MetaTPTrans: A Meta Learning Approach for Multilingual Code Representation Learning

Weiguo Pian[1], Hanyu Peng[2], Xunzhu Tang[1], Tiezhu Sun[1], Haoye Tian[1*], Andrew Habib[1], Jacques Klein[1], Tegawendé F. Bissyandé[1,3]

[1] SnT, University of Luxembourg, Luxembourg
[2] Baidu Inc., Beijing, China
[3] CITADEL, Université Virtuelle du Burkina Faso
weiguo.pian@uni.lu penghanyu@baidu.com {xunzhu.tang, tiezhu.sun, haoye.tian}@uni.lu
andrew.a.habib@gmail.com {jacques.klein, tegawende.bissyande}@uni.lu

Abstract

Representation learning of source code is essential for applying machine learning to software engineering tasks. Learning code representation from a multilingual source code dataset has been shown to be more effective than learning from single-language datasets separately, since more training data from multilingual dataset improves the model's ability to extract language-agnostic information from source code. However, existing multilingual training overlooks the language-specific information which is crucial for modeling source code across different programming languages, while only focusing on learning a unified model with shared parameters among different languages for language-agnostic information modeling. To address this problem, we propose MetaTPTrans, a meta learning approach for multilingual code representation learning. MetaTPTrans generates different parameters for the feature extractor according to the specific programming language type of the input code snippet, enabling the model to learn both language-agnostic and language-specific information with dynamic parameters in the feature extractor. We conduct experiments on the code summarization and code completion tasks to verify the effectiveness of our approach. The results demonstrate the superiority of our approach with significant improvements on state-of-the-art baselines.

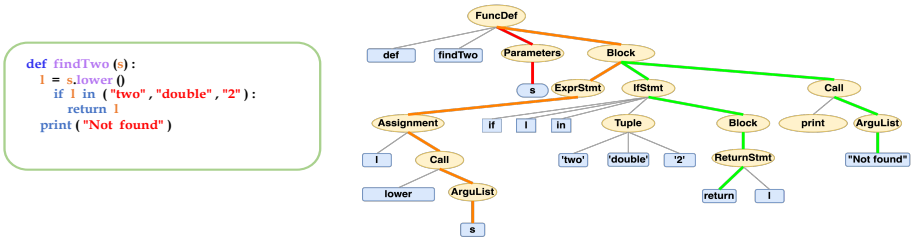
Introduction

Modeling source code aims at capturing both its syntax and semantics to enable applying machine learning to software engineering tasks such as code summarization (Zügner et al. 2021; Peng et al. 2021), code completion (Liu et al. 2020a,b), bug finding (Wang et al. 2016; Pradel and Sen 2018), patch correctness assessment (Tian et al. 2022a,b), etc. Inspired by the success of deep learning in the field of natural language processing (NLP) (Hochreiter and Schmidhuber 1997; Vaswani et al. 2017), modeling source code with deep learning techniques has attracted increasing attention from researchers in recent years (Alon et al. 2019a,b; Hellendoorn et al. 2020; Kim et al. 2021). Although programs are more repetitive and predictable (i.e. "natural" (Hindle et al. 2012)), unlike natural language text, they also contain rich structural information, e.g., ASTs, and data- and control-flow information. Therefore, a straightforward adoption of NLP models to source code struggles to capture structural information from the source code context (sequence of tokens).

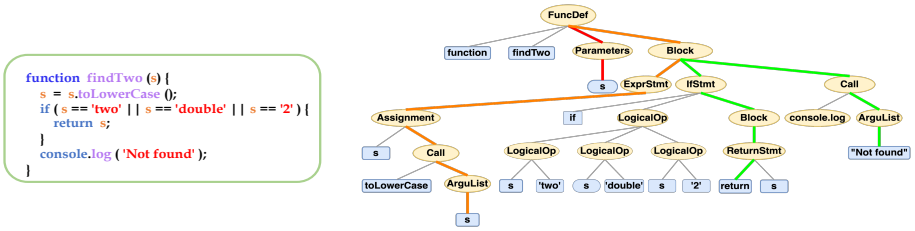
To alleviate the problem above, Alon et al. (2019a,b) proposed to incorporate the structural information by encoding pairwise paths on AST via LSTMs (Hochreiter and Schmidhuber 1997) to represent the source code. Inspired by the success of Graph Neural Networks (GNN) (Kipf and Welling 2017; Hamilton, Ying, and Leskovec 2017; Velickovic et al. 2018; Gasteiger, Groß, and Günnemann 2020) in modeling structural data, several works leverage GNNs on ASTs to capture the program structure. For instance, Allamanis, Brockschmidt, and Khademi (2018) used Gated Graph Neural Networks (GGNN) to learn program representation over ASTs and data flow graphs. Based on GGNN and programs graphs, Fernandes, Allamanis, and Brockschmidt (2019) proposed Sequence GNN for code summarization and Zhou et al. (2019) proposed a GNN-based approach to detect software vulnerabilities. However, GNN-based methods struggle to extract global structural information since GNNs focus on local message passing when aggregating information across nodes. Besides, the aforementioned approaches focus on structural information and ignore the contextual information when extracting features from source code. To model both structural and contextual information, Hellendoorn et al. (2020) combined the two kinds of information through Transformers (Vaswani et al. 2017) by biasing the self-attention process with relation information extracted from graph edge types. Zügner et al. (2021) proposed to bias the computation of self-attention with multiple kinds of pairwise distances between AST nodes and integrate them into the XLNet model (Yang et al. 2019). Lastly, Peng et al. (2021) introduced TPTrans, which biases the attention processes in Transformers with the encoding of relative and absolute AST paths.

In a different direction, Zügner et al. (2021) introduced their novel insight that multilingual training improves the performance of the model compared to training models on single-language datasets separately, since training the model with multilingual source code data enhances the model's ability to learn language-agnostic information (Zügner et al. 2021). The language-agnostic information is introduced in (Zügner et al. 2021), which means that the information can be di-

*Corresponding author.



(a) Python code snippet and its AST



(b) JavaScript code snippet and its AST

Figure 1: The same code snipped and its associated AST in (a) Python and (b) JavaScript.

rectly extracted from the source code or AST by a unified model across different languages, without relying on any language-specific features (Zügner et al. 2021). Such as structural information from the AST of the code, since these ASTs have a unified structural (tree) form across different programming languages (different programming languages have a unified rule of constructing an AST from the source code, and the structural modeling of ASTs is essentially to make the model understand such unified structural rule of them). In addition to language-agnostic information, other information is defined as language-specific information, e.g., the language-specific underlying syntactic rules, the language-specific API names, etc. As an example of language-agnostic information, Figure 1 shows two code snippets of the same function in (a) Python and (b) JavaScript. The two ASTs of the code snippets are shown on the right hand side of the respective figures. We can obviously see that the two ASTs have a unified structural (tree) form, besides, we also observe that they have some shared paths, e.g., the highlighted paths in red, orange, and green. Because of such language-agnostic information which can be extracted by a unified model directly across different languages, multilingual training shows superior performance compared to training models on single-language datasets separately.

