of code completion: Understanding the impact of the recommendation-list length on its efficiency. In *15th IEEE/ACM International Conference on Mining Software Repositories, MSR*, pages 70–73, 2018.

[KATF18] Htoo Htoo Sandi Kyaw, Shwe Thinzar Aung, Hnin Aye Thant, and Nobuo Funabiki. A proposal of code completion problem for java programming learning assistant system. In *12th Springer Conference on Complex, Intelligent, and Software Intensive Systems CISIS*, pages 855–864, 2018.

[KBR+20] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. Big code != big vocabulary: open-vocabulary models for source code. In *42th IEEE/ACM International Conference on Software Engineering ICSE*, pages 1073–1085, 2020.

[Ken38] Maurice Kendall. A new measure of rank correlation. *Biometrika*, 1938.

[KLHK09] Jinhan Kim, Sanghoon Lee, Seung-won Hwang, and Sunghun Kim. Adding examples into java documents. In *2009 IEEE/ACM International Conference on Automated Software Engineering, ASE*, pages 540–544, 2009.

[KMBS20] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *37th International Conference on Machine Learning, ICML*, pages 5110–5121, 2020.

[KR18] Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *8th Conference on Empirical Methods in Natural Language Processing, EMNLP*, pages 66–71, 2018.

[KS19] Rafael-Michael Karampatsis and Charles A. Sutton. Maybe deep neural networks are the best choice for modeling source code. *CoRR*, abs/1903.05734, 2019.

[KZTC21] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. Code prediction by feeding trees to transformers. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE*, pages 150–162, 2021.

[Lev66] Vladimir Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707, 1966.

[LH19] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, ICLR*, 2019.

[LHL+17] Sun-Ro Lee, Min-Jae Heo, Chan-Gun Lee, Milhan Kim, and Gaeul Jeong. Applying deep learning based automatic bug triager to industrial projects. In *11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, pages 926–931, 2017.

[liz] Lizard bu terru yin. `https://pypi.org/project/lizard/`. Accessed: 2023-05-31.

[LLD+19] Jie Lu, Anjin Liu, Fan Dong, Feng Gu, João Gama, and Guangquan Zhang. Learning under concept drift: A review. *IEEE Trans. Knowl. Data Eng.*, 31(12):2346–2363, 2019.

[LLZJ20] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. Multi-task learning based pre-trained language model for code completion. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE*, pages 473–485, 2020.

[LMM+17] Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. Déjàvu: a map of code duplicates on github. *Proc. ACM Program. Lang.*, 1(OOPSLA):84:1–84:28, 2017.

[LNNN15] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports (N). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE*, pages 476–481, 2015.

[LOG+19] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.

[LRA23] Kim Tuyen Le, Gabriel Rashidi, and Artur Andrzejak. A methodology for refined evaluation of neural code completion approaches. *Data Min. Knowl. Discov.*, 37(1):167–204, 2023.

[LSZ+20] Hui Liu, Mingzhu Shen, Jiaqi Zhu, Nan Niu, Ge Li, and Lu Zhang. Deep learning based program generation from requirements text: Are we there yet? *IEEE Transactions on Software Engineering*, 48(4):1268–1289, 2020.

[LWG+22] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu-Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In *NeurIPS*, 2022.

[LWN20] Yi Li, Shaohua Wang, and Tien N. Nguyen. Dlfix: context-based code transformation learning for automated program repair. In *42nd IEEE/ACM International Conference on Software Engineering, ICSE*, pages 602–614, 2020.

[MAL18] Pedro Martins, Rohan Achar, and Cristina V Lopes. 50k-c: A dataset of compilable, and compiled, java projects. In *15th International Conference on Mining Software Repositories, MSR*, pages 1–5, 2018.

[MBP+15] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. How can I use this method? In *37th IEEE/ACM International Conference on Software Engineering, ICSE*, pages 880–890, 2015.

[MBP+17] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. ARENA: an approach for the automated generation of release notes. *IEEE Trans. Software Eng.*, 43(2):106–127, 2017.

[MCB15] Mariana Marasoiu, Luke Church, and Alan F. Blackwell. An empirical investigation of code completion usage by professional software developers. In *26th Annual Workshop of the Psychology of Programming Interest Group, PPIG*, page 14, 2015.

[McN47] Quinn McNemar. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, 12(2):153–157, 1947.

