Empirical Software Engineering (2022) 27:91
https://doi.org/10.1007/s10664-022-10140-7

# A unified multi-task learning model for AST-level and token-level code completion

Fang Liu[1,2] · Ge Li[1,2] · Bolin Wei[1,2] · Xin Xia[3] · Zhiyi Fu[1,2] · Zhi Jin[1,2]

Accepted: 22 February 2022

**Abstract**
Code completion, one of the most useful features in the Integrated Development Environments (IDEs), can accelerate software development by suggesting the next probable tokens based on existing code in real-time. Recent studies have shown that recurrent neural networks based statistical language models can improve the performance of code completion tools through learning from large-scale software repositories. However, most of the existing approaches treat code completion as a single generation task in which the model predicts the value of the tokens or AST nodes based on the contextual source code without considering the syntactic constraints such as the static type information. Besides, the semantic relationships in programs can be very long. Existing recurrent neural networks based language models are not sufficient to model the long-term dependency. In this paper, we tackle the aforementioned limitations by building a unified multi-task learning based code completion model for both AST-level and token-level code completion. To model the relationship and constraints between the type and value of the code elements, we adopt a multi-task learning framework to predict the type and value of the tokens (AST nodes) simultaneously. To capture the long-term dependency in the input programs, we employ a self-attentional architecture based network as the base language model. We apply our approach to both AST-level and token-level code completion. Experimental results demonstrate the effectiveness of our model when compared with state-of-the-art methods.

**Keywords** Code completion · Deep learning · Multi-task learning

## 1 Introduction

As the complexity and scale of the software development continue to grow, code completion has become an essential feature of Integrated Development Environments (IDEs). It speeds

Communicated by: Yann-Gaël Guéhéneuc, Shinpei Hayashi and Michel R. V. Chaudron

This article belongs to the Topical Collection: *International Conference on Program Comprehension (ICPC)*

✉ Ge Li
lige@pku.edu.cn

Extended author information available on the last page of the article.

Published online: 18 April 2022

up the process of software development by suggesting the next probable token based on existing code. Based on the observation of source code's repeatability and predictability (Hindle et al. 2012), statistical language models are generally used for code completion in recent years. N-gram is one of the most widely used language models (Hindle et al. 2012; Tu et al. 2014; Hellendoorn and Devanbu 2017). Most recently, as the success of deep learning, source code modeling techniques have turned to Recurrent Neural Network (RNN)-based models (Bhoopchand et al. 2016; Li et al. 2018; Karampatsis et al. 2020). In these models, a piece of source code is represented as a source code token sequence or an Abstract Syntactic Tree (AST) node sequence. Given a partial code sequence, the model computes the probability of the next token or AST node and recommends the one with the highest probability. However, these models are limited from the following aspects:

a) **The hierarchical structural information is not fully utilized in the program's representation**. Existing neural code completion models mainly fall into two major categories, i.e., token-based models and AST-based models. The token-based models (Bhoopchand et al. 2016; Hellendoorn and Devanbu 2017) sequentially tokenize programs into token sequences as the input of models. The syntax and structure of code are not explicitly considered, so this information is underused. To exploit and utilize this information, programs are represented as ASTs, and AST-based code completion models are proposed (Liu et al. 2016; Li et al. 2018). In these models, programs are first parsed into ASTs. Then, ASTs are traversed to produce the node sequence as the representation of the programs. Although these models utilize ASTs in the program's representation, the hierarchical level of the AST nodes is ignored because the tree is traversed to flatten sequence. The tree's structural information is under-utilized.

b)**In programs, the long-term semantic dependency can not be well captured using RNN-based models**. In programs, semantic dependency can be very long. For example, when the model suggests calling a function that has been defined many tokens before (e.g., 500 tokens). Even worse, when the program is parsed into an AST, the tree can be very large when the code blocks are deeply nested, where the dependency will become longer. In such a case, recent code completion research which builds LSTM (Long Short-Term Memory) network based language models (Bhoopchand et al. 2016; Li et al. 2018) cannot work on modeling the very long-term dependency in the source code well. In LSTM-based models, the hidden state vector is served as a memory to store the input information and is updated recurrently to store the information of the contextual tokens. Thus, it is hard to carry the semantics along all time steps of the recurrent model. Existing research shows that LSTM-based language models use 200 context words on average (Khandelwal et al. 2018), which is not sufficient to model the long-term dependency in programs.

c)**The relationship between the type and value of the token/node is ignored.** Current code completion approaches treat the code completion as a single generation task, e.g., predicting the value of next token or AST node. However, the type information of the source code elements is underused. For the AST-level code completion, the node's type and value are two related attributes, where the type can serve as a constraint to the value, and vice versa. However, this correlation is not well considered in existing AST-level code completion models. Li et al. (2018) built two models to predict node's type and value separately, and they treated these two tasks independently. For the token-level code completion, which is more close to the setting that the code completion tools work in practice, the static type information of the identifiers plays an important role. Many IDEs heavily rely on the static types to make helpful suggestions for completing partial code. However, most of the existing token-level source code modeling techniques and code completion studies do not take the type information into consideration. We argue that the relationship between the type and value could provide effective constraints for code completion process.

In this paper, we propose a Multi-Task Learning (MTL) (Caruana 1997) based unified code completion model to address the aforementioned limitations. We consider both AST-level and token-level code completion. AST-level code completion considers code artifacts as abstract syntax trees, and make predictions based on information obtained from the given partial AST. The syntax and structural information of the programs can be better utilized by performing code completion on AST-level. Token-level code completion considers code artifacts as source token sequences, and make predictions based on information obtained from the existing token sequence. In most cases, the developers write code token by token, and line by line. Thus, represent the program as the token sequence can preserve the natural order of typing process. When represent the programs as AST node sequence by traversing the AST, the order of the node sequence are inconsistent with the token sequence. Thus, the natural order of the typing process of the developers are not preserved in the AST-level completion. Thus, predicting the next token is more close to the setting that a code completion tool will work in practice.

We design a unified multi-learning based code completion framework that allows us to perform a series of design decisions that can help us pick a good trade-off among the desired properties of a completion system. We design four main components in our framework and propose new solutions in these components to improve the adaptation of the framework: (1) Code Element Encoder which encodes an AST node or source code token into a distributed vector representation. **We propose two different token encoding approaches including word encoder and subword encoder**, where the subword encoder can capture the semantic of the identifier which is made up of several subwords better. (2) Contextual Code Encoder which encodes the contextual code into a distributed vector representation. We employ**both RNN-based and Transformer-based encoder and compare their results**. (3) Path2root Encoder that encodes the path from the predicting node to the root node, and (4) Code Element Predictor that takes the output of the previous encoders and produce the code completion results via multi-task learning. **We compare two ways of learning the two tasks, including type-first and jointly predicting.**

For the AST-level code completion, to bridge the gap between the sequential node sequences and the hierarchical structure of ASTs, we extract the path from the predicting node to the root node, which indicates the hierarchical level of the predicting node. Then we employ the Path2root Encoder to model the path information into the representation of the contextual program. In the Code Element Encoder, we try two different encoding approaches to encode the value of the node in the programs. To capture the long-term dependency in the programs, we employ Transformer-XL network (Dai et al. 2019) as the Contextual Code Encoder to encode the contextual program. For ASTs, each node has two attributes: type and value. The two attributes are naturally related. We employ multi-task learning to learn these two tasks jointly. We design two methods of learning the two tasks, i.e., predict the type and value jointly or first predict type then utilize the type prediction result to assist the value prediction.

For the token-level code completion, we explore two different token encoding approaches in the Code Element Encoder, i.e., word encoder and subword encoder, to encode each token in the programs. Same as AST-level code completion, we employ Transformer-XL network (Dai et al. 2019) as the Contextual Code Encoder to encode the contextual program. For source code tokens, the type information cannot be obtained directly from the code snippet. To utilize the static type information of the source code tokens, we extract the identifiers' type through static analysis or human annotation. Then we build a multi-task learning model to predict both the type and value of the next token. We also try two ways of learning the two tasks.

To evaluate the performance of our proposed model, we conduct experiments on real-world datasets on both AST-level and token-level code completion. For the AST-level code completion, we conduct experiments on three datasets, including Python, Java, and JavaScript. For the token-level code completion, we conduct experiments on Java and TypeScript programs. We compare our model with several state-of-the-art models, and the experimental results show that our model achieves the best performance.

This paper extends our preliminary study, which appears as a research paper in ICPC (Liu et al. 2020). In particular, we extend our preliminary work in the following direction:

1. We propose a Unified Multi-Task learning based neural Language Model for code completion called **UMTLM** that is an extended version of the model proposed in our preliminary work (Liu et al. 2020). In our previous work, the model was proposed to improve the performance of AST-level code completion. In this paper, we build a more general framework that can perform code completion on both token-level and AST-level. We extend the previous model by designing new model components that can support the requirements for the token-level code completion, and conduct extensive experiments on token-level completion to evaluate the performance of our framework.
2. We explore two ways of learning the two tasks in our multi-task learning framework, i.e., predict the type and value jointly or first predict type then utilize the type prediction result to assist the value prediction.
3. We further discuss how performance differs when adopting different token encoders and contextual code encoders. Specifically, for token encoders, we consider word encoder and sub-word encoder; for contextual code encoders, we consider RNN-based encoder and Transformer-based encoder.
4. We strengthen the experiments by adding more evaluation metrics, including identifier prediction accuracy (for AST-level completion) and type prediction accuracy (for token-level completion).

The main contributions of this paper, which form a super-set of those in our preliminary study, are summarized as follows:

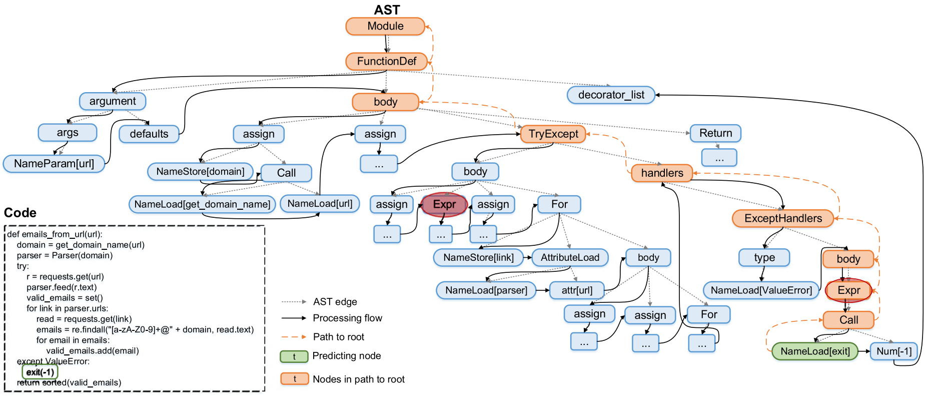
- We invent a unified multi-task learning based neural language model for both AST-level and token-level code completion, where the relationship between the type and value of the token (node) is utilized to offer syntactic and semantic constraints for code completion process.
- We propose a novel method that models the hierarchical structural information into the program's representation.
- We adopt the Transformer-XL network as the language model to capture the very long-range dependencies and the semantic relationship among the contextual tokens.
- We evaluate our proposed model on five real-world program datasets. Experimental results show that our model achieves the best performance on both AST-level and token-level code completion compared with the state-of-the-art models.

**Paper Organization** The remainder of this paper is organized as follows. We give motivating examples in Section 2 and provide background knowledge on statistical language model and multi-task learning in Section 3. Then we introduce our proposed model in Section 4. Section 5 presents experimental results. Section 6 analyzes the efficiency and quality of our model and discusses threats to validity. Section 7 highlights the related work. Finally, we conclude our study and mention future work in Sections 8 and 9.

## 2 Motivating Example

**Hierarchical structure of AST nodes** Figure 1 shows an AST of a Python code snippet. Each node in the AST contains a *Type* attribute, and the leaf nodes also contain an optional *Value* attribute. We use "Type[Value]" to represent each node. To make full use of the structural information of the AST in the program's representation, we take the path from the predicting node to the root node into consideration, which indicates the hierarchical level of the predicting node. For example, in Fig. 1, when predicting the node *NameLoad[exit]*, the contextual sequence contains all the previous nodes before *NameLoad[exit]* in the tree if the tree is flattened in the pre-order depth-first traversal (marked by solid black arrows in the figure). The hierarchical level of the predicting node is ignored. If the path from the predicting node *NameLoad[exit]* to root node (marked by orange arrows in the figure) is introduced into the program's representation explicitly, i.e., type, *Call, Expr, body, ExceptHandlers, handlers, TryExcept, body, FunctionDef, Module*, the structural level of the predicting node can be utilized. The model will realize that the predicting node is the child of a function call, which is a body of the *ExceptHander* of a *TryExcept* structure. This information would be helpful in code completion.

**Relationship between the type and value** In programs, the relationship between the type and value of the code elements is important for understanding the syntactic and semantic of the source code, especially in code completion task. For example, in AST-level code completion, as shown in Fig. 1, when the model is going to predict the node *Num[-1]*, the node's type "Num" conveys the message that the node's value is a number. The model will probably predict a number as the node's value. Likewise, if the model knows the node's value is a number, the model will probably predict "Num" as its type. Similarly, when predicting the node *NameLoad[url]*, the type "NameLoad" implies the information of object accessing, which helps the model to predict an existing object as the node's value. Conversely, For the nodes with the type *NameStore*, the corresponding value will be a newly defined object. For the token-level code completion, existing approaches build models to predict the value of tokens in the source code file. Modern IDEs for most languages heavily rely on



```
def emails_from_url(url):
    domain = get_domain_name(url)
    parser = Parser(domain)
    try:
        r = requests.get(url)
        parser.feed(r.text)
        valid_emails = set()
        for link in parser.urls:
            read = requests.get(link)
            emails = re.findall("[a-zA-Z0-9]+@" + domain, read.text)
            for email in emails:
                valid_emails.add(email)
    except ValueError:
        exit(-1)
    return sorted(valid_emails)
```

**Fig. 1** The AST of the given Python code snippet. Green node denotes the predicting node, i.e., *NameLoad[exit]*. Solid arrows indicate the nodes' processing order. Orange dotted arrows show the path from the predicting node to the root node

types to make helpful suggestions for completing partial code. For example, when accessing the field of an object in a Java IDE, code completion suggests suitable field names based on the object's type (Malik et al. 2019). For those dynamic languages, such as Python and JavaScript, IDEs often fail to make accurate suggestions because the types of code elements are unknown, which further demonstrates the importance of the type information. Thus, the type and value attributes of the code elements are closely related. However, most of the existing statistical language model based source code modeling techniques and code completion studies do not take the relationship between the type and value of the code elements into consideration.

