

# Toward Less Hidden Cost of Code Completion with Acceptance and Ranking Models

1<sup>st</sup> Jingxuan Li  
Huawei Technologies Co., Ltd  
Shenzhen, China  
lijingxuan1@huawei.com

2<sup>nd</sup> Rui Huang  
Huawei Technologies Co., Ltd  
Shenzhen, China  
huangrui27@huawei.com

3<sup>rd</sup> Wei Li  
Huawei Technologies Co., Ltd  
Shenzhen, China  
liwei563@huawei.com

4<sup>th</sup> Kai Yao  
Huawei Technologies Co., Ltd  
Shenzhen, China  
yaokai14@huawei.com

5<sup>th</sup> Weiguo Tan  
Huawei Technologies Co., Ltd  
Shenzhen, China  
tanweiguo@huawei.com

**Abstract**—Code completion is widely used by software developers to provide coding suggestions given a partially written code snippet. Apart from the traditional code completion methods, which only support single token completion at minimal positions, recent studies show the ability to provide longer code completion at more flexible positions. However, such frequently triggered and longer completion results reduce the overall precision as they generate more invalid results. Moreover, different studies are mostly incompatible with each other. Thus, it is vital to develop an ensemble framework that can combine results from multiple models to draw merits and offset defects of each model.

This paper conducts a coding simulation to collect data from code context and different code completion models and then apply the data in two tasks. First, we introduce an acceptance model which can dynamically control whether to display completion results to the developer. It uses simulation features to predict whether correct results exist in the output of these models. Our best model reduces the percentage of false-positive completion from 55.09% to 17.44%. Second, we design a fusion ranking scheme that can automatically identify the priority of the completion results and reorder the candidates from multiple code completion models. This scheme is flexible in dealing with various models, regardless of the type or the length of their completion results. We integrate this ranking scheme with two frequency models and a GPT-2 styled language model, along with the acceptance model to yield 27.80% and 37.64% increase in TOP1 and TOP5 accuracy, respectively. In addition, we propose a new code completion evaluation metric, Benefit-Cost Ratio(BCR), taking into account the benefit of keystrokes saving and hidden cost of completion list browsing, which is closer to real coder experience scenario.

**Index Terms**—Code completion, neural networks, acceptance model, ranking model, evaluation metrics

## I. INTRODUCTION

Research shows that code completion is one of the most frequently used functions in the IDEs [1]. Code completion tools such as IntelliSense was first released as the main feature of Visual Basic 5.0 in 1996 and activated by default in the Visual Studio 2005 installation package. In recent years, the development of machine learning raises interest in research of the intelligent code completion area. Microsoft and Kite successively released IntelliCode [2] and Intelligent Snippets [3] as code completion plug-ins for multi IDEs,

respectively. TabNine [4] also developed a code completion plug-in based on a deep learning model, which supports over 30 programming languages. When the developer writes code, the code completion engine can infer the potential following code fragments at the cursor position based on the written code context. Thus, it helps developer to improve the efficiency of coding by decreasing typing costs.

Traditional code completion methods embedded in IDEs rely on compile-time type information and specific grammatical rules to predict next tokens [1], [5], which are costly and could not capture human’s programming patterns well. In addition, they only trigger at the limited position with the completion results sorted in the alphabetic order. To address these problems, the concept of intelligent code completion was proposed [6]. With the naturalness of the programming languages had been proved [7], researchers started to apply various learning-based algorithms and train models to learn code characteristics from large-scale codebases. At an early stage, statistical language model [7], such as N-gram, is a promising approach for code completion tools [8], [9]. [10] proposed a Hidden Markov Model to complete multiple tokens at a time with abbreviated input. With the development of deep learning, researchers started to apply Recurrent Neural Networks (RNNs) styled models in source code modelling [5], [11]–[13] in the last five years. However, the performance of RNNs-styled models is limited by their vanishing/exploding gradients and sequential processing mechanism [14]. Most recently, transformers styled pretraining language models have overtaken RNNs and shown their advantages by achieving SOTA results in natural language processing (NLP) and natural language understanding (NLU) tasks [15]–[18]. [19] transfers source code into Abstract Syntactic Tree (AST) based node sequence and predict type and value of next node at the same time with Transformer-XL. [20] introduces IntelliCode based on GPT-2 models to provide up to entire line completion. However, with all the progress, code completion is not without drawbacks as a system.

**a) Displaying too many false-positive suggestions.** Intelligent code completion algorithms can provide completion

results at almost arbitrary positions within the code. Thus, the completion display frequency of these intelligent code completion algorithms increases a lot when the developer is typing code. Such frequently display results can improve coding efficiency by increasing the recall of completion. Meanwhile, it might also cause a drop of precision as too much false-positive suggestion is presented which reduces the coder experience. In general, user experience plays a more important role when developer choosing plug-ins, so it is vital to block the display of completion results when they are invalid.

**b) Lack of fusion ranking methods to combine results from multi code completion algorithms.** Due to the different interests of research, such as the different semantic representation of programming language [21]–[23] and the different completion targets (API calls [12], [24]–[27], AST node [5], [28], entire line [20]), each code completion algorithm has its distinct characteristics. In addition, the traditional code completion also has a different type of results sorted in alphabetical order. Current ensemble methods usually sort the candidates by the priority of the strategy. With all these comprehensive traditional and intelligent completion results, there is a lack of effective methods to integrate them, complement each other, and maximize completion efficiency. Thus, there is still a large room for completion result optimization, including the number of items listed in the completion result and the order of the items in the completion list.

**c) Lack of comprehensive assessment method to evaluate efficiency, including the hidden cost, of code completion tool.** Most tasks are to mask either code tokens or AST nodes in the code file according to certain rules, then using context to predict them. Since the targets of different researches are diverse, they proposed various evaluation metrics. For instance, top n accuracy and mean reciprocal rank (MRR) is commonly used for single token/AST node completion [11], [12], while BLEU-4 and edit similarity are used for multi tokens/full line completion [20]. Such various targets of code completion task make it difficult to compare their overall effects. Moreover, there is a gap between these offline evaluation metrics of the code completion model and the user experience in real scenarios. When using the code completion plug-in, it is impossible to know which part of the code should mask in advance. The accuracy evaluation metrics based on specific locations cannot fully reflect the actual use effect of the code completion tools. It ignores the time cost for developers to check the completion list and the negative impact of frequent invalid lists [29]. This increases the difficulty of promoting the intelligent code completion research achievements in real code completion plug-ins. Therefore, a general method to evaluate the code completion efficiency without adding specific constraints is necessary.

