Automated Software Engineering (2023) 30:11
https://doi.org/10.1007/s10515-023-00376-y

AdaComplete: improve DL-based code completion method's domain adaptability

Zejun Wang[1,2] · **Fang Liu**[3] · **Yiyang Hao**[4] · **Zhi Jin**[1,2]

Received: 31 August 2022 / Accepted: 17 January 2023 / Published online: 6 March 2023

Abstract
Code completion is an important feature in integrated development environments that can accelerate the coding process. With the development of deep learning technologies and easy-to-acquire open-source codebases, many Deep Learning based code completion models (DL models) are proposed. These models are trained using the generic source code datasets, resulting in poor domain adaptability. That is, these models suffer from performance loss when helping programmers code in a specific domain, e.g., helping to decide which domain-specific API to call. To solve the problem, we propose *AdaComplete*, a simple and effective framework that utilizes a local code completion model to compensate DL models' domain adaptability. The local code completion model is trained using the source codes of the target domain. When used in code completion, given the context, AdaComplete can adaptively choose the recommendations from either the DL model or the local code completion model based on our hand-crafted features. Experimental results show that AdaComplete outperforms state-of-the-art DL-based code completion methods on specific domains and can improve the accuracy by 7% on average.

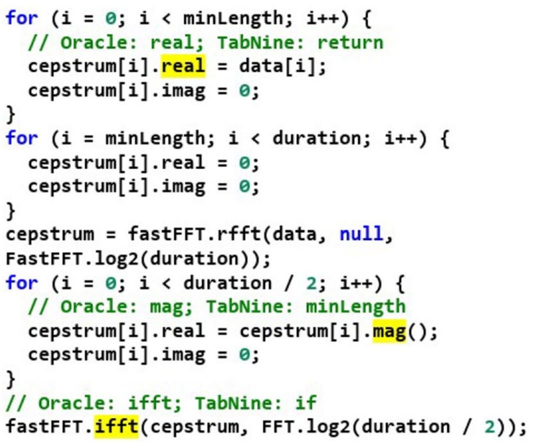
Keywords Code completion · Neural network · N-gram · Model integration · Domain adaptability

1 Introduction

Code completion is a commonly used feature in Integrated Development Environments (IDEs) that can predict the next token given existing code in the context. It is a critical tool in software development that benefits both quality and speed Bruch et al. (2009). Early code completion researchmainly used rule-based methods Robbes and Lanza (2010); Hou and Pletcher (2010) or statistical machine learning

✉ Zhi Jin
zhijin@pku.edu.cn

Extended author information available on the last page of the article



```
for (i = 0; i < minLength; i++) {
  // Oracle: real; TabNine: return
  cepstrum[i].real = data[i];
  cepstrum[i].imag = 0;
}
for (i = minLength; i < duration; i++) {
  cepstrum[i].real = 0;
  cepstrum[i].imag = 0;
}
cepstrum = fastFFT.rfft(data, null,
FastFFT.log2(duration));
for (i = 0; i < duration / 2; i++) {
  // Oracle: mag; TabNine: minLength
  cepstrum[i].real = cepstrum[i].mag();
  cepstrum[i].imag = 0;
}
// Oracle: ifft; TabNine: if
fastFFT.ifft(cepstrum, FFT.log2(duration / 2));
```

Fig. 1 Example of lacking domain adaptability

models Hindle et al. (2012); Tu et al. (2014); Nguyen et al. (2013) to perform code completion. With the development of deep learning and easy-to-acquire open-source code corpus, Deep Learning models (DL models) are generally used for code completion, for example, Recurrent Neural Network (RNN) and Transformer Allamanis et al. (2018); Bhoopchand et al. (2016); Li et al. (2018); Liu et al. (2020b). In general, a DL model for code completion is usually trained and tested on a large generic source code dataset Karampatsis et al. (2020); Liu et al. (2020b). For example, Liu et al. (2017) and Li et al. (2018) used 100k source code files for training and 50k for testing. Karampatsis et al. (2020) built a model for the C programming language based on 4,601 projects with 1.68B tokens. Liu et al. (2020b) trained and tested their method on a Java dataset with 800,983 source code files. Feng et al. (2020) used 2.1 M bimodal datapoints and 6.4 M unimodal codes across six programming languages. Researchers believe the more data they feed, the better the DL models would be.

However, current DL-based code completion models suffer from the performance loss when applied to a specific domain due to lack of 'Domain Adaptability'. They might make wrong predictions on domain-specific code, e.g., the invocation of domain-specific APIs. A domain in code completion refers to projects pointed at similar tasks or based on the same programming framework. For instance, the domain of 'Face Recognition of Java' refers to the Java projects that are used for face recognition. The domain of 'Spark' comprises the projects that utilize the Spark framework for large-scale data processing. The 'adaptability' means the ability of the DL models trained on generic source codes to adapt to a specific domain.

To better illustrate the problem that current DL models lack domain adaptability, we raise an example collected from GitHub in Fig. 1. The following Java code comes from the domain 'Fast Fourier Transformation (FFT).' Compared to code of other domains, this snippet involves many domain-specific APIs for FFT, for instance, ‘real,’ ‘imag,’ ‘mag,’ and ‘ifft’. Here we use a GPT-2 Radford et al. (2019) based code completion model TabNine Pro.[1] In Fig. 1, we highlight the tokens that TabNine fails to predict. TabNine’s wrong predictions are listed in the comments right above the highlighted tokens. TabNine fails to predict the domain-specific APIs: ‘real,’ ‘mag’ and ‘ifft.’On the contrary, TabNine gives predictions that are irrelevant to the domain: ‘return,’ ‘minLength’ and ‘if.’ This example shows that the DL-based code completion model, TabNine, fails to predict these domain-specific tokens and cannot adapt well to the domain ‘FFT.’

It isn’t easy to use the fine-tuning technique to improve the DL models’ performance on the specific domain, although using fine-tuning is intrinsic. Fine-tuning means tuning the DL model’s parameters to fit the specific domain. To successfully fine-tune a DL model, two conditions need to be fulfilled. The first one is the scale of the dataset used for fine-tuning, and it is very difficult to fine-tune the huge DL model on a dataset with limited scale Barone et al. (2017). The other one is the sufficient training time and computation resources. However, it is hard to meet both these requirements since the training data of a specific domain is always not big enough, and the computation resources for academic researchers are also limited.

To improve the DL-based code completion models’ performance on specific domains, we propose a novel general framework **AdaComplete**. It is light-weighted and aims at compensating any DL models’ domain adaptability by integrating a local domain-specific code completion method (local model in short for convenience). AdaComplete works as follows. Given the context, AdaComplete first produces the next token predictions from both the DL model and the local model. Then we balance these predictions with rules and a series of hand-crafted features and use the balanced result as the final prediction. AdaComplete is a general framework that can be applied to any DL model for code completion. In this paper, we instantiate AdaComplete with ‘Transformer-XL+N-Gram’. Transformer-XL network Dai et al. (2019) is adopted to instantiate the DL model following Liu et al. (2020a). As for the local model, we instantiate it with an N-gram model, which is trained on the domain-specific source code dataset.