Although the multilingual training strategy improves the model's performance significantly, it ignores the language-specific information hidden in the characteristics of each specific language. Considering the same example shown in Figure 1, we see that the context of the code snippets (the sequence of source code tokens) in different languages carries distinctive language-specific features, e.g., the different coding rules of the code context in different languages (language-specific underlying syntactic rules), the language-specific API names in different languages: `lower()` and `print()` in Python vs. `toLowerCase()` and `console.log()` in JavaScript. For the existing multilingual training, it struggles to learn such language-specific syntactic rules and language-specific token representation/projection with a unified model, *since a unified model is more inclined to learn the common characteristics of the input data*, i.e., the language-agnostic information under the scenario of multilingual code modeling.

To address this problem, in this paper, we propose MetaTPTrans, a meta learning based approach for multilingual code representation learning. Meta learning is a novel learning paradigm with the concept of learning to learn. There are several forms of learning to learn in meta learning, such as learning to initialize (Finn, Abbeel, and Levine 2017), learning the optimizer (Andrychowicz et al. 2016), learning hyperparameters (Li et al. 2021), learning to generate parameters of the feature extractor by another network (Bertinetto et al. 2016; Ha, Dai, and Le 2017; Chen et al. 2018; Wang et al. 2019; Pan et al. 2019, 2022). *The meta learning form we applied is the last one, i.e., learning to generate the model's parameters, in which the feature extractor can be named as Base Learner while the network for generating parameters is named as Meta Learner* (Wang et al. 2019; Pan et al. 2019, 2022). With this form of meta learning, our model can adjust its parameters dynamically regarding the programming language type of the input code snippet. This enables MetaTPTrans to not only extract language-agnostic information from multilingual source code data, but also capture the language-specific information, which is overlooked by standard multilingual training. Specifically, our approach consists of a language-aware Meta Learner and a feature extractor (Base Learner). The Meta Learner takes the programming language type (e.g., `Python`) as input and outputs a specific group of parameters for the feature extractor based on the associated language type. The language type is like an indicator to guide the Meta Learner to output specific parameters for a specific programming language. For the Base Learner, we apply TPTrans (Peng et al. 2021), a state-of-the-art code representation model, as the architecture of it. To the best of our knowledge, we are the first to leverage meta learning to learn to generate language-aware dynamic parameters for the feature extractor to learn both language-specific and

language-agnostic information from multilingual source code datasets. We evaluate MetaTPTrans on two common software engineering tasks: code summarization and code completion. The experimental results show that our approach outperforms state-of-the-art methods significantly on both tasks.

In summary, this paper contributes the following: (1) MetaTPTrans, a meta learning based approach for multilingual code representation learning, which learns both language-agnostic and language-specific information from multilingual source code data, (2) Three different schemes for generating parameters for different kinds of weights of the Base Learner in MetaTPTrans, and (3) Experimental evaluation on two important and widely-used software engineering tasks: code summarization and code completion, and the results show that our approach outperforms state-of-the-art baselines significantly. Our code is available at: https://github.com/weiguoPian/MetaTPTrans.

Technical Preliminaries

In this section, we introduce the foundations of absolute and relative position embedding in self-attention and the TPTrans model (Peng et al. 2021), upon which we build our model.

Self-Attention with Absolute and Relative Position Embedding

Self-attention (SA) is the basic module in Transformers (Vaswani et al. 2017). It maintains three projected matrices $\boldsymbol{Q} \in \mathbb{R}^{d_q \times d_q}$, $\boldsymbol{K} \in \mathbb{R}^{d_k \times d_k}$, and $\boldsymbol{V} \in \mathbb{R}^{d_v \times d_v}$ to compute an output that is the weighted sum of the input by attention score:

$$\begin{aligned} SA(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) &= softmax(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d}})\boldsymbol{V} \\ \textit{s.t.} \quad \begin{bmatrix} \boldsymbol{Q} \\ \boldsymbol{K} \\ \boldsymbol{V} \end{bmatrix} &= \boldsymbol{X} \begin{bmatrix} \boldsymbol{W}^Q \\ \boldsymbol{W}^K \\ \boldsymbol{W}^V \end{bmatrix} \end{aligned} \tag{1}$$

where $\boldsymbol{X} = (\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_n)$ is the input sequence of the self-attention module, $\boldsymbol{x}_i \in \mathbb{R}^d$, d is the dimension of the hidden state, and $\boldsymbol{W}^Q \in \mathbb{R}^{d \times d_q}$, $\boldsymbol{W}^K \in \mathbb{R}^{d \times d_k}$, $\boldsymbol{W}^V \in \mathbb{R}^{d \times d_v}$ are the learnable parameters matrices of the self-attention component. Here, we follow the previous works (Vaswani et al. 2017; Zügner et al. 2021; Peng et al. 2021) and set $d_q = d_k = d_v = d$. More specifically, the above equation can be reformulated as:

$$\begin{aligned} \boldsymbol{z}_i &= \sum_{j=1}^{n} \frac{\exp(\alpha_{ij})}{\sum_{k=1}^{n} \exp(\alpha_{ik})} (\boldsymbol{x}_j \boldsymbol{W}^V) \\ \textit{s.t.} \quad \alpha_{ij} &= \frac{(\boldsymbol{x}_i \boldsymbol{W}^Q)(\boldsymbol{x}_j \boldsymbol{W}^K)^{\mathrm{T}}}{\sqrt{d}} \end{aligned} \tag{2}$$

where $\boldsymbol{z}_i$ is the output of $\boldsymbol{x}_i$ calculated by self-attention operation. In Vanilla Transformer, Vaswani et al. (2017) used a non-parameteric absolute position encoding, which is added to the word vectors directly. Ke, He, and Liu (2021) proposed a learnable projection for absolute position for computing the attention score among words:

$$\alpha_{ij} = \frac{(\boldsymbol{x}_i \boldsymbol{W}^Q)(\boldsymbol{x}_j \boldsymbol{W}^K)^{\mathrm{T}}}{\sqrt{2d}} + \frac{(\boldsymbol{p}_i \boldsymbol{U}^Q)(\boldsymbol{p}_j \boldsymbol{U}^K)^{\mathrm{T}}}{\sqrt{2d}} \tag{3}$$

where $\boldsymbol{p}_i$ denotes the learnable real-valued vector of position i, and $\boldsymbol{U}^Q, \boldsymbol{U}^K \in \mathbb{R}^{d \times d}$ are the projection matrices of the position vectors $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$ respectively. To capture the relative position relationship between words, Shaw, Uszkoreit, and Vaswani (2018) proposed to use the relative position embedding between each two words:

$$\begin{aligned} \boldsymbol{z}_i &= \sum_{j=1}^{n} \frac{\exp(\alpha_{ij})}{\sum_{k=1}^{n} \exp(\alpha_{ik})} (\boldsymbol{x}_j \boldsymbol{W}^V + \boldsymbol{r}_{ij}^V) \\ \textit{s.t.} \quad \alpha_{ij} &= \frac{(\boldsymbol{x}_i \boldsymbol{W}^Q)(\boldsymbol{x}_j \boldsymbol{W}^K + \boldsymbol{r}_{ij}^K)^{\mathrm{T}}}{\sqrt{d}} \end{aligned} \tag{4}$$

where $\boldsymbol{r}_{ij}^K, \boldsymbol{r}_{ij}^V$ denote the learnable relative position embedding between positions i and j.