To address the challenges mentioned above, we have conducted extensive investigation and simulation to collect data from code context and different code completion strategies(cf. Section III and IV). Then, we apply the data to optimize the code completion tool from the perspective of making it more practical and user-friendly to developers. The following are

our three main contributions:

**First, we introduce and train an acceptance model which can dynamically control whether to accept the completion results and display them to the developer.** This model uses features extracted from code context and the features of the code completion results from different strategies as input and outputs the probability that whether a correct result is in the aggregate completion list(cf. Section V).

**Second, we design and construct a fusion ranking scheme that can automatically identify the priority of the completion results and reorder the candidates provided by different completion strategies.** This scheme is flexible in dealing with various code completion strategies, regardless of the type or the length of their completion results (cf. Section VI). In addition, our proposed scheme does not involve complex models, and the additional calculation and time-consuming are within a tolerable range. And it is easy to extend to new code completion algorithms and new development languages.

**Third, we design and implement a code completion evaluation method that is closer to real coder experience scenario.** It not only considers many factors such as keystrokes saving and completion list browsing, but also can be universally applied to the evaluation of different completion strategies (cf. Section VII).

Experiments have proved that after using the acceptance and fusion ranking models, even a simple frequency-style code completion model superimposed on a GPT-2 styled model can bring good benefits (cf. Section VIII and IX). At the end, we conclude our study and mention future work (cf. Section X).

## II. MOTIVATION EXAMPLE

Fig. 1 shows two examples to illustrate the motivation of our code completion scheme. Fig. 1(a) presents a code completion system that integrates two different completion strategies, Global Frequency and Local Frequency. These two strategies provide completion candidates with their corresponding scores in the table at the given cursor location. It should be noted that different strategies might provide scores in different scales and dimensions. However, these candidates are all incorrect. Thus, the acceptance model should evaluate the confidence of all the candidates and reject the display of the completion list. In Fig. 1(b), we integrate one completion strategy based on the deep learning model (GPT-2) together with two frequency strategies in the code completion system. The upper table presents a list of the completion candidates and scores provided by each strategy. As the correct results are involved in the candidate list, the acceptance model accepts the completion results and forwards the completion results to the fusion ranking model. The fusion ranking model ranks the completion candidates by their expected benefit shown in the lower table. This expected benefit is evaluated by the accuracy confidence and the token length of each completion candidate.

## III. DATASET

To generate the dataset, we take the Java-small dataset of Alon et al. [30], which is a reselect of the dataset of Allamanis

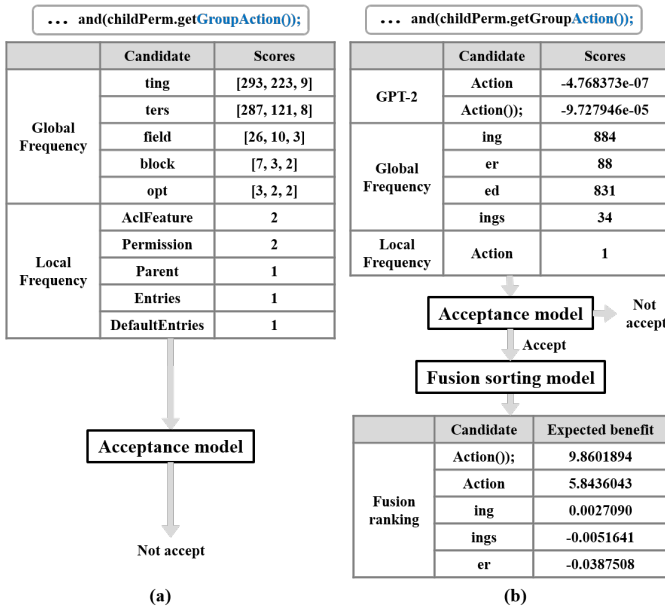


Fig. 1. Completion examples with acceptance and fusion ranking models. Example (a) shows a scenario that no candidate is correct; example (b) shows a scenario that two candidates are correct, the longer one is ranked at the top position

et al. [31]. It involves the most popular eleven Github projects, such as Cassandra, Elasticsearch, Gradle, etc. There are about 96.5k Java source files in total. It removes the overridden, abstract or class constructor methods in every Java file, and rewrite the names of camelCase into snake\_case. We further conduct the following filters:

- Remove the files which name contains the word “test”;
- Remove the methods named “toString”, “equals”, “finalize”, and “clone”;
- Remove the methods longer than twenty lines as they are normally used for configuration or initialization.

Finally, the dataset contains 89.3k/1.6k/3.5k training/simulation/test files. The training dataset is used to train each integrated code completion strategy. The simulation dataset is used to run the simulation scenario and collect completion results from each strategy to train the acceptance and fusion ranking models. Finally, the test dataset is used for the result evaluation.

#### IV. SIMULATION DATA GENERATION

To optimize the completion list from the user experience perspective, we use a simulation method to collect the candidate sets of each completion strategy. Then automatically label validated candidates according to the actual code content at completion position. Considering that the simulation process is relatively time-consuming, we speed up the entire data collection process through parallelism.

##### A. Data generation

In actual application, the more diverse strategies are integrated, the more significant effect of the model ensemble can be observed. We select three recall strategies for optimization:

- Global Frequency provides the completion of common words when developers define new variables or functions.
- Local Frequency captures the local repetition of the variables, classes and methods, which can infer the corresponding item based on the characters you have typed.
- GPT-2 styled language model trained from the large code-base can provide intelligent full line code completion.

Their implementation details will be described in the experiment setup. And the results section also shows the statistical data of their different characteristics.

When entering a code snippet, for instance, in the black font of the Java file in Fig. 2, each strategy will give a completion list. In addition to candidate items themselves, each strategy output scores for its candidates to support the sorting within the strategy. But the meanings of scoring in different strategies are diverse:

- Global Frequency has three scoring dimensions: word count, the number of occurrences in different documents, and projects involved. These statistics are counted from the entire codebase. Word count is used to sorting candidates within the strategy.
- Local Frequency only counts the token occurrences of the code before the cursor in the current file.
- The score of GPT-2 is the cumulative sum of the output of the log-softmax function at each step. The score is decreasing with the increase of the output token sequence length, so the top candidates tend to have a short result.

When different strategies suggest a candidate at the same time, such as “DefaultEntries” in Fig. 2, the scores of it will merge from each dimension for the subsequent tasks.