To prove that AdaComplete can improve DL models’ performance on a specific domain, we build several domain-specific datasets which are collected from GitHub.[2] Then we compare AdaComplete with several state-of-the-art DL-based code completion models on our proposed domain-specific datasets. We use prediction accuracy to evaluate the models’ performance. Experimental results demonstrate that AdaComplete outperforms the state-of-the-art generic code completion methods by a large margin. Besides, we also conducted experiments to compare AdaComplete with Fine-tuning techniques. The results show that AdaComplete can outperform Fine-tuning techniques while taking less time.

The main contribution of this paper is summarized as follows:

[1] https://www.tabnine.com/.

[2] https://github.com/.

- We are the first to raise the issue that DL-based code completion models suffer from performance loss when directly used for completing domain-specific source code.
- We propose AdaComplete, a novel method to increase the overall code completion accuracy via the light-weighted integration of a local domain-specific model.
- We create several Java domain-specific datasets, which can be used to evaluate the domain adaptation capability of the code completion model. The datasets are publicly available and are shared through Figshare.[3]
- We evaluate AdaComplete on the domain-specific datasets and compare it with several state-of-the-art generic code completion methods and Fine-tuning techniques. The results show that AdaComplete can outperform all the baselines by a large margin.

The reminder of this paper is organized as follows: Sect. 2 is the empirical study to investigate our motivation. Section 3 presents the background of this paper. Section 4 is the detailed description of AdaComplete, including the overall framework and description of AdaComplete's modules. Section 5 shows our experimental results and analysis. Section 6 is our case study to prove AdaComplete's effect by cases. Section 7 discusses method generalization, threats to validity, and how to choose pointer models. Section 8 describes the related work of deep code completion. Section 9 gives the conclusion of this paper.

2 Empirical study

In this section, we conducted an empirical study to investigate our motivation that current DL-based code completion models lack domain adaptability, resulting in performance loss when applied to a specific domain.

2.1 Domain-specific projects and generic projects

A project is a domain-specific project when we emphasize that it is developed for a specific task or is built on one or multiple specific programming frameworks, as introduced in the previous section. Under this definition, each project is domain-specific.

On the contrary, we have the concept of generic project and generic domain. The generic domain is an alias of 'cross-domain' for better narration. When we refer to a generic domain, we do not consider the corresponding projects' tasks or frameworks. For example, in this paper we use a dataset on the generic domain, we mean that this dataset is cross-domain. The projects within are collected randomly, covering various tasks and frameworks. Furthermore, when we refer to a generic project,

[3] https://figshare.com/s/c0bd0430cd4134ab07f4.

Table 1 Accuracy comparison on the generic domain and specific domains

Generic models	Accuracy on datasets				
	Generic (%)	jMonkeyEngine (%)	jFaceRec (%)	CodeGeneration (%)	j2me (%)
Transformer-xl	**72.12**	62.16	71.33	64.51	61.23
BPE	**70.29**	59.12	69.89	57.56	43.23
CugLM	**84.06**	75.19	74.84	74.65	67.66

The bold numbers in the Table means 'the best'

we mean the project belongs to the generic dataset and is collected randomly without considering its tasks and programming framework.

For our experiments, we collect four domain-specific datasets. Besides, the experiments involve a generic domain dataset from previous works. We select the domains carefully to guarantee that there is no overlapping between the domain-specific datasets and the generic domain dataset by checking the source code files

2.2 Identify domain-specific projects

For our experiments, we investigate the following four domains:

- jMonkeyEngine: Projects built on a game engine named 'jMonkeyEngine'
- jFaceRec: Projects built for face recognition but built on Java
- CodeGeneration: Projects built for auto code generation
- j2me: Projects built based on the platform j2me

To create the datasets for the domains, we search GitHub with the domain's name as keywords. We sort the matched projects by their stars, download the high-starred repositories manually and check the contents manually. Then we filter the files and keep the source code files only.

We choose these domains for the following reasons: Their clear boundaries make it easier for us to validate whether our crawled projects belong to these domains. The projects of these domains are helpful in our real life; thus, the experiment results from these projects are meaningful. These projects are rare, contributing little to the cross-domain datasets for training the DL-based code completion model. As a result, these domains can reveal the significant differences between the cross-domain and domain-wise performance.

2.3 Experiments and results

To verify our assumption, we evaluate the following state-of-the-art deep code completion models on the four specific domains

- Transformer-XL Dai et al. (2019): a self-attentional neural network-based language model for code completion.
- BPE Karampatsis et al. (2020): a GRU-based neural language model for code completion, which leverage Byte Pair Encoding (BPE) Gage (1994) algorithm to address the OoV (Out-of-Vocabulary) problem.
- CugLM Liu et al. (2020b): A Transformer-based pre-trained language model for code completion, which achieves state-of-the-art results.

The completion accuracies of these models are shown in Table 1. The results on the generic-domain dataset are also listed in the 'generic' column, and the other data columns are the domain-specific datasets' results.

According to Table 1, the accuracies of all the three DL models drop by 10% on average when applied to the domain-specific datasets. These results strongly support our motivation that current DL code-completion models are short of domain adaptability.

3 Background

3.1 Domain adaptability & domain shift

The domain adaptability is the ability for DL models to deal with domain shift, i.e., the difference of data distribution across the domains. So we introduce the background knowledge of domain shift here for readers to better understand the problem.

We borrow the annotations from Wang et al. (2021) to clarify the concept of 'domain' and 'domain shift' with formulas. Suppose that we have an input space $\mathcal{X}$, an output space $\mathcal{Y}$ and a joint distribution $\mathcal{P}_{XY}$. With $\mathcal{P}_{XY}$, we draw N data points from $\mathcal{X}$ and $\mathcal{Y}$, then build a multi-set $\mathcal{S}$ $\{(\mathrm{x}_i, y_i)\}_i^N$, where $\mathrm{x}_i \in \mathcal{X}$ and $y_i \in \mathcal{Y}$. We name $\mathcal{S}$ a domain. For convenience, we call $\mathcal{P}_{XY}$ the 'drawing distribution.' Further, we define the domain shift of two domains as the difference between their drawing distributions.

Domain shift severely damages the performance of data-driven-based methods in many research fields. Different fields have different methods to alleviate the domain shift. For example, Saenko et al. (2010) discussed domain shift in the context in object recognition. They adapted models to a new domain by learning a transformation that minimizes the difference in feature space. Kuang et al. (2020) focused on distribution in regression tasks. They presented a Decorrelated Weighting Regression (DWR) algorithm. DWR jointly optimizes a variable decorrelation regularizer and a weighted regression model. Kamath et al. (2020) found that QA systems make mistakes in unknown domains. So the authors designed a calibrator to monitor the QA system. When the QA system was going to make mistakes in the unknown fields, the calibrator would abstain from giving answers.

In the field of code completion, we are the first to discuss the threat of domain shift. The generic source code dataset forms a 'generic domain'. And the domain shift between the generic domain and the specific domain causes the performance loss of the DL-based code completion models.