[MKX+18] Qing Mi, Jacky Keung, Yan Xiao, Solomon Mensah, and Yujin Gao. Improving code readability classification using convolutional neural networks. *Inf. Softw. Technol.*, 104:60–71, 2018.

[MM04] Paul McNamee and James Mayfield. Character n-gram tokenization for european language text retrieval. *Information retrieval*, 7(1):73–97, 2004.

[MSC+21] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader-Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE*, pages 336–347, 2021.

[Mur19] Gail C. Murphy. Beyond integrated development environments: adding *context* to software development. In *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2019, Montreal, QC, Canada, May 29-31, 2019*, pages 73–76, 2019.

[MXBK05] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. In *27th ACM/SIGPLAN Conference on Programming Language Design and Implementation PLDI*, pages 48–61, 2005.

[NKZ17] Haoran Niu, Iman Keivanloo, and Ying Zou. API usage pattern recommendation for software development. *J. Syst. Softw. JSS*, 129:127–139, 2017.

[NN15] Anh Tuan Nguyen and Tien N Nguyen. Graph-based statistical language model for code. In *37th IEEE/ACM International Conference on Software Engineering, ICSE*, volume 1, pages 858–868, 2015.

[NNN+12] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *34th IEEE/ACM International Conference on Software Engineering, ICSE*, pages 69–79, 2012.

[NNN16] Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. A large-scale study on repetitiveness, containment, and composability of routines in open-source projects. In *13th IEEE/ACM International Conference on Mining Software Repositories, MSR*, pages 362–373, 2016.

[NSMB12] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. What makes a good code example?: A study of programming q&a in stackoverflow. In *28th IEEE International Conference on Software Maintenance, ICSM*, pages 25–34, 2012.

[NYN22] Anh Tuan Nguyen, Aashish Yadavally, and Tien N. Nguyen. Next syntactic-unit code completion and applications. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE*, pages 180:1–180:5, 2022.

[PANM16] Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini. Evaluating the evaluations of code recommender systems: a reality check. In *31st IEEE/ACM International Conference on Automated Software Engineering, ASE*, pages 111–121, 2016.

[PBL13] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. Seahawk: stack overflow in the IDE. In *35th International Conference on Software Engineering, ICSE*, pages 1295–1298, 2013.

[PBP+14] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Mining stackoverflow to turn the IDE into a self-confident programming prompter. In *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, pages 102–111, 2014.

[PGBG12] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed completion of partial expressions. In *34th ACM/SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, pages 275–286, 2012.

[PGL+21] David A. Patterson, Joseph Gonzalez, Quoc V. Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David R. So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training. *CoRR*, abs/2104.10350, 2021.

[PHD11] Daryl Posnett, Abram Hindle, and Premkumar T. Devanbu. A simpler model of software readability. In *8th International Working Conference on Mining Software Repositories, MSR*, pages 73–82, 2011.

[PLM15] Sebastian Proksch, Johannes Lerch, and Mira Mezini. Intelligent code completion with bayesian networks. *ACM Trans. Softw. Eng. Methodol.*, 25(1):3:1–3:31, 2015.

[pyt] Python website. `https://www.python.org/`. Accessed: 2023-10-08.

[qua] Qualtrics. `https://www.qualtrics.com`. Accessed: 2023-05-31.

[RC07] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queen's School of Computing TR*, 541(115):64–68, 2007.

[RC08] Chanchal Kumar Roy and James R Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *16th IEEE International Conference on Program Comprehension, ICPC*, pages 172–181, 2008.

[Rei07] Steven P. Reiss. Automatic code stylizing. In *22th IEEE/ACM International Conference on Automated Software Engineering, ASE*, page 74?83, 2007.

[repa] Replication package for "an empirical study on the usage of transformer models for code completion". `https://github.com/mciniselli/T5_Replication_Package.git`. Accessed: 2023-05-31.

[repb] Replication package for "deep learning-based code completion: On the impact on performance of additional contextual information". https://github.com/completion-context/completion-context-replication. Accessed: 2023-05-31.

[repc] Replication package for "on the generalizability of deep learning-based code completion across programming language versions". https://github.com/java-generalization/java-generalization-replication. Accessed: 2023-11-01.

[repd] Replication package for "source code recommender systems: The practitioners' perspective". https://code-recommenders.github.io. Accessed: 2023-09-23.

[repe] Replication package for "to what extent do deep learning-based code recommenders generate predictions by cloning code from the training set?". https://doi.org/10.5281/zenodo.6460983. Accessed: 2023-05-31.