##### B. Coding simulation

It would be great if the online usage data could be collected from the developers when coding. However, it isn’t easy to collect the candidate selected by the user in real scenarios and the corresponding context due to factors such as personal privacy. In addition, the cold start is also a problem when system builders want to implement new strategies, as they do not have any data for analysis in advance. So we simulate the developers’ coding process, including the typing and the completion selection. Thus we can collect the input code before the cursor, the prediction results from each completion strategy, and the matching labels close to the real scenarios. In addition, because the simulation is offline, the combination of various strategies is possible.

In the process of generating simulation data, we know the actual code context after the cursor, for example, the code snippet in blue font of Java file in Fig. 2. Therefore, we can label each candidate by the prefix matching method. The “Hit” column in the bottom table of Fig. 2 marks the correct completion results as 1 and keeps the others as 0.

For each file in our simulation dataset, we conduct the simulation by moving the cursor from the beginning to the end by one character per step. We generate one sample at each step. This sample includes the input code snippet (the code context

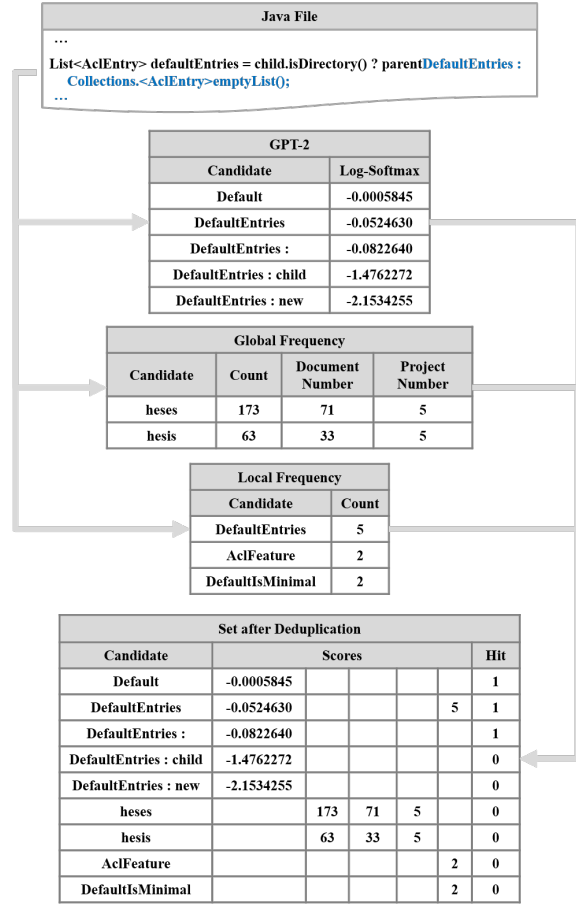


Fig. 2. Example simulation result at a certain location. The black font in the Java file is the code fragment before the cursor. The blue font is the code snippet after the cursor, which the target to predict. The three tables in the middle show the original output of the different strategies. The bottom table shows the aggregate completion candidates with a "Hit" column. If one completion candidate matches the target, we set its corresponding value in the "Hit" column as 1, otherwise 0.

before the cursor) and its corresponding set candidates from integrated strategies. Thus, the number of simulation samples generated from a file is equal to the length of the file.

### C. Critical position and Non-critical position

To emphasize the samples that most likely to appear in real scenarios, we define critical/non-critical positions in the code file and classify the simulation samples by the types of their corresponding trigger positions. Critical position means the position where the code completion request is more likely to be triggered. We filter the simulation data samples by dynamically skipping the non-critical positions where their corresponding characters are generated by the completion tool. The remaining positions are marked as critical positions as they are where the code completion is triggered in real scenarios. Fig. 3 presents the critical position with underlined characters. It should be noted that the critical positions can vary when applying different completion strategies as they depend on the completion results at each position. In this work, we mainly use simulation samples at the critical positions for further analysis as they are more representative. In our simulation, the proportion of critical positions is about 23.17%. We also

```
...
List<AclEntry> defaultEntries = child.isDirectory() ? parentDefaultEntries :
Collections.<AclEntry>emptyList();
...
```

Fig. 3. Training samples at critical position. The red underlined characters are the critical positions.

found that the completion accuracy is as high as 90% when considering all positions, while it drops to 55% when only considering critical positions. It indicates that the chosen critical positions are crucial for the overall code completion evaluation.

### D. Speed up through parallelism

Since the time to simulate a single file is at the level of minutes or more, we deploy each completion strategy as services that can be scalable. And the interaction between master and worker is asynchronous. Master will take out an idle worker from the pools and then send a file-level simulation task request. The choice of server cluster size is generally related to the number of code files involved in the simulation and the efficiency of the strategy to generate the completion list. We used 20 EC2s from cloud service with the machine specifications of s3.xlarge.2 to generate results for the simulation and test dataset. To simulate the data for the code files mentioned in Section III, it took about 2.5 days with the parallel mechanism, significantly shorter than 50 days with only one worker.

## V. ACCEPTANCE MODEL

In general, the increase in the number of integrated completion strategies will raise the trigger probability of code completion. Although the recall is improved, it also produces more false-positive completion results. These invalid results will superimpose additional costs that reduce development efficiency. So we design a module to judge whether or not to accept the completion list by evaluating its confidence. If the module accepts the completion result, the result will display at the corresponding position; otherwise, the module will block the completion result.

### A. Module design with candidate-set level samples

It is time-consuming to manually design a set of acceptance rules for this module based on expert experience. In addition, if the integrated completion strategies are changed, it is difficult to update the bespoke rules in time. Therefore, we design an acceptance model, adopting the method of binary classification modelling. This model is trained by the simulation samples at critical positions (Section IV-C) to better fit with coder experience. When the cumulative of "hit" columns (Fig. 2) in a candidate set is greater than 0, label this completion action as a positive sample; otherwise it is negative.

### B. Model algorithm

To construct this binary classification model, we use Autogluon package [32] in the pipeline of data preprocessing, model selection and hyperparameter tuning. Autogluon can train models with commonly used classification algorithms.

TABLE I

OFFLINE EVALUATION RESULTS OF DIFFERENT CLASSIFICATION MODELS TRAINED BY AUTOGLUON

Model	Accuracy	Pred Time (s)	Train Time (s)
Weighted_ensemble	93.49%	3.331	3881
RandomForestClassifier	92.98%	0.246	77
ExtraTreesClassifier	92.45%	0.446	62
CatboostClassifier	89.24%	0.114	23
NeuralNetClassifier	91.02%	1.836	3586
LightGBMClassifier	90.94%	0.069	5
KNeighborsClassifier	90.36%	4.643	488

Moreover, it can construct ensemble models with multiple algorithms and different hierarchical structures. Table I shows the offline evaluation results of six classification models and one ensemble model trained by Autogluon with our training dataset for the acceptance model. In addition, the table presents the classification accuracy, prediction time and training time of each model. Considering the time constraint in the real-time code completion scenario, we finally chose the LightGBM algorithm. LightGBM is a distributed gradient boosting framework based on the decision tree algorithm, providing fast and accurate prediction with a low memory footprint. To further improve the online inference performance, we apply the native LightGBM [33] in our online service and reduce the prediction time to less than 50ms.