## 3 Background

In this section, we present the background knowledge which will be used in this paper, including the statistical language models and multi-task learning.

### 3.1 Statistical Language Models

Statistical language models capture the statistical patterns in languages by assigning occurrence probabilities to a sequence of words in a particular sequence. Models are estimated on large corpora of text. For natural languages, a good language model should score a sentence high if it sounds natural, and score low if the sentence is unnatural or wrong. Similarly, programming languages are also languages that contain predictable statistical properties (Hindle et al. 2012), which can be modeled by statistical language models. The common way to score a code snippet $s$ of length $t$ is to first tokenize the source code into token sequence $s_1, s_2, ..., s_t$, and score each token $s_t$ given the previous tokens $s_1, ..., s_{t-1}$, i.e.,:

$$p(s) = p(s_1)p(s_2|s_1)p(s_3|s_1, s_2), ..., p(s_t|s_1, s_2, ..., s_{t-1}) \tag{1}$$

In the above equation, the probability of a token in a code snippet is calculated given all previous tokens. In general, once the corpus gets big enough, it is not practical to calculate based on all the previous tokens. There are two kinds of statistical language models to address this issue: explicit language models and implicit language models.

### 3.2 Explicit Language Models

Explicit language models add restriction to the contextual programs. N-gram models and dependency models are two generally used models.

**N-gram models** N-gram models are proposed based on the Markov assumption, where the probability of a token is conditioned only on the $n - 1$ most recent tokens:

$$p(s_t|s_1, s_2, ..., s_{t-1}) = p(s_t|s_{t-n+1}, ..., s_{t-1}) \tag{2}$$

For instance, in a 4-gram model for Java, the score for token "i" given the context of "for (int" would be very high since "i" frequently occurs behind "for (int" in the corpus. N-gram based models have been generally applied to code completion (Hindle et al. 2012; Tu et al. 2014; Hellendoorn and Devanbu 2017). These models have been proved to capture the repetitive regularities in the source code effectively.

**Dependency models** Different from the vanilla N-gram model which can only model the sequential dependencies between tokens, dependency models can also capture the syntax or semantic dependencies (Chelba et al. 1997, 1998), thus can be used for programming language modeling. In code completion, dependency models are proposed to model the syntax and semantic information of the source code, where the dependencies are extracted from the graphs (Nguyen and Nguyen 2015) or ASTs (Maddison and Tarlow 2014; Bielik et al. 2016).

### 3.3 Implicit Language Models

In the above explicit models, the token's occurrence patterns are learned by explicit counts of N-gram frequencies. In implicit models, neural networks are adopted to model the source code, where the code patterns are implicitly represented by an optimized high-dimensional real-valued vectors. The parameters of the networks are estimated using a gradient descent algorithm over training corpus. In recent years, the deep neural network has shown great performance on modeling programming languages. Two kinds of deep neural networks are popular for language modeling.

**Recurrent neural networks** Recurrent neural networks maintain a hidden state vector to store the information of the context. By using recurrent connections, information can cycle inside the networks for a long time, which loosens the fixed context size and can capture longer dependencies than the N-gram model. Tokens in a program $s_0, s_1, ..., s_t$ are fed into the network successively. The hidden state $h_t$ in the network stores the information of the inputs in previous time steps $s_0, ..., s_{t-1}$, and it is fed into the output layer to produce the predicted token. LSTM (Hochreiter and Schmidhuber 1997) and GRU (Gate Recurrent Unit) (Cho et al. 2014) networks are two common variants of RNN, which ease the vanishing gradient problem in RNN by employing powerful gate mechanisms to forget information about the context selectively, and allow room to take in more important information. LSTM network has been applied to source code modeling in recent years (Liu et al. 2016; Li et al. 2018; Bhoopchand et al. 2016).

**Self-attentional neural networks** Although recurrent neural networks, including GRU and LSTM, have achieved good performance in language modeling, the introduction of gating in LSTMs and GRUs might not be sufficient to address the gradient vanishing and explosion issue fully. Empirically, previous work has found that LSTM language models use 200 context words on average (Khandelwal et al. 2018), indicating room for further improvement. To ease this issue, attention mechanisms (Bahdanau et al. 2015; Vaswani et al. 2017) which add direct connections between long-distance word pairs are proposed, where the Transformer (Vaswani et al. 2017) is an architecture based solely on attention mechanism. Vaswani et al. (2017) proposed a multi-headed self-attention mechanism to replace the recurrent layers, and it can reduce sequential computation and capture longer-range dependency. But the Transformer networks are limited by a fixed-length context in the setting of language modeling. To address this issue, Transformer-XL (Dai et al. 2019) is proposed by introducing the notion of recurrence into the deep self-attention network. Thus it enables the Transformer networks to model the very long-term dependency in the source code. To the best of our knowledge, self-attention neural network based models have not been used for source code modeling. In this work, we adopt the Transformer-XL network as the language model for code completion.

### 3.4 Multi-task Learning

Multi-task learning is an approach for knowledge transfer across related tasks. It improves generalization by leveraging the domain-specific information contained in the training signals of related tasks (Caruana 1997). It acts as a regularizer by introducing an inductive bias. As such, it reduces the risk of over-fitting (Ruder 2017). There are two most commonly used ways to perform multi-task learning in deep neural networks: hard or soft parameter sharing of hidden layers. In soft parameter sharing, each task has its own hidden layers and output layer. To ensure the parameters of each task to be similar, the distance between the parameters of each task is regularized. Hard parameter sharing is the most commonly used way, where the hidden layers are shared among all tasks, and the output layers are task-specific. The shared hidden layers can capture the common features among all the tasks. Furthermore, by preferring the representation that all tasks prefer, the risk of over-fitting is reduced, and the model can be more general to new tasks in the future. To the best of our knowledge, MTL has not been applied to code completion. In this paper, we invent a novel MTL-based model to improve the performance of code completion.

## 4 Proposed Model

In this section, we first present an overview of the network architecture of our proposed model UMTLM. Then we introduce each component of UMTLM in detail.

### 4.1 Overall Architecture

We design a unified multi-learning based neural language model (UMTLM) for both AST-level and token-level code completion. Figure 2 shows the architecture of UMTLM. The source code file is first processed into AST node sequence or token sequence. The completion occurs at every location in the source code sequence, and the location to complete is

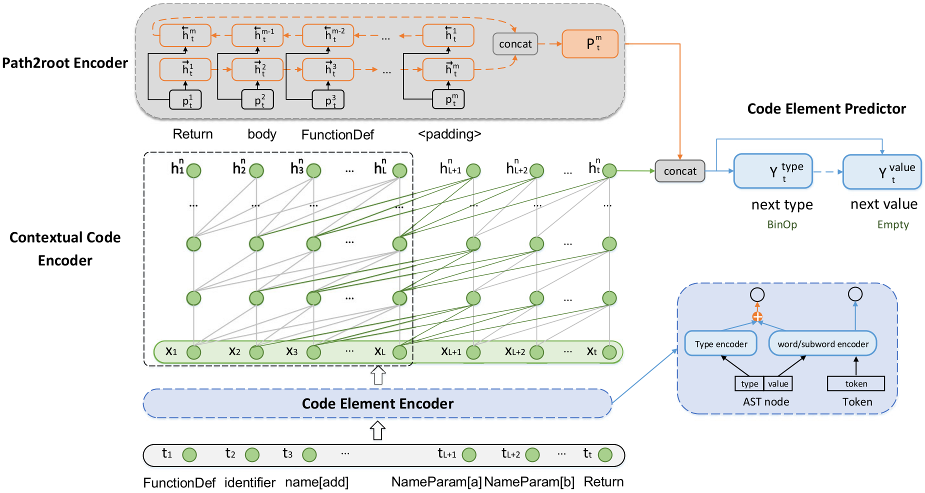
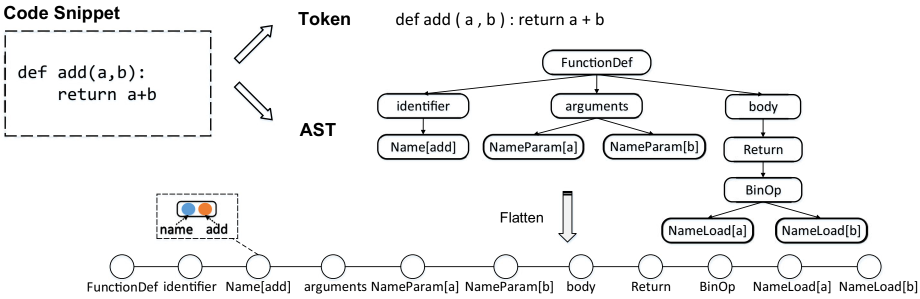


**Fig. 2** The architecture of UMTLM, including code element encoder, contextual code encoder, path2root encoder (for AST-level code completion), and code element predictor

sequentially chosen. At every point in the source code, our model gives a list of possible next code element (source code token or AST node) along with their probabilities that are estimated from the training corpus. For the location whose type or value is none, we use placeholder to represent them. For example, in AST-level completion, each node contains a type attribute, for those nodes which do not have value attribute, we use “EMPTY” to represent its value; In token-level completion, for tokens without type, we use “_” to represent its type. For each location, we make predictions on both type and value, where the completions of the placeholders are not counted in the results. There are four main components in our framework:

- **Code Element Encoder**: A neural network that encodes a code element (AST node or source code token) $t_i$ into a distributed vector representation $x_t$. For the token-level code completion, we explore two different token encoding approaches, i.e., word encoder and subword encoder, to encode each token in the programs. For the AST-level code completion, the representation of the AST nodes is produced by concatenating the type and value vector.
- **Contextual Code Encoder**: A neural network that encodes the program context $x_{1:t}$ into a distributed vector representation $h_t$. To capture the long-term dependency in the input programs, we apply Transformer-XL network (Dai et al. 2019) as the contextual code encoder. We also try RNN-based model in our experiments.
- **Path2root Encoder**: A neural network that encodes the AST path $p_t^{1:m}$ into the vector representation $P_t^m$. This is only used for the AST-level code completion to bridge the gap between the sequential node sequences and the hierarchical structure of ASTs.
- **Code Element Predictor**: A component that takes the output of the previous encoders and produces the code completion results. To capture and utilize the relationship between the type and value of the code elements, we leverage the type information to assist the code completion by employing MTL framework to predicting both the type and the value of the next token (node) jointly. We explore two ways of learning the two tasks, i.e., jointly predicting the type and value or first predicting the type and then predicting the value based on the type prediction results.

The input example of Fig. 2 is shown in Fig. 3. When predicting the node *BinOp*, the input node sequence includes [*FunctionDef, identifier, Name[add], arguments,..., body, Return*], each node is represented as “Type[Value]” (we omit the empty value for the non-leaf nodes. The detailed input node sequence should be [*FunctionDef[Empty], identifier[Empty], Name[add], arguments[Empty],..., body[Empty], Return*]). The input node



```
def add(a,b):
    return a+b
```

**Fig. 3** Token and AST representations for programs

sequence is fed into the Code Element Encoder to get the input vectors $x$. Then the Contextual Code Encoder encodes the input vector $x$ into a distributed vector representation $h$. The path from the predicting node (*BinOp*) to the root (*FunctionDef*) is [*Return, body, FunctionDef*], which is encoded by Path2root encoder to produce the path vector representation $P$. Finally, the contextual code vector representation $h$ and path vector $P$ are concatenated and fed into the Code Element Predictor, which is used to predict the type (*BinOp*) and the value of the next node (*Empty*). Two ways of learning the two tasks are explored, i.e., jointly predicting the type and value or first predicting the type and then predicting the value based on the type prediction results.

## 4.2 Code Element Encoder

We design a code element encoder to encode each AST node or source code token $t_i$ into a distributed vector representation $x_t$. We employ different ways to encode the code element for ASTs and tokens. For the AST-level code completion, each program is parsed into a unique AST since programming languages have an unambiguous context-free grammar. ASTs are widely used for processing programs to extract the syntax and structure of programs (Liu et al. 2016; Raychev et al. 2016; Li et al. 2018). They use ASTs to represent programs and then traverse them to node sequences. As shown in Fig. 3, each node contains a type attribute and the leaf node also has a value attribute.

For the token-level code completion, programs are tokenized into token sequences. Through static analysis or human annotations, we can get the type information for the identifiers in the token sequence. For Java programs, the type information of the identifiers are extracted through static analysis tools provided by aiXcoder[1]. For typescript, since it is a strict syntactical superset of JavaScript and adds optional static typing to the language. Thus, developers can specify the type of the variables, function parameters and object properties using :Type after the name of the them. Thus, when we collected the typescript programs from the github repositories, the type of some identifiers are already existed which is annotated by the developers during their development. Then we apply the approach in Hellendoorn et al. (2018) to extract type annotations for the identifiers.

Figure 4 shows the examples for Java and TypeScript code. The source code is shown in the left. In the source code, the tokens who have type are shown in bold. These bold tokens and their types are shown in the right, where the purple tokens next to the bold identifiers are the corresponding types.