[RGL+20] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *CoRR*, abs/2009.10297, 2020.

[RL08] Romain Robbes and Michele Lanza. How program history can improve code completion. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE*, pages 317–326, 2008.

[RL10] Romain Robbes and Michele Lanza. Improving code completion with program history. *Automated Software Engineering, ASE*, 17(2):181–212, 2010.

[RMWZ14] Martin P. Robillard, Walid Maalej, Robert J. Walker, and Thomas Zimmermann. *Recommendation Systems in Software Engineering*. Springer Publishing Company, Incorporated, 2014.

[RSR+20] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020.

[RTX09] Madhuri Marri R, Suresh Thummalapenta, and Tao Xie. Improving software quality via code searching and mining. In *31th IEEE/ACM ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, pages 33–36, 2009.

[RvDJ13] Steven Raemaekers, Arie van Deursen, and JoostVisser. The maven repository dataset of metrics, changes, and dependencies. In *10th Working Conference on Mining Software Repositories, MSR*, pages 221–224, 2013.

[RVY14] Veselin Raychev, Martin T. Vechev, and Eran Yahav. Code completion with statistical language models. In *36th ACM/SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, pages 419–428, 2014.

[SDFS20] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: code generation using transformer. In *28th ACM Joint European Software Engineering Conference and the ACM/SIGSOFT International Symposium on the Foundations of Software Engineering ESEC-FSE*, pages 1433–1443, 2020.

[SGM19] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in NLP. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL*, pages 3645–3650, 2019.

[SHB16] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *54th Annual Meeting of the Association for Computational Linguistics, ACL*, pages 1715–1725, 2016.

[sima] Simian by simon harris. `https://www.harukizaemon.com/simian`. Accessed: 2023-05-31.

[simb] Simscan. `http://www.blue-edge.bg/download.html`. Accessed: 2022-05-23.

[SLH+21] Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Vicente Franco, and Miltiadis Allamanis. Fast and memory-efficient neural code completion. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR*, pages 329–340, 2021.

[SLOP18] Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. A comprehensive model for code readability. *J. Softw. Evol. Process.*, 30(6):e1958, 2018.

[slp] Software language processing (slp) library for n-gram models. `https://github.com/SLP-team/SLP-Core`. Accessed: 2022-11-10.

[SMC74] Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[src] Srcml website. `https://www.srcml.org/`. Accessed: 2022-11-10.

[SSTS21] Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. You autocomplete me: Poisoning vulnerabilities in neural code completion. In *30th Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1559–1575, 2021.

[TC22] Sergey Troshin and Nadezhda Chirkova. Probing pretrained models of source codes. In *Fifth BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP, BlackboxNLP@EMNLP*, pages 371–383, 2022.

[TCC18] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. Ten years of jdeodorant: Lessons learned from the hunt for smells. In *25th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER*, pages 4–14, 2018.

[TDS+20] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers. *CoRR*, abs/2009.05617, 2020.

[TDSS22] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. Generating accurate assert statements for unit test cases using pretrained transformers. In *3rd IEEE/ACM International Conference on Automation of Software Test, AST*, pages 54–64, 2022.

[TMM+22] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. Using pre-trained models to boost code review automation. In *44th IEEE/ACM International Conference on Software Engineering, ICSE*, pages 2291–2302, 2022.

[TNAN11] Ahmed Tamrawi, Tung Thanh Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. Fuzzy set and cache-based approach for bug triaging. In *19th ACM Joint Meeting of the European Software Engineering Conference and ACM/SIGSOFT International Symposium on the Foundations of Software Engineering ESEC-FSE*, pages 365–375, 2011.

[TPT+21] Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. Towards automating code review activities. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE*, pages 163–174, 2021.

[TR16] Christoph Treude and Martin P. Robillard. Augmenting API documentation with insights from stack overflow. In *38th International Conference on Software Engineering, ICSE*, pages 392–403, 2016.

[TSD14] Zhaopeng Tu, Zhendong Su, and Premkumar T. Devanbu. On the localness of software. In *22nd ACM/SIGSOFT International Symposium on Foundations of Software Engineering, FSE*, pages 269–280, 2014.

[TT22] F. Tian and C. Treude. Adding context to source code representations for deep learning. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 374–378, 2022.

