Master

March 29, 2022

Assisted interactive programming

Generating context-aware single-line code from Natural Language

Alex Wolf

of Luzern, Switzerland (12-526-216)

supervised by

Prof. Dr. Harald C. Gall

Dr. Pasquale Salza & Marco Palma

University of Zurich[UZH]

s.e.a.l.

software evolution & architecture lab

Master

Assisted interactive programming

Generating context-aware single-line code from Natural Language

Alex Wolf

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Master

Author: Alex Wolf, alex.wolf@uzh.ch

Project period: July 30^{th}, 2021 - March 30^{th}, 2022

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich@citexnum

Acknowledgements

I would like to thank my supervisors Dr. Pasquale Salza and Marco Palma for all their insight and guidance given throughout the work as well as Prof. Dr. Harald C. Gall for making the work possible. I would also like to thank my family, my girlfriend, and friends for their support and faith in my abilities.

Abstract

The growing significance of programming languages is manifested by the change in educational curricula, which include programming lectures as early as in primary school. The growing significance also demonstrates that the means to acquire programming skills need to improve. We propose an interactive assistant to help novice programmers acquire knowledge through natural language during their programming tasks. Thus, *learning by doing* with an assistant that supports the user by providing source code in case of difficulties or missing know-how. Our approach is targeting single-line code generation while considering the natural language intent and an extensive context to provide accurate and relevant recommendations in the form of a single-line of source code. This approach is based on the idea that learning programming languages require solving programming exercises in addition to a fast feedback loop. Our approach aims to provide the learner with more opportunities to learn and understand the programming language by assisting them with minimal source code to help them continue with their task. Thus, only generating single-lines of source code instead of providing full functions. We implemented two models intending to contextualize single-line code generation using a custom context workflow. Our evaluations show that our approach is able to learn strong context representations from our custom context workflow as well as an option to improve context compression. In summary, we contribute a trained context-sensible model that takes natural language input, context, and predicts the next line. Using a custom workflow to deal with contexts of variable size, a new Java-based dataset, of over 200'000 samples tailored to our task of generating single-line Java code. Each sample contains context information, natural language intent, and the target line. Allowing us to contextualize our model, and we compare our work with other state-of-the-art approaches.

Zusammenfassung

Die wachsende Bedeutung der Informatik manifestiert sich schon in der Grundschule, hier werden einfache Grundkenntnisse der Programmierung bereits ab Primarschule in den Lehrplan integriert. Daraus folgt, dass die Mittel zum Erwerb von Programmierkenntnissen erweitert und altersgemäss adaptiert werden müssen. Wir schlagen einen interaktiven Assistenten vor, der Programmieranfängern hilft, sich während ihrer Programmieraufgaben Wissen durch natürliche Sprache anzueignen. Unser Ansatz verbindet *learning by doing* mit einem Assistenten, der die anwendende Person bei Schwierigkeiten oder fehlendem Fachwissen unterstützt, sich dieses anzueignen. Unsere Herangehensweise zielt auf die Generierung von einem einzeiligen Code ab und berücksichtigt dabei die mit natürlicher Sprache formulierte Absicht und einen umfangreichen Kontext für die Bereitstellung von akuraten und relevante Empfehlungen in Form einer einzigen Quellcodezeile. Dieser Grundsatz basiert auf der Idee, dass das Erlernen von Programmiersprachen neben einer schnellen Feedbackschleife auch das Lösen von Programmieraufgaben erfordert. Unser Ansatz zielt darauf ab, der lernenden Person mehr Möglichkeiten zum Erlernen und Verstehen der Programmiersprache zu bieten, um ihr bei der Fortsetzung der Aufgabe behilflich zu sein, indem wir minimalen Quellcode bereitstellen. Es werden also nur einzelne Zeilen Quellcode generiert, anstatt gänzliche Funktionen bereitzustellen. Wir haben zwei Modelle mit dem Ziel der kontextualisierten Generierung von einzeiligem Code implementiert, mithilfe eines benutzerdefinierten Kontext-Workflows. Unsere Auswertungen zeigen, dass unser Ansatz in der Lage ist, starke Kontextdarstellungen aus unserem benutzerdefinierten Kontextworkflow zu lernen, sowie eine Option zur Verbesserung der Kontextkomprimierung. Zusammenfassend liefern wir ein trainiertes kontextsensitives Modell, das Kontext und Eingaben in natürlicher Sprache verwendet und die nächste Zeile unter Verwendung eines benutzerdefinierten Workflows vorhersagt. Dieser individuelle Arbeitsablauf (Workflow) ermöglicht es aus Kontext variabler Grössen Informationen zu extrahieren. Zusätzlich haben wir ein neues Java-basiertes dataset erstellt, das auf unseren Ansatz massgeschneidert ist. Das Dataset enthält sowohl die Absicht, die Kontextinformationen als auch den einzeiligen Quellcode. Des Weiteren vergleichen wir unsere Arbeit mit anderen aktuellen Forschungsansätzen.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

While traditional introductory programming courses teach algorithmic problem-solving, where a program is just a sequence of consecutively executed steps, most applications in today's industry rely heavily on concurrency, parallelism, and multiple interacting components. However, learning to program is generally perceived as difficult as shown by several studies [10, 17, 30, 36].

The lack of personal instructions is one such challenge faced by novice programmers as stated by Lahtinen et al., this lack is even more prominent when a program is to be implemented from scratch without any hints, as stated by Terada and Watanobe, due to cognitive hurdles and missing know-how [17, 30]. Moreover, programming requires logical thinking and the ability to identify syntactical and conceptual errors, as stated by Watanobe, Yutaka et al. This impacts the time required by instructors to create and evaluate the tasks as well as the waiting time of students in this feedback loop. Additionally, even if a novice programmer may conceptually understand what needs to be done, they are still faced with the challenge of a concrete implementation, as stated by Xu et al. [41]. Therefore, learners are faced with an amalgamation of hurdles, namely the lack of personalized instructions and the missing know-how. The challenges faced by novices can be categorized according to the seven barriers identified by Wang et al.: (1) Decision Barrier: Should I ask for an example? (2) Search Barrier: How do I explain the example I want? (3) Integration Barrier: How do I integrate the example code into my own code? (4) Mapping Barrier: How do I map a property of the example code to my own code? (5) Understanding Barrier: How do I use an unfamiliar code block? (6) Modification Barrier: How to modify the example code to fit my own needs? (7) Testing Barrier: How to test the example code? [34]. These categories allow us to simplify and generalize the hurdles faced by novices. Hence, we attempt to categorize the identified hurdles into the aforementioned categories. Consequently, we argue that the lack of personal instructions is defined under the umbrella of the understanding barrier. Furthermore, understanding a problem conceptually plays into the search and decision barriers. However, it is difficult to attribute the missing know-how to just one of the categories. We do, however, argue that the categories are comprehensive enough that the problems can be attributed to one of the categories given enough context.

In spite of the many challenges faced by learners, who are in the process of acquiring programming skills, it is also generally agreed that it is paramount that students learn programming by exercising by themselves and doing practical exercises [17, 30]. Furthermore, this is supported by the perception of the students themselves, as shown by Lahtinen et al. in their study consisting of more than 500 students and teachers [17]. The increased need for programming skills is further manifested by educational curricula that include programming lectures, as early as in primary school [11, 36].

Thus, leading us to *interactive programming*, which allows a program to be adjusted while part of the program is already working. A trial and error approach to programming is supported, and a developer can get feedback on every new line produced. Several studies state that a fast feed-

back loop benefits the learning of programming languages (PLs) [5, 25]. Bai and Cui argue that the main path to improving programming skills is to solve programming exercises in addition to feedback. Thus, the combination of a fast feedback loop and actual programming leads to a better learning experience. This allows us to characterize an interactive programming language as follows: (1) the ability to respond on a line-level basis, (2) a fast feedback cycle, and (3) on-the-fly adjustability of variables and statements. While there are programming languages and tools that support this paradigm they usually provide feedback in the form of compiler errors, which need to be understood and analyzed by the learner. This leaves the novice programmer with no other option, but to search the web for examples or explanations. While there is no lack of examples (*e.g.*, on StackOverflow (SO)), these examples need to be understood, mapped, integrated, and modified to actually provide the novice programmer with the help they need. Even though the current interactive paradigm presents us with limitations, our intuition regarding the educational aspects of the paradigm is that it can be improved with an integrated natural language (NL) based generation option that provides the learner with contextualized examples. Therefore, empowering the learners with the ability to compare their own code to the proposed solution, in addition to reducing the initially required know-how. Thus, reducing the feedback loop and providing contextualized examples that can be explored by conceptual NL input. However, we also believe that fully functional examples lower the learning opportunities by taking away the need to think about the next required step.

The following simple use case should illustrate the intentions of our idea and intuitions: Suppose a novice developer or student that is unfamiliar with the codebase or the language itself is stuck on a problem but knows how to formulate his intentions with NL. Instead of searching for an example on the web, the user writes his intention as a comment and the model predicts the next line, which is already adapted to his previously written statements. Thus, eliminating the need to integrate, modify, and map an example to the current context. The learner is now able to analyze, evaluate, and synthesize the new bite-sized information and proceed with his task by contemplating the next necessary steps. This leads us to code generation, the task of generating PL from NL, which provides the ability to formulate the conceptual need with NL in order to create examples tailored to the learners' task. Thus, enabling us to combine the need for more in-depth problem-solving lectures with code generation.

We now present an excerpt of the currently available machine learning (ML) tools that provide recommendations and generate code based on NL input. For instance, one such tool is *GitHub Copilot*[1] which aims to provide support to the development process and is currently providing a technical preview. Furthermore, the current state-of-the-art (SOTA) approaches attend to some of the issues and ideas we mentioned and can be classified into the following categories: (1) generation without context consideration, such as the approaches proposed by Xu et al. and Wang et al. [35, 40], (2) special input approaches, such as the one presented by Phan et al. that requires Javadoc styled method comments [24], (3) comment generation approaches, such as Wong et al. and Wong et al. [38, 39], and (4) contextualized generation. However, generating full functions is a common theme among the generating approaches. Which leads to a reduction in learning opportunities and is not suited for educational purposes. On the other hand, we conjecture that single-line recommendations provide a more educational approach, granting learners the ability to improve their coding skills instead of providing full-fledged solutions for whole functions. There is a clear need for better ecosystems centered around feedback and assistants to help novices not be discouraged by gaps in their skill set [30, 31, 36]. Xu et al. performed an extensive user study of integrated development environment (IDE) code generation and demonstrated challenges and limitations in the current state. While their results were inconclusive they also show that the developers subjectively enjoyed the experience of using in-IDE tools. This leads us to the assumption that better tools would not just be accepted but could potentially be helpful.