TPTrans

We briefly introduced how absolute and relative position embeddings are integrated into the self-attention module of Transformers. In this subsection, we describe the TPTrans (Peng et al. 2021) which is based on the aforementioned position embedding concepts.

TPTrans modifies the relative and absolute position embedding in self-attention with AST paths encodings so that AST paths could be integrated into Transformers. Specifically, they first encode the relative and absolute path via a bi-directional GRU (Cho et al. 2014):

$$\begin{aligned} \boldsymbol{r}_{ij} &= GRU(\boldsymbol{Path}_{\boldsymbol{x}_i \rightarrow \boldsymbol{x}_j}) \\ \boldsymbol{a}_i &= GRU(\boldsymbol{Path}_{\boldsymbol{root} \rightarrow \boldsymbol{x}_i}) \end{aligned} \tag{5}$$

where $\boldsymbol{Path}_{\boldsymbol{x}_i \rightarrow \boldsymbol{x}_j}$ denotes the AST path from node $\boldsymbol{x}_i$ to node $\boldsymbol{x}_j$, $\boldsymbol{r}_{ij}$ is the relative path encoding between positions i and j, and $\boldsymbol{a}_i$ is the absolute path (from the $\boldsymbol{root}$ node to node $\boldsymbol{x}_i$) encoding of position i. Then, the two types of path encodings are integrated into Eq. (3) - (4) by replacing the absolute and relative position embeddings:

$$\begin{aligned} \boldsymbol{z}_i &= \sum_{j=1}^{n} \frac{\exp(\alpha_{ij})}{\sum_{k=1}^{n} \exp(\alpha_{ik})} (\boldsymbol{x}_j \boldsymbol{W}^V + \boldsymbol{r}_{ij} \boldsymbol{W}^{r,V}) \\ \textit{s.t.} \quad \alpha_{ij} &= \frac{(\boldsymbol{x}_i \boldsymbol{W}^Q)(\boldsymbol{x}_j \boldsymbol{W}^K + \boldsymbol{r}_{ij} \boldsymbol{W}^{r,K})^{\mathrm{T}}}{\sqrt{d}} \\ &\quad + \frac{(\boldsymbol{a}_i \boldsymbol{W}^{a,Q})(\boldsymbol{a}_j \boldsymbol{W}^{a,K})^{\mathrm{T}}}{\sqrt{d}} \end{aligned} \tag{6}$$

where $\boldsymbol{W}^{r,K}, \boldsymbol{W}^{r,V} \in \mathbb{R}^{d \times d}$ are the key and value projection matrices of the relative path encoding and $\boldsymbol{W}^{a,Q}, \boldsymbol{W}^{a,K}$ denote the query and key projection matrices of the absolute path encoding.

Approach

We introduce MetaTPTrans which consists of two components: a Meta Learner and a Base Learner. Specifically, the Meta Learner takes the language type as input and generates language-specific parameters for the Base Learner. We apply the TPTrans as the architecture of the Base Learner, which takes the source code as input, and uses the language-specific parameters generated from the Meta Learner to calculate the representation of the input source code snippet. The overview of our approach is shown in Figure 2.

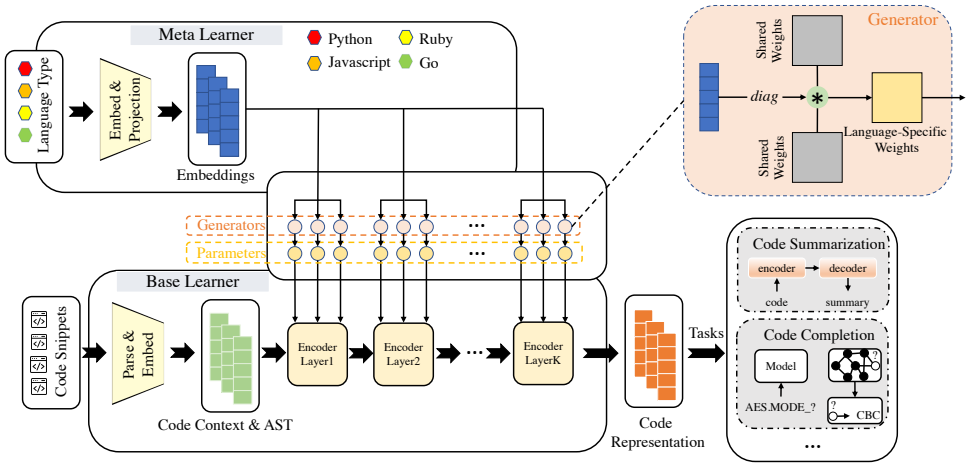


Figure 2: Architecture of MetaTPTrans.

Meta Learner

The Meta Learner generates parameters for the Base Learner according to the language type. Specifically, for a given source code snippet $\boldsymbol{X}_i$ and its corresponding language type t_i, the *Language Embedding Layer* $\mathcal{T}$ embeds the language type t_i into an embedding $\boldsymbol{T}_i \in \mathbb{R}^{d_T}$. Then, a projection layer scales the dimension of $\boldsymbol{T}_i$. The overall process can be presented as:

$$\begin{aligned} \boldsymbol{T}_i &= \mathcal{T}(t_i) \\ \boldsymbol{P}_i &= Projection(\boldsymbol{T}_i) \end{aligned} \tag{7}$$

where $Projection(\cdot)$ and $\boldsymbol{P}_i \in \mathbb{R}^{d_P}$ denote the projection layer and the projected language type embedding, respectively. After producing the projected language type embedding $\boldsymbol{P}_i$, we now present the parameter generation scheme for the Base Learner. For a weight matrix $\boldsymbol{W}^\lambda \in \mathbb{R}^{d \times d}$ in the parameters of Base Learner, we conduct it via a generator $\mathcal{G}_\lambda$, which can be denoted as:

$$\boldsymbol{W}^\lambda = \mathcal{G}_\lambda(\boldsymbol{P}_i) \tag{8}$$

As noted in (Bertinetto et al. 2016; Wang et al. 2019), using a linear projection layer to scale a d_P-dimension vector to a matrix of dimension $d \times d$ exhibits a large memory footprint, especially that there are many such weight matrices in the parameters of the Base Learner. To reduce the computational and memory cost, we adopt the factorized scheme for the weight generation procedure by factorizing the representation of the weights, which is analogous to the *Singular Value Decomposition* (Bertinetto et al. 2016). In this way, the original vector can be projected into the target matrix dimension space with fewer parameters in the generator. More specifically, we first apply the diagonal operation to transform the vector $\boldsymbol{P}_i$ to a diagonal matrix, then two projection matrices $\boldsymbol{M}_\lambda \in \mathbb{R}^{d_P \times d}$ and $\boldsymbol{M}'_\lambda \in \mathbb{R}^{d \times d_P}$ are defined to project the diagonal projected language type embedding matrix into the space of the target weight matrix. This operation can be formulated as:

$$\boldsymbol{W}^\lambda = \mathcal{G}_\lambda(\boldsymbol{P}_i) = \boldsymbol{M}'_\lambda diag(\boldsymbol{P}_i)\boldsymbol{M}_\lambda \tag{9}$$

where $diag(\cdot)$ is the non-parametric diagonal operation, and $\boldsymbol{M}_\lambda$ and $\boldsymbol{M}'_\lambda$ are the learnable parameters in generator $\mathcal{G}_\lambda$.