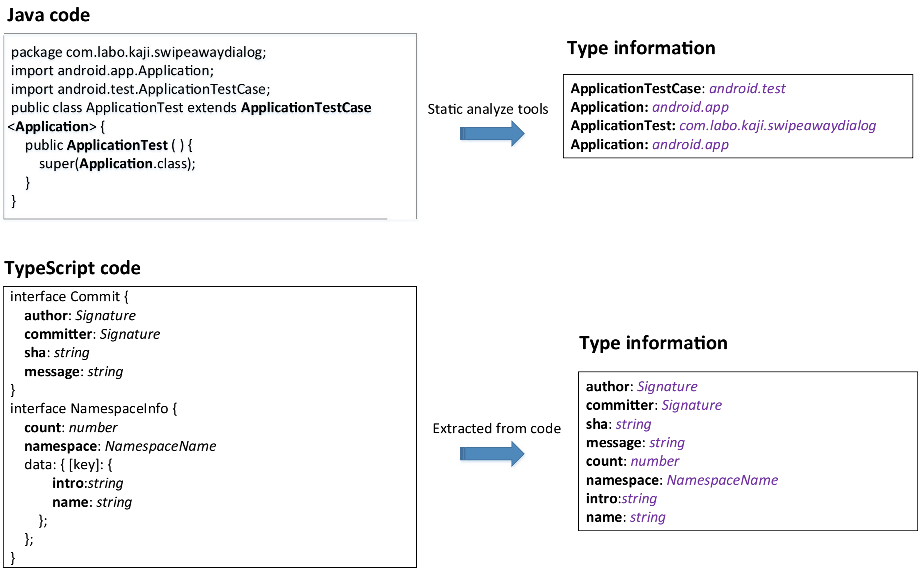
Since the number of the value for both the AST node and the token is large, we consider different token encoders to encode the value of the token and AST node.

**Word encoder** The simplest and most commonly used encoder that we consider is a word encoder. Word encoder learns an embedding of dimension D for the value of each token or AST node in a fixed vocabulary $V_t$ . This requires learning and storing an embedding matrix with $|V_t| \times D$ parameters. Token then performs a lookup:

$$E_{word}(t) = EmbeddingLookUp(t, V_t) \tag{3}$$

where $EmbeddingLookUp(t, V_t)$ returns the D-dimensional row of the embedding matrix that corresponds to $t$. If the lookup fails, then the learned embedding of a special unknown identifier ("UNK") is returned. The vocabulary $V_t$ is selected from the training data and

[1] https://www.aixcoder.com/#/



```
package com.labo.kaji.swipeawaydialog;
import android.app.Application;
import android.test.ApplicationTestCase;
public class ApplicationTest extends ApplicationTestCase
<Application> {
   public ApplicationTest ( ) {
      super(Application.class);
   }
}
```

```
ApplicationTestCase: android.test
Application: android.app
ApplicationTest: com.labo.kaji.swipeawaydialog
Application: android.app
```

```
interface Commit {
   author: Signature
   committer: Signature
   sha: string
   message: string
}
interface NamespaceInfo {
   count: number
   namespace: NamespaceName
   data: { [key]: {
         intro:string
         name: string
      };
   };
}
```

```
author: Signature
committer: Signature
sha: string
message: string
count: number
namespace: NamespaceName
intro:string
name: string
```

**Fig. 4** Code examples for type annotations

contains the most frequent value of the tokens/nodes and the UNK symbol. The size of the vocabulary is a hyper-parameter that needs to be tuned: smaller vocabularies reduce memory requirement at the cost of failing to represent many tokens and thus yielding less accurate suggestions.

**Subword encoder** The identifiers in the source code such as variable names, method names, are often made up of smaller sub words. For example, *set_Maximum_Time* is made up of three subwords (set, Maximum, Time). The subword encoder tokenizes the value of each token or AST node using Camelcase and underscore naming conventions, and the subwords are normalized to lowercase. For example, *set_Maximum_Time* will be tokenized into {*set, maximum, time*}. Then we obtain the embedding vectors for each sub-token $(e_{set}, e_{maximum}, e_{time})$ using an subtoken embedding matrix with size $|V_s| \times D$, where $V_s$ is the subword "vocabulary", which is much smaller than the full token embedding matrix. Finally, the subword encoder employs an aggregation operator to compose the representation from the subtoken embeddings that constitute the token single word from its subwords:

$$E_{subword} = \bigoplus_{t_s \in split(t)} EmbeddingLookUp(t_s, V_s) \tag{4}$$

where $EmbeddingLookUp(t_s, V_s)$ is defined analogously to the word-level case, $split()$ is a function that subtokenizes its input and returns a set of subtokens, and $\oplus$ is an aggregation operator that "summarizes" the meaning of a single word from its subwords. In our experiments, each token is split into 3 subwords, and the embedding size of the subword is set the same with the token embedding. We employ element-wise maximum operation for $\oplus$.

It should be noted that the subword encoder is not used to find the next subword. When predicting, the predicted tokens are chosen from token (not subtoken) vocabulary of candidate completions. The subword encoder can split the long and infrequent tokens into the frequent subtokens, where the subtokens are less sparse than tokens, the vocabulary size of subtokens can be much smaller than the tokens, and thus subtoken can afford a smaller embedding matrix. Besides, it will help to understanding the semantics of the rare tokens by composing the representation from the subtoken embeddings that constitute the token.

**AST Type encoder** Since each node of the AST has a type attribute, we also build a type encoder to encode the type attribute into the distributed vectors for the AST nodes. The type encoder directly learns an embedding of dimension $D_{type}$ for each type in a fixed vocabulary $V_{type}$ . This requires learning and storing an embedding matrix with $|V_{type}| \times D_{type}$ parameters. Then the type of AST node performs a lookup:

$$E_{type}(type) = EmbeddingLookUp(type, V_{type}) \tag{5}$$

where $EmbeddingLookUp(type, V_{type})$ returns the $D_{type}$-dimensional row of the embedding matrix that corresponds to $type$. Since the number of the node type is fixed and much less than the node's values, there is no unknown types. $|V_{type}|$ can be smaller than $|V_t|$, and thus can afford a smaller embedding matrix.

Finally, for the token-level code completion, we can get the representation for each token $x_i$ using the token encoder or subtoken encoder. For the AST-level code completion, we first flatten each AST in pre-order depth-first traversal to produce a sequence of nodes. Then we encode the *Type* into a vector using the AST Type encoder and employ word encoder or subword encoder to produce the representation for the value. Then we concatenate them as the final representation of the nodes $x_i = [T_i; V_i]$, where $T_i$ is the type vector, $V_i$ is the value vector, and ";" denotes the concatenation operation.

## 4.3 Contextual Code Encoder

The programs are represented as AST node sequences for the AST-level code completion and token sequences for token-level code completion. The completion happens at every point in the sequence, and the tokens/nodes before the point form as the contextual code. The context encoders are responsible for taking the completion context and encoding all information that is relevant for the current completion location into a vector representation $h_t$:

$$h_t = E_{cxt}(x_1, x_2, ..., x_t) \tag{6}$$

We consider the following two different encoders to encode the contextual code for both the AST-level and token-level code completion.

**RNN-based Context Encoder** Recurrent neural networks (RNNs) are commonly used for source code modeling, which can capture the long-range dependencies from the input sequence. The hidden state in the network stores the information of the inputs in previous time steps, and is updated recurrently:

$$h^i = RNN(h^{i-1}, x_{i-1}) \tag{7}$$

**Transformer-based Context Encoder** Compared with recurrent neural networks, Transformer network (Vaswani et al. 2017), which is based on multi-headed self-attention mechanism, can reduce sequential computation and capture longer-range dependency. Transformer-XL (Dai et al. 2019) is proposed to introduce a recurrence mechanism to the Transformer architecture, which enables Transformer networks to model very long-term dependency. In Transformer-XL architecture, the hidden states of each new input segment are obtained by reusing that of the previous segments, instead of computing from scratch. In this way, the recurrent connection is created, and the reused hidden states can serve as memories for the current segment, which enables the information to propagate through the recurrent connections. Thus the model can capture very long-term dependency.

Formally, let $s_\tau = [x_{\tau,1}, x_{\tau,2}, ..., x_{\tau,L}]$ and $s_{\tau+1} = [x_{\tau+1,1}, x_{\tau+1,2}, ..., x_{\tau+1,L}]$ represent two consecutive segments of length $L$. For the $\tau$-th segment $s_\tau$, the $n$-th layer hidden state sequence is denoted as $h_\tau^n \in \mathbb{R}^{L \times H}$, where $H$ is the dimension of the hidden units. The $n$-th layer hidden state for segment $s_\tau$ is computed as:

$$\begin{aligned}
&\widetilde{h}_{\tau+1}^{n-1} = [SG(h_\tau^{n-1}) \circ h_{\tau+1}^{n-1}] \\
&q_{\tau+1}^n, k_{\tau+1}^n, v_{\tau+1}^n = h_{\tau+1}^{n-1} W_q^T, \widetilde{h}_{\tau+1}^{n-1} W_k^T, \widetilde{h}_{\tau+1}^{n-1} W_v^T \\
&h_{\tau+1}^n = \text{Transformer-Layer}(q_{\tau+1}^n, k_{\tau+1}^n, v_{\tau+1}^n)
\end{aligned} \tag{8}$$

where $SG(\cdot)$ stands for stop-gradient, that is, we don't calculate gradients for the $\tau$-th segment. The notation $[h_u \circ h_v]$ indicates the concatenation of two hidden sequences along the length dimension, and $W_{\cdot}^T$ denotes model parameters. Compared to the standard Transformer, the critical difference lies in that the key $k_{\tau+1}^n$ and value $v_{\tau+1}^n$ are conditioned on the extended context $\widetilde{h}_{\tau+1}^{n-1}$ and hence $h_{\tau+1}^{n-1}$ cached from the previous segment. The Transformer-layer consists of multi-head self-attention mechanism and a position-wise fully connected feed-forward network. Besides, to keep the positional information coherent when we reuse the states, relative positional embedding is adopted, and the detailed computation procedure can be found in Dai et al. (2019).

## 4.4 Path2root Encoder

To model the hierarchical structural information of the predicting AST node for the AST-level code completion, we extract the path from the predicting node to the root node, i.e., $p_t^1, p_t^2, ..., p_t^m$, where $m$ is the length of the path, $p_t^i$ is the type of the $i$-th node in the path at time step $t$.[2] Taking the AST in Fig. 3 as an example, when predicting the last node *NameLoad[b]*, the path from it to the root node contains the nodes {*BinOp, Return, body, FunctionDef*}. We design a bidirectional-LSTM (Schuster and Paliwal 1997) based Path2root encoder, which encodes the nodes in the path to produce a path vector. The hidden states for both directions of the bi-LSTM are computed as follows:

$$\begin{aligned}
\overrightarrow{h_t^i} &= \overrightarrow{LSTM}(p_t^i, \overrightarrow{h_t^{i-1}}) \\
\overleftarrow{h_t^i} &= \overleftarrow{LSTM}(p_t^i, \overleftarrow{h_t^{i-1}})
\end{aligned} \tag{9}$$

$\overrightarrow{h_t^m}$ and $\overleftarrow{h_t^m}$ contain the path's forward information and backward information. We concatenate $\overrightarrow{h_t^m}$ and $\overleftarrow{h_t^m}$ to obtain the final path vector $P_t$ for each time step, i.e., $P_t = [\overrightarrow{h_t^m}; \overleftarrow{h_t^m}]$. In this way, we can reduce the chance that the model might forget the information of the top nodes or the bottom nodes when the path is long.

## 4.5 Code Element Predictor

We build a code element predictor to produce the results for both AST-level and token-level code completion, i.e., predicting the next node (token), including its type and value, based

[2] The nodes in the path are non-leaf nodes, and they do not have the value attribute. Thus, we use the node's type as the representation for the nodes in the path.

on the output of the previous encoders. Since these two attributes are closely related and interacted, we adopt multi-task learning to learn these two tasks together. We explore two ways of learning the two tasks, i.e., predicting the type and value jointly or first predicting the type, then the type prediction results are utilized to assist the value prediction.

**Jointly** In this approach, the type and value of the next token are predicted jointly based on the same hidden vector. The output of the contextual code encoder $h_t^n$ and path vector $P_t$ (for the AST-level code completion) are concatenated to compute the output vector $O_t$. *Sotfmax* function can take as input a vector of $N$ real numbers and normalizes it into a probability distribution consisting of $N$ probabilities proportional to the exponentials of the input numbers. We use the *softmax* function to produce the probability distribution of the outputs.

$$\begin{aligned} O_t &= \tanh(W^o(h_t^n; P_t)) \\ Y_t^{type} &= \mathrm{softmax}(W_y^{type} O_t + b^{type}) \\ Y_t^{value} &= \mathrm{softmax}(W_y^{value} O_t + b^{value}) \end{aligned} \tag{10}$$

where $W^o \in \mathbb{R}^{H \times (H+H_p)}$, $W^{type} \in \mathbb{R}^{V_{type} \times H}$, $W^{value} \in \mathbb{R}^{|V_{value}| \times H}$, $b^{type} \in \mathbb{R}^{|V_{type}|}$, $b^{value} \in \mathbb{R}^{|V_{value}|}$ are trainable parameters. $|V_{type}|$ is the vocabulary size for type, $V_{value}$ is the vocabulary size for value. $H_p$ is the hidden size of the Path2root encoder, and ";" denotes the concatenation operation.

**Type-first** Different from the previous approach, we first predict the type of the next token based on the output vector $O_t$. Then the predicted type vector $E_t^{type}$ is utilized for assisting the value prediction.