### C. Feature importance for acceptance model

The features we used are from two sources: the code context to capture position and structure features from the existing code and the completion results to capture features from each code completion model.

1) *Code context features*: Extract context features from the code before the position to be completed, mainly from the following aspects:

- Position: line number, token number of the line, etc.
- Uncompleted token: prefix length, capitalized token, etc.
- Adjacent tokens: last token, last symbol, etc.

2) *Completion result features*: Collect features from completion results of each integrated strategy. Some additional features are generated by aggregating the results from all strategies.

- Feature from each strategy: candidate number, score of each candidate, candidate length, etc.
- Aggregate features: the times of simultaneous occurrences of a candidate in multiple strategies, etc.

Fig. 4 presents the top fifteen critical features in the LightGBM model, the importance score denotes the number of times the feature used in this model. It can be noticed that the top three important features come from the current line code context. The adjacent token and symbol play an essential role as they are most related to the code to be predicted. On the other hand, eleven out of these fifteen features are from the completion results, especially from the length and scores of the top two candidates of each strategy. In addition, one aggregate feature, the times of simultaneous occurrences of a candidate in multiple strategies, is presented as the fourteenth important feature.

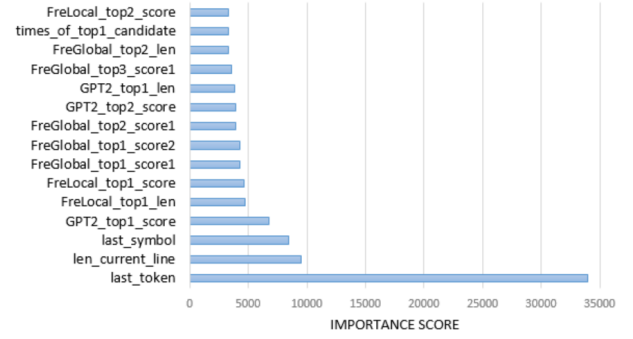


Fig. 4. Feature importance of the acceptance model

## VI. FUSION RANKING MODEL

When a correct candidate does not present at the top of the completion list, the developer has to increase the checking time to scan all the invalid candidates above it and increase the tap times to select it. Thus, the position of the correct candidate in the list significantly affects the coding efficiency. When the code completion system has multiple strategies, the candidate scores given by different strategies are usually on various scales, and sometimes one strategy might have a multi-dimensional score. To display the candidates from all integrated strategies in an ordered list, adjusting the candidates' priority is needed.

### A. Module design with candidate level samples

Unlike the candidate-set level sample granularity of the acceptance module, we extract candidate level samples from the simulation data at critical positions for this fusion ranking module. In this manner, one completion candidate is treated as a sample. Thus, the total number of samples will expand dramatically. Assuming the acceptance model has blocked the completion without any correct candidate, we only select the samples from the completion list with the correct candidate, which also solves the problem of the unbalanced samples.

Similar to the acceptance model's sample features, each candidate sample in the ranking model has code context features and features related to the candidate itself. The goal of the ranking model is to give a score as a unified measurement index to each candidate sample from different strategies. This score then ranks all the candidates in a completion list.

### B. Model algorithm

To rank candidates from multi strategies, we have explored the following two methods:

1) *Normalized Ranking*: The first method we considered is to normalize the candidate scores from each strategy by the StandardScaler function in the scikit-learn package [34]. It scales each input score separately by subtracting the mean and dividing by the standard deviation to shift the distribution to a mean of zero and a standard deviation. Then we rank all candidates by this normalized score. However, the top-ranking candidates tend to be more accurate and shorter in length with this method as the candidate length factor is not considered here.



```
import org.apache.hadoop.fs.permission.AclEntryScope;
```

Index	Candidate	Expected benefit
1	permission.Acl	13.5857870
2	permission.AclStatus	11.59410973
3	permission.AclStatus;	11.43501475
4	permission.	10.9402858
5	permission	9.9599077

Fig. 5. Example of fusion ranking module

2) *Fusion Ranking*: It is not easy to introduce a length factor [20], [35] into the ranking score and to achieve a compromise between the two indicators. We innovatively introduce the concept of expected benefit together with a regression model to deal with this problem. In this regression model, the input is the feature vector of a candidate, and the output is the expected completion length of this candidate. More specifically, for the correct candidate, use its length as the model's output; for the wrong candidate, set the output as 0 directly. The physical meaning of this method represents the potential benefits of choosing this candidate. If the candidate is wrong, there is no help to improve the coding efficiency, and when it turns out to be correct, the longer the candidate's length will bring more significant benefit.

We still choose the LightGBM algorithm to train this regression model with simulation data. The output of the regression model inherently considers both the correct probability and the length of the candidate. Thus, our fusion ranking module takes the output of the regression model as the final score and ranks candidates by this score. Fig. 5 is an example of the fusion ranking module. It can be seen that the output score from the regression model believes that compared with candidates 4 or 5, candidate 1 has a greater profit of giving longer completion compared to its risk of accuracy. In contrast, the risk of accuracy overrides the length benefits for candidates 2 and 3. In this manner, the fusion ranking module gives the top-ranking place to the correct candidate with the longest length.

## VII. EVALUATION METRICS CONSIDERING HIDDEN COST

In this section, we provide a novel comprehensive evaluation method. Unlike the simple accuracy metrics used in the previous code completion research, our metric takes into account the benefits from the code completion, meanwhile considers the hidden cost of it. Thus, this metric is closer to the real coder experience.

### A. Common metrics

The following performance indicators are commonly used to evaluate code completion models and algorithms: in academia, Accuracy of top K completion results (Accuracy@K) [36] and Mean Reciprocal Rank (MRR) [37], [38] are representative evaluation metrics. On the other hand, the keystroke is commonly used in industry to evaluate code completion systems, such as the well-known code auxiliary tools aiXcoder [39], Kite [3] and TabNine [4].

