Proceedings of the 5th International Conference on Computing and Data Science
DOI: 10.54254/2755-2721/22/20231222

A review on statistical language and neural network based code completion

Ze Gao

Christ Church Grammar School, Perth, Australia, 6010

eddiegaoze@gmail.com

Abstract. Code completion, also referred to as intellisense, is a prevalent feature of Integrated Development Environments (IDEs) and code editors. It aids developers by automatically recommending and inserting code segments, variable names, and method names, among other things. With the accelerated growth of the software industry and the process of digitalization in recent years, the demand for software engineers has reached a record-high level. Thus, the advancement of code completion is encouraged and has become a popular topic in software engineering. This paper examines and summarizes the development of a statistical language and neural network-based code completion system. The main contents consist of introducing the concepts of code completion system, summarizing the general process of code completion and the evaluation metrics used for performance benchmarking, reviewing and summarizing the existing work conducted on statistical language approach and neural network approach respectively, as well as the limitations and challenges of existing code completion method, and finally forecasting the future development of code completion techniques.

Keywords: code generation, code completion, statistical language model, neural network.

1. Introduction

Since the invention of computer systems, software engineers have been closely tied to tasks involving the writing of code. Improving the productivity and quality of software development consequently becomes one of the field's most pressing issues. This technology is capable of analyzing developers' input and existing code corpus to determine the name of a method, class, or variable. As a result, developers can spend less time memorizing unfamiliar method and class names, thereby increasing the development efficiency. In the past few decades, code completion has become one of the most popular topics in the field.

Currently, there are three primary forms of code completion: token completion, code snippet completion, and keywords/abbreviations completion.

The application of statistical language models and neural language models to the problem of intelligent code completion is investigated in depth. From 2009 to 2022, the most recent research papers on code completion from sources such as arXiv, IEEE, Springer, and various ACM conferences. This review's primary contribution is a summary of existing research on code completion models using a statistical language approach and neural network approach. In addition to examining the limitations of these approaches, the review also provides future development prospects.

DOI: 10.54254/2755-2721/22/20231222

2. Overview

2.1. Summary of research on code completion

Code completion is a problem involving the prediction and recommendation of the following token in code statements. The representation of code statements in the model is crucial to a code completion system as it affects how the dataset is processed and prepared and how the model is trained.

In 2012, Hindle et al. published the hypothesis that coding language and natural language share repetitive and predictable characteristics [1], allowing them to be predicted by a mathematical model. Hindle et al. proposed an N-gram-based statistical language model to predict the next token in a code statement [2]. The model outperformed the Eclipse code completion extension, which validates the use of statistical language models to represent and predict code statements.

Bengio et al. trained the statistical language model with a neural network in 2003, paving the way for the development of a code completion model [3]. Researchers are able to calculate the probability of a code token using neural networks to a greater extent as a result of the increased availability of computing power resulting from technological advancements. The statistical language approach and neural network approach will be discussed in greater depth in the sections that follow.

2.2. The general process of code completion

Different completion methods have distinct differences in how they represent code and how they evaluate its correctness. Both methods share a common structural and operational basis. Figure 5 depicts a general overview of the coding-completion process.

To begin extracting characteristics from the code for both the training and completion processes, the computer must first represent the code in a form it can understand. The extraction of data from the code and its structural representation constitute the method of code representation.

After training on the source code dataset, the model can predict and recommend code snippets with the highest likelihood and similarity values by comparing the context information of the to-be-completed code and other code segments.

Next, the model excludes the completion results that do not meet the syntax requirements from the preliminary code recommendations based on the type of completion and the specific syntax of the coding language.

The model ultimately provides the developers with a pop-up window containing a curated set of code recommendations.