1) Type prediction: The output of the contextual code encoder $h_t^n$ and path vector $P_t$ (for the AST-level code completion) are concatenated to compute the output vector $O_t^{type}$ for type prediction. Then we use the *softmax* function to produce the probability distribution of the type prediction output $Y_t^{type}$.
2) Value prediction: After predicting the token's type, we use the predicted type to assist the token prediction. We employ the token encoder to compute the vector representation of the predicted type $E_t^{type}$. Then use the predicted type to assist the token prediction. The vector of the predicted type $E_t^{type}$, the output of the contextual code encoder $h_t^n$, and path vector $P_t$ (for the AST-level code completion) are concatenated to compute the output vector for the value $O_t^{value}$. Then the output vector is fed into the output *softmax* layer to compute the output vector for the value $Y_t^{value}$:

$$\begin{aligned} O_t^{type} &= \tanh(W_o^{type}(h_t^n; P_t)) \\ Y_t^{type} &= \mathrm{softmax}(W_y^{type} O_t^{type} + b^{type}) \\ E_t^{type} &= E_{token}(Y_t^{type}, V_{type}) \\ O_t^{value} &= \tanh(W_o^{value}(h_t^n; P_t; E_{type})) \\ Y_t^{value} &= \mathrm{softmax}(W_y^{value} O_t^{value} + b^{value}) \end{aligned} \tag{11}$$

where $E_{token}$ is the Token encoder which is used for encoding the predicted type $Y_t^{type}$ into the vector representation $E_t^{type}$. $W_o^{type} \in \mathbb{R}^{H \times (H+H_p)}$, $W_o^{value} \in \mathbb{R}^{H \times (H+H+H_p)}$, $W_y^{type} \in \mathbb{R}^{V_{type} \times H}$, $W_y^{value} \in \mathbb{R}^{|V_{value}| \times H}$, $b^{type} \in \mathbb{R}^{|V_{type}|}$, $b^{value} \in \mathbb{R}^{|V_{value}|}$ are trainable parameters.

### 4.6 Training

To learn the type and value prediction tasks in our multi-task learning framework, we adopt a weighted sum over the task-specific losses as the final loss:

$$loss = \sum_{k=1}^{N} \alpha_k \times loss_k \tag{12}$$

where N is the number of tasks. $\alpha_k$ is the weight of the loss for the $k$-th task, and $\alpha_k \geq 0$, $\sum_{k=1}^{N} \alpha_k = 1$. In this paper, by default, we set the weights for the two tasks as 0.5 and 0.5, respectively. The effect of different weight settings will be discussed in Section 6.

## 5 Experiment and Analysis

### 5.1 Dataset and Vocabulary

#### 5.1.1 AST Data

For the AST-level code completion, we evaluate our model on three datasets: Python, Java, and JavaScript. Python and JavaScript datasets are collected from GitHub repositories by removing duplicate files, removing project forks, keeping only programs that parse and have at most 30,000 nodes in the AST. Each dataset contains 100,000 training programs and 50,000 test programs. Both source code files and their corresponding ASTs are provided. The parser used in Python to parse programs into ASTs is ast module in Python Standard Library[3]. The parser used in JavaScript is Acorn[4]. These two datasets have been used in Li et al. (2018) and Raychev et al. (2016). Java dataset are collected from Github. We use javalang[5] to parse the programs into ASTs. For all the datasets, each program is represented in its AST format, and the AST is serialized in pre-order depth-first traversal to produce the AST node sequence. Following Li et al. (2018), to enable the flattened ASTs can be converted back to the original tree structure thus converted back to the source code, each node type is allowed to encode two additional bits of information about whether the AST node has a child and/or a right sibling. That is, the node type value consists of three part (type name, whether has a child, whether has a right sibling). For example, the two “Expr” node circled in red in Fig. 1 has different type values since one “Expr” has a right sibling and another one does not has a right sibling. Thus, the node with the same type name can have different type values when the child or sibling number of the node is different. After this processing, the number of the node types are increased.

After this processing, the number of the types are increased. Specifically, the numbers of unique node types in JAVA, JS and PY are 65, 44 and 181 originally, by adding information about children and siblings, the number of the node types are increased to 175, 95 and 329. The datasets and processing code will be published for replicating our experiments. Then we generate queries used for training and test, one per AST node, by removing the node and all the nodes to the right from the sequence and then attempting to predict the node. The

[3] https://github.com/python/cpython/blob/3.9/Lib/ast.py

[4] https://github.com/acornjs/acorn

[5] https://github.com/c2nes/javalang

number of type attributes and value attributes of AST nodes, the queries of the programs, and the average length of the AST nodes in programs are shown in Table 1.

### 5.1.2 AST Vocabulary

Followed by Li et al. (2018), we choose the 50,000 most frequent values to build value's vocabulary for all the three datasets. For those values outside the vocabulary, we use *UNK* (unknown values) to represent them. The *UNK* rate of the value's vocabulary for Python, Java, and JavaScript are 11%, 13%, and 7%, respectively. All the types are used to build type's vocabulary.

### 5.1.3 Token Data

For the token-level code completion, we evaluate our model on two datasets: Java and TypeScript. The Java dataset are the same with the AST data.

The repositories are collected from GitHub. According to the statistics in Han et al. (2019), the number of stars can be used as the proxy for project popularity. Thus, we collected the repositories that have at least 10 stars aiming at filtering out low quality repositories. During the collections, we do not limit the library use or the project size. Among these projects, there are 571 projects that have more than 300 java files. Although some projects are small (with a few program files), it is still popular and considered as good with many stars, not just "using existing library without really defining anything". It is unreasonable to determine whether the projects are good enough or whether the projects use the existing library only depending on their size (the number of the java files). For example, in the dataset, Google Cloud Messaging (GCM) project[6] has 826 stars now, which is a service that lets developers send data from servers to users' devices, and receive messages from devices on the same connection) contains 53 java files, there are still many user-defined classes and methods in the code. Also, LruCache project[7] has 74 stars now, which is a tiny, thread safe memory cache implementation which uses a LRU policy) only contains 6 java files, there also exist user-defined classes and methods.

The reasons for using these two languages are as follows. These two languages are commonly used for software development, and we can get the identifiers' type through static analysis or through the developers' annotations. The programs in the corpus are collected from publicly available open-source GitHub repositories by removing duplicate files and project forks. Each program is tokenized into the token sequence. For Java programs, we extract the identifiers' type information through static analysis. For TypeScript programs, we apply the approach in Hellendoorn et al. (2018) to extract type annotations of the identifiers. Then we filter the programs to make sure at least 10% of type annotations are user-defined types in each TypeScript file. The detailed information is shown in Table 2. We split the projects into train/validation/test sets in the proportion 8:1:1.

### 5.1.4 Token Vocabulary

For token data, we choose $K$ (50,000) most frequent tokens in each training set to build the token vocabulary, which is the same as Li et al. (2018)'s study. For those tokens outside

[6] https://github.com/google/gcm

[7] https://github.com/hotchemi/LruCache

**Table 1** Statistics of AST datasets

| | Python | Java | JavaScript |
|---|---|---|---|
| # of Type | 330 | 178 | 95 |
| # of Value | $3.4 \times 10^6$ | $6.4 \times 10^6$ | $2.6 \times 10^6$ |
| # of Training Queries | $6.2 \times 10^7$ | $16.8 \times 10^7$ | $10.7 \times 10^7$ |
| # of Test Queries | $3.0 \times 10^7$ | $7.2 \times 10^7$ | $5.3 \times 10^7$ |
| # Identifier proportion | 28.61% | 19.29% | 20.86% |
| Avg. nodes in AST | 623 | 253 | 1730 |

the vocabulary, we use *UNK* to represent them. The size of type vocabulary is also set to 50,000. In both the training and test process, the predictions of the *UNK* targets are treated as wrong predictions. The *UNK* rates of the value's vocabulary for Java, and TypeScript test sets are 10%, 5%, and the *UNK* rates of the type's vocabulary are 9%, 1%, respectively.

## 5.2 Metrics

We use *accuracy* to evaluate the performance of our model. In the code completion task, the model provides an ordered list of suggestions for the next token/AST node given the context. We compute the top-1 accuracy on next token/AST node's value and type, i.e., the fraction of times the correct suggestion appears in the first of the predicted list.

In the experiments of most statistical language model based code completion research, every token in the program file is considered as the target for completion, including punctuation-like tokens such as operators, braces, etc. This is different from the real completion tools, for example, Visual Studio does not offer to complete punctuation and numerals. According to the findings in Karampatsis et al. (2020), more than two-thirds of the completion targets (tokens) are not identifiers. For statistical language model based code completion approach, every token in the program file is considered as the target for completion. Taking the following python program as an example:

```
for i in range(10):
    print(i, end=” ”)
}
```

**Code 1** Python code example.

**Table 2** Statistics of token datasets

| | Java | TypeScript |
|---|---|---|
| # of Files before processing | 804,470 | 419,024 |
| # of Files after processing | 800,983 | 227,424 |
| # of Lines | $5.4 * 10^7$ | $8.8 * 10^6$ |
| # of Tokens | $6.9 * 10^6$ | $1.1 * 10^6$ |
| # of Types | $6.4 * 10^6$ | $1.7 * 10^5$ |
| Annotated Identifier proportion | 21.04% | 9.74% |

Each token in this program is considered as the target for completion. First target is “i” given the context “for”, and the second target is “in” given the context “for i”, the third target is “range” given the context “for i in”, and so on. The last completing target is “)” given the context of all the previous token. Thus, during training and evaluation, the proportion of different data type in the data corpus has great impact on the model’s training process. According to the statistics in Hellendoorn et al. (2019), in most of the programs, more than 2/3 of the completion targets (tokens) are not identifiers. The majority (57%) of the tokens are punctuation-like tokens (e.g., operators, braces), followed by identifiers (30.4%), keywords and numerals (10.8% and 1.8% respectively). Thus, the model will perform better on the 2/3 of the non-identifier tokens since the training samples of predicting these targets is more than predicting identifiers. However, in practice, only completions pertaining to identifiers. The identifiers’ completion is more challenging and practical. Thus, to improve the performance on identifier’s completion, we take use of the static type information of the identifiers and also consider *identifier prediction accuracy (ID Accuracy)* as a metric to measure the identifiers’ completion performance.

Directly comparing accuracy by the difference or direct proportion may lead to inflated results (>100% improvement). Therefore, we also use *normalized improvement in accuracy* (*Imp. Accuracy*) (Costa et al. 2016) to measure the “the room for improvement”:

$$Imp.\ Accuracy = \begin{cases} \dfrac{Acc_x - Acc_y}{Acc_{ub} - Acc_y}, \text{if } Acc_x > Acc_y \\ \dfrac{Acc_x - Acc_y}{Acc_y}, \text{otherwise} \end{cases} \tag{13}$$

where $Acc_x$ represents the accuracy obtained by model $x$, $Acc_y$ represents the accuracy obtained by model $y$, and $Acc_{ub}$ represents the upper bound of the accuracy[8]. Thus, this metric can measure the room for improvement of model $x$ over model $y$.

Unlike ASTs, for the token-level code completion, only a part of the tokens have the type attribute since the type information is extracted through static analysis or human annotations. Thus we only report the results of the above metrics on the token’s value completion in our main experiments.

### 5.3 Experimental Setup

**AST-level code completion** The embedding sizes for type and value are 300 and 1,200, respectively. The size of the AST node vector is $300 + 1200 = 1500$. For Transformer-based context code encoder, we use a 6-layer Transformer-XL network (Dai et al. 2019). We employ $h = 6$ parallel heads, and the dimension of each head $d_{head}$ is set to 64. We set the segment length to 50, which is the same as the LSTM’s unrolling length (the length of the input sequence) in Li et al. (2018). The dimensionality of the hidden unit is $H = 1500$. Through the recurrent mechanism, we can cache previous segments and reuse them as the extra context when processing the current segment. Considering the GPU memory and training time, we set the length of cached hidden states $M$ to 256. In our experiment, as we increase $M$, the accuracy also increases. When $M$ is increased to 1024, the accuracy stops increasing, which demonstrates that our model can use up to about 1024 context

[8] For the next node’s type prediction, the upper bound of the accuracy is 100%. For the next node’s value prediction, since the UNK targets are treated as wrong predictions, the upper bound of the accuracy is less than 100%, which depends on the UNK rate of the dataset.

tokens. For the LSTM-based model, the accuracy stops increasing when the unrolling length increases to 256, which demonstrates that LSTM language models can only use less than 256 contextual tokens in this experiment, which is consistent with the findings in Khandelwal et al. (2018). The dimension of the feed-forward layer in the Transformer is set to 1024. For RNN-based context encoder, followed by Li et al. (2018), we use an attention enhanced single layer LSTM network with the hidden size of 1500, and we set the attention window length as 50.

For the Path2root encoder, we employ a single layer bidirectional-LSTM. In our model, we set the length of the path to $m$. For the nodes whose length is over $m$, we preserve $m$ nodes in the path from the predicting node to the root. For the nodes whose length is less than $m$, we pad the path to the length of $m$. Considering the trade-off between time cost and performance, we set the length of path $m$ to 5 and the hidden size of Path2root encoder and path vector size to 300, which can offer a considerable improvement and would not increase much time cost.

**Token-level code completion** The embedding sizes for the token is set to 600. For the transformer-based context code encoder, we use a 6-layer Transformer-XL network (Dai et al. 2019). We employ $h = 5$ parallel heads, and the dimension of each head $d_{head}$ is set to 64. We set the segment length to 50, which is the same as the LSTM's unrolling length (the length of the input sequence) in Li et al. (2018). The dimensionality of the hidden unit is $H$ = 600. The length of cached hidden states $M$ is set to 256. The dimension of the feed-forward layer in the Transformer is set to 1024. For the RNN-based context encoder, we use an attention enhanced single layer LSTM network with the hidden size of 1500, and the attention window length is set to 50.