Base Learner

The Base Learner is the module that actually learns the representation of code snippets, the parameters of which are generated from the Meta Learner. In our approach, we apply the TPTrans model as the architecture of the Base Learner. Recall from Eq. (6), the learnable parameters of TPTrans consist of the projection matrices of tokens $(\boldsymbol{W}^Q, \boldsymbol{W}^K, \boldsymbol{W}^V)$ and the projection matrices of path encodings $(\boldsymbol{W}^{r,K}, \boldsymbol{W}^{r,V}, \boldsymbol{W}^{a,Q}, \boldsymbol{W}^{a,K})$.

The Meta Learner generates the parameters of the Base Learner. First, we consider that the most obvious language-specific information is the language-specific underlying syntax rule in the context of the code snippet. Such contextual information is extracted from the input token sequences. Therefore, we first generate the projection matrices for token sequences. This procedure can be formulated as:

$$\begin{aligned} \boldsymbol{W}^\lambda_{t_i} &= \mathcal{G}_\lambda(\boldsymbol{P}_i) = \boldsymbol{M}'_\lambda diag(\boldsymbol{P}_i)\boldsymbol{M}_\lambda \\ &s.t. \quad \lambda \in \{Q, K, V\} \end{aligned} \tag{10}$$

where t_i denotes the corresponding language type of code snippet $\boldsymbol{X}_i$, and $\boldsymbol{W}^Q_{t_i}, \boldsymbol{W}^K_{t_i}, \boldsymbol{W}^V_{t_i}$ are the learnable weights of $\boldsymbol{W}^Q, \boldsymbol{W}^K, \boldsymbol{W}^V$ associated with language type t_i, and $\boldsymbol{P}_i = Projection(\mathcal{T}(t_i))$ denotes the projected language type embedding of t_i. After that, the generated weights matrices are assigned to the related parameters by replacing the unified related weights matrices in Eq. (6):

$$\begin{aligned} \boldsymbol{z}_i &= \sum_{j=1}^{n} \frac{\exp(\alpha_{ij})}{\sum_{k=1}^{n} \exp(\alpha_{ik})} (\boldsymbol{x}_j \mathcal{G}_V(\boldsymbol{P}_i) + \boldsymbol{r}_{ij} \boldsymbol{W}^{r,V}) \\ s.t. \quad \alpha_{ij} &= \frac{(\boldsymbol{x}_i \mathcal{G}_Q(\boldsymbol{P}_i))(\boldsymbol{x}_j \mathcal{G}_K(\boldsymbol{P}_i) + \boldsymbol{r}_{ij} \boldsymbol{W}^{r,K})^{\mathrm{T}}}{\sqrt{d}} \\ &+ \frac{(\boldsymbol{a}_i \boldsymbol{W}^{a,Q})(\boldsymbol{a}_j \boldsymbol{W}^{a,K})^{\mathrm{T}}}{\sqrt{d}} \end{aligned} \tag{11}$$

Further, the weights matrices for path encoding projection can also be generated by the Meta Learner for different language types. This process aims to integrate the language-agnostic

structural information (path encodings) into the contextual information dynamically according to the associated language type, which we believe is meaningful to investigate since the combination of these two kinds of information may also be influenced by the underlying language type. Similar to Eq. (10), this procedure can be expressed as:

$$\begin{aligned} \boldsymbol{W}_{t_i}^{\lambda} &= \mathcal{G}_{\lambda}(\boldsymbol{P}_i) = \boldsymbol{M}'_{\lambda} diag(\boldsymbol{P}_i) \boldsymbol{M}_{\lambda} \\ s.t.\ \ \lambda &\in \{\{r,K\},\{r,V\},\{a,Q\},\{a,K\}\} \end{aligned} \tag{12}$$

where $\boldsymbol{W}_{t_i}^{r,K}, \boldsymbol{W}_{t_i}^{r,V}, \boldsymbol{W}_{t_i}^{a,Q}, \boldsymbol{W}_{t_i}^{a,K}$ are the generated weights of $\boldsymbol{W}^{r,K}, \boldsymbol{W}^{r,V}, \boldsymbol{W}^{a,Q}, \boldsymbol{W}^{a,K}$ associated with language type t_i. After that, the generated weights matrices in Eq. (12) can be integrated into Eq. (6) by replacing the related weights matrices:

$$\begin{aligned} \boldsymbol{z}_i &= \sum_{j=1}^{n} \frac{\exp(\alpha_{ij})}{\sum_{k=1}^{n} \exp(\alpha_{ik})} (\boldsymbol{x}_j \boldsymbol{W}^V + \boldsymbol{r}_{ij} \mathcal{G}_{r,V}(\boldsymbol{P}_i)) \\ s.t. \quad \alpha_{ij} &= \frac{(\boldsymbol{x}_i \boldsymbol{W}^Q)(\boldsymbol{x}_j \boldsymbol{W}^K + \boldsymbol{r}_{ij} \mathcal{G}_{r,K}(\boldsymbol{P}_i))^{\mathrm{T}}}{\sqrt{d}} \\ &\quad + \frac{(\boldsymbol{a}_i \mathcal{G}_{a,Q}(\boldsymbol{P}_i))(\boldsymbol{a}_j \mathcal{G}_{a,K}(\boldsymbol{P}_i))^{\mathrm{T}}}{\sqrt{d}} \end{aligned} \tag{13}$$

Finally, we combine the above two kinds of weights generation schemes in which both context token projection and path encoding projection are generated by the Meta Learner:

$$\begin{aligned} \boldsymbol{z}_i &= \sum_{j=1}^{n} \frac{\exp(\alpha_{ij})}{\sum_{k=1}^{n} \exp(\alpha_{ik})} (\boldsymbol{x}_j \mathcal{G}_V(\boldsymbol{P}_i) + \boldsymbol{r}_{ij} \mathcal{G}_{r,V}(\boldsymbol{P}_i)) \\ s.t. \quad \alpha_{ij} &= \frac{(\boldsymbol{x}_i \mathcal{G}_Q(\boldsymbol{P}_i))(\boldsymbol{x}_j \mathcal{G}_K(\boldsymbol{P}_i) + \boldsymbol{r}_{ij} \mathcal{G}_{r,K}(\boldsymbol{P}_i))^{\mathrm{T}}}{\sqrt{d}} \\ &\quad + \frac{(\boldsymbol{a}_i \mathcal{G}_{a,Q}(\boldsymbol{P}_i))(\boldsymbol{a}_j \mathcal{G}_{a,K}(\boldsymbol{P}_i))^{\mathrm{T}}}{\sqrt{d}} \end{aligned} \tag{14}$$

In the above, we generate weights matrices from three perspectives: (i) For context token projection (Eq. (11)), (ii) For path encoding projection (Eq. (13)), and (iii) For both context token projection and path encoding projection (Eq. (14)). We name those three schemes of weights matrices generation: *MetaTPTrans-*α, *MetaTPTrans-*β, and *MetaTPTrans-*γ, respectively.