In this paper, we use Accuracy@K as the common metric for evaluating the effect of code completion. The definition formula is as follows:

$$Accuracy@K = \frac{\sum_{i \in W} isHit_{(i,K)}}{|W|} \quad (1)$$

$W$  represents the set of locations that have completion results in the test code file. For each location  $i$  in  $W$ , the value of  $isHit_{(i,K)}$  is 1 when the correct completion result is in the top  $K$  results of the completion list. Otherwise, the value is 0.

### B. Benefit-Cost Ratio

In the industry, when propagating the effect of the completion plug-in, it is more inclined to use the concept of the keystrokes, which is the number of times that a human needs to type to complete a document. Therefore, if the code completion tool can significantly reduce the keystrokes, coding can be more efficient.

Although the accuracy index can compare the performance of some models to some extent, it is incapable of evaluating models with different types of output. For example, the model with single token output generally has higher accuracy than the model with multi-token output. On the other hand, the keystrokes benefit reflects the extent to which code completion tools improve user efficiency. However, it ignores the hidden cost of browsing the completion list. To solve these issues, we propose our comprehensive evaluation metric, Benefit-Cost Ratio(BCR), for the code completion task.

1) *Benefit*: First we define the *Benefit*, which is the number of keystrokes reduced by using the code completion tool, with the following formula:

$$Benefit = N_{ori} - N_{cc} \quad (2)$$

where  $N_{ori}$  is the number of characters in the test file, more specifically, it means the keystrokes without using the code completion tool;  $N_{cc}$  represents the number of actual keystrokes with the usage of code completion tool. It should be noted that  $N_{cc}$  defined here does not involve the taps used to selecting the right answer in the completion list as this will be counted in the hidden cost.

2) *Hidden Cost*: Code completion tools not only improve development efficiency but also bring inevitable usage costs to developers. These extra costs, such as the time to browse the completion list and the keystrokes cost to select the correct answer, are often implicit and ignored by previous evaluation metrics. From a practical perspective, we only consider the time to browse the completion list in our work. The costs of the keystrokes to select the correct answer are negligible as they often happen during the browsing procedure and do not need much mental work.

Assuming developers scan the completion list in a top-down manner and the effect of evaluating each candidate is identical, we can define the browsing cost by the following rules:

- if the correct answer is in the completion list, the browsing cost is equal to the ranking of the correct answer in

the completion list as the developer would evaluate all the invalid answer above it. In case of more than one correct answers appeared in the completion list, we use the position of the longest correct answer;

- if none result in the completion list is correct, the browsing cost is equal to the length of the completion list, as the developer would have to evaluate all the invalid candidates in the list.

Thus, we define the *HiddenCost* with Eq. (3):

$$HiddenCost = \sum_{i \in Hitset} HitPos_i + \sum_{j \in (W - Hitset)} ListLen_j \quad (3)$$

where *Hitset* represents a set of locations in the test files where the correct result is in their completion lists; *W - Hitset* indicates the set of locations where all the completion results in the completion list are invalid; The *HitPos<sub>i</sub>* indicates the longest correct result position in the completion list at location *i* of the test file; *ListLen<sub>j</sub>* indicates the length of completion list when there is no correct result in the completion list at location *j* of the test files.

Based on Eq. (2) and (3), BCR can be defined with Eq. (4).

$$BCR = \frac{Benefit}{HiddenCost} \quad (4)$$

This equation considers the total benefit brought by the code completion tool together with its hidden cost. The physical meaning of BCR can be roughly regarded as the average saving keystrokes by browsing one candidate in the completion list during the coding process. Taking Fig. 5 as an example, the top 1 candidate is correct and has a length of 14. If the developer browses and select this candidate, the browsing cost is 1 and the tap times is 1. It saves 13 keystrokes comparing to manually tap 14 times to complete this token, so the benefit is 13. Thus, the BCR is 13/1 in this case. A higher value of BCR indicates the lower the effort required for the developer and the better the overall effect of the code completion tool.

## VIII. EXPERIMENT SETUP

This section introduces the implementation details of the three strategies (two frequency models and a GPT-2 model) and their corresponding parameter configuration during the experiment. The data set and the evaluation method have been introduced in Section III and Section VII) respectively. Thus, this section focuses on the setup of the code completion strategies. Each strategy provides up to five candidates to our acceptance and fusion ranking models for the subsequent tasks.

### A. Frequency strategies

Traditional code completion tools usually provide completion list sorted alphabetically. Such kind of list requires an extra cost to the developer to find the correct result. Sorting by the probability of the candidate occurrences in the historical codebase can improve the relevance of the suggestions, and it can automatically adapt to the changes in the codebase. Thus, we introduce two word-frequency-statistics methods in our experiment. They are integrated as different completion

strategies to provide candidates for our acceptance model and fusion sorting model.

1) *Global Frequency*: When defining variables or functions, developers usually name them with multi-words linked by camel-case or underscore to explain the specific meaning. If we perform code segmentation by token granularity, the long tail effect of the programming language will be more evident than the general natural language. Therefore, we adopt a finer-grained sub-token segmentation method [27] to further segment tokens when encountering underscore or camel case naming rules. In addition, we filter out the sub-tokens that only appear in one project, and the length is smaller than 5. As a result, the vocabulary size is dramatically reduced from 600,000 for the token level segmentation to 15,916 after sub-token level segmentation and the filter. Such vocabulary is sufficient to reflect the common words used in the coding process. Besides the keywords and basic types, the most frequently used tokens are (1) Terms that often appear in Java file header comments such as "License", "Version", and "Apache", (2) Words that often appear when importing packages such as "Service", "Manager", and "kernel". It is also noted that "result" has the highest probability as the variable name, and the "append" method has the highest usage rate.

We store the entire vocabulary with a tire tree to accelerate the online inference. For each token, the total number of occurrence in the training set, the number of involved files, and the number of involved projects are recorded as the three dimensions of token scores. Candidates are obtained by searching the tire tree and then sorted according to the total number of occurrences.

Global Frequency strategy is mainly used when defining new variables, function names, and classes. It can effectively complete the commonly used words in the code file.

2) *Local Frequency*: The programming language often has distinct localization characteristics [40]. When coding, developers may repeatedly call the variable or function defined in the previous code fragment or the custom class under the same project. Therefore, we use the Local Frequency strategy to capture and predict the local characteristics. Local Frequency strategy extracts the token-level vocabularies from the current code file, meanwhile, counts the number of occurrence of each token as the score. During the online inference, these vocabularies are dynamically updated with the written code file. Candidates are obtained by searching the vocabulary directly and then sorted according to the number of occurrences.