3.2 Transformer-XL

Transformer-XL Dai et al. (2019) is a powerful language model, which has been used as the base architecture or the strong baselines in many code completion research Liu et al. (2020a, b); Wang and Li (2021), which is also used in AdaComplete. Transformer-XL brings the recurrent mechanism to Transformer and a novel way for positional embedding. Compared with RNN and vanilla Transformer, it can capture longer dependency from the input sequences. Besides, The evaluation process is also much faster than vanilla Transformer. As for the performance, the Transformer-XL improves the state-of-art results of bpc/perplexity on many language modeling benchmarks, including enwiki8, text8, WikiText-103, One Billion Word, and Penn Treebank (without fine-tuning).

Unlike the vanilla Transformer, Transformer-XL involves the recurrent mechanism. It caches the previous hidden states. These hidden states join the computation of the hidden states as the key and value vectors directly or after some transformations. Suppose we have two consecutive segments of length L be s_τ and $s_{\tau+1}$ that are adjacent temporarily. Denoting the n-th layer hidden state produced for s_τ by $\mathbf{h}_\tau^n \in \mathbb{R}^{L*d}$, where d is the hidden dimension. Then, the n-th layer hidden state for segment $s_{\tau+1}$ is produced as follows:

$$\begin{aligned}
\widetilde{\mathbf{h}}_{\tau+1}^{n-1} &= [SG(\mathbf{h}_\tau^{n-1} \circ \mathbf{h}_{\tau+1}^{n-1})], \\
\mathbf{q}_{\tau+1}^n, \mathbf{k}_{\tau+1}^n, \mathbf{v}_{\tau+1}^n &= \mathbf{h}_{\tau+1}^{n-1}\mathbf{W}_q^\top, \widetilde{\mathbf{h}}_{\tau+1}^{n-1}\mathbf{W}_k^\top, \widetilde{\mathbf{h}}_{\tau+1}^{n-1}\mathbf{W}_v^\top, \\
\mathbf{h}_{\tau+1}^n &= Transformer - Layer(\mathbf{q}_{\tau+1}^n, \mathbf{k}_{\tau+1}^n, \mathbf{v}_{\tau+1}^n).
\end{aligned} \tag{1}$$

where $SG(\cdot)$ stands for 'Stop Gradient', $[\mathbf{h}_a \circ \mathbf{h}_b]$ stands for concatenating two matrix $\mathbf{h}_a$ and $\mathbf{h}_b$ along the temporal axis and $\mathbf{W}$. denotes model parameters. Then Transformer-XL uses the encoding matrix to represent the relative distances. Concretely speaking, the unnormalized attention score from position i to position j is:

$$\mathbf{A}_{\tau,i,j}^n = \mathbf{q}_{\tau,i}^{n\top}\mathbf{k}_{\tau,j}^n + \mathbf{q}_{\tau,i}^{n\top}\mathbf{W}_{k,R}^n\mathbf{R}_{\mathbf{i-j}} + u^\top\mathbf{k}_{\tau,j} + v^\top\mathbf{W}_{k,R}^n\mathbf{R}_{i-j} \tag{2}$$

where $\mathbf{W}_{k,R}^n, u, v$ are learnable parameters and $\mathbf{R}$ is the sinusoid encoding matrix. Learning the relative positions of the tokens is also important for code completion task, thus we also use the relative positional embedding in our model.

3.3 N-gram language model

The 'N-gram language model' is a model that estimates the appearing probabilities of all N-token tuples in the language by counts. And we call the N-token tuple ’the N-gram’. Given an N-gram model M_L for language L and an N-gram $s = w_1, w_2, \dots w_N$, we can get the appearing probability of s: $p_L(s) = M_L(s)$.

With Markov Assumption, N-gram model can predict the next token based on $N - 1$ most recent tokens. For a sequence $w_1, w_2, \dots, w_{i-1}$ and any candidate prediction w_i, the Markov Assumption tells us

$$p(w_i \| w_1, w_2, \dots, w_{i-1}) = p(w_i \| w_{i-N+1}, \dots, w_{i-1}) \tag{3}$$

Because $p(w_{i-N+1}, \dots, w_{i-1})$ is constant, we only need to estimate $p(w_i, w_{i-N+1}, \dots, w_{i-1})$ for all possible w_i and pick up the w_i with the largest value. We can use the N-gram language model to estimate the values:

$$p(w_i, w_{i-N+1}, \dots, w_{i-1}) = M_L(w_i, w_{i-1}, \dots, w_{i-N+1}) \tag{4}$$

N-gram models need smoothing methods to help improve their generalization. It is crucial because many N-grams could not be observed in the training corpus. Given the previous $N - 1$ tokens, if the N-gram formed by word w and the context has not been observed, word w will never have the chance to be the output. This phenomenon harms N-gram models’ generalization ability. To deal with this issue, researchers use linear interpolation to calculate the probabilities rather than counting directly Chen and Goodman (1999). Generally, following Brown et al. Brown et al. (1992), we have the following expression:

$$p(w \| \mathbf{x}_{N-1}) = \lambda_N p_0(w \| \mathbf{x}_{N-1}) + (1 - \lambda_N) p(w \| \mathbf{x}_{N-2})$$

where $\mathbf{x_i}$ is the context of length i, λ_N is the N’s confidence score and $p_0(w \| \mathbf{x}_{N-1})$ is the discounted probability of order $N - 1$ that estimated directly from the counts. In this way, p is calculated recursively and stops when the context is empty.

Methods vary in the way of calculating λ_N. For example, Hellendoorn and Devanbu (2017) use the Jelinek-Mercer smoothing method in their code completion model, whose confidence scores are all 0.5. Witten-bell considers how likely a few particular words follow a context. The more likely the situation is, the higher the confidence score.

4 AdaComplete

4.1 Overall framework

To compensate the DL models’ domain adaptability, we propose AdaComplete, a generic light-weighted framework that integrates the DL-based code completion model with a domain-specific local code completion model. It balances the predictions of the DL model and local model by our designed hand-crafted features. Firstly, we briefly define two main concepts in our model.

Generic Model A generic model is the DL-based code completion model for which we want to compensate the domain adaptability. It is called ‘generic’

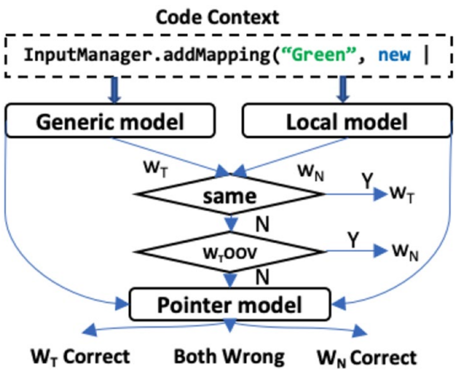


Fig. 2 AdaComplete overall architecture

because it is trained on the source code from the 'generic domain.' And the 'generic domain' is defined in Sect. 3.1.

Local Model A local model is a code completion model to compensate the generic model's domain adaptability. It is called 'local' because it is built on the local domain-specific dataset.