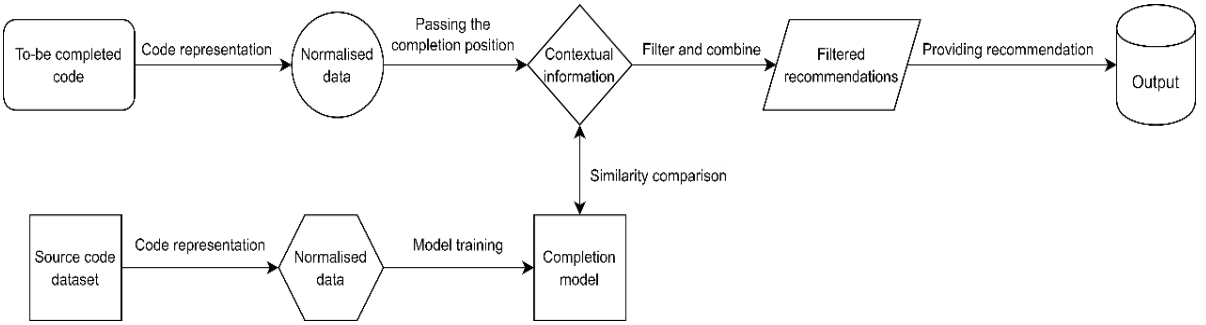

Figure 1. General code completion process (original).

2.3. Performance evaluation metrics for the model

As the code completion model typically suggests the K most probable options to the user, the following performance metrics are typically used to evaluate its effectiveness: Mean Reciprocal Rank (MRR), Accuracy@K, Precision@K, Recall@K, and F-measure@K, where @K indicates the calculation is based on the completion model's top K recommendations.

Mean Reciprocal Rank (MRR) is commonly employed for benchmarking the efficacy of information indexing and recommendation systems. This benchmark establishes the position of the first current

DOI: 10.54254/2755-2721/22/20231222

answer as the measuring standard; it is calculated as follows: for a code completion recommendation, if the position of the correct answer in the list when it appears for the first time is n, then the MRR value is $\frac{1}{n}$. As the MRR approaches one, the correct code completion moves closer to the head of the list of recommendations. Consequently, the multiplicative inverse of MRR indicates the average position of the correct result in the recommendation list, and MRR is the mean of the reciprocal rankings of results for the sample of queries Q.

$$\mathrm{MRR} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{\mathrm{rank}_i} \tag{1}$$

Accuracy@K is the ratio of the correct code completion recommendations to the top K recommendations.

$$\mathrm{Accuracy} = \frac{\mathrm{recommendations}_{\mathrm{Correct}}}{\mathrm{recommendations}_{\mathrm{Made}}} \tag{2}$$

Precision@K is the ratio of the relevant code completion recommendations to the top K recommendations.

$$\mathrm{Precision} = \frac{\mathrm{recommendations}_{\mathrm{made} \cap \mathrm{relevant}}}{\mathrm{recommendations}_{\mathrm{made}}} \tag{3}$$

Recall@K is the ratio of the relevant code completion recommendations in the top K recommendations to the total number of relevant options.

$$\mathrm{Recall} = \frac{\mathrm{recommendations}_{\mathrm{made} \cap \mathrm{relevant}}}{\mathrm{recommendations}_{\mathrm{relevant}}} \tag{4}$$

F-measure@K is a customised metrics calculated by precision and recall, which enables research to adjust the weights of precision and recall by changing the value of β.

$$\mathrm{F} = \frac{(1+\beta^2) \times \mathrm{precision} \times \mathrm{recall}}{\beta^2 \times \mathrm{precision} + \mathrm{recall}} \tag{5}$$

2.4. Validation method of the model

Typically, the validation method involves dividing the data set into a training set and a validation set in a predetermined ratio. K-Fold Cross-Validation, in which the data is divided into more than two categories, is one of the most well-known and utilized methods in the field of code completion. In this method, 'K' is the number of groups into which the data is divided, one group is set aside as validation data, and K-1 groups are used to train the model. Each group is then used to validate the code completion model, with the average output of K number of operations serving as the model's evaluation metric.

3. Approaches

3.1. Statistical language model approach

This part examines recent advancements in code completion utilizing statistical language models.

