Advancements and Challenges of Large Language Model-Based Code Generation and Completion

Zheer Wang
College of Engineering, University of Kentucky, Lexington, Kentucky, 40506-0107, U.S.A.

Keywords: Large Language Models, Code Generation, Code Completion.

Abstract: This paper provides an in-depth review of the recent advancements and applications of large language models (LLMs) in the field of code generation and code completion. Since deep learning and transformer architectures have advanced so quickly, LLMs have shown previously unheard-of powers in producing source code from natural language, revolutionizing software development procedures. The underlying ideas of these models are first explained in the review, with particular attention to how large models such as Generative Pre-trained Transformer (GPT)-3 and Codex use pre-training and fine-tuning techniques to produce sophisticated code from descriptions in simple language. These models produce high-quality outputs by autonomously learning programming syntax and semantics and using attention techniques to capture contextual dependencies in code, contrasting with conventional rule-based or heuristic approaches. This paper also demonstrates how well LLMs perform in a variety of applications, including code translation, code completion, and error detection, as well as how effectively they function in multi-language programming environments. Additionally, models like PolyCoder and Program and Language Bidirectional and Auto-Regressive Transformers (PLBART) are emphasized because they outperform traditional methods, particularly in cross-language tasks. Although LLMs show great promise, the study also discusses some of their current drawbacks, such as their high memory consumption, opaque training data, and difficulties with generalizing to new codebases. In summary, while LLMs provide unparalleled prospects for software engineering advancement, further investigation is required to overcome current obstacles and expand their relevance to broader fields.

1 INTRODUCTION

In today's digital world, code generation is growing more and more significant and is essential to driving innovation across a range of sectors. These days, programming is a crucial component of many other technical domains and is not only the technical expertise of software engineers. Fast and effective code development is becoming more and more necessary as many sectors need automated processes to handle massive volumes of data and increase productivity. This has led to the creation of cutting-edge technologies, particularly large language models (LLMs) that automate coding operations. With the use of Transformer architecture and deep learning technology, LLMs have pushed advancements in the field of code generation, allowing machines to comprehend and produce language that is comparable to that of humans. These models can interpret natural language instructions and translate them into executable code; well-known examples of these models include Generative Pre-Trained (GPT)-3 and Codex. Compared to traditional programming methods, which call for the manual creation of certain grammars, grammatical rules, and algorithms, this change is very different. LLMs can automate complicated programming jobs by learning programming languages in a manner akin to human learning through the training of vast amounts of data.

In particular, LLMs are capable of carrying out a wide range of activities previously assigned to human programmers. These jobs include code translation, which is translating code between different programming languages, and code completion, which is the model's prediction and recommendation of the subsequent line or segment of code based on contextual prompts. Furthermore, LLMs have demonstrated noteworthy outcomes in debugging and mistake detection, significantly decreasing the time and effort needed for human error checks. Large models can handle difficult tasks with little human intervention, which eases the burden on developers

Wang, Z.
Advancements and Challenges of Large Language Model-Based Code Generation and Completion.
DOI: 10.5220/0013271800004558

In *Proceedings of the 1st International Conference on Modern Logistics and Supply Chain Management (MLSCM 2024)*, pages 208-213
ISBN: 978-989-758-738-2

and makes it possible for non-professionals to generate functional code using natural language input. This is one of the key benefits of employing large models for code generation. In addition to their versatile applications, LLMs show remarkable flexibility in handling multiple programming languages. Global development environments can benefit greatly from the accuracy of code translations provided by models such as Program and Language Bidirectional and Auto-Regressive Transformers (PLBART) and PolyCoder, which are well-suited for cross-language work. These models can also be adjusted for particular domains, which improves their accuracy in particular tasks. Because of their scalability and versatility, LLMs are essential tools for software engineers.

In summary, this paper examines the design and capabilities of the big language model and reviews its evolution in the field of code generation. This evaluation looks at these models' benefits and drawbacks as well as how they might be used to further the field of programming.

2 LLM-BASED CODE GENERATION

The automatic production of code fragments or entire programs from a high-level specification—like plain language descriptions—is referred to as code generation. Significant advancements in automating this procedure have been made possible by deep learning and large-scale language models. First, large language models have shown impressive new abilities to generate natural language text and to solve a rapidly expanding set of modeling and reasoning tasks. Second, over the past decade, machine learning approaches have been applied to source code text to yield a variety of new tools to support software engineering (Austin, 2021). Unlike traditional methods that require manual coding of specific rules and grammar, large models learn to generate code by understanding large amounts of data. With the help of tokenized datasets, large language models for code generation may efficiently capture the syntax and semantics of many programming languages without the need for explicit programming, and this method allows these models to handle a variety of complex coding tasks with minimal human intervention.