Algorithm 1 AdaComplete Overall Framework

```
Context: x
Data: generic model $M_T$, N-gram Language Model $M_N$, Pointer Model $M_P$
Output: Prediction $w_o$
  $w_T \Leftarrow M_T(\mathbf{x})$
  $w_N \Leftarrow M_N(\mathbf{x})$
  if $w_T == w_N$ then
      $w_o \Leftarrow w_T$                                    ▷ Rule $r_1$
  else if $w_T$ is OOV then
      $w_o \Leftarrow w_N$                                    ▷ Rule $r_2$
  else
      $h_T$ is the inner state of $M_T$ when generating $w_T$;
      $h_N$ is the inner state of $M_N$ when generating $w_N$;
      $c \Leftarrow M_p(h_T, h_N)$;
      if $c == M_T$ then
          $w_o \Leftarrow w_T$
      else if $c == M_N$ then
          $w_o \Leftarrow w_N$
      else
          $w_o \Leftarrow NULL$
      end if
  end if
```

We instantiate the generic model with Transformer-XL and the local model with the N-gram model. We present AdaComplete's overall framework in Algorithm 1. The working process is also illustrated in Fig. 2. We have a generic model and a local N-gram code completion model in the framework. We balance their outputs to get the final prediction. In detail, we collect the two models' predictions and hidden states given the code context. Then we use the following workflow to produce the final prediction as the final output:

- r_1 *(Same Output Check):* If $w_T = w_N$, that is, the outputs from M_T and M_N are the same, we just use w_T to be the output. Else we move on to r_2.
- r_2 *(OOV Output Check):* If w_T is Out of the Vocabulary (OoV), which is a meaningless token that cannot satisfy the users need, we use w_N as the output as it has the chance to be the correct prediction. Else we move on to use M_p.
- M_p *(The Pointer Model):* M_p is a simple classifier that takes part of the inner states of M_T and M_N as its input and decides which output to choose. Significantly, if M_p thinks that both of the outputs are wrong, it will tell the programmer that the code completion tool can not give a correct prediction.

In the following subsection, we will introduce the components of AdaComplete in detail.

4.2 Model instantiation

We use the Transformer-XL network Dai et al. (2019) as the generic DL model, which is a powerful language model and has been used as the base architecture or the strong baselines in many code completion research Liu et al. (2020a, 2020b); Wang and Li (2021). We train our Transformer-XL with the cross-domain dataset from Liu et al. (2020b).

For the local model, we apply the N-gram language model to summarize the local data distribution based on the local vocabulary. It is light-weighted to build and is friendly to a personal computer running an IDE. As it can handle extensive vocabulary, it helps deal with the OoV problem. Besides, Hellendoorn and Devanbu (2017) proves that the N-gram model can have acceptable accuracy performance in small datasets compared with DL-based code completion models.

To train the local model, first, we need to download the N-gram toolkit from the GitHub repo of Hellendoorn and Devanbu (2017).[4] Then we split the dataset of one domain into three splits for training, validation, and test. We use the training split to train the local N-gram model, following the instructions on the toolkit's GitHub homepage. We use Jelinek-Mercer smoothing method in the N-gram model following Hellendoorn and Devanbu (2017).

4.3 Pointer model

We design a pointer model to balance the outputs of the generic and local models. The pointer model is a classification model. It takes a few specifically designed

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

features as its input and adaptively chooses the output from M_T and M_N, the same as classifying the situations. In the following sections, we introduce our designed input features, the labels for classification and the choices of the model instantiation.

4.3.1 Input features of the pointer model

We use the confidence scores of both the generic and local N-gram models as the features. The confidence score is the name for a group of metrics. These metrics measure how likely the models believe that their predictions are correct.

We choose confidence scores as the features for the following reasons: (1) The confidence score suggests the possibility for a model to make mistakes. The lower the confidence score, the more chance a model will make mistakes Corbière et al. (2019). (2) The confidence score is data-free. It is unrelated to the actual form of inputs and the ground truth, and only cares about the inner states of the deep model and N-gram model. Thus the input's space is narrowed, and the training is simplified. (3) They are expandable. Model Confidence is crucial because it helps increase an AI system's robustness by warning the user when the model is not confident about its output and needs human intervention. As for the relevance to choosing the prediction, it is proved by our experiments in the following section.

For the actual features of the confidence scores, we first introduce the features that could be extracted from both the generic and local model. We extract the models' maximum class probability Corbière et al. (2019) and output distribution entropy. Then we introduce the features individually designed for each of the models. For N-gram, we follow Bakhtin et al. (2018) and extract each order's discount probabilities as the features. And for Transformer-XL, we design the confidence score based on the attention scores. We use each of the last layer's attention head entropy summarized from the normalized attention score as the additional confidence measurements. This design is originated from the motivation of self-attention. The model is supposed to pay attention to a few tokens. If the attention scores are averaged over the previous inputs, the model has no clue about what to focus on, that is to say, lacking confidence. The Shannon Entropy is an excellent way to estimate the confidence scores.

The computation of the Transformer-XL's confidence score is illustrated below. For the token in position i, we denote its unnormalized attention scores of one attention head:

$$\widetilde{\mathbf{A}}_{i,j} = \mathbf{A}^{n}_{\tau,i,j} \tag{5}$$

Then we use a softmax function to normalize the attention score $\mathbf{p_{i,j}}$ and calculate the entropy:

$$
\begin{aligned}
p_{i,j} &= \frac{e^{\widetilde{\mathbf{A}}_{i,j}}}{\sum_k e^{\widetilde{\mathbf{A}}_{i,k}}} \\
E_i &= \sum_j p_{i,j} log(\frac{1}{p_{i,j}})
\end{aligned}
\tag{6}
$$

If we denote the attention head as k, we can rewrite the entropy as E_i^k.

All the features are summarized as follows:

- DP-NG: The discount probabilities of the N-gram model.
- MCP-NG: The maximum class probability generated by the N-gram
- VENT-NG: The entropy of the probability distribution over the vocabulary generated by the N-gram
- HENT-TX: The multi-head attention score entropy of the Transformer-XL language model
- MCP-TX: The maximum class probability generated by the Transformer-XL language model
- VENT-TX: The entropy of the probability distribution over the vocabulary generated by the Transformer-XL language model

4.3.2 Output labels of the pointer model

Finally, we introduce the output labels of our pointer model. The outputs of the pointer model are divided into three classes: (1) NN ONLY: The prediction of the generic model is correct while that of the N-gram model is not; (2) N-GRAM ONLY: The prediction of the N-gram model is the correct while that of the generic model is not; (3) BOTH BAD: Both of the predictions are wrong.

4.3.3 Choices of the pointer model

To make AdaComplete both light-weighted and effective, In this paper, we choose two statistical classifiers as the pointer model: the Support Vector Machine, the Random Forest.

(1) *Support Vector Machine (SVM)* Cortes and Vapnik (1995) proposed SVM as a robust linear classifier. It inserts a maximum-margin hyperplane to divide different kinds of data points. We consider SVM because training SVM is very simple. Because it is a convex optimization problem, then convergence is guaranteed.

(2) *Random Forest (RF)* The Random Forest Model is a statistical classification model of ensemble learning first introduced by Ho (1995). The model builds decision trees using the Bagging strategy. However, the bagging strategy is not only applied to the data points but also the features. When predicting, the majority of the votes is the final result. We consider RF because it can decrease the

variance without increasing the bias thanks to the ensemble mechanism. Besides, constructing decision trees is simple.