As an illustration, we select a representative file ("AclStorage.java") from the testing dataset with 689 tokens and about 8000 characters. Among 689 tokens, 488 tokens have appeared in the code before their location. In other words, the recall rate is 69.91%. Table II shows the results sorted by the number of token occurrences. It can be seen that the top-ranking tokens repeatedly appear in the code file. In addition, further analysis shows that in the case of the Local Frequency strategy hits, there is about half chance that the correct candidate can be found at the top one position of the completion list of Local Frequency strategy.

TABLE II  
NUMBER OF TOKEN OCCURRENCES IN A JAVA FILE

Index	Token	Count	Index	Token	Count
1	AcEntry	30	6	FsPermission	15
2	accessEntries	27	7	import	14
3	List	20	8	permission	14
4	inode	19	9	featureEntries	13
5	if	19	10	new	13

TABLE III  
GPT-2 MODEL ARCHITECTURE HYPER-PARAMETERS

Hyper-parameter	Explanation	Value
n_layer	Number of transformer layers	12
n_model	Dimension of hidden states	768
n_embd	Dimension of embedding	768
n_head	Number of attention heads	12
n_positions	Max code token length	256
p_drop	Drop probability	0.1
vocab_size	Vocabulary size	30000

### B. Sentence-level language model

With their self-attention and paralleled processing mechanism, the newly designed transformers present superiority over RNNs in modelling the long-term dependency and the inference speed. GPT-2 [16], as a transformers styled pre-train language model, has demonstrate its strong ability in both text and code generation area [20]. Thus, we use GPT-2 styled model as the sentence-level language model in our research.

1) *Model structure*: GPT-2 consists of a multi-layer transformer decoder stack, which maps input token embedding and position embedding into an output vector. The output vector is then multiplied with the token embedding matrix and forward into a log-softmax function to calculate the prediction score for each vocabulary token. We train our GPT-2 model from scratch with the training dataset.

Considering the trade-off between accuracy and performance, we select a small GPT-2 124M model and apply a minor modification to build our own GPT-2 model for the code completion task. The maximum sequence length is reduced from 1024 to 256, and the vocabulary size is reduced from 50257 to 30000. Final model hyper-parameters are shown in Table III with 108M parameters in total.

2) *Preprocessing*: We apply the following preprocessing steps to the source code, including:

- Partially remove comments, only keep comment lines before the function definition and docstring below the function definition;
- Remove non-English letters and symbols;
- Replace long string/number with special placeholders;

3) *Tokenizer*: The tokenizer is used to encode a code literal into a sequence of tokens before feeding it into GPT-2 model and decode the model output back to a code literal. We use the same tokenization method, Byte-Pair Encoding (BPE), with the original GPT2 model. It is an unsupervised tokenizer, which recursively replaces the most frequently occurring pair of Unicode characters with a new character in the vocabulary. We retrain the BPE tokenizer from scratch using our source code data and set the vocabulary size as 30000. Since this

customized tokenizer provides a better fit with source code, it can consider more extended source code than the original GPT-2 tokenizer with the same number of tokens.

4) *Inference*: The maximum online inference time for GPT-2 model is set as 200ms to ensure displaying completion results to the developer fluently. We apply the beam-search method (Fig. 6) in the GPT-2 model inference procedure to generate the code completion result. Beam-search reduces the risk of missing high probability token sequences by keeping the top  $k$  ( $k$  is the beam size) highest aggregate probability sequences at each step. To reduce the redundant calculation and accelerate the inference during the beam-search, we apply the following output sequence termination criterion to adjust the beam size dynamically at each step:

- if its aggregate score is lower than a score threshold  $t$ .
- if it has end-of-line token (" $\backslash$ n").
- if it has an annotation symbol.
- if it runs into closed loop.
- if it reaches the maximum inference step.

Fig. 6 illustrates one of our beam-search example with beam size  $k$  and score threshold  $t$  being set as 5 and -3 respectively. At step 0, the top 5 score candidates are selected and combined as a batch for the step 1 inference. At step 1, we also take the top 5 candidates with the highest aggregate score, in which only three candidates have an aggregate score higher than the threshold  $t$ . Thus, only three candidates are selected for the next step inference, which means the batch size of step 1 input is 3. The rest inference steps are conducted in the same manner until our criterion terminate all beam search process. Thus, instead of doing beam-search with fixed batch size inference at each step, we dynamically adjust the batch size according to the previous step result to reduce the inference cost.

Given the increment sequence typing nature of the coding process, we also implement two caching mechanisms in the inference stage. Firstly, we cache the input sequence with its attention keys and values of the GPT-2 blocks. If a new inference request is received, we only calculate the keys and values for the additional transformer block if the input sequence partially matches the items in the cache. This caching speeds up inference by up to 40%. Secondly, we cache the inference sequence results for the current line. When the beam-search procedure cannot finish in the time constraint (200ms), we return the results from the processed step. Meanwhile, we keep the beam-search running in the background until it reaches the termination criterion. The untapped results are saved in cache and used if a new inference with a matching prefix is received.

## IX. RESULT

This section conducts a series of experiments of three chosen code completion strategies with our acceptance model and fusion sorting model and presents a comprehensive analysis based on the experimental results. The data used for the evaluation is generated from the test dataset introduced in Section III and IV.



TABLE IV  
CHARACTERISTICS OF DIFFERENT CODE COMPLETION STRATEGIES

Strategy	Occurrence rate	Hit position 90% quantile	Prefix length 90% quantile	Completeness	Accuracy@1	Accuracy@5
Global Frequency	47.33%	TOP2	12	65.80%	9.74%	11.18%
Local Frequency	12.48%	TOP4	2	95.03%	31.82%	52.67%
GPT-2	8.20%	TOP5	9	88.78%	58.96%	62.79%

#### A. Single strategy evaluation

Table IV shows the comparison of the characteristics of different strategies from the following aspects.

- Occurrence rate: the proportion of non-empty completion list.
- Hit position 90% quantile: when the completion list has the right candidate, collect the positions of the longest hit candidate. Sort the collected position in ascending order and take the value of 90th quantile to avoid discrete points.
- Prefix length 90% quantile: prefix length means how many characters of a token you need to tap before the correct result appears in the completion list. We also use the value of the 90th quantile.
- Completeness: the proportion of the chosen candidate is a full token.
- Accuracy@K: as described in Eq. (1).

1) *Global Frequency*: Global Frequency is able to provide results at almost 50% of the locations while has the lowest accuracy among these three strategies. The long requirement of the prefix length helps the Global Frequency strategy hit the correct result at a relatively top position as the extended prefix limits the number of candidates with solid constraint. When there is a hit item in the completion list, the correct completion result is ranked in the top 2 positions in 90% of the cases. In addition, since the Global Frequency strategy segments the code with sub-token level granularity, its completeness is lower than the other two strategies. Most of the correct results of the Global Frequency strategy are the keywords, common types, and the customary name of variables and methods.