[1] Github Copilot [1]

We propose an interactive approach that allows learners to formulate their intention/problem with NL and predicts the next line of code while also considering the current context. Thus, reducing the integration, mapping, and modification barriers by providing contextualized target code, that is already adjusted according to the context. Therefore, removing the need to adjust pre-existing examples to their own code context, while also providing learning opportunities by only generating single-line target code instead of full functions. Due to the nature of single-line code generation, this approach should also provide feedback and hints by allowing learners to compare their own solution with the generated one. Hence, alleviating the lack of feedback and hints. While full function generation may reduce the required time to write a solution it also reduces the learning opportunities as the thinking process of the learner is reduced. We believe that single-line code generation provides learning opportunities in addition to a time reduction. Our approach is based on the pre-trained model from Wang et al. and leverages the PyTorch [23] and HuggingFace's transformers (transformers) [37] libraries. In order to validate our approach, we compared our model to a SOTA approach [13] using two metrics, namely bilingual evaluation understudy (BLEU) and Code BLEU (CodeBLEU). We achieved SOTA results of 8.726 using the CodeBLEU score, improving the Natural Language-Guided Programming (NLGP) model by 3.643 points [13]. However, our model only achieves a BLEU score of 1.303, which is 0.767 points lower than the NLGP model.

In summary, this work contributes:

- a trained context-sensible model that takes NL input, context, and predicts the next line. Using a custom workflow to deal with contexts of variable size,
- a new Java-based dataset, of over 200'000 samples tailored to our task of generating single-line Java code. Each sample contains context information, NL intent, and the target line. Allowing us to contextualize our model,
- and we compare our work with other SOTA approaches.

The rest of this paper is structured as follows: Chapter 2 goes through different approaches currently used for code generation. Chapter 3 details the architecture, data mining, and exploratory data analysis (EDA) process. Chapter 4 showcases how our model was trained and what hardware was used. Chapter 5 compares the results of our work using two metrics. And we conclude our work in Chapter 6.

Chapter 2

Related Work

In this chapter, we describe approaches that are related to our own, as well as SOTA approaches that compete with ours.

2.1 Comment generating approaches

We consider comment generation and code generation as correlating tasks, which is why we assume they could provide insight into our own task.

AutoComment is one such comment-generating approach, implemented by Wong et al. that generates comments with SO data. The generated comments are grouped by code segments, according to snippets from SO. *AutoComment* leverages code clone detection techniques, as well as, natural language processing (NLP) techniques to generate comments in a two-component process. These two major components consist of generating code-description mappings and a second component for generating comments based on code clone detection techniques, where similar code is associated with the comments [39].

More recently Wong et al. improved their previous work (*viz.*, *AutoComment*) and show the difficulties NLP encounters with project-specific comments, which are sometimes neither complete sentences nor describing the code without project-specific jargon. They also show the need for better code clone detection techniques, as the similarity of the code might be high but hidden with context-specific code surrounding it or different ordering [38]. These difficulties could impact our approach, however analogous to their previous work this approach generates comments instead of code.

2.2 Context-less approaches

Context-less approaches are the basis of code generation and were studied extensively in recent years using many different technologies [2, 8, 16, 35, 40]. They provide us with valuable information regarding code generation and techniques. Thus, we consider them related to our work.

A recently proposed model-agnostic approach, by Xu et al., based on the idea of data augmentation, retrieval, and data resampling to incorporate external resources combines code generation with additional data to improve predictions. The approach implements the model in two steps, where the model is pre-trained on automatically extracted external data (from resources such as SO and API documentation) and then fine-tuned on small manually curated datasets. They implemented their approach on top of a syntax-based method for code generation, TranX [43], with additional hypothesis reranking and were able to improve the previous methods with the incorporation of external resources [40].

Phan et al. presented a multi-task approach based on both NL and programming language with an encoder-decoder model with attention using both bimodal (both NL and code) as well as unimodal data (only code) called *CoTexT*. While *CoTexT* generates code from text, they generate the code from Javadoc style method comments, which is one of the reasons why this approach is not considered for comparison. Moreover, this approach does not consider the context. The approach could be classified into the category of special input approaches, but for simplicity, we added the approach to the context-less category.

The current SOTA approach (at the time of writing) proposed by Wang et al. uses a novel unified encoder-decoder approach as well as a novel identifier-aware pre-training objective that considers token type information from code. The novel pre-training objective enables the model to distinguish between identifier code tokens and recover them when they are masked. While many similar approaches produce source code, such as [2, 4, 8, 9, 26], CodeT5 produces SOTA results (at the time of writing), which is why we decided to base our approach on the CodeT5 model. While we base our approach on the CodeT5 model we do not compare our results with their approach, as the CodeT5 model generates full functions, which conflicts with our goal of providing learning opportunities. Furthermore, all of these approaches do not consider the context and thus do not alleviate the integration, mapping, and modification barriers.

2.3 Context-aware approaches

The third category of generative approaches is the most similar to what we intend to do. We assume that this class of approaches is able to provide learners with code that is tailored to their needs with respect to (w.r.t.) the barriers they face.

One such approach is presented by Iyer et al., which considers member variables and methods to generate source code. They employed a multi-step decoder that starts by attending the NL intent and only afterwards attends to the variables and methods [15]. Granted that this approach considers both NL and context, they also produce full functions and use production rules as an intermediary output that are then used to generate the source code. Hence, conflicting with our goals.

Some emerging works in the recent literature also explore interactive approaches to generate contextualized code from NL intent. An interactive approach to generate code from a triplet containing the code context, a NL intent, and a target, was introduced by Heyman et al. Their approach uses language models (LMs) to generate text and is based on the GPT-2 model from Radford et al. [26]. They demonstrated how natural language-guided programming could be implemented and developed three language model variants to study the impact of natural language intent on prediction quality. They also provided a research agenda for natural language-guided programming, as well as open key research questions. The model was trained on data gathered from Jupyter notebooks [13]. While the approach is similar to ours we opted to base our approach on the CodeT5 model [35] while they decided to use GPT-2 [26]. Furthermore, our approach provides a more detailed code context extracted from multiple parts of the preceding context instead of taking everything before the intent as context. This approach was published during our research and is viewed as the SOTA approach in the field. We decided to use the *NLGP* approach as a base comparison to our approach since they consider context and are based on an interactive paradigm.

Chapter 3

Approach

In this section, we describe the model architectures, as well as the fine-tuning and the configurations used.

3.1 Model architectures

We chose to base our approach on an already pre-trained model to bypass the need for large amounts of training data, and base our approach on SOTA models that would increase the likelihood of better results. We considered multiple models that were pre-trained on source code data such as CodeT5 [35], CodeGPT-2 [26], PLBART [2], and CodeBERT [9]. While all of the models including CodeT5 are available on HuggingFace for further usage we chose CodeT5 with the simple reasoning that they outperformed all of the aforementioned models.

We shall now proceed to briefly describe the architecture of the CodeT5 model as well as other relevant information required to understand the architecture. Pre-trained language models such as BERT, GPT, and T5 have greatly improved the performance for a wide range of NLP tasks. The underlying architecture for all of these models is the Transformer architecture (Vaswani et al.). The prominent rise in performance of these models can be attributed to the architecture that enables significantly more parallelization and relies heavily on the attention mechanism to associate input and output dependencies. The Transformer architecture leverages multiple attention mechanisms referred to as ”Multi-head attention”, allowing the architecture to attend to different representation subspaces at different positions. Additionally, allowing the decoder in an encoder-decoder model to reference the input sequence in full. Equally important, the parallelization allows for a notable reduction of training time. Thus, permitting models to train on even larger datasets [33]. The Transformer-styled models are typically pre-trained using a combination of the following pre-training objectives: (1) denoising objectives, such as masked language modeling (MLM), usually manipulate the input sequence and train the model to predict the correct sequence. In the case of MLM, a percentage of the input sequence is masked with a special symbol and then the model is trained to predict these masked tokens, and (2) next sentence prediction (NSP) that provides the model with two sentences and the model is to determine whether or not the sentence is the subsequent one [8].

The CodeT5 model extends the pre-training objectives by employing a novel identifier-aware denoising pre-training objective called masked identifier prediction (MIP) and an identifier tagging (IT) task. In detail the MIP objective masks all identifiers (*e.g.*, method names) in such a manner that each unique identifier is assigned a unique token for all occurrences. Whereas the objective of the IT task is similar to sequence labeling, namely to determine whether or not part of the sequence is an identifier. Thus, the pre-training objectives of the CodeT5 model are comprised

of these two objectives in conjunction with MLM. Moreover, the loss of each of these pre-training objectives is alternately optimized with an equal probability. Consequently, the CodeT5 model is able to capture more of the code syntax and semantics [35].

The usage of the transformers library[1][37] in combination with the PyTorch library[2] [23] allowed us to reuse the CodeT5 model. Moreover, easily extend the model with our own approach. Due to our small dataset (see Section 3.2.1) we decided to adapt the training process to use a 3-fold cross-validation (CV) approach. We implemented a training procedure that allows us to train, test, and validate our models using the produced folds. Additionally, we persisted our folds; thus, allowing us to consistently reproduce our results. Consequently, allowing us to verify that our model is able to generalize. Besides, we also use a linear scheduler to decrease the learning rate of our models in order to hopefully arrive at better weights (see also Section 4.1.2). We implemented two models for our experiments: the first model is based on the CodeT5 model from Wang et al. and the second is an extension of the first model that tries to improve the context information with a bidirectional LSTM (BiLSTM) layer (Hochreiter and Schmidhuber [14]). Using BiLSTMs to improve models is nothing new, but they have proven to be useful for long sequences since they retain long-term information by propagating only the relevant information back [14, 15, 20, 28]. We will call the models' context-enhanced generation (CEG) and CEG_{BiLSTM}, respectively.

In the following, we characterize the specifics of the two models we implemented.

3.1.1 Context-enhanced generation

Our first model is based on the intuition that additional context information benefits the generation process since the model learns the representation of already existing methods, imports, and member variables. Thus, enabling the model to reuse previous methods, imported application programming interfaces (APIs), and member variables. We hold the assumption that the additional information helps the model to deal with the barriers faced by learners. Specifically, with the integration, mapping, and modification barriers, as mentioned previously (see Chapter 1), since the provided information could enable the model to directly adapt the prediction of the intent to the context. Hence, producing integrated, mapped, and modified code.

In order to deal with contexts of variable sizes, we defined a custom context workflow that only extracts the necessary information (see also Section 3.2.1). We considered what information an actual programmer would have during the process of writing code and based our definitions on that assumption; *i.e.*, the method signatures, the member variables, the code preceding the comment, and the name of the current method. Accordingly, our model takes the aforementioned information in addition to the NL intent and the target line for training purposes. This results in a tuple of string vectors $(\overrightarrow{nli}, \overrightarrow{mth}, \overrightarrow{cdb}, \overrightarrow{sig}, \overrightarrow{imp}, \overrightarrow{fld}, \overrightarrow{trg})$ as an input. Thus, the model has the same information as an actual programmer would. To deal with this information we added a new module to prepare our data in such a way that the model is able to use our input in accordance with the CodeT5 model (*BERT-style concatenation* [35]). Although the inputs are concatenated they are separated by a $[SEP]$ token. We assume that the model will learn to attend each of these parts and is able to decide which of the inputs is more important. The equations eqs. (3.1) to (3.13) formally define the inputs and output of our model.