4.3.4 Build and use the pointer model

To build the pointer model, we first run the inference process of both the generic model and the local model on the validation split of the domain-specific dataset. We collect their intermediate states during the inference and extract the features introduced in 4.3.1. After the inference, we compare the models' output with the ground-truth and build labels according to 4.3.2. The situations can be summarized into three classes as introduced in 4.3.2. In practice, we use integers to represent these situations, 0 for NN ONLY, 1 for NGRAM ONLY, 2 for BOTH BAD. As a result, we create a dataset for the pointer model. We use this dataset to train the pointer model in the same way as training a classification model.

To put the pointer model into use, given the previous code $\mathbf{x}$, we input $\mathbf{x}$ into both the generic and local models to get their inputs and intermediate states. Then we extract features from these states and input them into the local model to get its output. If the output is 0, we use the generic model's output; if the output is 1, we use the local model's output. Otherwise (the output is 2), we warn the user that both the outputs are wrong and the prediction is failed.

5 Experiments and analysis

5.1 Baselines

To prove AdaComplete's effectiveness, except for Transformer-XL, we introduce another two state-of-the-art generic models as our baselines. These code completion models incorporate specific techniques to modify the simple Transformer-family-based deep code completion models. They are stronger than the Transformer-XL-based code completion model we use in AdaComplete. One is GRU BPE (BPE for short) Karampatsis et al. (2020) from ICSE 2020. This work uses the BPE technique to represent tokens. The other is CugLM Liu et al. (2020b) from ASE 2020, which uses the multi-task technique to pre-train their Transformer backend. These baselines cover the main technology trends of code completion: BPE, Transformer, and Pre-training.

5.2 Data preparation

The generic dataset used for training the generic model is the Java dataset used by Liu et al. (2020b), and we call it *Java_full*. It is a large dataset with 9708 projects and 800,983 files. To reproduce the testing results, we follow the partition scheme

Table 2 Statistics of the collected datasets

	Train_token	Valid_token	Test_token	Unk rate (%)
jMonkeyEngine	666,019	291,465	248,495	23.00
jFaceRec	119,8077	542,909	447,096	23.70
CodeGeneration	1,583,434	638,783	494,019	24.34
j2me	2,126,631	929,536	967,579	30.54

described in Liu et al. (2020b) that 94% of the projects go to the training set and the rest are equally divided for validation and test.

The local dataset is the collection of projects in one specific domain. We choose four domains and collect projects in each domain correspondingly to form four domain-specific datasets. The domains are:

- *jFaceRec*: Java projects about face recognition
- *jMonkeyEngine*: Java projects about a game engine called jMonkeyEngine
- *CodeGeneration*: Java projects about automatic code generation
- *j2me*: Java projects about j2me

All the projects in these datasets are collected from the public open-source GitHub repositories. We exclude the files that appear in *Java_full*. After filtering, we reserve projects with more than one source file and tokenize each program into a token sequence with Python package *javalang*. Then we randomly permute each project's files and split them into the training, validation, and test sets. We list the statistics about the datasets in Table 2.

5.3 Experimental setup

5.3.1 Experiment procedure

The experiment procedure is divided into the following steps:

- *Step 1:* We train a Transformer-XL-based deep code completion model on a global dataset.
- *Step 2:* We choose a domain and draw a local dataset from it.
- *Step 3:* We train an N-gram language model on the train part of the local dataset.
- *Step 4:* We train the pointer models on the validation part of the local dataset. All modules in AdaComplete are ready.
- *Step 5:* We test AdaComplete and the baselines on the test part of the local dataset then compare the results.

5.3.2 Models and parameters

We describe the models and their configurations in this section.

Generic Models Because we use the dataset in Liu et al. (2020b) as the generic dataset to train these baselines, and these baseline methods are also tested in Liu et al. (2020b), we directly follow their model setup. We utilize a Transformer-XL implemented by HuggingFace[5] with 6 layers, 516-dimensional hidden states, and 6 attention heads. The hidden inner size of the feed-forward layer is 3072. The memory length is 256. For GRU BPE, we set the hidden size to be 1500, and the other part in their code unchanged.[6] As for CugLM, because it involves pre-training and its training process is complex, we directly employ their implementation and model setup.[7]

Local N-gram Model Our implementation of the local N-gram model is a modification of Hellendoorn's implementation Hellendoorn and Devanbu (2017).[8] We get the N-gram model's inner states as mentioned in the method. Besides that, everything is the same as they were. Following the setup reported in Hellendoorn and Devanbu (2017), we use the Jelinek-Mercer smoothing method and set the N-gram order as 6.

Statistical Classifier The SVM and the Random Forest are implemented by Pedregosa et al. (2011). For both models, we use their default structure.

5.3.3 Experimental environment

We used 2 NVIDIA Tesla V100 GPUs to train our generic Transformer-XL, CugLM, and BPE model. Our CPU is Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz. For the Random Forest, we manually set its parallel jobs as 16. For the other implementations of the pointer model, the numbers of parallel jobs are the same as the default.

5.3.4 Metrics

We use top-1 accuracy as the metric, matching accuracy between the oracle and the token predicted with the highest probability. If the baselines think their output is OOV, we regard this prediction as a failure.

5.3.5 Vocabulary

The vocabularies for the generic models and the local N-gram language model are different. We built the vocabulary for the generic models based on the training set of *Java_full*. We followed Liu et al. (2020b) to keep the most frequent 50,000 tokens. We built the N-gram language model's vocabulary based on these local datasets' training parts and kept every token we met. Based on the generic models'

[5] https://huggingface.co/transformers/index.html.

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

[7] https://github.com/LiuFang816/CugLM.

[8] https://github.com/SLP-team/SLP-Core.

Table 3 Upper-bound statistics

	Generic only (%)	Upper bound (%)
jFaceRec	71.33	87.11
jMonkeyEngine	62.16	80.62
CodeGeneration	64.51	79.66
j2me	61.23	80.19

vocabulary, we counted the frequency of the OOV words of the local datasets, denoted as ‘unk rate’. These statistics are presented in Table 2.

5.4 Research questions and results

To evaluate our method, we conducted experiments to investigate the following research questions:

- *RQ1: Performance Upper Bound* The proposed method does not change the outputs of the generic and local models. It just learns to choose from the predictions. AdaComplete can achieve the upper bound of the performance if it can always select the suitable model, and we compute the upper bound performance in this research question.
- *RQ2: Overall Method Performance* We propose AdaComplete to improve the performance of the generic code completion model. So we conduct experiments to compare AdaComplete with the baselines that do not utilize AdaComplete.
- *RQ3: Contribution of Each Component* AdaComplete utilizes two rules and one pointer model to make the choices. How much does each of them contribute to the overall performance?
- *RQ4: Difference among Pointer Models* We propose two kinds of pointer models: the SVM and the RF with the corresponding reasons. What are their performances individually? What causes these differences?
- *RQ5: Comparison with Fine-tuning* To adapt the generic model to a specific domain, using fine-tuning is intrinsic. However, we analyze that applying fine-tuning is difficult because of lacking training data. Besides, fine-tuning consumes too much time. So we design experiments to prove our analysis.

5.4.1 RQ1: performance upper-bound

To calculate the performance upper-bound of AdaComplete in each domain, we assume that we have a perfect pointer model which can make every choice correctly. Then we calculate the accuracy based on the perfect pointer model.