[TWB+19] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Trans. Softw. Eng. Methodol.*, 28(4):19:1–19:29, 2019.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *30th Advances in Neural Information Processing Systems NIPS*, pages 5998–6008, 2017.

[wan] Weights and biases website. `https://www.wandb.com/`. Accessed: 2022-11-10.

[WAN+21] Fengcai Wen, Emad Aghajani, Csaba Nagy, Michele Lanza, and Gabriele Bavota. Siri, write the next method. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE*, pages 138–149, 2021.

[WDS+19] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and Jamie Brew. Huggingface's transformers: State-of-the-art natural language processing. *CoRR*, abs/1910.03771, 2019.

[Wil45] Frank Wilcoxon. Individual comparisons by ranking methods. *International Biometric Society, Wiley*, 1(6):80–83, 1945.

[WLG+23] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. Codet5+: Open code large language models for code understanding and generation. *CoRR*, abs/2305.07922, 2023.

[WRG21] Kexin Wang, Nils Reimers, and Iryna Gurevych. TSDAE: using transformer-based sequential denoising auto-encoderfor unsupervised sentence embedding learning. In *Findings of the Association for Computational Linguistics: EMNLP*, pages 671–688, 2021.

[WSLJ20] Wenhan Wang, Sijie Shen, Ge Li, and Zhi Jin. Towards full-line code completion with neural language models. *CoRR*, abs/2009.08603, 2020.

[WTM+20] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful assert statements for unit test cases. In *42nd IEEE/ACM International Conference on Software Engineering, ICSE*, pages 1398–1409, 2020.

[WVVP15] Martin White, Christopher Vendome, Mario Linares Vásquez, and Denys Poshyvanyk. Toward deep learning software repositories. In *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR*, pages 334–345, 2015.

[WWJH21] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *11st Conference on Empirical Methods in Natural Language Processing, EMNLP*, pages 8696–8708, 2021.

[WWW+22] Xin Wang, Yasheng Wang, Yao Wan, Fei Mi, Yitong Li, Pingyi Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. Compilable neural code generation with compiler feedback. In *Findings of the Association for Computational Linguistics: ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 9–19, 2022.

[XLD+17] Xin Xia, David Lo, Ying Ding, Jafar M. Al-Kofahi, Tien N. Nguyen, and Xinyu Wang. Improving automated bug triaging with specialized topic model. *IEEE Trans. Software Eng.*, 43(3):272–297, 2017.

[XP06] Tao Xie and Jian Pei. MAPO: mining API usages from open source repositories. In *3rd IEEE/ACM International Workshop on Mining Software Repositories, MSR*, pages 54–57, 2006.

[XVN22] Frank F. Xu, Bogdan Vasilescu, and Graham Neubig. In-ide code generation from natural language: Promise and challenges. *ACM Trans. Softw. Eng. Methodol.*, 31(2):29:1–29:47, 2022.

[YHL16] Di Yang, Aftab Hussain, and Cristina Videira Lopes. From query to usable code: an analysis of stack overflow code snippets. In *P13th International Conference on Mining Software Repositories, MSR*, pages 391–402, 2016.

[YX19a] Yixiao Yang and Chen Xiang. Improve language modelling for code completion by tree language model with tree encoding of context (S). In *31st International Conference on Software Engineering and Knowledge Engineering, SEKE*, pages 675–777, 2019.

[YX19b] Yixiao Yang and Chen Xiang. Improve language modelling for code completion through learning general token repetition of source code. In *31st International Conference on Software Engineering and Knowledge Engineering, SEKE*, pages 667–777, 2019.

[YY95] Benjamini Yoav and Hochberg Yosef. Controlling the false discovery rate: A practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, 57(1):289–300, 1995.

[Zar72] Jerrold H. Zar. Significance testing of the spearman rank correlation coefficient. *Journal of the American Statistical Association*, 67(339):pp. 578–580, 1972.

[zie] Github copilot: Parrot or crow? by albert zieger. `https://docs.github.com/en/github/copilot/research-recitation`. Accessed: 2023-05-31.

[ZJW+23] Wangchunshu Zhou, Yuchen Eleanor Jiang, Ethan Wilcox, Ryan Cotterell, and Mrinmaya Sachan. Controlled text generation with natural language instructions. In *International Conference on Machine Learning, ICML*, pages 42602–42613, 2023.

[ZKL+22] Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. Productivity assessment of neural code completion. In *International Symposium on Machine Programming*, pages 21–29, 2022.