We define the NL intent as

$$\overrightarrow{nli} = (nli_0, nli_1, \ldots, nli_i) \tag{3.1}$$

[1]transformers version 4.12.5 – `https://huggingface.co/`
[2]PyTorch version 1.10.0+cu113 – `https://pytorch.org/`

The method signature of the current method is denoted by

$$\overrightarrow{mth} = (mth_0, mth_1, \ldots, mth_j) \tag{3.2}$$

The code preceding the comment is denoted by

$$\overrightarrow{cdb} = (cdb_0, cdb_1, \ldots, cdb_k) \tag{3.3}$$

The method signatures of all other methods are denoted by

$$\overrightarrow{sig} = (sig_0, sig_1, \ldots, sig_l) \tag{3.4}$$

The imports defined in the java file are denoted by

$$\overrightarrow{imp} = (imp_0, imp_1, \ldots, imp_m) \tag{3.5}$$

The global members/fields

$$\overrightarrow{fld} = (fld_0, fld_1, \ldots, fld_n) \tag{3.6}$$

The next line of code after the comment

$$\overrightarrow{trg} = (trg_0, trg_1, \ldots, trg_o) \tag{3.7}$$

In order to truncate the inputs to the maximum allowed length we define the following:
Let Z be the set of $\{i, j, k, l, m, n\}$, which denote the respective indexation of the input vectors and all elements of Z are truncated to the respective maximum input length.

$$\begin{aligned} \forall e \in Z : e \in \mathbb{N} \\ 0 \leq Z \leq max_{inp} \end{aligned} \tag{3.8}$$

Whereas o is truncated to the maximum output length max_{trg} (see Section 4.1.2 for more details on the max length).

$$0 \leq o \leq max_{trg} \tag{3.9}$$

The input parts are concatenated by a pre-defined token ($[SEP]$):

$$\hat{\sum} \text{ is used to denote the aggregation of concatenations} \tag{3.10}$$

$$\oplus \text{ denotes a single concatenation} \tag{3.11}$$

These definitions allow us to define the input for our model as the concatenation of the NL intent and all context vectors, truncated to the maximum input size allowed by the model, as shown in Equation 3.12.

$$\begin{aligned} x &= \hat{\sum}_{i=1}^{6} Z_i \oplus [SEP] \\ z &= tokenize(x, max_{inp}) \end{aligned} \tag{3.12}$$

Similarly, our output is defined by

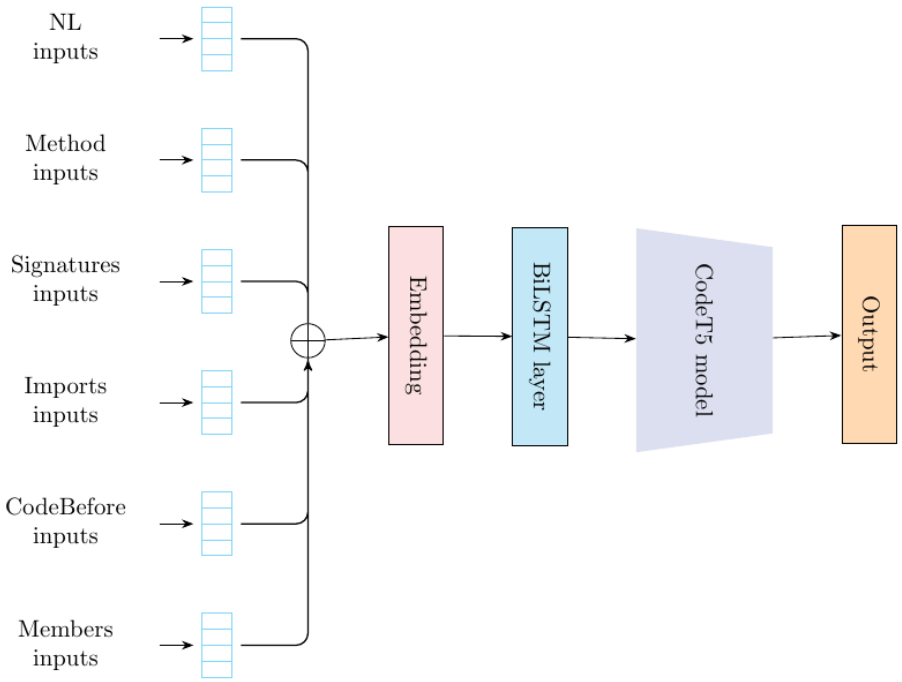
$$t = detokenize(trg, max_{trg}) \tag{3.13}$$

3.1.2 Context-enhanced generation BiLSTM

While our initial approach does consider the context we assume that a better context representation would benefit the model. We initially implemented the standard approach of truncating the input to the expected size (as demonstrated in Section 3.1.1). Granted that this approach is a commonplace method to deal with large inputs and does produce results, we decided to extend the model with an additional compression layer. Hence, our second model tries to enhance the input by compressing it with an additional BiLSTM layer before passing the input to the transformer model as shown in Figure 1.

In order to add the BiLSTM layer, we defined a new PyTorch module that embeds the original CodeT5 model after the newly added BiLSTM layer. This allows us to change the model efficiently and we can gather the necessary information regarding the dimensions from the CodeT5 configuration. Thus, configuring the BiLSTM layer to produce output with the expected input dimension of the model. In addition to the new PyTorch module, we also needed to adjust the transformers library, as the current implementation of the "generate" function only allows proper generation with input ids instead of embeddings. This adjustment was necessary, as we generate the embeddings from the *BiLSTM* layer and pass it to the original model. However, the "generate" method requires input ids and the "forward" method of the model throws an exception if both ids and embeddings are passed. To circumvent this problem we added a simple check for this case and remove the input ids within the forward call of the original *t5* model. Thus, allowing us to pass both input ids and embeddings to the "generate" function while still complying with the requirements of the "forward" function.

Figure 1: High-level CEG_{BiLSTM} architecture



We suspect that the CEG_{BiLSTM} is able to compress the large context information better since the BiLSTM layer should learn more compact representations of the context. In other words, achieve better results than the CEG model.

While our approach is model-agnostic we chose a pre-trained model, expecting better results than training from scratch with randomly initialized weights. This decision is further supported by recent surveys and papers [12, 13, 21].

3.2 Dataset

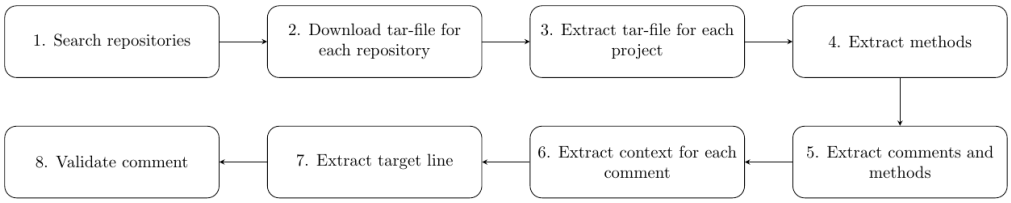
This section describes how we obtained and processed the data that we use for our approach. Furthermore, it provides statistical information's regarding the produced dataset.

3.2.1 Data mining process

While there are datasets containing NL-code pairs and some even containing context information such as CONCODE [15], CoNaLa [42], WikiSQL [45], and CodeXGLUE [19], none of these were in accordance with our goal of generating single-line code. Moreover, none of the aforementioned datasets are tailored to single-line source code targets and provide the full context we require for our custom context workflow. Therefore, we decided to produce our own dataset containing the context information, the NL intent, and a single next-line of source code as our target.

We downloaded 19'397 repositories with 6'359'942 Java files from GitHub (GitHub). The data mining process is split into (1) gathering Java repositories, (2) method extraction, (3) comment extraction, (4) comment validation, and (5) target extraction and was executed over a course of 12 days and 15 hours. The data mining pipeline is illustrated in the following figure:

Figure 2: Data mining pipeline



The data mining pipeline yields a JSON file where each line correlates to a correct JSON object (illustrated in Listing 3.2) created with the jsonlines[3] library.

Repository mining

We downloaded the Java repositories using the GitHub-API. We decided to limit our approach to one programming language, namely Java. Furthermore, we decided to limit our search to projects with a minimum of 90 GitHub stars and sort the resulting repositories by the number of stars in descending order. This decision is based on the paper from Borges and Valente, who state that the number of stars is a key metric regarding the evolution of GitHub projects, however, they also state that the number of stars may be concentrated in a short time period, due to marketing

[3]jsonlines version 3.0.0 – `https://jsonlines.org/`

and advertisement, and consequently may not follow solid software engineering principles and practices [7]. In summary, we filter by

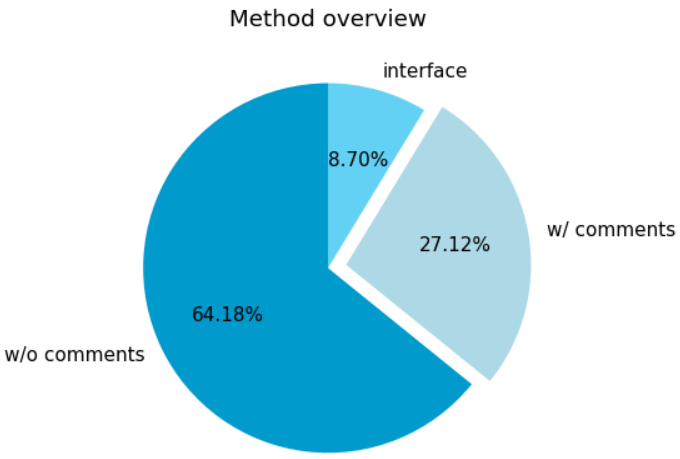
- the programming language Java, and
- projects that have more than 90 GitHub stars.

Method, comment, and target extraction

We parsed all of the downloaded Java files using ANother Tool for Language Recognition (ANTLR4)[4] resulting in 1'185'549 extracted methods. Out of those we removed the methods that did not contain any comments resulting in 321'486 methods that contain "in-line comments". Furthermore, a total of 3'518 interfaces were omitted from the extraction, due to there being no method body. However, this is only true for projects prior to Java 8 as it is since possible to add *default* implementations of interface methods. The default methods in an interface are fully implemented methods and not just signatures without a method body [22].

In a second step, we iterated through all of the "in-line" comments in order to extract the next line following the comment as the target code for the training. The extraction process yielded 216'280 viable comments, that are used to fine-tune the model. The distribution of methods with (w/) and without (w/o) comments is illustrated in Figure 3.