Table 4 Comparisons against the state-of-the-arts

		Accuracy on datasets				
		Generic (%)	jMonkey-Engine (%)	jFaceRec (%)	CodeGeneration (%)	j2me (%)
Baselines	Transformer-xl	72.12	62.16	71.33	64.51	61.23
	BPE	70.29	59.12	69.89	57.56	43.23
	CugLM	84.06	75.19	74.84	74.65	67.66
	N-gram	40.26	65.68	81.35	63.99	61.47
AdaComplete	UPPER-BOUND	74.86	80.62	87.11	79.66	80.19
	AdaComplete(svm)	72.12	76.90	86.57	77.88	78.80
	AdaComplete(rf)	72.12	**78.43**	**86.93**	**77.97**	**78.84**

The bold numbers in the Table means 'the best'

We report the upper-bounds of every dataset's test set in Table 3. They are presented in the 'UPPER BOUND' column, while the second column shows the base deep code generator's actual performance, that is, the performance only using the generic model.

According to Table 3, we find that if our pointer models are perfect, AdaComplete can bring improvements by 20% on average. It is nearly one-third of the original accuracy performance. Besides, the portion of tokens in the charge of the pointer model is not small. Thus carefully designing and training the pointer models are necessary.

5.4.2 RQ2: overall method performance

We compare our model with several state-of-the-art code completion baselines, and the results are shown in Table 4. The first four rows are for the three baseline generic models and our local N-gram model. The following four lines are for our method, AdaComplete. The first one of them is for the upper-bounds reported in Table 3, and the others are for AdaComplete with different pointer models: SVM, and Random Forrest (RF).

According to Table 4, all of the implementations of AdaComplete have outperformed the generic baselines and the local N-gram model in accuracy on all the four domain-specific datasets.. Compared with the best method of the baselines, our improvements range from the percentage of 3 to 12. Considering that the generic model is based on a Transformer-XL without training tricks and AdaComplete can still beat the strong baseline of CugLM, the effectiveness of AdaComplete is convincing.

Specifically, we conduct experiments on cross-domain level datasets. We present the results in column 'generic.' We find that our local N-gram model has relatively poor performance for the projects in the generic domain compared with the generic DL-based model. Thus, the upper bound of AdaComplete is low. Moreover, the automatic pointer model limits the overall performance since they can not avoid

Table 5 Absolute accuracy contributed by the components

	SAME (%)	UNK (%)	Pointer		
			SVM (%)	RF (%)	NN-ONLY (%)
jMonkeyEngine	47.37	12.03	16.49	**19.03**	14.94
jFaceRec	65.62	13.40	7.55	**7.91**	5.77
CodeGeneration	48.84	10.60	18.43	**18.52**	15.67
j2me	42.52	15.75	20.54	**20.58**	18.72

The bold numbers in the Table means 'the best'

making mistakes (Tabel 11). So following the design principle of AdaComplete, we force the pointer to choose the generic DL-based model. So as reported in the chart, the performance of AdaComplete is equal to the generic DL-based model, transformer-XL.

5.4.3 RQ3: contribution of each component

In the overall picture, we have shown that our method is effective. We further evaluate the performance of each component of AdaComplete (two rules and one pointer model) in this research question.

Each of the three components contributes to the final result of the accuracy, and we list the results in Table 5. They are (1) Rule 1: Whether the outputs are the same; (2) Rule 2: Whether the output from the generic model is OOV; (3) The Pointer Model. Except for Column 'NN-ONLY,' all the numbers are all the percentages of the successfully predicted tokens by the modules. Column 'SAME' is for Rule 1: If the predictions are the same, just give it to the user. Column 'UNK' is for Rule 2: If the generic model thinks the prediction is OOV, just use N-gram's output as the final prediction. Columns of 'POINTER' are for the three types of pointer models. As for Column 'NN-Only,' it is the percentage of the correct predictions if we do not use the pointer model and the N-gram language model. We list it here for comparison and hope that the maximum number of the pointer models on each dataset can exceed the numbers in 'NN-ONLY.' If so, the pointer model is effective, or it would be better to keep using the generic model's predictions. Based on the results, we have the following analysis.

- Rule 1 contributed the largest proportion of the overall accuracy. Therefore, though add a rule makes AdaComplete complicated compared with simply utilizing the pointer model, it is still worthy. The pointer model can not avoid making mistakes. By adding Rule 1, we reduce much deadweight loss.
- The contribution of Rule 2 reveals AdaComplete's ability to solve the OOV problem. Besides, it emphasize the importance of solving the OOV problem in code completion. Because it can increase the performance upper-bound of deep code completion models.

Table 6 Statistics of recall on jMonkeyEngine and jFaceRec

	jMonkeyEngine		jFaceRec	
	NN ONLY (%)	NGRAM ONLY (%)	NN ONLY (%)	NGRAM ONLY (%)
SVM	83.25	80.69	94.58	90.14
RF	**90.48**	**87.80**	**98.06**	**96.90**

The bold numbers in the Table means 'the best'

- The contribution of the pointer model is greater than the original deep code completion model. The result not only proves that the pointer model is effective, but further confirm the existence of domain shift in code completion.

5.4.4 RQ4: difference among pointer models

We compare the pointer model's choices horizontally and give some insights from several aspects. On all four datasets, RF performs better than SVM.

To further investigate the cause, we first calculate each label's recall score of the pointer models, then we count the label proportions in the whole dataset. The recall scores are posted in Tables 6 and 7.

From Tables 6 and 7, we find that RF completely outperforms SVM on jMonkeyEngine and jFaceRec. On CodeGeneration and j2me, although SVM outperforms RF on NN ONLY, it still behaves much worse on NGRAM ONLY, not to mention that the performance on NN ONLY is not a considerable advantage.

RF performs better on skewed data, which results in its higher accuracy than SVM. We investigate the label proportion of these datasets (please refer to Table 8. NGRAM ONLY is the minor label, much less than NN ONLY. SVM adapts to the

Table 7 Statistics of recall on CodeGeneration and j2me

	CodeGeneration		j2me	
	NN ONLY (%)	NGRAM ONLY (%)	NN ONLY (%)	NGRAM ONLY (%)
SVM	**93.31**	83.84	**95.46**	83.17
RF	93.19	**86.30**	94.97	**87.44**

The bold numbers in the Table means 'the best'

Table 8 Ratio of label *NN ONLY* and label *NGRAM ONLY*

	jFaceRec	jMonkeyEngine	CodeGeneration	j2me
Ratio $\frac{NN\_ONLY}{NGRAM\_ONLY}$	2.47	2.37	3.45	5.83

Table 9 Accuracy comparison with fine-tuning

	jMonkey-Engine (%)	jFaceRec (%)	CodeGen-eration (%)	j2me (%)
Before FT	62.16	71.33	64.51	61.23
After FT	69.22	76.73	68.88	63.07
AdaComplete	**78.43**	**86.93**	**77.97**	**78.84**

The bold numbers in the Table means ‘the best’

Table 10 Time consumed comparison

	jMonkeyEngine	jFaceRec	CodeGeneration	j2me
AdaComplete	1 m 6 s	52.58 s	4 m 40 s	3 m 58 s
Fine-tuning	1 h 9 m	1 h 2 m	1 h 5 m	1 h 47 m