LLMs have significantly advanced in automating the generation of source code from natural language descriptions. These models, often referred to as Code LLMs. Transformer-based models have demonstrated significant progress in handling complex code generation tasks, surpassing earlier rule-based systems and heuristics, allowing models to handle complex code generation tasks more effectively (Jiang, 2024). These models have demonstrated the capability to not only generate code that meets functional requirements but also to learn from feedback and improve over time, as seen with techniques such as reinforcement learning. The self-attention mechanism enables the transformer model to focus on different parts of the input sequence, understanding both the context and the relationships between words, regardless of their distance (Chen, 2024). By using attention ratings to give varying levels of priority, this technique enables each word or token in the input sequence to consider every other word or character. Practically speaking, this means that the model can assess the connections between words in a phrase regardless of where they are located, which is essential for comprehending intricate directives and subtle contextual cues. This approach, when used in conjunction with code generation, allows the model to comprehend the context of a variable or function definition and how it will be used later in the code. By capturing these dependencies, the model can generate code that is not only syntactically correct but also logically coherent, ensuring that the generated code adheres to the intended functionality and structure. Typically, a multi-stage process is involved in the transformer-based LLM code generation workflow to fully utilize these models. The first phase is called pre-training, Transformer models use pre-training on large datasets of source code to learn general programming patterns, which can be further fine-tuned for specific tasks (Chen, 2024). To help the model learn typical programming patterns and syntax, this training involves exposing it to a variety of programming languages, coding styles, and problem-solving techniques. The model is adjusted to more precise activities or situations during fine-tuning. For example, a model can be tailored especially for Structured Query Language (SQL) query generation or Python scripting by utilizing datasets that include code samples that match descriptions in plain language. Through this process, the model can become more specialized and enhance its relevance and accuracy when producing particular kinds of code. Lastly, the model uses its learnt representations to translate natural language inputs into executable code during the generation phase. To produce the intended result, this phase entails applying learnt programming logic in addition to comprehending the input context. The generated code can range from simple utility

functions to more complex algorithms, depending on the input provided.

A significant example of using the Transformer model is the PLBART model, which is a unified pre-training model specially designed for program understanding and generation tasks. PLBART employs denoising sequence-to-sequence pre-training, where the model is trained to recover corrupted input sequences, helping it learn syntax and semantics across programming languages (Ahmad, 2021). The pre-training involves a denoising autoencoding approach, where the model is trained to reconstruct original input sequences that have been corrupted by random noise. In the pre-training phase, the model is trained to recover original input sequences that have been tainted by random noise using a denoising autoencoding technique. This method helps in the model's acquisition of programming language syntax and semantics, as well as their correspondence with descriptions of natural language, allowing it to function successfully in a variety of tasks.

The application of large models in code generation has also become an important field in machine learning research. These models are primarily categorized into three types: language models, transducer models, and multimodal models. Language models, like natural language processing (NLP), are made to represent the process of creating code as a sequence prediction issue. These models, which include neural network-based and n-gram models, forecast the subsequent token in a series depending on the tokens that came before it. Programming language syntax and structure can be effectively learned by language models, allowing them to carry out operations like code completion. However, the assumption made by n-gram models simplifies the dependency on context, thus failing to handle long-range dependencies, making them ineffective in capturing information such as variable scoping in code generation (Allamanis, 2018).

Transducer models, which are based on statistical machine translation, are used to translate code between different programming languages or from pseudocode to source code, among other representations. These models are inspired by statistical machine translation, map code between different languages or representations, making them ideal for tasks such as code migration or refactoring because they learn mappings between various syntactic or semantic components in code (Allamanis, 2018).

Multimodal models combine natural language and code production with different modalities. The goal of these models is to produce code from a variety of inputs, including written descriptions, visual clues, and other non-code data. For instance, the assumption made by n-gram models simplifies the dependency on context, thus failing to handle long-range dependencies, making them ineffective in capturing information such as variable scoping in code generation.