**Training** To train the model, we employ the cross-entropy loss and Adam optimizer (Kingma and Ba 2015). In both the training and test process, the predictions of the *UNK* targets are treated as wrong predictions as in Li et al. (2018). The hyper-parameters (the parameters whose value are used to control the learning process, including the learning rate: 2.5e-4, dropout rate: 0.1, batch size: 60, training epochs: 6, etc.) are selected on the validation set, that is, we choose the hyper-parameters settings associated with the best validation performance. We implement our model using Tensorflow (Abadi et al. 2016) and run our experiments on a Linux server with the NVIDIA GTX TITAN Xp GPU with 12 GB memory.[9]

### 5.4 Research Question and Results

To evaluate our proposed approach, in this section, we conduct experiments to investigate the following research questions:

**RQ1: How does our proposed approach perform in AST-level code completion when compared with state-of-the-art models?** For the AST-level code completion, we compare our model with the Pointer Mixture Network (PMN) (Li et al. 2018): an attention and pointer-generator network-based code completion model. Besides, we also compare our model with widely used deep neural network based language models: vanilla LSTM and Transformer-XL network.

[9] The datasets and code are publicly available in https://figshare.com/s/7eb8819f2a04e8163224

**Table 3** AST-level code completion accuracy comparison of state-of-the-art approaches and our proposed model

| | Python | | | Java | | | JavaScript | | |
|---|---|---|---|---|---|---|---|---|---|
| | Type | Value | ID | Type | Value | ID | Type | Value | ID |
| PMN | 80.6% | 70.1% | 44.2% | 79.9% | 74.3% | 41.9% | 88.6% | 81.0% | 59.7% |
| LSTM | 79.2% | 67.0% | 42.7% | 78.3% | 72.7% | 40.0% | 86.9% | 78.2% | 58.9% |
| Transformer-XL | 82.3% | 69.8% | 46.6% | 79.8% | 75.5% | 42.4% | 88.5% | 80.1% | 60.7% |
| UMTLM (RNN) | 84.2% | 71.3% | 47.0% | 79.7% | 74.2% | 45.1% | 88.9% | 81.1% | 63.4% |
| UMTLM (Trans) | **87.1%** | **73.8%** | **55.1%** | **83.4%** | **76.8%** | **49.9%** | **91.4%** | **82.7%** | **68.9%** |

The results are shown in Table 3. The last two rows show the results of UMTLM with Transformer-based context encoder and RNN-based context encoder, respectively. To compare with PMN, we downloaded their publicly available source code[10]. Since the python and JavaScript datasets are the same as them, we directly report the results in their paper for these two datasets. For java dataset, we use the same approach to process the code into the AST format and train the model using their source code.

As can be seen from the results, on all the three datasets, our model outperforms all the baselines on both the next node's type and value prediction. For the next node's type prediction, our best model achieves the accuracy of 87.1%, 83.4%, and 91.4% on these three datasets respectively, which improves state-of-the-art approach, i.e., PMN, by 34%, 17%, and 25%[11], in terms of *normalized improvement in accuracy*. For the next node's value prediction, our best model achieves the accuracy of 73.8%, 76.8%, and 82.7% on three datasets, which improves PMN by 20%, 20%, and 14%[12], in terms of *normalized improvement in accuracy*. In the value prediction, the predictions of the *UNK* targets are treated as wrong predictions. The *UNK* rates for Python, Java, and JavaScript are 11%, 13%, and 7%. Therefore, when computing the *normalized improvement in accuracy*, the upper bounds of the accuracy for the three datasets are 89%, 87%, and 93%, not 100%. In PMN, Pointer Network is adopted to address the OoV issue in the value prediction. Actually, Li et al's pointer network based model does not address the OoV issue. The pointer network can only support the cases where the predicted token has been occurred in the local context window size of 50. Only 3.2%, 3.9%, 1.9% of the OoV tokens in Python, Java, and JS can meet the requirements. And there is no guarantee that their model can correctly predicted these tokens. Thus, most of the OoV tokens can still not be correctly predicted in PMN. Compared with them, our model employ a powerful backbone language model and introduce MTL and path2root encoder, which can outperform their model on the in-vocabulary-tokens, especially on the identifiers prediction.

The performance improvement on predicting next node's type and value is relatively small compared to the baseline Transformer-XL. However, for the value of the identifier node's completion, which is more challenging, our model outperforms the baselines including the powerful Transformer-XL by a large margin. Specifically, for the next identifier node's value prediction, our best model achieves the accuracy of 55.1%, 49.9%, and 68.9%

[10] https://github.com/jack57lee/neuralCodeCompletion

[11] 34% = (87.1%-80.6%) / (100%-80.6%), 17% = (83.4%-79.9%) / (100%-79.9%), 25% = (91.4%-88.6%) / (100%-88.6%)

[12] 20% = (73.8%-70.1%) / (89%-70.1%), 20% = (76.8%-74.3%) / (87%-74.3%), 14% = (82.7%-81.0%) / (93%-81.0%)

**Table 4** Token-level code completion accuracy comparison of state-of-the-art approaches and our proposed model

| | Java | | | TypeScript | | |
|---|---|---|---|---|---|---|
| | Type | Value | ID | Type | Value | ID |
| PMN | 61.5% | 68.3% | 38.4% | 72.2% | 68.8% | 33.8% |
| BPE NLM | 61.6% | 69.2% | 44.7% | 71.0% | 67.4% | 37.2% |
| LSTM | 60.7% | 64.3% | 33.8% | 69.8% | 64.4% | 28.1% |
| Transformer-XL | 62.6% | 72.1% | 43.6% | 73.1% | 73.9% | 37.5% |
| UMTLM (RNN) | 63.1% | 69.4% | 46.1% | 72.7% | 69.8% | 31.0% |
| UMTLM (Trans) | **66.7%** | **75.0%** | **51.9%** | **76.4%** | **74.9%** | **45.9%** |

on these three datasets respectively, which achieve absolute accuracy improvements of 9%, 8%, and 8% compared to Transformer-XL.

We apply the Wilcoxon Rank Sum Test (WRST) (Wilcoxon 1945) to test whether the improvements of our model over baselines are statistically significant, and all the p-values are less than 1e-5, which indicates significant improvements. We also use Cliff's Delta (Macbeth et al. 2011) to measure the effect size, and the values are non-negligible. From Table 3, we also notice that the improvements on the JavaScript are not as good as the other two datasets. The reason might lie in that the correlation between the node's type and value in JavaScript is weaker than Python and Java. As shown in Table 1, the category of the node's type for JavaScript is much less (only 95 types) compared with Python or Java, but one type can correspond to many values, which could result in the limited improvement.

When comparing with vanilla LSTM or Transformer-XL models, UMTLM with corresponding context encoders, i.e., UMTLM (RNN) and UMTLM (Trans), can achieve better performance, which demonstrates the components proposed are effective for improving the performance of AST-level code completion.

**RQ2: How does our proposed approach perform in token-level code completion when compared with state-of-the-art models?** For the token-level code completion, we compare our model with the following state-of-the-art models:

- Pointer Mixture Network (PMN) (Li et al. 2018): an attention and pointer-generator network-based code completion model.
- Byte Pair Encoding based Neural Language Model (BPE NLM) (Karampatsis et al. 2020): a large-scale open-vocabulary NLM for code completion, which leverage BPE (Gage 1994) algorithm to keep vocabulary size low and successfully predict OoV (Out-of-Vocabulary) tokens.

The results are shown in Table 4. The performance of Vanilla LSTM and Transformer-XL network are also presented for comparison. To compare with PMN, we downloaded their publicly available source code[13]. In their original model, the programs in the datasets are parsed into ASTs, and they build the model to perform code completion on AST node sequences. In our corpus, the programs are tokenized into token sequences. To compare

[13] https://github.com/jack57lee/neuralCodeCompletion

with them, we train their model within our tokenized programs using the command line arguments given in the artifact's README file[14].

As shown from the results, on the Java test set, our best model achieves 75.0%, 66.7%, and 51.9% in terms of value, type and identifier prediction accuracy. On the TypeScript test set, our best model achieves 74.9%, 76.4% and 45.9% in terms of value, type and identifier prediction accuracy. On both of the datasets, our best model outperforms PMN on Java and TypeScript datasets by a large margin, especially in identifier completion. Specifically, our best model achieves absolute accuracy improvement of 14% and 12% in identifier completion compared with PMN. We can find that the general improvements on the TypeScript dataset are smaller than Java, especially in identifier completion. The reason lies in that, the identifier proportion in TypeScript (9.74%) is smaller than Java (21.04%) because the type information in TypeScript is annotated by developers, and only a part of the identifiers are annotated. During code completion, the type information of these identifiers is used to assist the identifiers' prediction. Due to the lower identifier proportion, the multi-task learning procedure can offer less information than Java, thus resulting in smaller improvements.

To compare with BPE NLM (Karampatsis et al. 2020), we downloaded their publicly available source code[15] and train their model on our datasets. They use a single layer GRU NLM with an unrolling length of 200 built upon sub-word units learned from BPE. The embedding size and the hidden unit size are both set to 512 in their model. To keep the number of parameters comparable with our model and other baselines, we increase the hidden unit size and the embedding size of their model to 1500. There are three scenarios: static, dynamic, and maintenance in BPE NLM, we only present the results of the static scenario to make the comparison fair. As shown from the results, BPE NLM performs best on completing identifiers among all the baseline models on both datasets, which proves the power of the open vocabulary language model for predicting the identifiers. Even though, our best model still outperforms BPE NLM on completing identifiers. When evaluating on completing all kinds of tokens, the performance of BPE NLM is not as well as the identifier completion. Our model outperforms BPE NLM on completing all kinds of tokens substantially.

When comparing with vanilla LSTM or Transformer-XL models, UMTLM with corresponding context encoders can achieve better performance, which further demonstrates the components proposed are effective for improving the performance of token-level code completion.

**RQ3: What is the effectiveness of each component in our framework for AST-level and token-level code completion?** For the AST-level code completion, we perform an ablation study to examine the effects of the proposed components used in our best model (transformer-based): multi-task learning mechanism, the subword encoder, and the Path2root encoder. We conduct experiments without each of these components. Besides, to verify whether capturing the long-range dependency from the input programs helps, we also conduct an experiment of removing the recurrent mechanism from the Transformer-XL architecture. The recurrent mechanism means the recurrent connection in the Transformer-XL network, which enables the model to reuse the hidden states of the previous input segment. When remove the recurrence mechanism, the input sequence length is 50, which

[14] Since the PMN also makes use of the additional information derived from ASTs, the results of using the token sequence as input might understate the accuracy of the plain PMN.

[15] https://github.com/mast-group/OpenVocabCodeNLM

**Table 5** Effectiveness of each component in our proposed model for the AST-level code completion

| | Python | | | Java | | | JavaScript | | |
|---|---|---|---|---|---|---|---|---|---|
| | Type | Value | ID | Type | Value | ID | Type | Value | ID |
| UMTLM (Trans) | **87.1%** | **73.8%** | **55.1%** | **83.4%** | **76.8%** | **49.9%** | **91.4%** | **82.7%** | **68.9%** |
| - MTL | 84.3% | 71.8% | 48.5% | 80.6% | 76.2% | 43.6% | 89.6% | 80.8% | 62.0% |
| - Subword Encoder | 86.9% | 73.2% | 53.6% | 82.8% | 75.1% | 47.8% | 91.3% | 82.5% | 67.9% |
| - Path2root Encoder | 84.8% | 70.9% | 51.3% | 74.8% | 72.5% | 46.1% | 90.6% | 81.8% | 65.5% |
| - Recurrence | 80.5% | 67.9% | 45.1% | 77.7% | 69.2% | 41.6% | 85.7% | 78.0% | 60.2% |

is the same as the LSTM's unrolling length (the length of the input sequence) in Pointer Mixture Network.

The results are shown in Table 5. The first row shows the results of our full model. The second row presents the results of removing MTL from the full model. The third row shows the results of using the word encoder to encode the value of the AST node instead of the subword encoder. The fourth row removes the Path2root encoder from the full model. The results of removing the recurrent mechanism from the Transformer-XL architecture are shown in the last row. When removing MTL, the accuracies of type, value and identifier prediction on three datasets drops a lot, especially in the identifier prediction (from 55.1% to 48.5% in Python, from 49.9% to 43.6% in Java, from 68.9% to 62.0% in JavaScript, which drops 6.6%, 6.3% and 6.9% respectively). Besides, removing Path2root Encoder also lead to a severe accuracy drop, where the identifier prediction accuracy decrease about 4% on average (from 55.1% to 51.3% in Python, from 49.9% to 46.1% in Java, from 68.9% to 65.5% in JavaScript). When removing the recurrent mechanism from our full model, the accuracy drops a lot, even lower than the vanilla Transformer-XL network. Specifically, the type, value, and identifier prediction accuracy in vanilla Transformer-XL network are 82.3%, 69.8%, and 46.6% on Python. After removing the recurrence mechanism from our model, the type, value, and identifier prediction accuracy on Python is 80.5%, 67.9%, and 45.1%, which is lower than vanilla Transformer-XL. The results on the other two datasets is also similar. The recurrent mechanism enable the model to cache the memory of the previous contextual information, and thus can capture the long-range dependency in the programs. Thus, the results of removing recurrence mechanism further demonstrate that capturing long-range dependency is of great importance and necessity for programming language modeling. When removing the Subword Encoder, the accuracy also drops on all the datasets. However, the drop is smaller than removing other components.

To sum up, removing each component results in a drop in the accuracy, and removing MTL drops more, which demonstrates that all these proposed components are necessary to improve the performance of AST-level code completion, and MTL contributes more to the improvements.

For the token-level code completion, we perform an ablation study to examine the effects of the components used in our best model (transformer-based): the multi-task learning mechanism and the subword encoder. We conduct experiments without either MTL or subword encoder, and also conduct experiment of removing the recurrent mechanism from the Transformer-XL architecture. The results are shown in Table 6. As seen from the results, similar to AST-level completion, removing either MTL or subword encoder results in a drop in the accuracy, and removing MTL drops more, which demonstrates that both the

**Table 6** Effectiveness of each component in our proposed model for the token-level code completion

| | Java | | | TypeScript | | |
|---|---|---|---|---|---|---|
| | Type | Value | ID | Type | Value | ID |
| UMTLM (Trans) | 66.7% | 75.0% | 51.9% | 76.4% | 74.9% | 45.9% |
| - MTL | 63.2% | 72.1% | 43.6% | 73.5% | 73.9% | 37.5% |
| - Subword Encoder | 65.2% | 74.2% | 50.5% | 74.0% | 74.2% | 43.2% |
| - Recurrence | 58.2% | 66.6% | 39.9% | 68.1% | 63.9% | 31.2% |

multi-task learning mechanism and the subword encoder are necessary to improve the performance of token-level code completion, and MTL contributes more to the improvements. Besides, removing the recurrent mechanism from our full model, the accuracy also drops a lot, which further demonstrates that capturing long-range dependency is of great importance and necessity for token-level source code modeling.