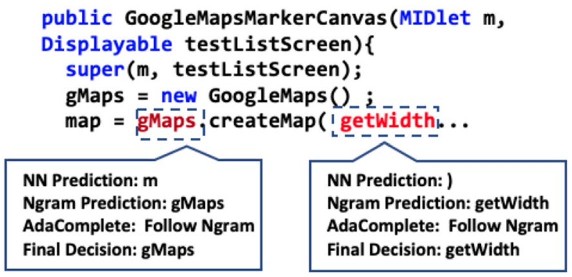

```
public GoogleMapsMarkerCanvas(MIDlet m,
Displayable testListScreen){
  super(m, testListScreen);
  gMaps = new GoogleMaps() ;
  map = gMaps.createMap( getWidth...
```

Fig. 3 Case 1 and Case 2

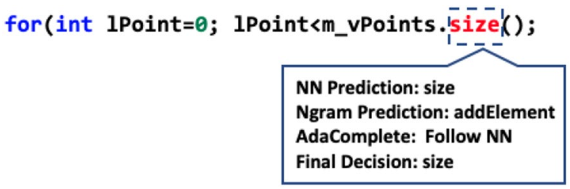
skewed data worse than RF, has lower recall values on the minor label, and performs worse than RF looking from the whole dataset.

5.4.5 Comparison with fine-tuning

In practice, people use the **fine-tuning** technique to solve one domain’s domain shift, which is to keep training the generic model on the specific domain. However, there might not be enough data for fine-tuning for a specific domain. Besides, fine-tuning requires too many computation resources and too much time, like training the generic models.

To prove the analysis above, we conduct the fine-tuning experiments on Transformer-XL. The training dataset is the same as the N-gram language model for fair. We kept training the Transformer-based code completion model until convergence. We present the final test accuracy in Table 9 and the time these experiments consumed in Table 10. In Table 9, each column is for a domain. As for the rows, ‘Before FT’ is for Transformer-XL’s accuracy before fine-tuning. ‘After FT’ is the accuracy after fine-tuning. ‘AdaComplete’ is the accuracy of AdaComplete with RF as the

Fig. 4 Case 3


```
for(int lPoint=0; lPoint<m_vPoints.size();
```

pointer model. In Table 10, Row ‘Fine-tuning’ is for the time consumed fine-tuning the generic model. Row ‘AdaComplete’ is a summation for training the N-gram language model and the RF pointer model. We report the RF pointer model’s training time because it consumes more time in training than SVM.

As seen from the results, it is apparent that the fine-tuned Transformer-XL has worse performance than AdaComplete and consumes much more time for training. So we can conclude that AdaComplete is more practical and effective than fine-tuning dealing with the domain-shift introduced by domains.

6 Case study

This section presents several real cases to analyze AdaComplete’s performance.

Figure 3 shows two cases, i.e., Case 1 and Case 2, on the domain of GoogleMap. The two cases show AdaComplete compensates the generic model’s domain adaptability on the domain of GoogleMap by choosing the local model’s predictions.

Case 1 A Google Map object ‘map’ is created by the factory ‘gMaps,’ where ‘gMaps’ is expected. ‘gMaps’ is a domain-specific entity, whose semantic cannot be captured by the generic model, thus is not correctly predicted. Instead, it conservatively predicts the method parameter ‘m’ as the final result. Since the local model is trained on the specific domain source code, the semantic information of the domain-specific entities can be successfully captured and correctly predicted. AdaComplete measures the generic model and the local models’ confidence score and chooses to trust N-gram. As a result, AdaComplete successfully predicts ‘gMaps’ here.

Case 2 The parameter list of method ‘create map’ needs to be filled. However, the generic model doesn’t know the parameter list. So it predicts the close parenthesis conservatively. On the contrary, the local model knows the parameter list and gives its prediction ‘getwidth.’ AdaComplete measures the generic model and the local models’ confidence score and chooses to trust N-gram. As a result, we successfully predict ‘getwidth’ here.

We present Case 3 in Fig. 4. This case shows that AdaComplete will follow the generic model when the token predicted is not domain-relevant.

Case 3 Case 3 is to set the loop length. The variable ‘m_vPoints’ is a Java Array. Because the method ‘size’ of a Java Array is usually called to set the loop length, the generic model predicts ’size.’ However, in the domain of this case, the method ‘addElement’ is called more frequently than ‘size.’ Hence, the local model predicts ‘addElement.’AdaComplete measures the generic model and the local models’

Table 11 Pointer recall on generic domain

	NN ONLY (%)	NGRAM ONLY (%)
SVM	87.87	27.34
RF	78.61	33.47

confidence score and chooses to trust the generic model. As a result, we successfully predict 'size' here.

In the above cases, we show that by applying AdaComplete, we can predict domain-specific tokens while keeping the generic model's prediction when the correct prediction is not domain-specific.

7 Discussion

7.1 Method generalization

AdaComplete can integrate all kinds of generic models and improve their performances as long as we find a model-specific metric to measure the model confidence. For transformer-based methods, in this paper, the Transformer-XL, and the other works (Ciniselli et al. 2021; Alon et al. 2020), we draw 'HENT-TX' as the model-specific metric. For RNN-based methods Salton et al. (2017), we can use the entropy of its attention scores. For graph neural networks(Yang et al. 2022; Wang and Li 2021), we can use attention scores for those using self-attention, or involve the node feature distance for the others Vashishth et al. (2019).

Our method can also be applied to DL-based models with pre-training(Liu et al. 2020b; Ciniselli et al. 2021) because pre-training is just a technique for training a deep code completion model. The measurements of their confidence scores are not different from the non-pre-trained code completion models.

7.2 Conservative strategy

We use the conservative strategy in our method and our experiment for strong proof of AdaComplete's effectiveness. Our detail setups are:

- *BOTH BAD* We set the label, BOTH BAD, as the conservative recommendation strategy for AdaComplete. We choose not to give token recommendations when the models believe their recommendations are not likely to be correct. Previous works utilize the aggressive recommendation strategy in that they give recommendations under any circumstances, regardless of the probability of making mistakes. AdaComplete can have better results using the aggressive strategy. But it still outperforms the baselines, which strongly proves its effectiveness.

Table 12 Performance comparison for anti-data skew methods

	jMonkey-Engine (%)	jFaceRec (%)	CodeGen-eration (%)	j2me (%)
SVM-raw	76.90	86.57	77.88	78.80
SVM-balanced	77.20	85.97	75.81	77.54
RF-raw	78.43	86.93	77.97	78.84
RF-balanced	75.89	86.07	75.43	76.70

- *Backbone Selection* We use Transformer-XL as the backbone deep learning model rather than CugLM as a conservative experiment design. All the experiments intended to prove the effectiveness of AdaComplete, which is a general framework for any deep learning models. So it is more persuasive that with the help of AdaComplete, a weaker model defeats a more robust model. In our experiments, AdaComplete with a weaker Transformer-XL outperforms all the baselines, which proves that AdaComplete works well.