Direct translation of natural language instructions into executable code is a major capability of large models. Developers that work with traditional programming typically need to be fluent in programming languages and possess in-depth understanding of algorithms. However, complicated code can be generated by users even without programming skills thanks to huge models. These models, such as OpenAI's GPT-3, have demonstrated the capacity to generate human-like text, including programming code, by training on vast datasets of human language. In order to comprehend the syntax and semantics needed for diverse programming tasks, the procedure usually entails training a model on a sizable corpus of code and related textual data. With the use of natural language descriptions, these models may produce code snippets, find and repair errors in existing code, and even recommend code completions. These models' performance in code generation tasks is frequently assessed in three different learning scenarios: zero-shot, one-shot, and few-shot. Zero-shot learning involves the model generating code without any examples given during the job; one-shot learning provides one example; and few-shot learning provides a small number of instances. GPT-3 achieves promising results in the zero- and one-shot settings, and in the few-shot setting is sometimes competitive with or even occasionally surpasses state-of-the-art (Mann, 2020). A major discovery in the research on huge language models, such as GPT-3, is that these models can function as effective meta-learners. That implies they don't need a lot of retraining to swiftly adjust to new jobs, which is especially useful for code generation. The feature known as "in-context learning" allows these models to make use of their substantial pre-training by inserting examples right into the input context, enabling them to comprehend and produce relevant replies.

3 LLM-BASED CODE COMPLETION

Code completion is an automatic code generation approach that is based on context and aims to foresee and finish code fragments that a developer is currently typing.

Code completion can take one of the following forms, depending on the level of operation:

Token-level completion: In this case, code completion suggests for the next line of code based on a partially entered word or symbol by the developer. For example, the tool may propose the entire variable name after you type the first few letters.

Line-level completion: This is anticipating and finishing a line of code from the partially written code and the context of the current line. For example, depending on a conditional phrase entered, it might automatically finish a semicolon or closing bracket.

Block-level completion: Code completion can anticipate and inserting more complex code, such loops, methods, or whole class hierarchies. A deeper comprehension of the logic and structure of code is necessary for this kind of completion.

Deep learning approaches for sequence prediction are typically the foundation of code completion algorithms. Take Codex as an illustration. Codex is trained on large-scale public datasets containing code, mainly sourced from GitHub repositories, and is evaluated on its ability to complete or generate code based on natural language descriptions (Chen, 2021). With its transformer architecture, large-scale sequence data processing is possible. The model learns the syntax, organization, variable dependencies, and common programming patterns of the code by being trained on vast amounts of code data that are made available on open-source platforms like GitHub. Code completion tasks are especially well-suited for the language model's auto-regression generation method since the logic and structure of the code are comparable to those of natural language. The code must be transformed into a format that the model can understand for it to be able to complement the code. Millions of code repositories on GitHub provide the training data, which has been preprocessed, cleaned, and copied to create a sizable code base. Most of the training data in Codex is Python code, and each code file has been tokenized to enable the model to recognize and learn every element in the code, including function names, variable names, operator names, etc. In this manner, Codex can gradually produce code fragments while learning the syntax, variable binding, function calling, and other patterns of other languages. Code completion relies on a model that utilizes the input portion of the code to anticipate the next most likely code fragment. Codex generates code in a recursive manner, producing one or more tokens each time. It then keeps predicting the next token by using the created content that has already been produced. This procedure keeps going until the terminator, which could be a comment symbol, a newline character, or the function's ending symbol.

Codex will produce several potential completion techniques for user-inputted code fragments depending on the current situation. The model learns common programming patterns during training, like function declaration, loop structure, conditional judgment, etc., on which these completions are typically based. For instance, Codex might automatically finish the function's parameters, body, and even return value if the user only enters the function definition's beginning. Nevertheless, it is challenging to further enhance or apply models like Codex to other sectors because their internal workings and training data are not publicly available. The PolyCoder model fills many of the gaps in the existing research when compared to the Codex model. Based on the GPT-2 architecture, PolyCoder is a model with 2.7B parameters, 249GB of training data, and support for 12 programming languages (Xu, 2020). PolyCoder even outperformed Codex in the C language code creation task, exhibiting superior performance in a particular language. PolyCoder's multilingual training data set, which includes C, C++, Python, Java, JavaScript, and other languages, is one of its advantages. Because of its multilingual training, PolyCoder is better able to handle multiple programming languages and benefit from this shared characteristic. Besides, researchers and developers can utilize PolyCoder's model parameters and training data without restriction because it is an entirely open-source model. By studying more about their model architecture, data selection, and training procedure, researchers can enhance and optimize code completion technology in subsequent studies.