Figure 3: Method/comment distribution



Context extraction

We implemented a custom workflow extraction process that extracts each of the following parts: (1) the method signatures within the class, (2) the name of the current method, (3) the imports used in the class, (4) member variables, (5) and the code preceding the current comment inside of the current method, allowing us to handle each of the respective parts separately. This enables us

[4] ANTLR4 version 4.9.2 – `https://www.antlr.org/`

to extract the context information without *boilerplate code*. For example, if we are considering the context of an inline comment, we can ignore the method body of other methods. As we only need information pertaining to the internals of the current method. We hope that our model learns to extract the relevant information from each part of the context. We then concatenate all of the parts as described in Section 3.1 to get a *BERT-style* concatenated input [8] following the CodeT5 model [35].

In order to give a detailed illustration of the context extraction process we will use the following example: Let the current inline comment be *line 24* in Listing 3.1. Using ANTLR4 we extract the imports from the file, which yields *lines 3-6*. In a similar fashion also using ANTLR4 we extract member variables as well as all method signatures excluding the current method, resulting in *line 9* and *lines 11 and 15*, respectively. The current method signatures *line 19* is handled separately, as we assume that the method name could reveal information regarding the intent of the comment. With the positional information provided by ANTLR4, we extract the code preceding the comment by selecting the lines between the start of the method body and the comment omitting all other comments contained within this section *lines 21 through to 23*, which we call code before.

Listing 3.1: "Context extraction"

```
1  package com.google.gson;
2
3  import com.google.gson.internal.GsonPreconditions;
4  import java.math.BigDecimal;
5  import java.math.BigInteger;
6  import com.google.gson.internal.LazilyParsedNumber;
7
8  public final class JsonPrimitive extends JsonElement {
9   private final Object value;
10  ...
11  public JsonPrimitive(String string) {
12    value = GsonPreconditions.checkNotNull(string);
13  }
14  @Override
15  public char getAsCharacter() {
16    return getAsString().charAt(0);
17  }
18  @Override
19  public boolean getAsBoolean() {
20    // check type
21    if (isBoolean()) {
22      return ((Boolean) value).booleanValue();
23    }
24    // Check to see if the value as a String is "true" in any case.
25    return Boolean.parseBoolean(getAsString());
26  }
27 ...
```

Imports

Fields

Signature

Signature

Method name

Code before

The extraction process yields the following example JSON object:

Listing 3.2: "Example JSON"

```
{
  "comment": "// Check to see if the value as a String is "true" in any case.",
  "methodName": "public boolean getAsBoolean()",
  "codeBefore": "if (isBoolean()) { return ((Boolean) value).booleanValue(); }",
  "signatures": "public JsonPrimitive(String string) public char getAsCharacter()",
  "imports": "import com.google.gson.internal.GsonPreconditions import java.math.
    BigDecimal ...",
  "fields": "Object value",
  "target": "return Boolean.parseBoolean(getAsString());",
  "file": ".../JsonPrimitve.java"
}
```

This extraction process should simulate the behavior of writing code and provide all possible information w.r.t. the current NL intent. At the time of writing a comment, the programmer would not know any information following the comment. Therefore, omitting everything after the comment. However, we include all other method signatures as it is possible to add a new method anywhere within a class.

3.3 Exploratory data analysis

In order to better understand the mined data, we performed some simple EDA on the data. We created boxplots for each of our input variables in order to analyze the length distribution of each variable. This analysis showed us how much of our context data would be lost regarding the models' maximum input size (see Table 3). For the following figures, note that the box represents the interquartile range (IQR), the line within the box represents the median, and all points outside the whiskers are considered outlier. Also, note the logarithmic scale on the x-axis.

Figure 4: Context variable boxplots

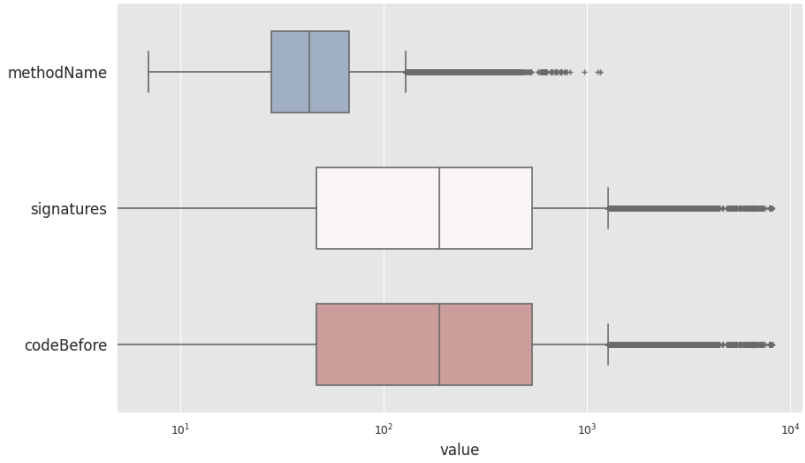


Figure 5: Context variable boxplots

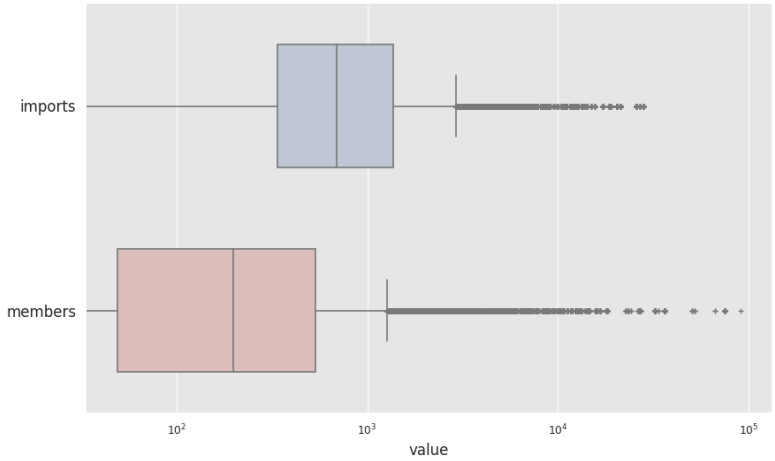
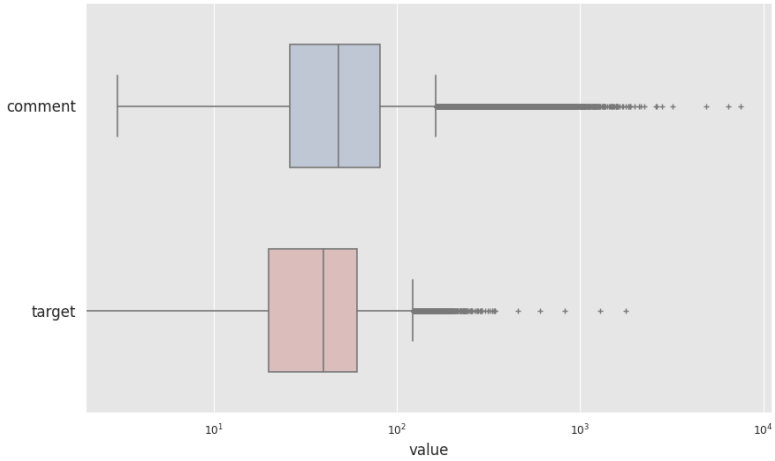


Figure 6: Comment & target variable boxplots



Looking at the distribution we can see that majority of the targets, comments, and method names have lengths of less than 100 characters. However, the outliers reach up to a length of 1773, 7502, and 1164, respectively. Whereas the majority of the signatures, codeBefore, members, and imports are larger than 100 characters with an average length of 474, 729, 533, and 1158, respectively. This provides us with valuable information regarding the maximum length for our model (w.r.t. our memory limitations see Section 4.1.2). However, as the pre-trained *CodeT5* tokenizer uses byte pair encoding (BPE) [29] it is difficult to determine the effective number of tokens the concatenated sentence represents. For example, when considering our toy example

from Listing 3.2 we receive 93 tokens without padding for our concatenated input with a length of 346 characters.

N-Gram analysis

In order to gain some insight into what kind of data examples are in our data, we ran an n-gram analysis shown in Figure 7, Figure 8, and Figure 9. From the bi-grams, we can deduce that the most common n-grams from the comment data are in English. The import bi-grams can be used to see the most common libraries in our dataset. The target bi-grams reveal that they pertain either to structural information (*e.g., for, if, else*), exceptions, or return values.