Experimental Setup and Results

In this section, we present our experimental setup and results. We evaluate MetaTPTrans on two common and challenging tasks: code summarization and code completion. For the number of parameters and training time cost compared with the baseline (TPTrans), we present them in the Appendix.

Code Summarization Code summarization aims at describing the functionality of a piece of code in natural language and it demonstrates the capability of the models in capturing the semantics of source code (Alon et al. 2019b; Zügner et al. 2021; Peng et al. 2021). Similar to previous work (Zügner et al. 2021; Peng et al. 2021), we consider a complete method body as the source code input and the method name as the target prediction (i.e. the NL summary) while predicting the method name as a sequence of subtokens. Following Zügner et al. (2021); Peng et al. (2021), we evaluate the performance of our approach and baselines on the code summarization task using the metrics of precision, recall, and F1 scores over the target sequence.

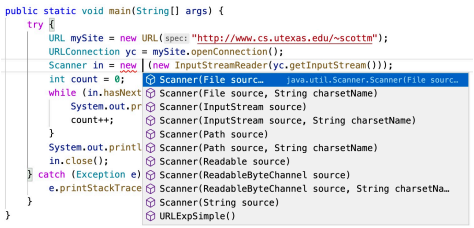


Figure 3: An example of code completion

Code Completion Code completion is another challenging downstream task in source code modeling. As the settings in (Liu et al. 2020a,b), the model aims to predict a missing token by taking the incomplete code snippet as input. An example of code completion in realistic scenario is shown in Figure 3, in which the IDE predicts a list of possible tokens for a missing token in an incomplete code snippet. To generate data for code completion task, we randomly replace a token with a special token `<MASK>` in a given code snippet. And then, the new code snippet with the special token `<MASK>` is used to predict the replaced missing token. We report Top-1 and Top-5 prediction accuracy as the evaluation metrics for the code completion task.

Dataset We conduct our experiments on the CodeSearchNet (Husain et al. 2019) dataset. Following Zügner et al. (2021); Peng et al. (2021), we consider four programming languages in the dataset: Python, Ruby, JavaScript, and Go. Please see the Appendix for the details and the pre-processing of the dataset.

Baselines For code summarization, we compare MetaTPTrans against code2seq (Alon et al. 2019a), GREAT (Hellendoorn et al. 2020), CodeTransformer (Zügner et al. 2021) and TPTrans (Peng et al. 2021). For code completion, we compare MetaTPTrans against Transformer (Vaswani et al. 2017), CodeTransformer (Zhou et al. 2019) and TPTrans (Peng et al. 2021). Please see the Appendix for more details.

Implementation Details For both tasks, following Peng et al. (2021), we set the embedding sizes of the word, path node, and hidden size of the Transformer to 512, 64, and 1024, respectively. A linear layer projects the word embedding into the size of the hidden layer of the Transformer. We use one bidirectional-GRU (Cho et al. 2014) layer of size 64 to encode the paths, and concatenate the final states of both directions as output. We use the Adam (Kingma and Ba 2015) optimizer with a learning rate of $1e^{-4}$. We train our models for 10 and 40 epochs for the code summarization and code completion tasks, respectively on 4 Tesla V100 GPUs with batch size of 128 and dropout of 0.2. For the Base Learner in the code summarization task, we use the same hyperparameters setting of TPTrans (Peng et al. 2021) for a fair comparison. Specifically, we set the number of encoder and

Table 1: Precision, recall, and F1 for the code summarization task. The bold part denotes the overall best results, and the underlined part denotes the best results of baselines.

Model	Python			Ruby			JavaScript			Go		
	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1
code2seq (Single-language)	35.79	24.85	29.34	23.23	10.31	14.28	30.18	19.88	23.97	52.30	43.43	47.45
GREAT (Single-language)	35.07	31.59	33.24	24.64	22.23	23.38	31.20	26.84	28.86	50.01	46.51	48.20
CodeTransformer (Single.)	36.40	33.66	34.97	31.42	24.46	27.50	35.06	29.61	32.11	55.10	48.05	51.34
TPTrans (Single-language)	38.39	34.70	36.45	33.07	28.34	30.52	33.68	28.95	31.14	55.67	51.31	53.39
code2seq (Multilingual)	34.49	25.49	29.32	23.97	17.06	19.93	31.62	22.16	26.06	52.70	44.36	48.17
GREAT (Multilingual)	36.75	31.54	33.94	30.05	24.33	26.89	33.58	27.78	30.41	52.65	48.30	50.38
CodeTransformer (Multi.)	38.89	33.82	36.18	33.93	28.94	31.24	36.95	29.98	33.10	56.00	50.44	53.07
TPTrans (Multilingual)	39.71	34.66	37.01	39.51	32.31	35.55	34.92	30.01	32.33	56.48	52.02	54.16
MetaTPTrans-α	40.22	**36.22**	**38.12**	**40.62**	**34.01**	**37.02**	37.87	31.92	34.64	58.12	53.82	55.89
MetaTPTrans-β	39.97	36.12	37.94	40.44	33.69	36.76	**38.87**	**32.66**	**35.50**	**58.86**	**54.24**	**56.45**
MetaTPTrans-γ	**40.47**	35.19	37.65	40.58	32.04	35.81	37.90	30.11	33.56	58.20	53.38	55.68

decoder layers to 3 and 8, the number of attention heads to 3, and the dimension of the feed-forward layer to 4096. In the Meta Learner, the dimension of the language type embedding (d_T) and its projection (d_P) are set to 1024 and 2048, respectively. Following Zügner et al. (2021); Peng et al. (2021), we add the pointer network (Vinyals, Fortunato, and Jaitly 2015) to the decoder. For the code completion task, we set the number of encoder layers, number of heads, and the dimension of feed-forward layers to 5, 8 and 2048 respectively for all the baselines and our approaches. As we mentioned above, we follow the code completion settings in (Liu et al. 2020a,b) that predict a missing token amid a incomplete code snippet, which is not a sequence-to-sequence task. Thus, we apply a fully connected layer after the encoder rather than using a decoder to generate the predicted token. In the Meta Learner, we set both the dimension of the language type embedding (d_T) and its projection (d_P) to 512.