Initially, the code completion problem was raised to reduce the need for developers to remember programming language syntax and API (application programming interface); this led to the discovery of prospective methods for improving the performance of the completion model. In 2012, Hindle et al. drew inspiration from the use of statistical language models in speech recognition and NLP (natural language processing) and based their work on the premise that programming languages [1], despite being complex and sophisticated, are repetitive and simple in nature due to the fact that they are written by humans. Based on a language dataset, their work builds an N-gram language model, allowing them to complete code through the calculation of cross-entropy [4]. A year later, Nguyen et al. proposed a tool called SLAMC to address the issue of the N-gram model only being able to extract features within a range of N elements [5], where it also labeled the code token with data type (i.e. integer) and type (i.e. constant) to assist the model in capturing the contextual information of the source code. The SLANG

DOI: 10.54254/2755-2721/22/20231222

completion utility was developed by Raychev et al. in 2014 [6]. The SLANG tool extracts the sequence of API calls from a partial program containing holes and applies them to a statistical language model (N-gram, RNN, or a combination of the two) to return the completion with the highest probability that also satisfies the constraints imposed by the holes. In ninety percent of instances, the model returns the desired candidate among the top three options.

In 2015, Bettenburg et al. proposed that the datasets obtained from open-source repositories are highly variable, which has a negative effect on the validity of the model [7]. They proposed to divide the data into smaller, homogenous subsets and train each statistical language model separately. A case study of a dataset in the field of software engineering demonstrates that models constructed from local datasets outperform traditional models in terms of model fitting and predictive performance. Nonetheless, the clustering algorithm and parameter selection will have detrimental effects on the quality of the model. Nguyen et al. proposed a statistical language dubbed AUTOSC that integrates program analysis and a statistical language model with the principle of software naturalness to complete incomplete code statements [1, 8]. AUTOSC employs the N-gram model to learn the most likely candidate templates from code statements that contain a sequence of special annotations called Extended Code Tokens. These Extended Code Tokens contain information regarding the token's type and/or data type, allowing the language model to anticipate the next token with greater contextual knowledge of the code statement. Such a design facilitates AUTOSC obtains 38.1%-41.3% top-1 accuracy and 48.2-50.1% top-5 accuracy in statement completion and outperforms a conventional approach from 9X-69X in top-1 accuracy.

Code completion models based on statistical language continue to improve performance by optimizing the N-gram model; however, the limitations of the N-gram model, namely the difficulty in dealing with long-distance relations and code statement regularities, cannot be entirely resolved. The Recurrent neural network was then applied to the code completion issue in order to enhance performance.

3.2. Neural network model approach

This section discusses recent developments in code completion utilizing neural network models.

Various deep learning-based approaches have been proposed to enhance and stabilize the performance of code completion with multiple long-range dependency inferences. In 2015, White et al. proposed a model based on Recurrent neural network (RNN) and applied deep learning techniques to enhance the quality of abstraction of code features and the quality of representation [9], which were key factors in determining the quality of code completion. The model extended the neural network model to incorporate more long-range dependencies and outperformed the standard N-gram model in terms of code completion performance. Bhoopchand et al. created a model based on pointer network with a sparse attention mechanism later in 2016 [10,11]. This study compiled a large corpus of Python code suggestions and discovered that standard neural networks are constrained by a hidden state bottleneck. As all contextual information must be recorded in a fixed-dimensional vector representation, it restricts the model to local features of the code corpus and hinders its ability to capture long-range relationships. The model maintains a pointer structure to global vocabulary and calculates pseudo-sparse distribution to increase token prediction precision by 25%.

Transformer is the most prevalent state-of-the-art deep neural network architecture, which has lately been utilized in the field of code generation and completion. In 2020, Kim et al. proposed a novel application of transformers as the neural architecture's backbone [12]. The PathTrans and TravTrans models were used to teach the transformer structure the syntactic structure of the source code, allowing the work to outperform the accuracy of some state-of-the-art next token prediction models by a significant margin of 14% to 18%. The study concludes that transformer outperforms the conventional RNN in predicting the token sequence of a given source document. A year later, Matteo et al. utilized the Roberta model to evaluate the capabilities of cutting-edge deep learning models in the field of code completion at various granularity levels [13]. Roberta (Robustly Optimized BERT Pre-training Approach) is a BERT model whose input sentences have been arbitrarily masked out using a special MASK> token. Bidirectional Encoder Representations from Transformers is what BERT refers to. The

DOI: 10.54254/2755-2721/22/20231222

Roberta model was able to predict code token, code construct, and code blocks, but it has the limitation that the number of concealed tokens must equal the value of n, which creates an unrealistic scenario in which the completion system is informed of the number of tokens to be generated.