Figure 7: N-gram analysis part 1

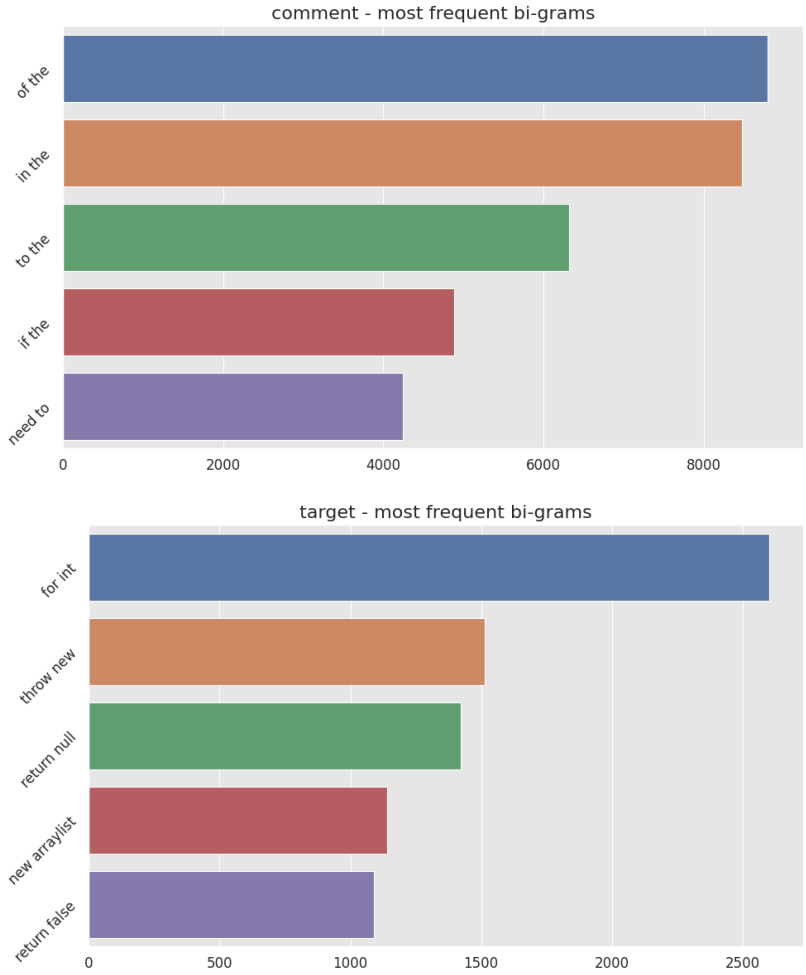


Figure 8: N-gram analysis part 2

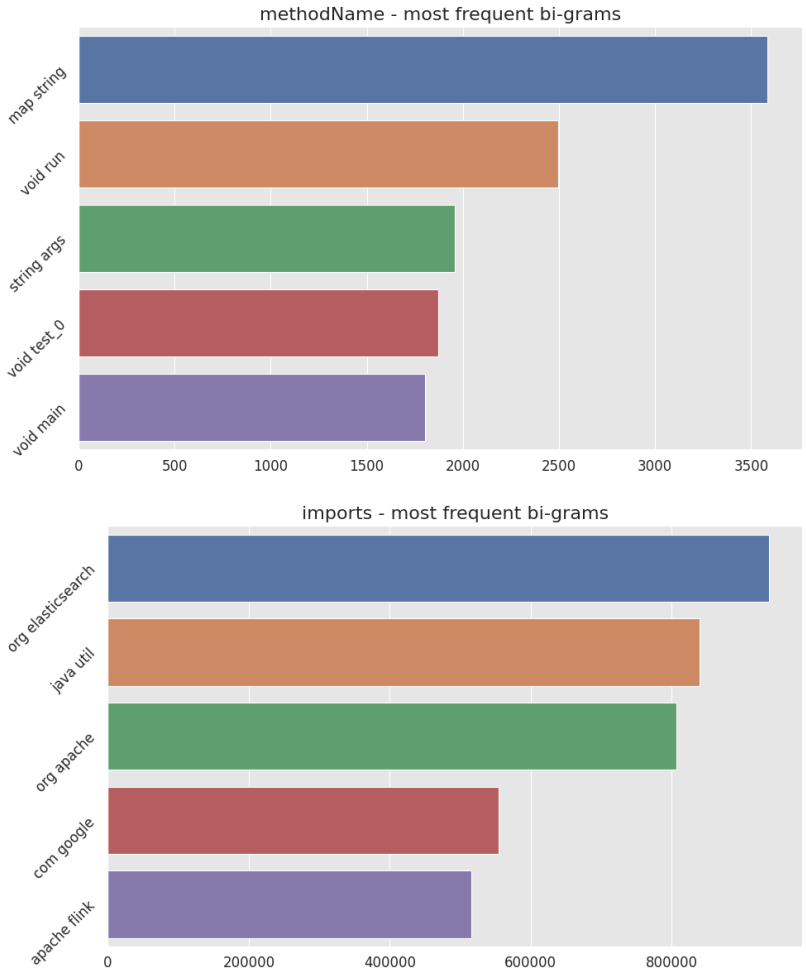
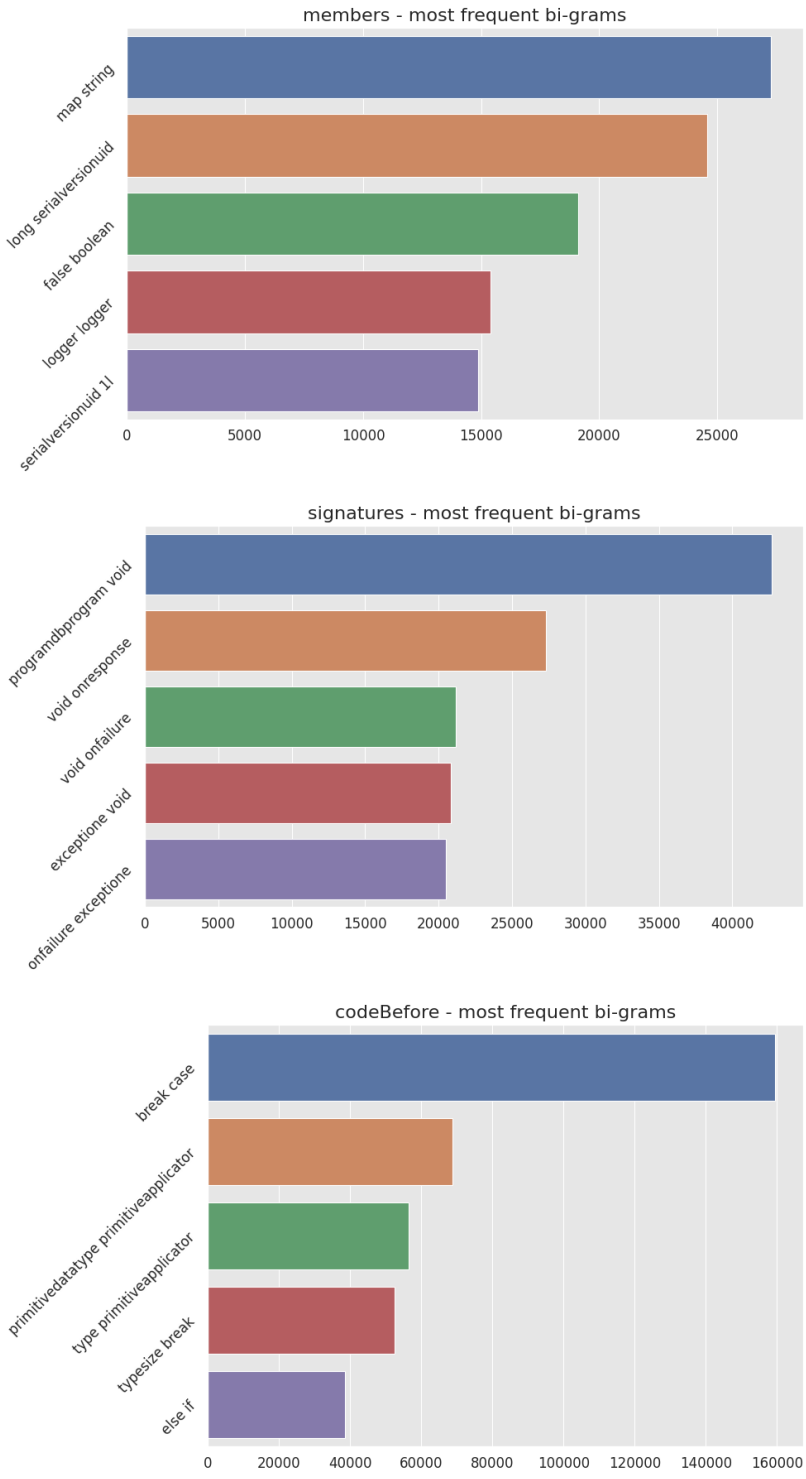


Figure 9: N-gram analysis part 3



3.4 Data mining statistics

The following table provides an overview of the data yielded from the data mining process:

TABLE 1: Data statistics

Name	Data	
	AVG	Min/Max
Comment lengths	69.188	3/7'502
Target lengths	42.053	0/1'773
Signature lengths	473.702	1/8'092
Field lengths	532.982	0/90641
Method name lengths	54.983	7/1'164
Import lengths	1'157.938	0/28'062
Code before lengths	729.428	0/106'354
Imports per class	17.59	0/434
Fields per class	6.38	0/1'284
Methods per class	2.86	1/182
Comments per method	2.02	0/375

Chapter 4

Experimental design

This chapter aims to address how we intend to answer our research questions and is closely tied to Chapter 5. Furthermore, this chapter describes how we trained our model, the training dataset, and threats to the validity of our approach.

4.1 Experimental design

To examine the applicability and performance of our approach, we construct several experiments while keeping the research questions in mind.

> **RQ1** *Does our CEG_{BiLSTM} model compress the context information more efficiently than the CEG model?*

In order to assess if our CEG_{BiLSTM} model benefits from the BiLSTM layer and whether or not it outperforms our initial CEG model, we intend to compare their respective BLEU and CodeBLEU scores.

> **RQ2** *How does our approach compare to the current SOTA approach?*

While there are many ways to compare approaches (*e.g.*, ROUGE, Recall) we chose to use our defined metrics since these metrics are commonly used to measure the performance of code generation tasks. Therefore, we intend to use the metrics to compare our model to the current SOTA approach. As stated previously we chose to compare our models to the NLGP model from Heyman et al., since their approach is the most similar to our approach. Furthermore, the technologies they used in addition to the contextualization are comparable to ours, with the exception of the underlying model used and our custom context workflow. Specifically, we chose the best performing $NLGP_{natural}$ model for a fair comparison [13]. Additionally, these metrics allow us to verify if the models are overfitting or not.

> **RQ3** *Do our models learn strong context representation from our custom context workflow?*

While it is crucial to assess the learning capabilities of our model and see if our models converge, it is equally important to further analyze the usage of the context. While that may be true, these metrics do not allow us to analyze how the context is used. Therefore, we intend to use a sample of predictions to get a preliminary intuition regarding the applicability of our approach by manually comparing the predictions. Hence, they should provide us with a definite answer regarding our single-line goal as well as an indication of whether or not we are able to learn strong context representations. Even though such a comparison is time-consuming and not exhaustive

we assume that the example comparison provides us with insight into the performance of our models.

RQ4 *Are we able to produce single-line source code from the given inputs?*

The metrics are based on provided target sentences they also indicate if we are able to produce single-line source code or not. While that may be true, we chose to verify the prediction of single-line source code by analyzing the generated predictions. Thus, allowing us to review the output in detail.

In summary, we intend to assess our questions as follows:

- Using the defined metrics we aim to answer **RQ1** and **RQ2** by comparing the respective results of our models
- We try to answer **RQ3** and **RQ4** using an exploratory example analysis.

In the following, we describe the training method, briefly introduce training-specific details regarding the dataset, metrics, and introduce possible threats to the validity of our work.

4.1.1 Training dataset and metrics

For all of our experiments and models, we chose to use our own dataset (see Section 3.2.1). The dataset was prepared from more than 19 thousand repositories downloaded from GitHub that were filtered by programming language and sorted by the number of stars. We parsed all the files using ANTLR4 to extract comments contained within methods. Thus, only extracting ”in-line” comments.

To evaluate our models, we apply three-fold cross-validation to our dataset by splitting the dataset into three equal folds and using two for training and one for testing. After the initial split, we further split the training data into a training and validation set, 90% and 10%, respectively. Resulting in the following folds:

TABLE 2: Dataset

Fold	Training samples	Validation samples	Test samples	Total
1	129’767	14’419	72’094	216’280
2	129’768	14’419	72’093	
3	129’768	14’419	72’093	

We use BLEU as our main metric as it is used in most of the SOTA papers and allows for easier comparison. However, Tran et al. and Ren et al. showed that BLEU does not reflect the semantic accuracy of code, which is why in addition to BLEU we also use CodeBLEU as a more informative metric to evaluate the models [27, 32].

CodeBLEU

We will briefly introduce the difference between BLEU and CodeBLEU: BLEU measures the percentage of n-gram overlaps between the reference and the candidate sentence. Whereas CodeBLEU additionally considers syntactic abstract syntax tree (AST) matching, semantic dataflow matching, and a weighted BLEU score [27].