Code Summarization

Table 1 shows the results of the code summarization task. The top part of the table shows the results of the baselines trained on single-language datasets. The middle and bottom parts of the table show the results of the baselines trained on multilingual dataset and our MetaTPTrans respectively. MetaTPTrans-α (Eq. 11), MetaTPTrans-β (Eq. 13) and MetaTPTrans-γ (Eq. 14) outperform all the baseline methods significantly. Specifically, compared with the state-of-the-art results on the Python, Ruby, JavaScript, and Go datasets, our approach improves the F1 score by 1.11, 1.47, 2.40 and 2.29 respectively. Overall, MetaTPTrans improves precision by 0.76–2.38, recall by 1.52–2.65, and F1 by 1.11–2.40 across the four programming languages. For the experiment results without pointer networks, please see Appendix for details.

Code Completion

Table 2 shows the results for the code completion task where the top and middle parts denote the testing results of the baselines trained on single-language datasets and multilingual dataset respectively. And bottom part of the table demonstrates the results of our MetaTPTrans. MetaTPTrans-α improves the Top-1 (Top-5) prediction accuracy over the best baseline, by 7.32 (10.18), 5.91 (11.95), 6.07 (11.71) and 7.02 (12.12) for Python, Ruby, JavaScript, and Go, respectively. Among the three variants of MetaTPTrans, the MetaTPTrans-α consistently achieves the best results over the four programming languages. As described before, the language-specific weights matrices generated by the Meta Learner in MetaTPTrans-α are only assigned to the parameters of the code context token projection in the Base Learner (Eq. 11) while MetaTPTrans-β and MetaTPTrans-γ assign the generated weights to AST path encodings. This means that for code completion, compared with generating language-specific weights for the projection of structural information, generating weights for context token projection is more conducive for the extraction of language-specific information, which leads to a significant performance improvement.

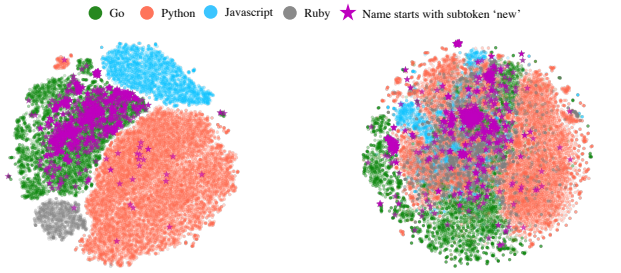


Figure 4: t-SNE visualization of the representation learned by MetaTPTrans-α (left) and TPTrans (right).

Visualization of Learned Representation

In figure 4, we show the t-SNE (Van der Maaten and Hinton 2008) visualization of the learned representations of all the code snippets from the validation set of the code summarization task. The left and right parts of Figure 4 show the code representation generated by the encoder of MetaTPTrans-α and the best baseline TPTrans respectively. We see that MetaTPTrans-α learns a distributed code representation that also respects the type of programming language of the code snippet as data points from the same language group together.

Table 2: Top-1 and Top-1 accuracy for the code completion task. Underlined, bold values denote the best results in the baselines and the overall best results respectively.

Model	Python		Ruby		JavaScript		Go	
	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5	Top-1	Top-5
Transformer (Single-language)	47.57	69.86	44.39	62.24	37.57	53.15	40.21	59.65
CodeTransformer (Single.)	62.45	76.73	51.63	69.96	47.56	68.88	47.71	61.35
TPTrans (Single-language)	63.71	77.99	64.42	72.50	64.67	73.42	57.15	67.81
Transformer (Multilingual)	47.02	78.82	47.16	77.32	38.77	70.84	42.01	72.95
CodeTransformer (Multi.)	68.19	82.98	67.67	83.47	59.32	80.07	57.12	77.67
TPTrans (Multilingual)	69.81	84.10	72.14	82.27	67.45	81.17	60.45	79.03
MetaTPTrans-α	**77.13**	**94.28**	**78.05**	**95.42**	**73.52**	**92.88**	**67.47**	**91.15**
MetaTPTrans-β	71.75	86.26	73.82	86.85	72.71	86.90	66.74	85.21
MetaTPTrans-γ	67.12	90.72	71.89	93.61	69.55	90.97	61.60	88.99

This demonstrates that our model learns languages-specific features much better than the baseline model, where code snippets from the same language do not necessarily group together. Moreover, as an example, we mark all the code snippets whose names start with the subtoken `new` by the symbol ★. We see that MetaTPTrans-α achieves a much better grouping of those code snippets compared to the baseline model emphasizing our model ability to learn a better semantic representation of source code, while also respecting the language-specific features.

Related Work

Learning Representation of Source Code Source code representation learning has seen many developments. Early works mainly focus on learning language models from raw token sequences (Wang et al. 2016; Dam, Tran, and Pham 2016; Allamanis, Peng, and Sutton 2016; Iyer et al. 2016). More recent works explore the effectiveness of leveraging structural information to model source code. Mou et al. (2016) apply the convolutional operation on ASTs to extract structure features to represent source code. Alon et al. (2019b,a) extract paths from ASTs and use RNNs to encode them to represent the source code. Allamanis, Brockschmidt, and Khademi (2018); Fernandes, Allamanis, and Brockschmidt (2019); Zhou et al. (2019) use Graph Neural Networks to capture the structural information from carefully designed code graphs. Hellendoorn et al. (2020); Zügner et al. (2021); Peng et al. (2021) use Transformer-based models to represent source code by capturing both context and structural information, in which the structural information is integrated into the self-attention module by replacing the position embedding with encoding from AST. Specifically, Hellendoorn et al. (2020) bias the self-attention process with different types of correlations between nodes in code graph. Zügner et al. (2021) use several pair-wise distances on ASTs to represent the pair-wise relationships between tokens in code context sequence and find that multilingual training improves the performance of language models compared to single-language models. Peng et al. (2021) encode AST paths and integrate them into self-attention to learn both context and structural information. Compared to these works, ours is the first to use meta learning to learn multilingual source code models that are capable of learning language-specific in addition to language-agnostic information and improves on several of the aforementioned models yielding state-of-the-art results.

Meta Learning for Parameters Generation Meta learning is a novel learning paradigm with the concept of learning to learn. There are several types of meta learning such as learning to initialize (Finn, Abbeel, and Levine 2017), learning the optimizer (Andrychowicz et al. 2016), and learning hyperparameters (Li et al. 2021). The most related meta learning method with ours is learning to generate model's parameters. Bertinetto et al. (2016) propose a method called learnet to learn to generate the parameters of the pupil network. Ha, Dai, and Le (2017) propose Hypernetworks to generate parameters for large models through layer-wise weight sharing scheme. Chen et al. (2018) propose a meta learning-based multi-task learning framework in which a meta network generates task-specific weights for different tasks to extract task-specific semantic features. Wang et al. (2019) use a task-aware meta learner to generate parameters for classification models for different tasks in few-shot learning. Pan et al. (2019, 2022) apply meta learning to generate parameters for models in spatial-temporal data mining to capture spatial and temporal dynamics in urban traffic. Our approach is a form of meta learning to generate the model's parameters according to the programming language type of the input code snippet.