**RQ4: How does our proposed components perform in AST-level and token-level code 830 completion?** For Code Element Encoder, we try two different encoders: word encoder and subword encoder to encode the value of the token and AST node. As shown from Tables 5 and 6, when we replace the subword encoder with the word encoder, the performance becomes a little worse for both AST-level and token-level code completion. However, the subword encoder provides competitive results at a higher computational cost (needed for composing the representation of each subword, increase about 0.05ms on predicting a single suggestion). Considering the trade-off between the performance and cost, we suggest to use word encoder to encode the value of the token and AST node.

For contextual code encoder, we conduct experiments on using transformer-based contextual code encoder and RNN-based contextual code encoder. As seen from Tables 3 and 4, on both AST-level and token-level code completion, the transformer-based model works better than the RNN-based model, especially in token-level.

For Multi-task Learning, experimental results demonstrate that multi-task learning can bring improvements on both token-level code completion and AST-level code completion. The improvements token-level completion are larger than AST-level, especially on identifiers completion. The reason lies in that the correlation between the type and value prediction tasks are closer in token-level completion. For the AST-level code completion, the type and value are extracted from AST node's attributes, where the type is more general and one type can correspond to many values, which results in low correlation. For the token-level code completion, we extract the identifiers' type based on static analysis or developers' annotations, where the type of the token is more specific and contains more information about the token's value. Thus, multi-task learning can work better in token-level completion since more precise knowledge can be shared between tasks.

**RQ5: Computational Cost Per-Suggestion** To measure the computational cost per-suggestion, we compute the average time needed for our proposed model (RNN-based and Transformer-based) and baseline models to predict a single suggestion. We evaluate the running time on a Linux server with a NVIDIA GeForce GTX 1080 Ti GPU with 12 GB memory. We compute the cost for both the RNN-based setting and Transformer-based setting. For baseline models, we evaluate the baseline models used in our experiments except for BPE-NLM (Karampatsis et al. 2020). Since they split each token into sub-tokens with Byte Pair Encoding algorithm, and then use a variation of the beam search algorithm to combine the sub-tokens to complete tokens, which is very time-consuming. Thus, it will be

**Table 7** Computational cost per-suggestion

| Model | Time(ms) |
|---|---|
| PMN | 0.092 |
| Transformer-XL | 0.07 |
| UMTM (RNN) | 0.17 |
| UMTM (Trans) | 0.11 |

unfair for them to directly compare the time for predicting a complete token. The results are shown in Table 7. For baseline models, PMN is slower than Transformer-XL model because of the large amount of computations. For our proposed models, Transformer-based setting are faster than the RNN-based setting. Overall, our models make predictions in under 1 ms which makes them eligible for real-time code completion systems.

## 6 Discussion

### 6.1 Learning Process Analysis

To find out why our proposed model performs better, we analyze the learning process of the state-of-the-art baseline model (Pointer Mixture Network Li et al. 2018) and our proposed model. Figure 5 shows the loss of predicting the next node's type after every epoch on Python's training and test set for the two models. As seen from the figure, the difference between the training loss and test loss is large in the baseline model, which is obviously the result of over-fitting. While in our model, the difference is much smaller. Furthermore, the test loss of our model is lower than the baseline model at each epoch. The reason lies in three aspects: (1) by utilizing the hierarchical structural information of AST and the information contained in the training signals of related tasks, our proposed model can extract more accurate and common features from programs, and thus can achieve better performance; (2) adopting the Transformer-XL architecture to model the long-range dependency in the programs helps our model capture more information from the context and thus improves

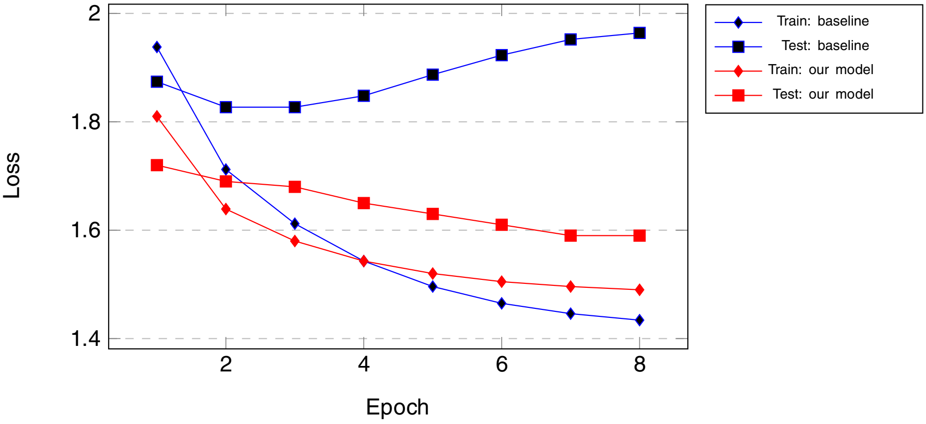


**Fig. 5** The cross-entropy loss on training and test set for baseline model and our model

model's performance; (3) multi-task learning provides an effective regularization method through knowledge sharing among tasks, thus can improve the model's performance by decreasing the difference between training and test loss, which to some extent prevents the model from over-fitting. For another two datasets, i.e., Java and JavaScript, we have the same observations and findings.

## 6.2 Training Cost Analysis

To evaluate the cost of the improvements, we count the number of parameters and record the training time of our best model and PMN (Li et al. 2018). To evaluate the cost of our proposed components, we also present these statistics data of the vanilla Transformer-XL network and removing one of the components from our model. We take the training time in the Python dataset as an example. The run-time in the test process is very fast (about 0.1 milliseconds per query), and the difference in the test time among different models is little. Thus, we do not compare the test time. The number of trainable parameters and the training time are presented in Table 8.

For the number of training parameters, the 6-layer Transformer-XL network uses only 59% of the parameter budget compared to PMN (Li et al. 2018) but can achieve comparable performance with them. In UMTLM, we adopt Transformer-XL as the language model and apply Multi-task Learning to learn two tasks jointly and propose a new Path2root encoder, which leads to an increase of the trainable parameters compared with the vanilla Transformer-XL networks. In our framework, code element encoder, contextual code encoder, and Path2root encoder are shared among all tasks, and only the output layers are task-specific. Thus, the parameter increasing is slight, only by 3.2% (from 95.8M to 98.9M). However, the number of trainable parameters of our model is only 60.8% of the number of trainable parameters in PMN. Besides, we also count the number of the parameters of removing MTL or Path2root encoder from our model, and the results are presented in the last two rows in Table 8. The results demonstrate that the additional parameters of integrating these two components into Transformer-XL increase a small number of parameters.

For the training time, UMTLM spends 74% of the time compared to PMN (Li et al. 2018). In PMN, they adopt LSTM as the language model, where most of the recurrent computations are performed during the hidden states' updating process. While in UMTLM, Transformer-XL (Dai et al. 2019) is used as the language model. In Transformer-XL, the representations of each input for each segment are computed relying on the self-attention layers, and the recurrence only happens between segments. Thus, it allows for substantially more parallelization and requires less time to train. When removing MTL, the training time decreases slightly (from 25 hours to 22 hours) because most of the parameters are shared between tasks. Thus, applying MTL will not introduce much additional training time during

**Table 8** Training cost analysis in the Python dataset

| Model | # of Parameters | Training time |
|---|---|---|
| PMN | 162.6M | 34 hours |
| Transformer-XL | 95.8M | 15 hours |
| UMTLM (Trans) | 98.9M | 25 hours |
| - MTL | 96.8M | 22 hours |
| - Path2root Encoder | 97.6M | 20 hours |

**Table 9** The results of different weight settings in our AST-level model

| $\alpha_1$ | $\alpha_2$ | Python | | | Java | | | JavaScript | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Type | Value | ID | Type | Value | ID | Type | Value | ID |
| 0.7 | 0.3 | **87.1%** | 54.4% | 71.6% | **83.4%** | 76.4% | 48.7% | **91.4%** | 80.5% | 67.7% |
| 0.5 | 0.5 | 85.6% | 72.1% | 54.7% | 83.0% | 76.6% | 49.4% | 91.0% | 81.3% | 68.0% |
| 0.3 | 0.7 | 84.1% | **73.8%** | **55.1%** | 82.7% | **76.8%** | **49.9%** | 89.8% | **82.7%** | **68.9%** |

the training process. Adding a Path2root encoder into our model is an improvement towards the model's structure. It increases the model's complexity, which leads to increased training time. When removing the Path2root encoder from our full model, the training time is reduced by 5 hours. Compared to vanilla Transformer-XL, applying the MTL and Path2root encoder will increase the training time, but considering the improvements, the increase is acceptable.

In summary, our model uses 59% of the parameter budget and spends 74% of the runtime to train compared to PMN (Li et al. 2018), and can still outperform them statistically significant and by a substantial margin. We also have the same observations and results for the other two datasets, i.e., Java and JavaScript.

### 6.3 Effect of Weights for Task-specific Loss.

In UMTLM, we use a weighted sum over task-specific losses as the final loss. By default, we set the weights for the two tasks as 0.5 and 0.5. The performance of the model is related to the choice of weighting between the tasks' loss. To show the effect of the weights, we present the results of different weight settings on our model in Tables 9 and 10. $\alpha_1$ is the weight of the loss for the type prediction task, and $\alpha_2$ is the weight of the loss for the value prediction task. As expected, when giving more weight to a task's loss, the accuracy of this task will be increased.

### 6.4 The effect of different learning mechanisms in our MTL framework

In our approach, we adopt the multi-task learning framework to predict the next token's (node's) type and value. We explore two ways of learning these two tasks, i.e., Jointly and Type-first. The results of the different ways for AST-level and token-level code completion are shown in Tables 11 and 12. As seen from the results, the performance of the different learning ways differs in these two completion tasks. For the AST-level code completion, predicting two tasks jointly performs better than type-first. For the token-level completion, the results are reversed, where the type-first approach achieves better performance. The

**Table 10** The results of different weight settings in our token-level model

| $\alpha_1$ | $\alpha_2$ | Java | | | TypeScript | | |
|---|---|---|---|---|---|---|---|
| | | Type | Value | ID | Type | Value | ID |
| 0.7 | 0.3 | **66.7%** | 74.7% | 51.2% | **76.4%** | 73.8% | 44.0% |
| 0.5 | 0.5 | 64.5% | 74.2% | 51.7% | 73.3% | 74.5% | 45.2% |
| 0.3 | 0.7 | 64.8% | **75.0%** | **51.9%** | 71.8% | **74.9%** | **45.9%** |

**Table 11** AST-level code completion accuracy comparison of different learning mechanisms

| | | Python | | | Java | | | JavaScript | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Type | Value | ID | Type | Value | ID | Type | Value | ID |
| UMTLM (Trans) | Jointly | **87.1%** | **73.8%** | **55.1%** | **83.4%** | **76.7%** | **49.9%** | **91.4%** | **82.7%** | **68.9%** |
| | Type-first | 84.9% | 71.4% | 53.4% | 79.6% | 72.2% | 48.4% | 86.5% | 78.0% | 68.1% |

reason lies in that the correlation between the type and value on the AST node is weaker than source code token. As shown in Tables 1 and 2, the category of the AST node's type is much less compared with the token's type. In ASTs, one type can correspond to many values, which could result in low correlation. When first predicting the node's type, and utilizing the predicted type to assist the value prediction, the model will add a more restrictive constraint between the type and value, which might not be helpful in some cases. When predicting type and value jointly, the learning process of the two tasks is more flexible, where the constraint between the type and value prediction is less restricted. For the token-level code completion, the type of the token (which is extracted through static analysis or human annotation) is more specific and contains more information about the token's value. Thus, the correlation between the type and value for the source code tokens is more close. The predicted type can offer more relevant and precise information for the value completion process. Thus, the type-first approach can achieve better performance for the token-level code completion.

## 6.5 Qualitative Analysis

**Difficult type predictions** Predicting the structure of the code, such as loops, if statements, and exception handling statements, is overall a very hard task (Raychev et al. 2016). Raychev et al. (2016) define a set of types on JavaScript that are hard to predict and name them as “difficult type prediction”. We evaluate our model's performance on these types' prediction and compare our model with PMN (Li et al. 2018) on the same test set. The results are shown in Table 13. As seen from the table, our model outperforms PMN by a large margin in all these types. Besides, in our model, the variance of the accuracies for predicting each token is much smaller than the PMN. The accuracies are mostly distributed in the range of 88% - 93%. In PMN, the accuracies of those low-frequency tokens are very low. For example, “SwitchStatement” only appears 2625 times in the test set, the accuracy is only 45.9% in PMN. While in our model, the accuracy is 88.2%, which is much higher than the PMN. These results demonstrate that our model can discover the structure of programs and achieve an excellent generalization performance on structure predictions.