4.1.2 Model configurations

We build two contextualized models based on the CodeT5 model [35]. However, due to memory limitations, we adjusted the original hyperparameters that we took from the CodeT5 GitHub repository [5] from their code generation task. We reduced the batch size and target length to 4 and 100, respectively.

We trained the model on a single NVIDIA Tesla T4 GPU with 16 GB of memory, with a three-fold CV approach.

TABLE 3: Pre-training hyperparameters vs. CodeT5

Parameter	CEG	CEG_{BiLSTM}	$CodeT5$
Optimizer	AdamW	AdamW	AdamW
Learning rate	0.0001	0.0001	0.0001
Learning rate warm up steps	1'000	1'000	1'000
Early stopping	True	True	True
Early stopping patience	3	3	3
Beam size	10	10	10
Max input length	300	300	300
Batch size	**4**	**4**	16
Max target length	**100**	**100**	150
Epochs	**3**	**3**	30

The original hyperparameters were taken from the CodeT5 (Wang et al. [35]) GitHub repository[5].
The changed hyperparameters compared to the original CodeT5 parameters are marked in bold.

4.1.3 Training

We trained the models with the following criteria (1) the BLEU score does not increase with a patience of 3, and (2) the loss does not decrease with a patience of 3, or (3) the model converges. This resulted in a total of three epochs per fold. The total training time per contextualized model is approximately nine days each. During the training process, we collected information on the loss, BLEU, and CodeBLEU scores. The respective results for the training and validation data are illustrated in Figure 10 and Figure 11. Even though the model converges after a few epochs the validation and training losses do not cross each other. Thus, we assume that no *overfitting* takes place. In both models, the CodeBLEU scores are higher than the BLEU scores and both models greatly improve through training in comparison with the initial scores (epoch zero) as seen in Figure 11.

[5]CodeT5 GitHub repository: `https://github.com/salesforce/CodeT5`

Figure 10: Training & validation loss

(a) CEG

(b) CEG_{BiLSTM}

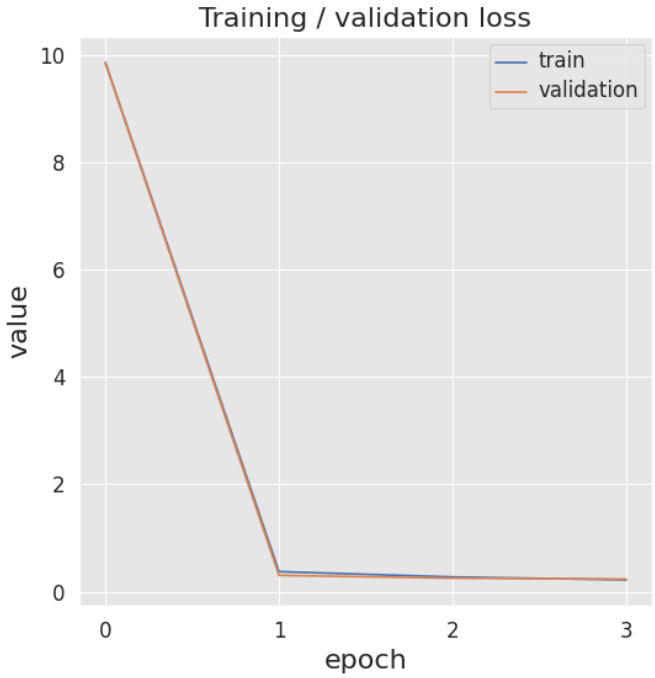
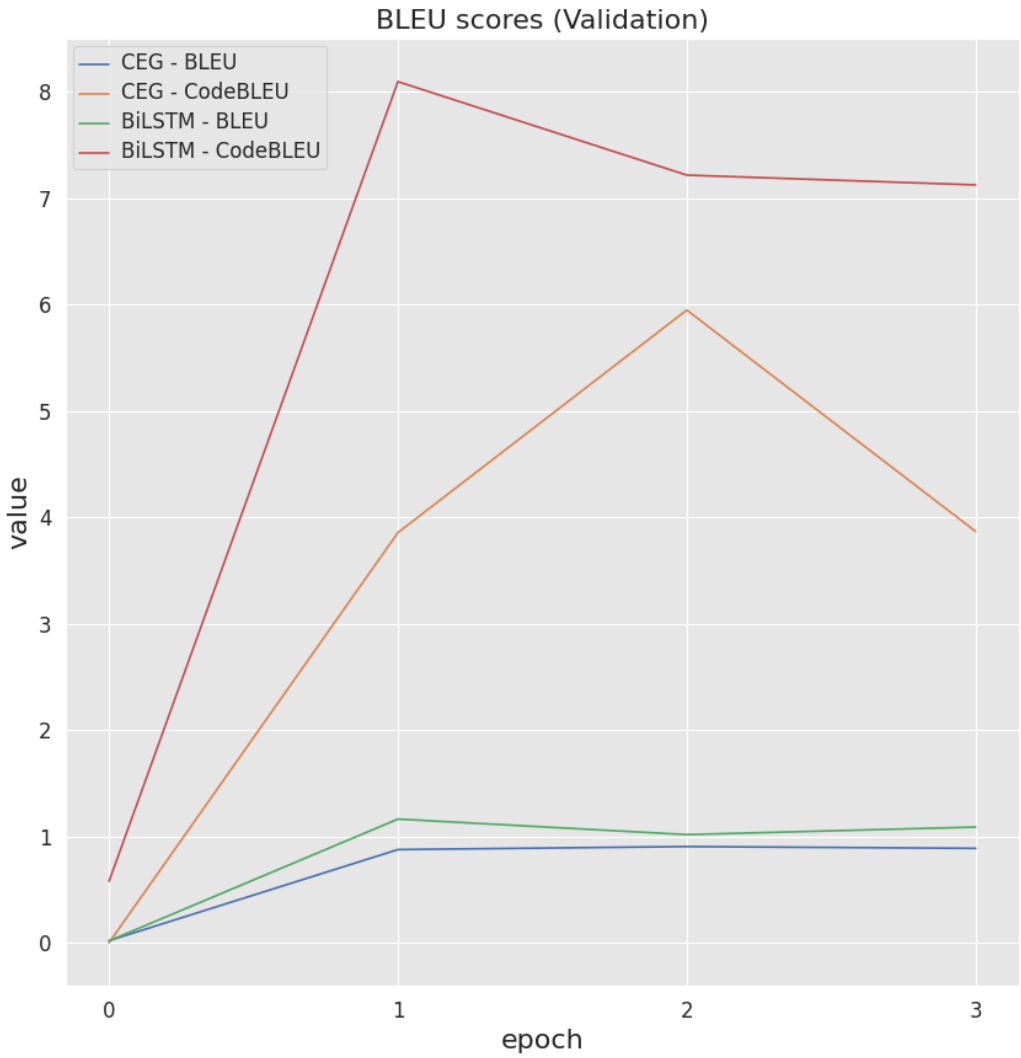


Figure 11: BLEU & CodeBLEU scores



4.2 Threats to validity

Internal validity. The most significant limitation to our design emerges from the dataset we created. We assume that our data contains a considerable amount of noise and this crosses over into the evaluation of our models as the targets, as well as, the input may contain noise. Thus, the model may learn too much from the noise and this potentially impacts the performance of the model. We also note that the size of our dataset is rather small (see Section 3.2.1), which could impact the generalization of the model, however, we mitigated some of the potential impacts by using a 3-fold CV approach. Furthermore, our data cleaning was kept to a minimum in order to see how our model deals with noisy data. Consequently, it is not guaranteed that all of our comments are written in English nor if the target line actually makes sense.

Equally important, is the fact that we manually evaluated the generated predictions for our exploratory example analysis. Our selection and comparative criteria may not represent the actual performance of the models. However, we tried our best to be objective and provide the reader with the specific examples used in the evaluation.

External validity. The first external threat is the choice of the programming language used to train our model. It is thus unclear if our approach can be generalized to other languages and whether or not a similar performance can be achieved. The reason for choosing Java can be summarized as follows. First, Java is and has been, consistently so, one of the most popular programming languages. Secondly, the familiarity of the language allowed us to be more objective with regard to our manual evaluation of the examples. In the future, we intend to verify the performance of our model with other programming languages (such as JavaScript, Python, C++).

The second external threat is the limited comparison to other SOTA approaches. Due to the nature of this emerging field, the comparable approaches were limited to a small number. However, we hope that future works are able to use the current approaches as a basis of comparison.

Chapter 5

Results

This chapter presents our results from the experiments described in Chapter 4 and aims to answer our questions proposed in the experimental design.

5.1 Comparative results using the metrics (RQ1 and RQ2)

We now present the results of the experiments involving the comparison of the models using the defined metrics (BLEU and CodeBLEU). To achieve a fair comparison of the model performance, we trained all the models for an equal amount of time (185.62 hours) and ran the SOTA model from Heyman et al. [13] on the same hardware as our models (converges after two epochs see Appendix A). The result comparison for the *test set* using the metrics BLEU and CodeBLEU of the models is shown in Table 4.

TABLE 4: Model comparison

Model	**Metrics**	
	BLEU score	**CodeBLEU score**
CEG	0.849	5.798
CEG_{BiLSTM}	1.303	**8.726**
$NLGP_{natural}$	**2.070**	5.083

Looking at the results from Table 4 we can clearly see that the CEG_{BiLSTM} achieved an improvement over the CEG model of $\pm$ 0.454 and $\pm$ 2.928 points in both metrics BLEU and CodeBLEU, respectively. Furthermore, the CodeBLEU metric considers AST matching and semantic dataflow matching in addition to the BLEU score. Consequently, we can view the higher relative difference between the two models as a further indication of better contextualization. Therefore, confirming our initial assumption regarding the retention of context information of the CEG_{BiLSTM} model.

> **RQ1 - In summary:** The CEG_{BiLSTM} model is able to improve the CEG model and retains more context information.

We now refer to the results pertaining to the SOTA model (*viz.*, $NLGP_{natural}$) in comparison to our best model. Comparing the results shown in Table 4 we see that our best model outperformed the $NLGP_{natural}$ model by 3.643 points in the CodeBLEU metric. Contrarily, the $NLGP_{natural}$ model achieved a higher BLEU score of 2.070 with an increase of 0.767 points. Still, we conjecture that a more detailed context collection improves the contextualized code generation process with respect to the CodeBLEU metric. However, looking at both metrics we have to concede that the results are inconclusive regarding the performance of our model in comparison with the current SOTA approach. Consequently, the inconclusive results do not provide us with a definite answer for our research question, albeit we do argue that our approach is not inferior to other approaches w.r.t. CodeBLEU and provides further research avenues regarding the use of context.

> **RQ2 – In summary:** Our best model achieves a higher CodeBLEU score, while the $NLGP_{natural}$ model achieves a higher BLEU score. Thus, the results are inconclusive.

Thus far, we have verified that our model benefits from the BiLSTM layer and achieved better results than without the additional layer. But, there is no conclusive answer regarding the performance comparison with the SOTA approach.