Conclusion

We propose MetaTPTrans, a meta learning approach for multilingual code representation learning. Instead of keeping the feature extractor with a fixed set of parameters across different languages, we adopt meta learning to generate different sets of parameters for the feature extractor according to the language type of the input code snippet. This enables MetaTPTrans to not only extract language-agnostic information, but to also capture language-specific features of source code. Experimental results show that our approach outperforms the state-of-the-art baselines significantly. Our work provides a novel direction for multilingual code representation learning.

Acknowledgments

This work is supported by the NATURAL project, which has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant No. 949014).

References

Allamanis, M.; Brockschmidt, M.; and Khademi, M. 2018. Learning to Represent Programs with Graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*.

Allamanis, M.; Peng, H.; and Sutton, C. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48, 2091–2100.

Alon, U.; Brody, S.; Levy, O.; and Yahav, E. 2019a. code2seq: Generating Sequences from Structured Representations of Code. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*.

Alon, U.; Zilberstein, M.; Levy, O.; and Yahav, E. 2019b. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL): 40:1–40:29.

Andrychowicz, M.; Denil, M.; Colmenarejo, S. G.; Hoffman, M. W.; Pfau, D.; Schaul, T.; and de Freitas, N. 2016. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, 3981–3989.

Bertinetto, L.; Henriques, J. F.; Valmadre, J.; Torr, P. H. S.; and Vedaldi, A. 2016. Learning feed-forward one-shot learners. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, 523–531.

Chen, J.; Qiu, X.; Liu, P.; and Huang, X. 2018. Meta Multi-Task Learning for Sequence Modeling. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, 5070–5077. AAAI Press.

Cho, K.; Van Merriënboer, B.; Bahdanau, D.; and Bengio, Y. 2014. On the properties of neural machine translation: Encoder-decoder approaches. arXiv:1409.1259.

Dam, H. K.; Tran, T.; and Pham, T. 2016. A deep language model for software code. arXiv:1608.02715.

Fernandes, P.; Allamanis, M.; and Brockschmidt, M. 2019. Structured Neural Summarization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*.

Finn, C.; Abbeel, P.; and Levine, S. 2017. Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, 1126–1135. PMLR.

Gasteiger, J.; Groß, J.; and Günnemann, S. 2020. Directional Message Passing for Molecular Graphs. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*.

Ha, D.; Dai, A. M.; and Le, Q. V. 2017. HyperNetworks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*.

Hamilton, W. L.; Ying, Z.; and Leskovec, J. 2017. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, 1024–1034.

Hellendoorn, V. J.; Sutton, C.; Singh, R.; Maniatis, P.; and Bieber, D. 2020. Global Relational Models of Source Code. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*.

Hindle, A.; Barr, E. T.; Su, Z.; Gabel, M.; and Devanbu, P. T. 2012. On the naturalness of software. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, 837–847. IEEE Computer Society.

Hochreiter, S.; and Schmidhuber, J. 1997. Long Short-Term Memory. *Neural Computation*, 9(8): 1735–1780.

Husain, H.; Wu, H.-H.; Gazit, T.; Allamanis, M.; and Brockschmidt, M. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. arXiv:1909.09436.

Iyer, S.; Konstas, I.; Cheung, A.; and Zettlemoyer, L. 2016. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics.

Ke, G.; He, D.; and Liu, T. 2021. Rethinking Positional Encoding in Language Pre-training. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*.

Kim, S.; Zhao, J.; Tian, Y.; and Chandra, S. 2021. Code Prediction by Feeding Trees to Transformers. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, 150–162. IEEE.

Kingma, D. P.; and Ba, J. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Kipf, T. N.; and Welling, M. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017*.

Li, S.; Gong, K.; Liu, C. H.; Wang, Y.; Qiao, F.; and Cheng, X. 2021. MetaSAug: Meta Semantic Augmentation for Long-Tailed Visual Recognition. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2021, virtual, June 19-25, 2021*, 5212–5221.

Liu, F.; Li, G.; Wei, B.; Xia, X.; Fu, Z.; and Jin, Z. 2020a. A Self-Attentional Neural Architecture for Code Completion with Multi-Task Learning. In *ICPC '20: 28th International*

Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020, 37–47. ACM.

Liu, F.; Li, G.; Zhao, Y.; and Jin, Z. 2020b. Multi-task Learning based Pre-trained Language Model for Code Completion. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, 473–485. IEEE.

Mou, L.; Li, G.; Zhang, L.; Wang, T.; and Jin, Z. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, 1287–1293. AAAI Press.

Pan, Z.; Liang, Y.; Wang, W.; Yu, Y.; Zheng, Y.; and Zhang, J. 2019. Urban Traffic Prediction from Spatio-Temporal Data Using Deep Meta Learning. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, 1720–1730. ACM.

Pan, Z.; Zhang, W.; Liang, Y.; Zhang, W.; Yu, Y.; Zhang, J.; and Zheng, Y. 2022. Spatio-Temporal Meta Learning for Urban Traffic Prediction. *IEEE Transactions on Knowledge and Data Engineering*, 34(3): 1462–1476.

Peng, H.; Li, G.; Wang, W.; Zhao, Y.; and Jin, Z. 2021. Integrating Tree Path in Transformer for Code Representation. *Advances in Neural Information Processing Systems*, 34.

Pradel, M.; and Sen, K. 2018. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA): 1–25.

Shaw, P.; Uszkoreit, J.; and Vaswani, A. 2018. Self-Attention with Relative Position Representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 2 (Short Papers)*, 464–468.

Tian, H.; Li, Y.; Pian, W.; Kaboré, A. K.; Liu, K.; Habib, A.; Klein, J.; and Bissyandé, T. F. 2022a. Predicting Patch Correctness Based on the Similarity of Failing Test Cases. *ACM Trans. Softw. Eng. Methodol.*, 31(4): 77:1–77:30.

Tian, H.; Liu, K.; Li, Y.; Kaboré, A. K.; Koyuncu, A.; Habib, A.; Li, L.; Wen, J.; Klein, J.; and Bissyandé, T. F. 2022b. The Best of Both Worlds: Combining Learned Embeddings with Engineered Features for Accurate Prediction of Correct Patches. arXiv:2203.08912.

Van der Maaten, L.; and Hinton, G. 2008. Visualizing data using t-SNE. *Journal of machine learning research*, 9(11).

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L.; and Polosukhin, I. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems, NeurIPS 2017, December 4-9, 2017, Long Beach, CA, USA*, 5998–6008.

Velickovic, P.; Cucurull, G.; Casanova, A.; Romero, A.; Liò, P.; and Bengio, Y. 2018. Graph Attention Networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*.