Contrary to some misconceptions, code generation and completion tasks are not limited to text; code can be generated from an image, such as a screenshot. Beltramelli et al. proposed the pix2code model in 2017 based on CNN (Convolutional neural network) and RNN (Recurrent neural network) [14]. CNN was used to transform the image into a fixed-length output vector, which RNN then analyzed to generate the code generation output.

Allamanis et al. proposed that the structure and syntactical information of source code can be depicted by a graph [15], and that graph neural network outperforms convolutional neural network in variable completion performance. However, the mechanism could only be used to complete variable names and could not complete tokens.

Prediction of Out-of-Vocabulary (OOV) words is presently one of the most significant issues with code completion and code generation. Li et al. proposed a pointer network for improved OOV word prediction performance [16]. The pointer mixture network can learn to generate within-vocabulary words using the RNN component with LSTM or to generate outside-of-vocabulary words using the pointer component.

4. Limitations and prospects

In terms of limitations, the first is the model's compatibility with various programming languages. Although numerous code completion models have been developed in recent years, the majority of them focus on Java and Python. This is not only due to the availability of datasets and the syntax of those languages. Java and Python have advantages in terms of code corpus quantity and quality, and they are both object-oriented programming languages that, compared to other programming languages (such as C or C++), have a more comprehensive library and API (application programming interface) support. Recent work on the development of a multilingual code corpus may hold the key to enhancing the mobility of the code completion model [17,18].

Manually evaluating the performance of the code completion procedure is the second step. As code completion technology is developed to aid in the development process, these models must be manually evaluated to accurately reflect their performance and contributions to projects. However, the absence of a uniform standard for evaluation and the higher costs associated with it have resulted in only a small number of works opting for human evaluation [19,20].

The third lacks a standardized metric for evaluating model efficacy. Most existing works only use one of the commonly used evaluation metrics (i.e., accuracy@K, Recall@K, and MRR) for evaluation, making it difficult to compare the performance of models evaluated by different metrics.

The last one lacks a high-quality, standard-compliant code dataset. Typically, the code corpus used to train a code completion model is obtained from open-source repositories or explicitly generated by DSL (domain specific language). Code corpora generated by DSL typically have straightforward syntax characteristics and shorter statement lengths, and are consequently simpler to train and validate. However, models trained on DSL-generated code corpus are frequently incompatible with other programming languages. Code corpora obtained from online repositories are more representative of actual software development, but neither the quality nor syntax can be guaranteed. Models trained on these low-quality code corpora frequently suffer from increased noises that negatively influence the model's performance [21,22].

Regarding the expectations of this discipline, the vocabulary size must be expanded. Most existing works limited vocabulary size to 100k, resulting in insufficient coverage for each token. This increases the incidence of out-of-vocabulary (OOV) words and impacts the model's performance. Therefore, a larger vocabulary inventory should be utilized for the training of the completion model [23].

Moreover, it is an effective method for integrating compiler technology to enhance output quality. Existing models frequently do not take into account whether the code snippets they predict correspond with the syntax of the programming language, resulting in code that cannot be compiled [24]. By

DOI: 10.54254/2755-2721/22/20231222

incorporating the syntax requirements of the programming language into the code completion model and refining the recommendation output, it is possible to maintain the output's quality and consistency.

5. Conclusion

In this review, code completion tasks using statistical language models and neural network models are highlighted. First, code completion system-related concepts are introduced. Then, a summary of the general process of code completion and commonly employed evaluation metrics follows. Following a review of the extant literature on statistical language approach and neural network approach, the authors present their findings. Finally, the system's limitations and future prospects are discussed.