5.2 Example exploration and analysis (RQ3 and RQ4)

We now present the results from our exploratory example analysis with the aim of providing a more elaborate comparison. Analyzing the example predictions produced by the three models (*viz.*, CEG, CEG_{BiLSTM}, and $NLGP_{natural}$) shown in tables 5 to 8 we can see that the CEG model produces the most accurate predictions. Considering the results of the CEG_{BiLSTM} model a pattern emerges, where the model often predicts *for loops*. This behavior could be attributed to the BiLSTM layer, which assumably learns representations that favor *loops*. Another reason for this pattern could be that the underlying pre-trained model is not able to discern the different parts of the new embeddings created by the BiLSTM layer. In comparison, the $NLGP_{natural}$ models' predictions worsen as the context size increases. Therefore, we can deduce that our custom context extraction workflow results in a more compact context; thus, alleviating issues regarding large contextual input. Even though we need to place special considerations w.r.t. to the prediction of our CEG model in the case of example 4 (Table 8). We note that the reference does not match the prediction, however, the model produces code that overlaps with the intent. This mismatch could be linked to a noisy dataset and further consideration needs to be placed on data cleaning.

> **RQ3 – In summary:** Our custom context workflow allows the models to deal with substantial context inputs and provide accurate predictions. Hence, the models are able to learn strong context representations.

The examples provide us with a non-exhaustive overview of the predictions our models produce, but they clearly show that both of the models consistently predict single-line output, which is in alignment with our goals. Even though the examples are by no means exhaustive, we previously mentioned that the metrics compare the prediction to single-line references that further support our claim that our models are producing single-line source code.

> **RQ4 – In summary:** Our models are able to produce single-line source code from the given input as demonstrated by the exploratory example analysis.

Listing 5.1: "Example 1 - Context"

```
import java.util.HashMap;
import java.util.Map;
import com.alibaba.excel.metadata.data.ImageData.ImageType;
public class FileTypeUtils {
  private static final char[] DIGITS = {'0','1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e','f'};
  private static final int IMAGE_TYPE_MARK_LENGTH = 28;
  private static final Map<String, ImageType> FILE_TYPE_MAP;
  public static ImageType defaultImageType = ImageType.PICTURE_TYPE_PNG;
  static {FILE_TYPE_MAP = new HashMap<>();
  FILE_TYPE_MAP.put("ffd8ff", ImageType.PICTURE_TYPE_JPEG);
  FILE_TYPE_MAP.put("89504e47", ImageType.PICTURE_TYPE_PNG);
}
public static int getImageTypeFormat(byte[] image) {
  ImageType imageType = getImageType(image);
  if (imageType != null) {
    return imageType.getValue();
  }
  return defaultImageType.getValue();
}
public static ImageType getImageType(byte[] image) {
  if (image == null || image.length <= IMAGE_TYPE_MARK_LENGTH) {
    return null;
  }
  byte[] typeMarkByte = new byte[IMAGE_TYPE_MARK_LENGTH];
  System.arraycopy(image, 0, typeMarkByte, 0, IMAGE_TYPE_MARK_LENGTH);
  return FILE_TYPE_MAP.get(encodeHexStr(typeMarkByte));
}
private static String encodeHexStr(byte[] data) {
  final int len = data.length;final char[] out = new char[len << 1];
```

TABLE 5: Example 1 - Comparison

Intent		
// two characters from the hex value.		
Predictions		
CEG	**BiLSTM**	**NLGP**
for (int i = 0; i < len; i++) {	for (int i = 0; i < data.length; i++) {	\n
Reference		
for (int i = 0, j = 0; i < len; i++) {		

Listing 5.2: "Example 2 - Context"

```
package smoketest.websocket.undertow.snake;
```

```
import java.awt.Color;
import java.util.Random;
public final class SnakeUtils {
  public static final int PLAYFIELD_WIDTH = 640;
  public static final int PLAYFIELD_HEIGHT = 480;
  public static final int GRID_SIZE = 10;
  private static final Random random = new Random();
  private SnakeUtils() {}
  public static String getRandomHexColor() {
      float hue = random.nextFloat();
```

TABLE 6: Example 2 - Comparison

Intent		
// sat between 0.1 and 0.3		
Predictions		
CEG	**BiLSTM**	**NLGP**
float saturation = (random.nextInt(2000) + 1000) / 10000f;	for (int i = 0; i < 4; i++) {	\n
Reference		
float saturation = (random.nextInt(2000) + 1000) / 10000f;		

Listing 5.3: "Example 3 - Context"

```
package smoketest.bootstrapregistry.app;
import smoketest.bootstrapregistry.external.svn.SubversionBootstrap;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplicationpublic class SampleBootstrapRegistryApplication {
  public static void main(String[] args) {
```

TABLE 7: Example 3 - Comparison

Intent		
// This example shows how a Bootstrapper can be used to register a custom SubversionClient that still has access to data provided in the application.properties file		
Predictions		
CEG	**BiLSTM**	**NLGP**
SpringApplication application = new SpringApplication(Application.class);	SpringApplication .run(Application.class,args);	SpringApplication .run(SampleBootstrapRegistryApplication.class, args);
Reference		
SpringApplication application = new SpringApplication(SampleBootstrapRegistryApplication.class);		

Listing 5.4: "Example 4 - Context"

```
package com.google.gson;
import com.google.gson.internal.GsonPreconditions;
import java.math.BigDecimal;
import java.math.BigInteger;
import com.google.gson.internal.LazilyParsedNumber;
public final class JsonPrimitive extends JsonElement {
 private final Object value;
 public JsonPrimitive(Boolean bool) {
  value = GsonPreconditions.checkNotNull(bool);
 }
 public JsonPrimitive(Number number) {
  value = GsonPreconditions.checkNotNull(number);
 }
 public JsonPrimitive(String string) {
  value = GsonPreconditions.checkNotNull(string);
 }
 public JsonPrimitive(Character c) {
  value = GsonPreconditions.checkNotNull(c).toString();
 }
 @Override
 public JsonPrimitive deepCopy() {
  return this;
 }
 public boolean isBoolean() {
  return value instanceof Boolean;
 }
 @Override
 public boolean getAsBoolean() {
  if (isBoolean()) {
   return ((Boolean) value).booleanValue();
  }
  return Boolean.parseBoolean(getAsString());
 }
 public boolean isNumber() {
  return value instanceof Number;
 }
 @Override
 public Number getAsNumber() {
  return value instanceof String ? new LazilyParsedNumber((String) value) : (Number) value;
 }
 public boolean isString() {
  return value instanceof String;
 }
 @Override
 public String getAsString() {
  if (isNumber()) {
   return getAsNumber().toString();
  } else if (isBoolean()) {
   return ((Boolean) value).toString();
  } else {
   return (String) value;
  }
 }
 @Override
```

```
public double getAsDouble() {
  return isNumber() ? getAsNumber().doubleValue() : Double.parseDouble(getAsString());
}
@Override
public BigDecimal getAsBigDecimal() {
  return value instanceof BigDecimal ? (BigDecimal) value : new BigDecimal(value.toString());
}
@Override
public BigInteger getAsBigInteger() {
  return value instanceof BigInteger ?
      (BigInteger) value : new BigInteger(value.toString());
}
@Override
public float getAsFloat() {
  return isNumber() ? getAsNumber().floatValue() : Float.parseFloat(getAsString());
}
@Override
public long getAsLong() {
  return isNumber() ? getAsNumber().longValue() : Long.parseLong(getAsString());
}
@Override
public short getAsShort() {
  return isNumber() ? getAsNumber().shortValue() : Short.parseShort(getAsString());
}
@Override
public int getAsInt() {
  return isNumber() ? getAsNumber().intValue() : Integer.parseInt(getAsString());
}
@Override
public byte getAsByte() {
  return isNumber() ? getAsNumber().byteValue() : Byte.parseByte(getAsString());
}
@Override
public char getAsCharacter() {
  return getAsString().charAt(0);
}
@Override
public int hashCode() {
  if (value == null) {
    return 31;
  }
```

TABLE 8: Example 4 - Comparison

Intent		
// Using recommended hashing algorithm from Effective Java for longs and doubles		
Predictions		
CEG	**BiLSTM**	**NLGP**
int h = value.hashCode();	for (int i = 0; i < values.length; i++) {	}
Reference		
if (isIntegral(this)) {		

Chapter 6

Conclusion & Future work

In this paper, we propose a contextualized deep learning architecture for code generation. We incorporated a custom workflow to model contextual information, which encapsulates multiple aspects of source code by extracting *imports, signatures, preceding code, and member variables* and is used by both of the models that we implemented. While it is true that both models use the custom workflow, our second model additionally compresses the input with an additional BiLSTM layer.

We also contribute a dataset with over 200'000 samples of NL intent, context, and Java code tuples. In addition to our own work, we compare our results to the work of Heyman et al. [13] by replicating their results with our own data. The replication was used in order to compare their results (BLEU and CodeBLEU), as well as example predictions with our own models. Our experimental results reveal that the proposed model achieves better results than the previous SOTA model [13] using the CodeBLEU metric, however, the NLGP model achieved better results using the BLEU metric. Our results show that: (1) our models learn strong context representation from our custom context workflow, (2) our model benefits from the additional BiLSTM layer w.r.t. context retention, and (3) that our approach is not inferior to the current SOTA with regard to the CodeBLEU metric.

In the future, we plan to apply our contextualized approach to models designed for long input sequences such as BigBird [44], Longformer [6], and ETC [3] to capture more context information. We assume this would improve the model further.

We also plan to expand our dataset by adding multiple programming languages to show that our approach applies to a wide variety of languages and is able to be generalized. Integrating the proposed model into an IDE would provide us with the possibility of a study with user reviews to see how well the model enhances the learning process.

We propose to set up a case study with the goal of qualitatively accessing the usefulness of our model. To this end, we intend to integrate our model into an IDE in order for the study participants to use the model in a familiar environment as stated previously. Each of the participants will receive several randomized exercises from an example pool. Half of the exercises are to be solved using the assistant and the other half without the assistant. This allows us to compare the time spent per exercise as well as the correctness of the solution provided by the participants in comparison with the predictions of the assistant. Another time measurement is to be implemented between the query completion and the recommendation to analyze the time performance of the model. We also intend to collect ratings regarding the correctness, usefulness, and contextualization of the recommended code statements using a Likert scale [18] styled feedback option. In order to measure the time of an exercise, the participants are to initiate a timer at the start and end of each task. In order to analyze differences between query patterns w.r.t. gender, we intend to collect gender data from the participants. This could provide useful information pertaining to model biases, that should be addressed.

We intend to present the participants with a Likert-styled questionnaire, which is to be an-

swered after they have completed the series of exercises. A preliminary selection of possible questions is shown in Table 9.