Vinyals, O.; Fortunato, M.; and Jaitly, N. 2015. Pointer Networks. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, 2692–2700.

Wang, S.; Chollak, D.; Movshovitz-Attias, D.; and Tan, L. 2016. Bugram: bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, 708–719. ACM.

Wang, X.; Yu, F.; Wang, R.; Darrell, T.; and Gonzalez, J. E. 2019. TAFE-Net: Task-Aware Feature Embeddings for Low Shot Learning. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, 1831–1840. IEEE / Computer Vision Foundation.

Yang, Z.; Dai, Z.; Yang, Y.; Carbonell, J. G.; Salakhutdinov, R.; and Le, Q. V. 2019. XLNet: Generalized Autoregressive Pretraining for Language Understanding. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, 5754–5764.

Zhou, Y.; Liu, S.; Siow, J. K.; Du, X.; and Liu, Y. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, 10197–10207.

Zügner, D.; Kirschstein, T.; Catasta, M.; Leskovec, J.; and Günnemann, S. 2021. Language-Agnostic Representation Learning of Source Code from Structure and Context. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*.

Appendix

Summary of the CodeSearchNet Dataset

We show the summary of the CodeSearchNet (Husain et al. 2019) dataset used in our experiments in Table 3, which contains four programming languages: Python, Ruby, JavaScript, and Go. The dataset has been deduplicated by the creators to avoid data leakage from training set.

Table 3: Dataset statistics

Language	Samples per partition		
	Train	Valid	Test
Python	412,178	23,107	22,176
Ruby	48,791	2,209	2,279
JavaScript	123,889	8,253	6,483
Go	317,832	14,242	14,291
Total	902,690	47,811	45,229

Table 4: Dataset Statistics

Language	Samples per partition		
	Train	Valid	Test
Python	69,527	11,112	10,565
Ruby	11,524	1,075	1,048
JavaScript	26,626	4,353	3,504
Go	36,948	3,434	4,563
Total	144,175	19,974	19,680

Preprocessing

We parse code snippets using Tree-Sitter[1], an open-source parser that can parse multiple programming languages into AST. Besides, we follow the token splitting rule in (Alon et al. 2019a; Zügner et al. 2021; Peng et al. 2021) that splits each code token into sub-tokens regarding to the code name convention. For instance, the code token `sendDirectOperateCommandSet` is split into `[send, direct, operate, command, set]`. For the vocabulary of sub-tokens, we limit it with the least occurrence number of 100 in the training set, and we also restrict the max length of the token sequence to 512 after removing all punctuation. For code summarization task, we follow Zügner et al. (2021); Peng et al. (2021) and remove all the anonymous functions in the javaScript dataset which can not be used for this task. For the code completion task, we randomly select a subset from the whole CSN dataset with 144,175, 19,974 and 19,680 code snippets for training, validation and testing respectively. Please see the next section for details of the dataset used in the code completion task. For the paths, we set the max length to 32, and we make padding for the path shorter than max length while sampling nodes with equal intervals to maintain max length following (Peng et al. 2021).

[1]https://github.com/tree-sitter/

Summary of the CSN Subset in Code Completion

For the code completion task, we randomly select a subset from the whole CSN dataset with 144,175, 19,974 and 19,680 code snippets for training, validation and testing respectively. We show the summary of it in Table 4.

Baselines

In our experiments, we set the following methods as baselines:

- **Transformer:** Transformer (Vaswani et al. 2017) is a language model based on multi-head attention, which can only model contextual information. Transformer is the basic backbone of GREAT, CodeTransformer and TPTrans.
- **code2seq:** Code2seq (Alon et al. 2019a) is an LSTM-based method that utilizes the pairwise path information in AST to model code snippets.
- **GREAT:** GREAT (Hellendoorn et al. 2020) is a Transformer-based approach that utilizes manually designed edges, such as dataflow, 'computed from', 'next lexical use' edges.
- **CodeTransformer:** CodeTransformer (Zügner et al. 2021) is a Transformer-based model that combines multiple pairwise distance relation between nodes in AST to integrate the structure information into the context sequence of code
- **TPTrans:** TPTrans (Peng et al. 2021) is a recent state-of-the-art approach for code representation learning based on Transformer and tree paths in AST, which integrate the encoding of tree paths into self-attention module by replacing the relative and absolute position embedding.

Table 5: Number of Parameters of TPTrans and MetaTPTrans

Model	Task	
	Code Summarization	Code Completion
TPTrans	113.47M	101.74M
MetaTPTrans-α	178.48M	109.87M
MetaTPTrans-β	118.65M	102.73M
MetaTPTrans-γ	181.56M	110.59M

Table 6: Average One Epoch's Training Time Cost of TPTrans and MetaTPTrans

Model	Task	
	Code Summarization	Code Completion
TPTrans	4h15min	32min
MetaTPTrans-α	6h15min	35min
MetaTPTrans-β	4h28min	32min
MetaTPTrans-γ	6h44min	35min

Table 7: Experimental results of code summarization task w/o pointer network. The bold part denotes the best results, the underlined part denotes the best results of baselines.

Model	Python			Ruby			JavaScript			Go		
	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1
CodeTransformer (Multi.)	<u>38.91</u>	33.12	35.78	34.52	27.31	30.50	<u>37.21</u>	29.75	<u>33.07</u>	56.07	50.76	53.28
TPTrans (Multilingual)	38.78	<u>34.72</u>	<u>36.64</u>	<u>38.05</u>	<u>32.35</u>	<u>34.97</u>	36.35	<u>30.06</u>	32.90	<u>56.49</u>	<u>51.99</u>	<u>54.15</u>
MetaTPTrans-α	**39.26**	36.57	**37.87**	**39.22**	34.55	36.74	37.29	32.50	34.73	**57.14**	54.48	55.78
MetaTPTrans-β	38.34	**37.32**	37.82	38.87	**36.07**	**37.42**	37.35	**34.06**	**35.63**	56.56	**55.14**	**55.84**
MetaTPTrans-γ	38.50	36.96	37.71	38.38	33.99	36.05	**37.72**	32.62	34.98	56.49	54.30	55.38

Number of Parameters and Training time Cost

We show the number of parameters of our approaches and TPTrans in Table 5. In code summarization task, compared with TPTrans, our MetaTPTrans-α, MetaTPTrans-β and MetaTPTrans-γ have 57.29%, 4.57% and 60.01% more parameters respectively. For code completion task, MetaTPTrans-α, MetaTPTrans-β and MetaTPTrans-γ have 7.99%, 0.97% and 8.70% more parameters than TPTrans respectively. In Table 6, we show the average one epoch's training time cost of TPTrans and our approaches.

Code Summarization Results w/o Pointer Network

In our experiments on code summarization task, we apply a pointer network (Vinyals, Fortunato, and Jaitly 2015) in the decoder following (Zügner et al. 2021; Peng et al. 2021). Here, we conduct an ablation study in which we remove the pointer network and the experiment results are shown in Table 7. We can see that, our approaches still have better performance compared with the baselines.