2) *Local Frequency*: Local Frequency represents the localization characteristic of the program coding. Table IV shows that, although the occurrence rate of the Local Frequency model is lower than that of the Global Frequency model, an input with only two prefix characters can help the Local Frequency strategy to find the correct completion result in 90%

of the cases. It also has the highest completeness, 95% of the time, Local Frequency Strategy can complete an entire token.

3) *GPT-2*: The completion list of GPT-2 has the highest Accuracy@1 and Accuracy@5, with Accuracy@5 reaching 62.79% compared to 52.67% and 11.18% for the Local and Global Frequency strategies, respectively. The occurrence rate is relatively low as we manually limit the trigger condition of the GPT-2 strategy. The ranking method for GPT-2 itself is to sort by the log-softmax score of each candidate, while the shorter candidate generally has a higher log-softmax score. Our fusion ranking model will amend this improper ranking method. In addition, by blocking the low-confidence completion list with our acceptance model, the accuracy can be further optimized.

#### B. Benefit-Cost Ratio (BCR)

BCR is the bespoke evaluation metric (defined in Section VII) we proposed for the code completion task. It considers both the benefit and the hidden cost when using the code completion tool in a real scenario. The last row ("Acceptance+Fusion Ranking") of Table V presents the Accuracy@k and BCR results of our code completion scheme with the acceptance model and fusion ranking model. Our code completion scheme achieves a BCR value of 3.65, which is higher than any single strategy. It means that the developer can averagely save 3.65 keystrokes by browsing one candidate in the completion list when using our code completion scheme.

It is noted that the BCR of the Global Frequency strategy is very poor. Such performance is consistent with the actual experience. The high trigger and low accuracy completions from the Global Frequency are dazzling. The other interesting finding is that the BCR value of our completion scheme (3.65) is higher than the sum of the BCR value of the three integrated strategies themselves (3.34), indicating that our completion scheme can draw merits and offset defects from each integrated strategy.

Fig. 7 shows the completion effect of a code snippet in a test file. The abscissa of each subgraph represents the locations

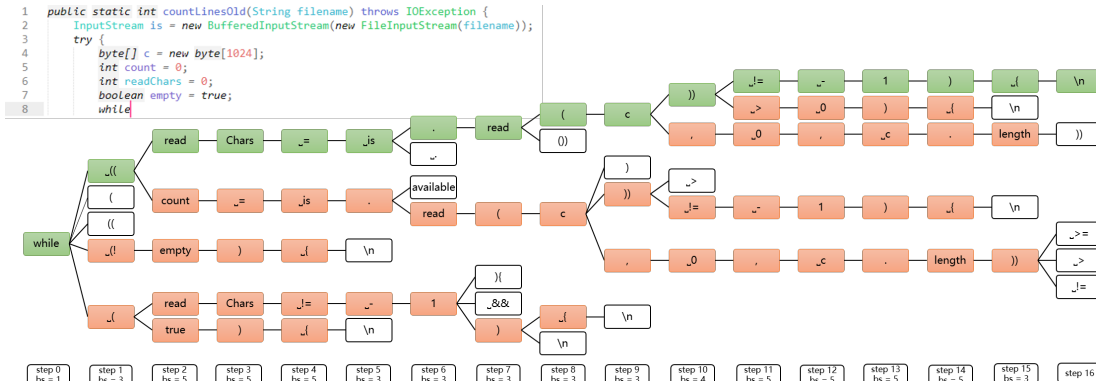


Fig. 6. GPT-2 beam-search. The green branch is the one with highest aggregate probability; the white blocks are the ones match the termination criterion

TABLE V  
EXPERIMENTAL RESULT OF THE EVALUATION METRICS

Strategy	Accuracy@1	Accuracy@5	Benefit-Cost Ratio
Global Frequency	9.74%	11.18%	0.09
Local Frequency	31.82%	52.67%	1.11
GPT-2	58.96%	62.79%	2.14
Acceptance+Fusion Ranking	<b>62.61%</b>	<b>82.55%</b>	<b>3.65</b>

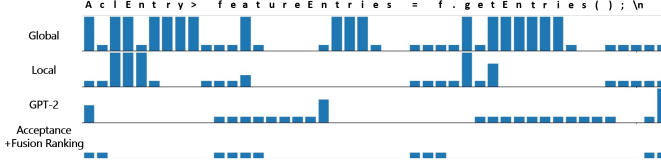


Fig. 7. The real cost to write a code snippet

of successive code characters in the code snippet. The height of the histogram represents the coding cost of the current position, and the cost value ranges from 0 to 6. The number 0 means no need to perform any action at this position as this character is completed by the previous position. Value 1 may mean that this character is manually tapped, or the top 1 candidate in the completion list is hit. Value 2 to 5 are similar to value 1, and they are just different in the order of the hit candidate position in the completion list. Value 6 means that there is no correct option in the completion list. The developer has to browse five invalid candidates and manually type this character. It can be seen that our completion scheme with the acceptance model and the fusion sorting model can significantly reduce the cost when writing this code snippet.

### C. Ablation study

Since our proposed code completion scheme involves two subtasks, we conduct a set of comparative experiments to evaluate the effect of each task. The results are presented in Table VI. Following strategies are used in this study:

- Normalized Ranking: a baseline ranking method introduced in Section VI-B
- Fusion Ranking: only use fusion ranking model
- Acceptance+Normalized Ranking: use acceptance model and Normalized Ranking method
- Acceptance+Fusion Ranking: use acceptance model and fusion ranking model

From *FusionRanking* to *Acceptance + FusionRanking*, there is a dramatic improvement in Accuracy@K and BCR metrics, which explains the function of the acceptance model. To verify the effect of the fusion ranking model, we compare *Acceptance + NormalizedRanking* with *Acceptance + FusionRanking*. By replacing the Normalized Ranking with Fusion Ranking, BCR increases from 2.14 to 3.65, which shows that the fusion ranking model also plays an essential role in our code completion scheme.