TABLE 9: Questions

Question/Statements
You are likely to use the assistant again in the future
The assistant was helpful in solving the exercises
The recommendations were contextualized and could be used directly
The recommendation appeared in a sufficiently short time, after the completion of the query
Single-line recommendations increase the learning effect
I was able to learn new knowledge from the recommendations

Each question has the following options: 1) Strongly disagree; (2) Disagree; (3) Neither agree nor disagree; (4) Agree; (5) Strongly agree.

The participants are to be selected according to the following criteria:

- they should have a basic understanding of programming akin to introductory programming courses and
- be familiar with the IDE.

These criteria ensure that the participants (1) have the necessary knowledge to formulate their requirements in a comment, (2) are able to create a simple piece of code without any assistance, (3) do not waste time on IDE specific issues, and (4) are able to compare the proposed implementation to a correct solution.

Bibliography

[1] (2021). Github copilot · your ai pair programmer. `https://copilot.github.com/`.

[2] Ahmad, W., Chakraborty, S., Ray, B., and Chang, K.-W. (2021). Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online. Association for Computational Linguistics.

[3] Ainslie, J., Ontañón, S., Alberti, C., Pham, P., Ravula, A., and Sanghai, S. (2020). ETC: encoding long and structured data in transformers. *CoRR*, abs/2004.08483.

[4] Andonian, A., Biderman, S., Black, S., Gali, P., Gao, L., Hallahan, E., Levy-Kramer, J., Leahy, C., Nestler, L., Parker, K., Pieler, M., Purohit, S., Songz, T., Wang, P., and Weinbach, S. (2021). GPT-NeoX: Large scale autoregressive language modeling in pytorch.

[5] Bai, X. and Cui, Y. (2016). Enhancing the learning process in programming courses through an automated feedback and assignment management system.

[6] Beltagy, I., Peters, M. E., and Cohan, A. (2020). Longformer: The long-document transformer. *CoRR*, abs/2004.05150.

[7] Borges, H. and Valente, M. T. (2018). What's in a github star? understanding repository starring practices in a social coding platform. *CoRR*, abs/1811.07643.

[8] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*.

[9] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., and Zhou, M. (2020). CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

[10] Garcia, R. and Al-Safadi, L. (2013). Comprehensive assessment on factors affecting students' performance in basic computer programming course towards the improvement of teaching techniques. *International Journal for Infonomics*, 6:682–691.

[11] Gupta, N., Tejovanth, N., and Murthy, P. (2012). Learning by creating: Interactive programming for indian high schools. In *2012 IEEE International Conference on Technology Enhanced Education (ICTEE)*, pages 1–3.

[12] Han, X., Zhang, Z., Ding, N., Gu, Y., Liu, X., Huo, Y., Qiu, J., Yao, Y., Zhang, A., Zhang, L., Han, W., Huang, M., Jin, Q., Lan, Y., Liu, Y., Liu, Z., Lu, Z., Qiu, X., Song, R., Tang, J., Wen, J.-R., Yuan, J., Zhao, W. X., and Zhu, J. (2021). Pre-trained models: Past, present and future.

[13] Heyman, G., Huysegems, R., Justen, P., and Cutsem, T. V. (2021). Natural language-guided programming. *CoRR*, abs/2108.05198.

[14] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9:1735–80.

[15] Iyer, S., Konstas, I., Cheung, A., and Zettlemoyer, L. (2018). Mapping language to code in programmatic context. *CoRR*, abs/1808.09588.

[16] Kusupati, U., Ravi, V., and Ailavarapu, T. (2018). Natural language to code using transformers.

[17] Lahtinen, E., Ala-Mutka, K., and Järvinen, H.-M. (2005). A study of the difficulties of novice programmers. volume 37, pages 14–18.

[18] Likert, R. (1932). A technique for the measurement of attitudes. *Archives of Psychology 140: 5-55.*

[19] Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong, M., Zhou, M., Duan, N., Sundaresan, N., Deng, S. K., Fu, S., and Liu, S. (2021). Codexglue: A machine learning benchmark dataset for code understanding and generation. *ArXiv*, abs/2102.04664.

[20] Mangal, S., Joshi, P., and Modak, R. (2019). LSTM vs. GRU vs. bidirectional RNN for script generation. *CoRR*, abs/1908.04332.

[21] Norouzi, S. and Cao, Y. (2021). Semantic parsing with less prior and more monolingual data. *CoRR*, abs/2101.00259.

[22] Oracle Corporation (2014). Compatibility guide for jdk 8. Accessed: 2022-03-12.

[23] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E. Z., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. *CoRR*, abs/1912.01703.

[24] Phan, L., Tran, H., Le, D., Nguyen, H., Anibal, J., Peltekian, A., and Ye, Y. (2021). Cotext: Multi-task learning with code-text transformer.

[25] Pinheiro, A., Barbosa, A., Carvalho, R., Freitas, F., Tsai, Y.-S., Gasevic, D., and Ferreira, R. (2021). Automatic feedback in online learning environments: A systematic literature review. *Computers and Education: Artificial Intelligence*, 2:100027.

[26] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language models are unsupervised multitask learners.

[27] Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A., and Ma, S. (2020). Codebleu: a method for automatic evaluation of code synthesis. *CoRR*, abs/2009.10297.

[28] Santhanam, S. (2020). Context based text-generation using LSTM networks. *CoRR*, abs/2005.00048.

[29] Sennrich, R., Haddow, B., and Birch, A. (2016). Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany. Association for Computational Linguistics.

[30] Terada, K. and Watanobe, Y. (2021). Code completion for programming education based on deep learning. *International Journal of Computational Intelligence Studies*, 10:78.

[31] Teshima, Y. and Watanobe, Y. (2018). Bug detection based on lstm networks and solution codes. pages 3541–3546.

[32] Tran, N. M., Tran, H., Nguyen, S., Nguyen, H., and Nguyen, T. N. (2019). Does BLEU score work for code migration? *CoRR*, abs/1906.04903.

[33] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. *CoRR*, abs/1706.03762.

[34] Wang, W., Kwatra, A., Skripchuk, J., Gomes, N., Milliken, A., Martens, C., Barnes, T., and Price, T. (2021a). *Novices' Learning Barriers When Using Code Examples in Open-Ended Programming*, page 394–400. Association for Computing Machinery, New York, NY, USA.

[35] Wang, Y., Wang, W., Joty, S., and Hoi, S. C. (2021b). Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*.

[36] Watanobe, Yutaka, Intisar, Chowdhury, Cortez, Ruth, and Vazhenin, Alexander (2020). Next-generation programming learning platform: Architecture and challenges. *SHS Web Conf.*, 77:01004.

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

[38] Wong, E., Liu, T., and Tan, L. (2015). Clocom: Mining existing source code for automatic comment generation. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 380–389.

[39] Wong, E., Yang, J., and Tan, L. (2013). Autocomment: Mining question and answer sites for automatic comment generation. *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 562–567.

[40] Xu, F. F., Jiang, Z., Yin, P., Vasilescu, B., and Neubig, G. (2020). Incorporating external knowledge through pre-training for natural language to code generation. In Jurafsky, D., Chai, J., Schluter, N., and Tetreault, J. R., editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 6045–6052. Association for Computational Linguistics.

[41] Xu, F. F., Vasilescu, B., and Neubig, G. (2021). In-ide code generation from natural language: Promise and challenges. *CoRR*, abs/2101.11149.

[42] Yin, P., Deng, B., Chen, E., Vasilescu, B., and Neubig, G. (2018). Learning to mine aligned code and natural language pairs from stack overflow. *CoRR*, abs/1805.08949.

[43] Yin, P. and Neubig, G. (2018). TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 7–12, Brussels, Belgium. Association for Computational Linguistics.

[44] Zaheer, M., Guruganesh, G., Dubey, A., Ainslie, J., Alberti, C., Ontañón, S., Pham, P., Ravula, A., Wang, Q., Yang, L., and Ahmed, A. (2020). Big bird: Transformers for longer sequences. *CoRR*, abs/2007.14062.

[45] Zhong, V., Xiong, C., and Socher, R. (2017). Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103.

A NLGP training details

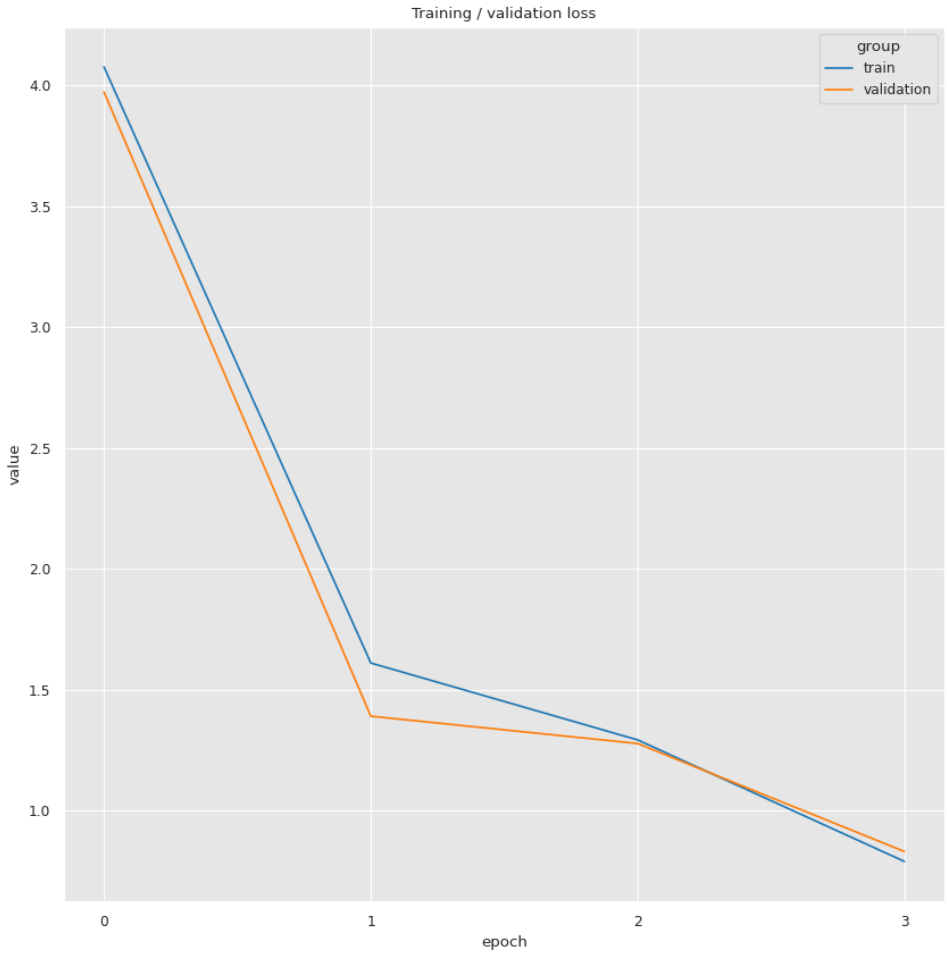


Figure 12: NLGP training & validation loss