Recent advancements in code completion are driven by the integration of large-scale language models, such as those used in neural-based code suggestion systems. Traditional approaches have limitations due to their high memory consumption and difficulty generalizing across new codebases or unseen APIs. Current methods for code completion use neural models in conjunction with static analysis to increase prediction accuracy and memory efficiency. These models optimize code completion by reranking suggestions rather than generating completions from scratch, allowing for faster

predictions with a lower memory footprint. The best neural reranking model consumes just 6 MB of RAM, 19× less than previous models, and achieves 90% accuracy in its top five suggestions (Svyatkovskiy, 2021). Furthermore, recent advancements in sequence-to-sequence (Seq2Seq) models, including Sequence Span Rewriting (SSR), indicate the possibility of improving code completion even more. SSR bridges the gap between pre-training and fine-tuning, because many downstream Seq2Seq tasks like summarization and paraphrase generation are naturally sequence span rewriting tasks (Zhou, 2021). By training models to rewrite machine-generated imperfect spans into ground truth text, SSR improves earlier text-infilling techniques. This method works particularly well with smaller models or in limited contexts since it not only widens the range of learning signals in the model but also narrows the gap between pre-training and fine-tuning.

4 DISCUSSIONS

The discussion of this paper highlights both the strengths and limitations of LLMs in code generation. Significant progress has been made by these models, especially in automating debugging, translation, and code completion activities while lowering the need for human participation. Software development has been transformed by its capacity to produce high-quality code from natural language inputs, particularly for non-experts. Nevertheless, LLMs continue to encounter significant obstacles, such as excessive memory consumption, ambiguous training data, and trouble generalizing to unknown codebases. Their wide application and scalability are restricted by these problems. Future studies need to concentrate on overcoming these restrictions. Important next stages include increasing the transparency of LLMs' training procedures and strengthening their capacity to manage a variety of unfamiliar programming settings. Reducing the computational resources needed for these models will also enhance their usability and facilitate their incorporation into different programming workflows. LLMs can reach even higher potential in software engineering and other fields by overcoming these obstacles.

5 CONCLUSIONS

This paper has provided an in-depth review of LLMs in the field of code generation, highlighting their methods, results, and future potential. Deep learning and transformer architectures underpin models like GPT-3 and Codex, which have demonstrated impressive efficacy in automating code completion, translation, and debugging activities. These models can efficiently learn the syntax and semantics of several programming languages by employing pre-training and fine-tuning procedures, producing executable code from natural language inputs. The findings show that LLMs considerably decrease the amount of time needed for manual debugging and error detection, and they increase coding efficiency, particularly in multilingual situations. However, LLMs still face notable limitations. Challenges to their wider implementation include high memory usage, opaque training data, and difficulty in generalizing to new and unfamiliar codebases. These problems restrict their use in a variety of specialized programming contexts and impede their scalability. In the future, research should concentrate on lowering the processing power needed by LLMs and enhancing the clarity of their training procedures. Increasing their applicability will need improving their capacity to adapt to new programming languages and environments. With further development, LLMs could completely transform automated software development and become indispensable to programming in the future.

REFERENCES

Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K. W. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*.

Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, *51*(4), 1-37.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., ... & Sutton, C. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Chen, L., Guo, Q., Jia, H., Zeng, Z., Wang, X., Xu, Y., ... & Zhang, S. 2024. A Survey on Evaluating Large Language Models in Code Generation Tasks. *arXiv preprint arXiv:2408.16498*.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., ... & Zaremba, W. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Jiang, J., Wang, F., Shen, J., Kim, S., & Kim, S. 2024. A Survey on Large Language Models for Code Generation. *arXiv preprint arXiv:2406.00515*.

Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., ... & Amodei, D. 2020. Language

models are few-shot learners. *arXiv preprint arXiv:2005.14165*, *1*.

Svyatkovskiy, A., Lee, S., Hadjitofi, A., Riechert, M., Franco, J. V., & Allamanis, M. 2021. Fast and memory-efficient neural code completion. In 2021 IEEE/ACM 18th International Conference on Mining Software Repositories, 329-340.

Xu, F. F., Alon, U., Neubig, G., & Hellendoorn, V. J. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 1-10.

Zhou, W., Ge, T., Xu, C., Xu, K., & Wei, F. 2021. Improving sequence-to-sequence pre-training via sequence span rewriting. *arXiv preprint arXiv:2101.00416*.