TABLE VI  
COMPARISON RESULTS OF DIFFERENT COMBINATIONS

Strategy	Accuracy@1	Accuracy@5	Benefit-Cost Ratio	Invalid list
Normalized Ranking	34.81%	44.91%	1.20	55.09%
Fusion Ranking	35.86%	50.35%	1.57	49.65%
Acceptance+Normalized Ranking	57.31%	71.80%	2.14	28.20%
Acceptance+Fusion Ranking	<b>62.61%</b> (27.80% ↑)	<b>82.55%</b> (37.64% ↑)	<b>3.65</b>	<b>17.44%</b>

We also count the probability of the invalid completion lists with the value shown in the *Invalid list* column. If the acceptance model is not integrated, the likelihood of invalid triggers is nearly 50%. Such frequently display of the false result increases the expense of declined user experience. After using the acceptance model, the invalid trigger rate reduces significantly. Combining with the fusion ranking model, the probability of *Invalid list* is further reduced to 17.44%.

Compared with the baseline strategy "Normalized Ranking", Accuracy@1 and Accuracy@5 of "Acceptance+Fusion Ranking" are increased by 27.80% and 37.64%, respectively.

## X. CONCLUSION

We have introduced and deployed a code completion scheme capable of integrating the results from multiple completion strategies. Within this scheme, we introduced two models: the acceptance model and the fusion ranking model. The acceptance model uses features extracted from the code context and the results from different code completion strategies to predict whether a correct result is in the completion list. It can dynamically control whether to accept the completion results and display them to the developer. The fusion ranking model can automatically identify the priority of the completion results and reorder the candidates provided by different completion strategies. This scheme is flexible in dealing with various code completion strategies, regardless of the type or the length of their completion results. In addition, we have proposed a comprehensive code completion evaluation metric BCR, which considers both the benefit of the keystrokes saving and the cost of completion list browsing.

We have integrated our code completion scheme with two frequency style models and a GPT-2 style language model to conduct a set of comprehensive experiments. The results prove that our code completion scheme can achieve solid improvements. With the acceptance and fusion ranking models, the proportion of invalid completion list reduces from 55.09% to 17.44%. Meanwhile, the TOP1 and TOP5 accuracy increase by 27.80% and 37.64%, respectively. The BCR value is 3.65, which indicates the developer can averagely save 3.65 keystrokes by browsing one candidate in the completion list when using our code completion scheme.

To the best of our knowledge, we are the first to optimize the multi-path recall strategies in the code completion task, showing a significant improvement over the single completion strategy. Our exploration may not be comprehensive enough, but we hope to expand new ideas for research in this area. For example, integrating more code completion strategies with different characteristics in the ensemble task may bring more apparent benefits. Trying multi-task joint training or reinforcement learning when optimizing the ensemble model is also an exciting research task. In terms of evaluation methods, we are also expecting to discover more potential influencing factors and combinations. A suitable evaluation method can, in turn, promote the development of algorithms.

## REFERENCES

- [1] Gail C Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the eclipse ide? *IEEE software*, 23(4):76–83, 2006.
- [2] Amanda. Introducing visual studio intellcode. <https://devblogs.microsoft.com/visualstudio/introducing-visual-studio-intellcode/>, 2018. Accessed: 2021-06-16.
- [3] Kite: Free ai coding assistant and code auto-complete plugin. <https://www.kite.com/>. Accessed: 2021-06-16.
- [4] Tabnine: Code faster with ai completions. <https://www.tabnine.com/>. Accessed: 2021-06-16.
- [5] Jian Li, Yue Wang, Michael R Lyu, and Irwin King. Code completion with neural attention and pointer networks. *arXiv preprint arXiv:1711.09573*, 2017.
- [6] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 213–222, 2009.
- [7] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016.
- [8] Daqing Hou and David M Pletcher. Towards a better code completion system by api grouping, filtering, and popularity-based ranking. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, pages 26–30, 2010.
- [9] Vincent J Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 763–773, 2017.
- [10] Sangmok Han, David R. Wallace, and Robert C. Miller. Code completion from abbreviated input. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 332–343, 2009.
- [11] Yixiao Yang and Chen Xiang. Improve language modelling for code completion through learning general token repetition of source code. In *SEKE*, pages 667–777, 2019.
- [12] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. Pythia: Ai-assisted code completion system. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2727–2735, 2019.
- [13] Gareth Ari Aye and Gail E Kaiser. Sequence model design for code completion in the modern ide. *arXiv preprint arXiv:2004.05249*, 2020.
- [14] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.
- [15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
- [16] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.
- [17] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.
- [18] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.
- [19] Fang Liu, Ge Li, Bolin Wei, Xin Xia, Zhiyi Fu, and Zhi Jin. A self-attentional neural architecture for code completion with multi-task learning. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 37–47, 2020.
- [20] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellcode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1433–1443, 2020.
- [21] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. *ACM SIGPLAN Notices*, 53(4):404–419, 2018.
- [22] Marc Brockschmidt, Miltiadis Allamanis, Alexander L Gaunt, and Oleksandr Polozov. Generative code modeling with graphs. *arXiv preprint arXiv:1805.08490*, 2018.
- [23] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.
- [24] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642, 2016.
- [25] Jaroslav Fowkes and Charles Sutton. Parameter-free probabilistic api mining across github. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pages 254–265, 2016.
- [26] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. Exploring api embedding for api usages and applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 438–449. IEEE, 2017.
- [27] Alexey Svyatkovskoy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Franco, and Miltiadis Allamanis. Fast and memory-efficient neural code completion. *arXiv preprint arXiv:2004.13651*, 2020.
- [28] Pavol Bielik, Veselin Raychev, and Martin Vechev. Phog: probabilistic model for code. In *International Conference on Machine Learning*, pages 2933–2942. PMLR, 2016.
- [29] Xianhao Jin and Francisco Servant. The hidden cost of code completion: Understanding the impact of the recommendation-list length on its efficiency. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 70–73, 2018.
- [30] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.
- [31] Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*, pages 2091–2100. PMLR, 2016.
- [32] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. Autogluon-tabular: Robust and accurate autogl for structured data. *arXiv preprint arXiv:2003.06505*, 2020.
- [33] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems*, 30:3146–3154, 2017.
- [34] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [35] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google’s neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.
- [36] Stephen Boyd, Corinna Cortes, Mehryar Mohri, and Ana Radovanovic. Accuracy at the top. 2012.
- [37] Dragomir R Radev, Hong Qi, Harris Wu, and Weiguo Fan. Evaluating web-based question answering systems. In *LREC. Citeseer*, 2002.
- [38] Ellen M Voorhees et al. The trec-8 question answering track report. In *Trec*, volume 99, pages 77–82. Citeseer, 1999.
- [39] AIXCoder: Leave artificial intelligence to AIXCoder. leave real intelligence to human. <https://www.aixcoder.com/>. Accessed: 2021-06-16.
- [40] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 269–280, 2014.