As for the limitations, the review has relied solely on research papers as its primary source of information and has not conducted any interviews with industry developers, making its evaluation of performance from the developer's perspective less reflective. Future work will address this issue by interviewing and surveying developers currently employed in the software industry regarding their experience with the codec completion utility.

References

[1] Abram Hindle, E. T. (2012). On the Naturalness of Software. IEEE, In: Proc. of the 2012 34th Int'l Conf. on Software Engineering (ICSE).

[2] Lalit Bahl, F. J. (1983). A maximum likelihood approach to continuous speech recognition. IEEE Trans. on Pattern Analysis & Machine Intelligence, 5(2):179−190.

[3] Yoshua Bengio, R. D. (2003). A Neural Probabilistic Language Model. Journal of Machine Learning Research, 3(6):1137−1155.

[4] Christopher Manning, H. S. (1999). Foundations of Statistical Natural Language Processing. The MIT Press.

[5] Nguyen, T. T. (2013). A statistical semantic language model for source code. ESEC/FSE.

[6] Raychev V, V. M. (2014). Code completion with statistical language models. ACM SIGPLAN Notices, 49(6):419−428.

[7] Nicolas Bettenburg, M. N. (2015). Towards improving statistical modeling of software engineering data: think locally, act globally! Empir Software Eng, 20:294–335.

[8] Son Nguyen, T. N. (2019). Combining Program Analysis and Statistical Language Model for Code Statement Completion. Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering Conference (ASE 2019).

[9] Martin White, C. V.-V. (2015). Toward Deep Learning Software Repositories. 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR), IEEE.

[10] Oriol Vinyals, M. F. (2015). Pointer Networks. Advances in Neural Information Processing Systems 28 (NIPS 2015).

[11] Avishkar Bhoopchand, T. R. (2016). LEARNING PYTHON CODE SUGGESTION WITH A SPARSE POINTER NETWORK. arXiv:1611.08307.

[12] Seohyun Kim, J. Z. (2021). Code Prediction by Feeding Trees to Transformers. IEEE, In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) (pp. 150-162).

[13] Matteo Ciniselli, N. C. (2021). An Empirical Study on the Usage of BERT Models for Code Completion. IEEE, In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) (pp. 150-162).

[14] Tony Beltramelli. (2017). pix2code: Generating Code from a Graphical User Interface Screenshot. arXiv:1705.07962.

[15] Miltiadis Allamanis, M. B. (2017). Learning to represent programs with graphs. arXiv preprint arXiv:1711.00740.

[16] Jian Li, Y. W. (2018). Code Completion with Neural Attention and Pointer Networks. arXiv:1711.09573v2.

[17] Georgios Nikitopoulos, K. D. (2021). CrossVul: A Cross-Language Vulnerability Dataset with Commit Data. ESEC/FSE2021.

DOI: 10.54254/2755-2721/22/20231222

[18] Tao Yu, R. Z. (2018). Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. arXiv:1809.08887v5 cnncs.CL].

[19] Frank F. Xu, B. V. (2022). In-IDE Code Generation from Natural Language: Promise and Challenges. ACM Trans. Softw.Eng. Methodol, vol. 31, no. 2, pp. 29:1–29:47.

[20] Eman Abdullah AlOmar, A. I. (2021). AntiCopyPaster: Extracting Code Duplicates As Soon As They Are Introduced in the IDE. arXiv:2112.15230v2 [cs.SE].

[21] Pengcheng Yin, G. N. (2017). A Syntactic Neural Model for General-Purpose Code Generation. Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, pages 440–450.

[22] Pengcheng Yin, G. N. (2018). TRANX: A Transition-based Neural Abstract Syntax Parser for Semantic Parsing and Code Generation. Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (System Demonstrations), pages 7-12.

[23] Maxim Rabinovich, M. S. (2017). Abstract Syntax Networks for Code Generation and Semantic Parsing. Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, pages 1139–1149.

[24] Ashish Vaswani, N. S. (2017). Attention is all you need. Advances in neural information processing systems, vol. 30.