**Example completion** Here, we present code completion examples on Python AST-level code completion to analyze the performance of UMTLM. We take several positions in a Python code snippet to test the performance of UMTLM and the baseline model. We show

**Table 12** Token-level code completion accuracy comparison of different learning mechanisms

| | | Java | | | TypeScript | | |
|---|---|---|---|---|---|---|---|
| | | Type | Value | ID | Type | Value | ID |
| UMTLM (Trans) | Jointly | 64.9% | 74.0% | 50.5% | 75.8% | 74.1% | 42.0% |
| | Type-first | **66.7%** | **75.0%** | **51.9%** | **76.4%** | **74.9%** | **45.9%** |

**Table 13** Difficult type predictions on JavaScript

| Difficult type | PMN | UMTLM (Trans) |
|---|---|---|
| ContinueStatement | 65.6% | **88.5%** |
| ForStatement | 65.5% | **89.0%** |
| WhileStatement | 79.8% | **88.9%** |
| ReturnStatement | 61.4% | **89.0%** |
| SwitchStatement | 45.9% | **88.2%** |
| ThrowStatement | 54.1% | **88.0%** |
| TryStatement | 57.3% | **88.9%** |
| IfStatement | 68.3% | **89.0%** |

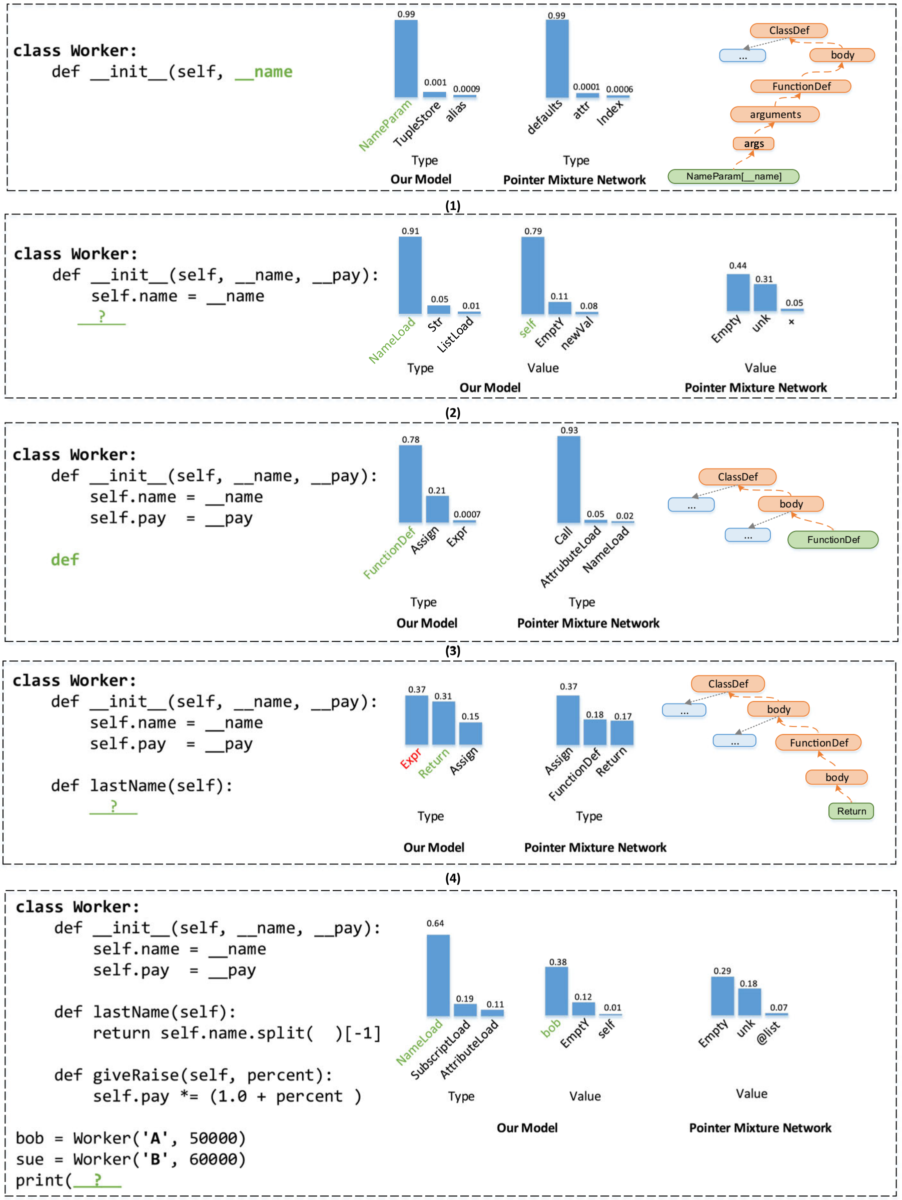
the top three predictions of our model and the baseline model of PMN (Li et al. 2018). The results are shown in Fig. 6. We divide the cases of the prediction into two situations:

a) **The effect of the path information**. In the first example, the target prediction *__name* is a parameter for the function *__init__*, and its corresponding node's type is *NameParam*. The path from it to the root node (shown on the right side of the example) implies the information that the prediction is a parameter of a function, thus it can help our model to make the correct prediction on the node's type. For the baseline model, it can only learn from the sequential context and fail to produce the right prediction. Similarly, in the third example, the target prediction *def* means a function definition, where its corresponding node's type is *FunctionDef*. With the information contained in the path, our model can make the correct prediction, while the baseline model fails. In the fourth example, both of our model and the baseline model fail to produce the correct prediction *return*. In this case, the path cannot provide accurate information because there exist many possible children for a function's body. Thus, our model produces *Expr*, which is also a grammatical child. The correct prediction is ranked second in our model and is ranked third in the baseline model. In cases like this, our model might make wrong predictions.

b) **The effect of MTL**. In the second example, the target prediction *self* is not a new variable and has been used in the previous context. By correctly predicting *NameLoad* in the node's type prediction task, our model can realize the value of the node is an already used value in the previous context. Thus it can identify the value from the context. For the baseline model, it may not realize the prediction is a variable accessing operation without the help of the auxiliary task. Thus, it just predicts *EMPTY*, which is the most frequent node's value in our corpus. The last example is also in the same way.

### 6.6 Threats to Validity

**Threats to external validity** relate to the quality of the datasets we used and the generalizability of our results. For the AST-level code completion, we use Python, Java and JavaScript datasets. Python and JavaScript datasets are two benchmarked datasets that have been used in previous code completion work (Raychev et al. 2016; Liu et al. 2016; Li et al. 2018). Java dataset we used is from Hu et al. (2018). For token level code completion, we evaluate our model on Java and TypeScript datasets. The reasons for using these two languages are as follows. These two languages are commonly used for software development, and we can get the identifiers' type through static analysis or through the developers' annotations. All of the programs in the datasets are collected from GitHub repositories and contain the



```
class Worker:
    def __init__(self, __name
```

(1)

```
class Worker:
    def __init__(self, __name, __pay):
        self.name = __name
        ?
```

(2)

```
class Worker:
    def __init__(self, __name, __pay):
        self.name = __name
        self.pay  = __pay

    def
```

(3)

```
class Worker:
    def __init__(self, __name, __pay):
        self.name = __name
        self.pay  = __pay

    def lastName(self):
        ?
```

(4)

```
class Worker:
    def __init__(self, __name, __pay):
        self.name = __name
        self.pay  = __pay

    def lastName(self):
        return self.name.split(  )[-1]

    def giveRaise(self, percent):
        self.pay *= (1.0 + percent )

bob = Worker('A', 50000)
sue = Worker('B', 60000)
print( ? 
```

(5)

**Fig. 6** Code completion examples

large-scale amount of source code files. However, further studies are needed to validate and generalize our findings to other programming languages. For the dataset splits, following existing research (Li et al. 2018), we split the programs into train/test set randomly. We did not control for time added to the repository, which might lead to small chance where the programs in the training set are created after the programs in the test set. Furthermore, our case study is small scale. More user evaluation is needed to confirm and improve the usefulness of our code completion model.

**Threats to internal validity** include the influence of the weightings between each task's loss i.e., $\alpha_k$. The performance of our model would be affected by the different weights (discussed in Section 6.3), which are tuned by hand in our experiments. However, the default weight settings of 0.5 and 0.5 for the next node's type and value prediction loss can still achieve a considerable performance increase. Take the experiments on the Python dataset as an example, default weight setting achieves 5% (from 80.6% to 85.6%) improvements in accuracy on the next node's type prediction compared with Li et al. (2018), which are only 1.5% lower than the best weight settings. The results in the next node's value prediction are also similar. Another threat to internal validity relates to the errors in the implementation of the baseline methods. For Hellendoorn and Devanbu (2017), we directly used their published jars. Thus, there is little threat to approach implementation. For PMN, which is originally used for the AST-level code completion, we also consider it as a baseline of the token-level code completion. In their original model, the additional information derived from ASTs is utilized to improve the performance. The results of using the token sequence as input might understate the accuracy of the plain PMN. However, we have tried our best to make fair comparison with PMN by only changing the format of the input, and keeping the model unchanged. For BPE NLM (Karampatsis et al. 2020), we compare our model with the static setting of their model considering the fairness of the comparison. We realize that evaluating dynamically may improve accuracy. The dynamic and maintenance scenarios are not implemented and compared in this work, which will be considered as our future work.

**Threats to construct validity** relate to the suitability of our evaluation measure. We use accuracy as the metric which evaluates the proportion of correctly predicted next token's type or value, and ID accuracy to measure the proportion of correctly predicted identifiers. Accuracy is a classical evaluation measures for code completion and have been used in many the previous code completion work (Hindle et al. 2012; Tu et al. 2014; Raychev et al. 2016; Hellendoorn and Devanbu 2017; Li et al. 2018; Karampatsis et al. 2020).

Besides, there is another threat on the performance comparison between AST-level and token-level completion. The experiment design for the token-level and AST-level is not exactly the same, including the input format and model components. Thus, the results of the AST-level and token-level code completion (accuracy numbers from Tables 3 and 4 in the Java dataset) cannot be strictly compared with each other even in the same dataset. The details are shown as follows:

- Input format: for token-level completion, the input token sequence is directly produced by tokenize the program sequentially, and the type information of the identifiers are extracted through static analysis tools; for AST-level completion, the input node sequence is produced by traversing the AST in depth-first order, each node contains the type attribute and leaf node also contains the value attribute. The AST node and token is not a not one-to-one correspondence. For example, in the AST, there are no corresponding node for the braces and semicolon in the token sequence. Besides, the order of the input is also different. Represent the program as the token sequence preserves the order of typing process. When represent the programs as AST node sequence by traversing the AST, the order of the node sequence are inconsistent with the token sequence. Furthermore, the type of the node and token also means different. In AST nodes, the type refer to the type attribute of the tree node, which represent the type of the node's corresponding program structure, and each node contains the type attribute. for example, ClassDeclaration, MethodInvocation, Assignment, etc. In tokens, the type corresponds to the single token's type, for example, java.lang.String, java.util.ArrayList, android.widget.TextView, etc. The type information is extracted through static analysis tools, where only part of the identifiers have the type information.

- Model components: Considering the characteristics of the AST data, we design the path2root encoder to capture the hierarchical structural information of the tree, which cannot be applied to token-level completion since there is no explicit tree structure in the token sequence.

Thus, it is hard to make the results of the AST-level and token-level completion comparable due to the above differences.

The last threat is that it is hard to apply our model in TDD scenario, where the model might need to predict a piece of code that does not exist anywhere. That is, the predicted token is not in the vocabulary. Under this situation, our model cannot predict it correctly. Existing deep learning based code completion models including our model aim at improving the quality and efficiency of software development by suggesting the common usage by training the model with the huge popular projects. These models can ease the programmer's burden greatly, and the programmers can pay more attention on implementing the complex functionality. The OOV rates in the test dataset (5-13%) can also demonstrate that in most cases, our model can adapt well since our model is trained on the huge popular projects. For the projects whose domain is significantly different from the general projects, the domain-specific and the local information should be considered and which is now studied in our another research.

## 7 Related Work

**Code Completion** Code completion is a hot research topic in the field of software engineering. Early work in code completion bases on on heuristic rules and static type information to make suggestions (Hou and Pletcher 2010), or bases on similar code examples (Bruch et al. 2009) and program history data (Robbes and Lanza 2008). Since Hindle et al. (2012) found that source code contained predictable statistical properties, statistical language models began to be used for modeling source code (Nguyen et al. 2013; Tu et al. 2014; Hellendoorn and Devanbu 2017; Li et al. 2018), where N-gram is the most widely used model. Tu et al. (2014) observed that source code has a unique property of localness, which could not be captured by the traditional N-gram model. They improved N-gram by adding a cache mechanism to exploit localness and achieved better performance than other N-gram based models. Hellendoorn and Devanbu (2017) introduced an improved N-gram model that considered the unlimited vocabulary, nested scope, locality, and dynamism in source code. Their evaluation results on code completion showed that their model outperformed existing statistical language models, including deep learning based models. Thus we choose their model as a baseline. Raychev et al. (2016) proposed a probabilistic model based on decision tree and domain-specific grammars. They performed experiments to predict AST nodes on Python and JavaScript datasets.

In recent years, deep recurrent neural network-based language models have been applied to learning source code and have made great progress (White et al. 2015; Bhoopchand et al. 2016; Li et al. 2018). Liu et al. (2016) proposed a code completion model based on a vanilla LSTM network. Bhoopchand et al. (2016) proposed an RNN model with a sparse pointer mechanism aiming at capturing long-range dependencies. Li et al. (2018) proposed a PMN to address the OoV issue. For the next node's type prediction, their model outperforms (Raychev et al. 2016) on both Python and JavaScript datasets. For the next node's value prediction, their model outperforms (Raychev et al. 2016) on Python and achieves comparable performance on JavaScript. Svyatkovskiy et al. (2019) proposed a code completion system based on LSTM for recommending Python method calls. Their system is deployed as part of the Intellicode extension in Visual Studio Code IDE. Karampatsis et al. (2020) proposed a large-scale open-vocabulary neural language model for source code, which leverages the BPE algorithm, beam search algorithm, and cache mechanism to both keep vocabulary size low and successfully predict OoV tokens. The experimental results demonstrate that their open vocabulary model outperforms both N-gram models and closed vocabulary neural language models, and achieve state-of-the-art performance on token-level code completion. Liu et al. (2020) proposed a multi-task learning based neural language model for AST-level code completion. They employed Transformer-XL as their backbone model, and introduced multi-task learning mechanism to predict next node's type and value jointly. Svyatkovskoy et al. (2020) implemented and evaluated a number of neural code completion models, which offer varying trade-offs in terms of memory, speed and accuracy. They provided a well-engineered approach to deep-learning based code completion, which is important to the software engineering community. Most recently, Liu et al. (2020) proposed CugLM, a pre-trained language model for code understanding and generation. They pre-trained their model on two massive datasets and with three objective functions and then fine-tune it on token-level code completion task. They also utilized the static type information enhance the performance of identifier completion.

In this paper, we build a unified framework that can perform code completion on both token-level and AST-level, which allows us to perform a series of design decisions that can help us pick a good trade-off among the desired properties of a completion system. Besides, we also explore whether different model settings or techniques perform the same for different input formats (tokens and ASTs), thus can provide a reference for later code completion research.

**Multi-task Learning** Multi-task learning has been used successfully across many fields including natural language processing (Liu et al. 2015; Guo et al. 2018; Devlin et al. 2018), speech recognition (Deng et al. 2013) and computer vision (Long and Wang 2015; Lu et al. 2017). In the natural language processing area, MTL has been proven effectively in many tasks, such as machine translation (Luong et al. 2016; Dong et al. 2015; Zaremoodi et al. 2018), text summarization (Isonuma et al. 2017; Guo et al. 2018), and sequence labeling (Peng and Dredze 2017; Lin et al. 2018).

In the field of source code modeling, there are also some studies focus on learning multiple tasks, including code retrieval, code generation, code comment generation, etc. Yao et al. (2019) adopted reinforcement learning to learn code summarization and code retrieval jointly, and design rewards based on these two tasks. Wei et al. (2019) proposed a dual learning network to optimize two tasks of code generation and code summarization simultaneously. They designed two regular term constraints based on the duality of the two tasks. Feng et al. (2020) proposed CodeBERT, a bimodal pre-trained model for natural language and programming language, aiming at capturing the semantic connection between natural language and programming language. They trained CodeBERT with masked language modeling task and replaced token detection task, and evaluated it on two downstream tasks, including natural language code search and code documentation generation.

In this paper, we apply MTL to code completion to predict the type and value of the next code element jointly and improve the state-of-the-art statistically significant and substantially.

## 8 Future Work

In our paper, the training is done per language. For each language, we use a large program data corpus to train the code completion model, aiming at capturing the general code patterns as much as possible. Most of existing deep learning based code completion research adopt this way to train and evaluate their approaches, where project-specific information is not explicitly considered. Since the data corpus used to train the model are big which covers the popular projects in the open source platform (github), when applying the trained model to the test programs, the model can perform well in most cases. According to the statistics in our experiment, the OoV rates in the test dataset are 5-13% for different languages.

When the model is applied to the unpopular projects, where the classes, APIs, identifiers are significantly different from those of training datasets, the results might be poor since the data distribution between the testing project and the training projects differs largely. In this situation, to obtain good results, the local code pattern need to be considered explicitly. This is caused by “domain shift”. To address this issue, in the future, we plan to build a new model which integrates a local model and a global model, where a light-weighted local model is used to learn the project-specific local code pattern to make for the lack of the domain knowledge.

## 9 Conclusion

In this paper, we propose a unified multi-task learning based framework for both AST-level and token-level code completion. We design a code element encoder to encode each code element into a distributed vector representation. To capture the long-term dependency in the programs, we build a Transformer-XL network based contextual code encoder to encode the contextual code into a distributed vector representation. To model the hierarchical information of the program explicitly, we propose a novel Path2root encoder to encode the AST paths from the predicting node to the root node. To utilize the relationship between the type and value of the code elements, we apply MTL framework to predict the type and value of the next code element jointly, which enables knowledge sharing between these two tasks. Experimental results demonstrate that the proposed model achieves better results than previous state-of-the-art models on both AST-level and token-level code completion. In the future, we plan to improve the effectiveness of our proposed model by introducing domain-specific customizations to make for the lack of the domain knowledge.

**Acknowledgments** This research is supported by the National Natural Science Foundation of China under Grant Nos. 62072007, 62192731, 62192733, 61832009, 62192730. Zhi Jin and Ge Li are corresponding authors.

## References

Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M et al (2016) Tensorflow: A system for large-scale machine learning. In: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), pp 265–283

Bahdanau D, Cho K, Bengio Y (2015) Neural machine translation by jointly learning to align and translate

Bhoopchand A, Rocktäschel T, Barr ET, Riedel S (2016) Learning python code suggestion with a sparse pointer network. CoRR arXiv:1611.08307

Bielik P, Raychev V, Vechev MT (2016) PHOG: probabilistic model for code. In: Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, JMLR Workshop and Conference Proceedings, vol 48. JMLR.org, New York City, pp 2933–2942

Bruch M, Monperrus M, Mezini M (2009) 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, pp 213–222

Caruana R (1997) Multitask learning. Mach Learn 28(1):41–75

Chelba C, Engle D, Jelinek F, Jimenez V, Khudanpur S, Mangu L, Printz H, Ristad E, Rosenfeld R, Stolcke A, Wu D (1997) Structure and performance of a dependency language model. In: Fifth European Conference on Speech Communication and Technology, EUROSPEECH 1997, Rhodes

Chelba C, Jelinek F (1998) Exploiting syntactic structure for language modeling. In: Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics-Volume 1. Association for Computational Linguistics, pp 225–231

Cho K, van Merrienboer B, Bahdanau D, Bengio Y (2014) On the properties of neural machine translation: Encoder-decoder approaches, pp 103–111

Costa C, Figueiredo J, Murta L, Sarma A (2016) Tipmerge: recommending experts for integrating changes across branches. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, pp 523–534

Dai Z, Yang Z, Yang Y, Carbonell JG, Le QV, Salakhutdinov R (2019) Transformer-xl: Attentive language models beyond a fixed-length context. In: Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers, pp 2978–2988

Deng L, Hinton GE, Kingsbury B (2013) New types of deep neural network learning for speech recognition and related applications: an overview. In: IEEE international conference on acoustics, speech and signal processing, ICASSP 2013, vancouver, bc, canada, may 26-31, 2013. IEEE, pp 8599–8603

Devlin J, Chang M-W, Lee K, Toutanova K (2018) BERT: pre-training of deep bidirectional transformers for language understanding. CoRR arXiv:1810.04805

Dong D, Wu H, He W, Yu D, Wang H (2015) Multi-task learning for multiple language translation. In: Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, Volume 1: Long Papers. The Association for Computer Linguistics, Beijing, pp 1723–1732

Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D et al (2020) Codebert: A pre-trained model for programming and natural languages. arXiv:2002.08155

Gage P (1994) A new algorithm for data compression. C Users J 12(2):23–38

Guo H, Pasunuru R, Bansal M (2018) Soft layer-specific multi-task summarization with entailment and question generation. In: Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Volume 1: Long Papers. Association for Computational Linguistics, Melbourne, pp 687–697

Han J, Deng S, Xia X, Wang D, Yin J (2019) Characterization and prediction of popular projects on github. In: 2019 IEEE 43rd annual computer software and applications conference (COMPSAC), vol 1. IEEE, pp 21–26

Hellendoorn VJ, Bird C, Barr ET, Allamanis M (2018) Deep learning type inference. In: Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering, pp 152–162

Hellendoorn VJ, Devanbu PT (2017) Are deep neural networks the best choice for modeling source code? In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017. ACM, pp 763–773

Hellendoorn VJ, Proksch S, Gall HC, Bacchelli A (2019) When code completion fails: A case study on real-world completions. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, pp 960–970

Hindle A, Barr ET, Su Z, Gabel M, Devanbu PT (2012) On the naturalness of software. In: 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland. IEEE Computer Society, pp 837–847

Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural Comput 9(8):1735–1780

Hou D, Pletcher DM (2010) 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, pp 26–30

Hu X, Li G, Xia X, Lo D, Lu S, Jin Z (2018) Summarizing source code with transferred API knowledge. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018. ijcai.org, Stockholm, pp 2269–2275

Isonuma M, Fujino T, Mori J, Matsuo Y, Sakata I (2017) Extractive summarization using multi-task learning with document classification. In: Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP 2017. Association for Computational Linguistics, Copenhagen, pp 2101–2110

Karampatsis R-M, Babii H, Robbes R, Sutton C, Janes A (2020) Big code!= big vocabulary: Open-vocabulary models for source code. ICSE

Khandelwal U, He H, Qi P, Jurafsky D (2018) Sharp nearby, fuzzy far away: How neural language models use context, pp 284–294

Kingma DP, Ba J (2015) Adam: A method for stochastic optimization. In: Bengio Y, LeCun Y (eds) 3rd International Conference on Learning Representations, ICLR 2015, Conference Track Proceedings, San Diego

Li J, Wang Y, Lyu MR, King I (2018) Code completion with neural attention and pointer networks. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018. ijcai.org, Stockholm, pp 4159–4165. https://doi.org/10.24963/ijcai.2018/578

Lin Y, Yang S, Stoyanov V, Ji H (2018) A multi-lingual multi-task architecture for low-resource sequence labeling. In: Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Volume 1: Long Papers. Association for Computational Linguistics, Melbourne, pp 799–809

Liu C, Wang X, Shin R, Gonzalez JE, Song D (2016) Neural code completion

Liu F, Li G, Wei B, Xia X, Fu Z, Jin Z (2020) A self-attentional neural architecture for code completion with multi-task learning. In: Proceedings of the 28th International Conference on Program Comprehension, pp 37–47

Liu F, Li G, Zhao Y, Jin Z (2020) Multi-task learning based pre-trained language model for code completion. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, pp 473–485

Liu X, Gao J, He X, Deng L, Duh K, Wang Y-Y (2015) Representation learning using multi-task deep neural networks for semantic classification and information retrieval. In: NAACL HLT 2015, the 2015 conference of the north american chapter of the association for computational linguistics: Human language technologies. The Association for Computational Linguistics, Denver, pp 912–921

Long M, Wang J (2015) Learning multiple tasks with deep relationship networks. CoRR arXiv:1506.02117

Lu Y, Kumar A, Zhai S, Cheng Y, Javidi T, Feris RS (2017) Fully-adaptive feature sharing in multi-task networks with applications in person attribute classification. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017. IEEE Computer Society, Honolulu, pp 1131–1140

Luong M-T, Le QV, Sutskever I, Vinyals O, Kaiser L (2016) Multi-task sequence to sequence learning

Macbeth G, Razumiejczyk E, Ledesma RD (2011) Cliff's delta calculator: A non-parametric effect size program for two groups of observations. Univ Psychol 10(2):545–555

Maddison C, Tarlow D (2014) Structured generative models of natural source code. In: International Conference on Machine Learning, pp 649–657

Malik RS, Patra J, Pradel M (2019) Nl2type: inferring javascript function types from natural language information. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, pp 304–315

Nguyen AT, Nguyen TN (2015) Graph-based statistical language model for code. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol 1. IEEE, pp 858–868

Nguyen TT, Nguyen AT, Nguyen HA, Nguyen TN (2013) A statistical semantic language model for source code. In: Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13. ACM, Saint Petersburg, pp 532–542

Peng N, Dredze M (2017) Multi-task domain adaptation for sequence tagging. In: Proceedings of the 2nd Workshop on Representation Learning for NLP, Rep4NLP@ACL 20107. Association for Computational Linguistics, Vancouver, pp 91–100

Raychev V, Bielik P, Vechev MT (2016) Probabilistic model for code with decision trees. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016. ACM, Amsterdam, pp 731–747

Robbes R, Lanza M (2008) How program history can improve code completion. In: 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. IEEE, pp 317–326

Ruder S (2017) An overview of multi-task learning in deep neural networks. CoRR arXiv:1706.05098

Schuster M, Paliwal KK (1997) Bidirectional recurrent neural networks. IEEE Trans Signal Process 45(11):2673–2681

Svyatkovskiy A, Zhao Y, Fu S, Sundaresan N (2019) Pythia: Ai-assisted code completion system. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp 2727–2735

Svyatkovskoy A, Lee S, Hadjitofi A, Riechert M, Franco J, Allamanis M (2020) Fast and memory-efficient neural code completion. arXiv:2004.13651

Tu Z, Su Z, Devanbu PT (2014) On the localness of software. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22). ACM, Hong Kong, pp 269–280

Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser L, Polosukhin I (2017) Attention is all you need. In: Advances in neural information processing systems, pp 5998–6008

Wei B, Li G, Xia X, Fu Z, Jin Z (2019) Code generation as a dual task of code summarization. In: Advances in Neural Information Processing Systems, pp 6563–6573

White M, Vendome C, Vásquez ML, Poshyvanyk D (2015) Toward deep learning software repositories. In: 12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015. IEEE Computer Society, Florence, pp 334–345

Wilcoxon F (1945) Individual comparisons by ranking methods. Biometr Bullet 1(6):80–83

Yao Z, Peddamail JR, Sun H (2019) Coacor: code annotation for code retrieval with reinforcement learning. In: The World Wide Web Conference, pp 2203–2214

Zaremoodi P, Buntine WL, Haffari G (2018) Adaptive knowledge sharing in multi-task learning: Improving low-resource neural machine translation. In: Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Volume 2: Short Papers. Association for Computational Linguistics, Melbourne, pp 656–661

## Affiliations

**Fang Liu**[1,2] · **Ge Li**[1,2] · **Bolin Wei**[1,2] · **Xin Xia**[3] · **Zhiyi Fu**[1,2] · **Zhi Jin**[1,2]

✉ Zhi Jin
zhijin@pku.edu.cn

Fang Liu
liufang816@pku.edu.cn

Bolin Wei
bolin.wbl@gmail.com

Xin Xia
xin.xia@acm.org

Zhiyi Fu
ypfzy@pku.edu.cn

[1] Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing, China

[2] School of Computer Science, Peking University, Beijing, China

[3] Huawei, China