7.3 Limitation and future work

The first threat of the effectiveness of AdaComplete is the effectiveness of the local model. Empirically, although the generic models have performance loss in a specific domain, the accuracy will not be too bad to use. And the pointer model does not need too many data points to train. So the threat to AdaComplete's validity is the local model. According to Tabel 4, our local models, the N-gram models, perform worse than the generic models. Besides, the upper-bounds of AdaComplete in our experiments have not reached 100%, which is determined by N-gram's performance. To improve AdaComplete's performance, we need to design our local model better.

The second threat to the effectiveness of AdaComplete is the performance of the pointer model. Although AdaComplete is explicitly designed for domain-specific code completion tasks, we hope it could perform reasonably well with the generic DL-based model with the help of automatic pointer models. However, what we expect is not realized because of the poor performance of the auto-pointer models (see Table 11). We manually switch to the generic DL-based model for completion on the generic domain, but it is worth thinking about how to build pointer models that perform well in the generic domain.

7.4 Impact of data skew

Our well-designed features relieve the threat of data skew. In Table 6, we mention that data skew is serious in our datasets. Thus we apply more robust versions of SVM and RF to see the differences. The robust version of SVM is the weighted SVM Yang et al. (2007). They give each misclassified sample a price. The sample in the minority class receives higher prices than the majority class. The robust version of RF is proposed by Chen and Breiman (2004). They proposed to draw samples from the majority class and the minority class separately to keep the data balance within a subset. The results are shown in Table 12. SVM-raw and RF-raw are the pointer models we use in AdaComplete. SVM-balanced and RF-balanced are their corresponding robust version.

The results show that the robust versions have little difference from their raw versions. Data skew does not have a destructive negative effect. Considering that the pointer models are simple statistical classifiers, we conclude that our hand-crafted features are separable and well-designed.

8 Related work

An N-gram-based code completion model uses the N-gram language model to estimate the output probabilities for all possible outputs. The basic N-gram language models explicitly count the n-grams in the training set and use these counts during inference. The works using N-gram for code completion develop a series of methods to help N-gram language models to adapt to code nature. Raychev et al. (2014) trains an RNN to adjust the N-gram language model's output probabilities. Roos (2015) tries multiple smoothing methods to find the best one for code completion. Tu et al. (2014) add a 'cache' component to help the N-gram language model to adapt to the 'localness of source code'. Hellendoorn and Devanbu (2017) helps the N-gram language model with a much more expansive, multi-level notion of locality that is well-suited for modeling software.

A DL-based code completion model uses the deep learning method to estimate the output probabilities for all possible outputs. Inherent from neural networks, deep code completion models have compact parameters, and large model capacity to learn from extensive program corpus effectively and efficiently Liu et al. (2017, 2020b); Li et al. (2018). Furthermore, because neural networks do well in modeling non-linear relations and mining from a larger dataset, deep code completion models perform better than those using statistical machine learning methods in general Karampatsis et al. (2020), not to mention the rule-based code completion methods.

Researchers have introduced many neural network architectures, training tricks, and program representations into code completion. Dam et al. (2016) used Long Short Term Memory architecture to build a language model for code completion. Kim et al. (2020) introduced Transformer Vaswani et al. (2017) to code completion. To alleviate the big vocabulary issue in code completion, Li et al. (2018) introduced the copy mechanism Vinyals et al. (2015) that learns to copy tokens from the context as the output. Karampatsis et al. (2020) uses Byte-pair Encoding to split the tokens into sub-tokens. As for the program representation, Kim et al. (2021) tried to feed ASTs to a transformer for code completion, hoping the syntax information could help to improve the performance. When the pre-training technique is on-trend in the field of Natural Language Processing Devlin et al. (2018), Liu et al. (2020b) used this technique to improve the performance of deep code completion models further.

9 Conclusion

We are the first to raise the issue that applying generic models to a specific domain harms the models' performance in code completion. And we propose AdaComplete, a novel method integrating the generic model and the local model to reduce the loss by balancing the models' predictions. The tests show that our method can significantly increase the overall accuracy and beat the strong generic models. Further, we find that AdaComplete beat fine-tuning in both time and accuracy. Compared with fine-tuning, AdaComplete is more convenient and powerful.

In the future, we'll try to improve the performance of both local and pointer models. We will investigate and design a more robust and light-weighted model for the local models to improve AdaComplete's upper bound performance. We will design more meaningful and model-free features for the pointer models to improve the pointer models' accuracy. Then we will introduce Online Learning, trying to build AdaComplete on the fly and regard a single project as a unique domain.

Author contributions All authors contributed to the proposal of the idea. ZW: conducted the experiments. ZW and FL: wrote the manuscript. All authors reviewed the manuscript.

Declarations

Conflict of interest The authors declare no competing interests.

References

Allamanis, M., Barr, E.T., Devanbu, P., et al.: A survey of machine learning for big code and naturalness. ACM Comput. Surv. (CSUR) **51**(4), 1–37 (2018)

Alon, U., Sadaka, R., Levy, O., et al.: Structural language models of code. In: Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event, Proceedings of Machine Learning Research, vol 119. PMLR, pp 245–256, (2020) http://proceedings.mlr.press/v119/alon20a.html

Bakhtin, A., Szlam, A., Ranzato, M., et al.: Lightweight adaptive mixture of neural and n-gram language models.(2018) arXiv e-prints arXiv–1804

Barone, AVM., Haddow, B., Germann, U., et al.: Regularization techniques for fine-tuning in neural machine translation. (2017) arXiv preprint arXiv:1707.09920

Bhoopchand, A., Rocktäschel, T., Barr, E., et al.: Learning python code suggestion with a sparse pointer network. (2016) arXiv preprint arXiv:1611.08307

Brown, P.F., Della Pietra, S.A., Della Pietra, V.J., et al.: An estimate of an upper bound for the entropy of english. Comput. Linguist. **18**(1), 31–40 (1992)

Bruch, M., Monperrus, M., Mezini, M.: Learning from examples to improve code completion systems. In: van Vliet H, Issarny V (eds) Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009. ACM, pp 213–222, (2009) https://doi.org/10.1145/1595696.1595728,

Chen, C., Breiman, L.: Using Random Forest to Learn Imbalanced Data. University of California, Berkeley (2004)

Chen, S.F., Goodman, J.: An empirical study of smoothing techniques for language modeling. Comput. Speech Lang. **13**(4), 359–394 (1999)

Ciniselli, M., Cooper, N., Pascarella, L., et al.: An empirical study on the usage of bert models for code completion. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), IEEE, pp. 108–119 (2021)

Corbière, C., Thome, N., Bar-Hen, A., et al.: Addressing failure prediction by learning model confidence. In: Wallach HM, Larochelle H, Beygelzimer A, et al (eds) Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada, pp. 2898–2909, (2019) https://proceedings.neurips.cc/paper/2019/hash/757f843a169cc678064d9530d12a1881-Abstract.html

Cortes, C., Vapnik, V.: Support-vector networks. Mach. Learn **20**(3), 273–297 (1995). https://doi.org/10.1007/BF00994018

Dai, Z., Yang, Z., Yang, Y., et al.: Transformer-xl: Attentive language models beyond a fixed-length context. (2019) arXiv preprint arXiv:1901.02860

Dam, HK., Tran, T., Pham, T.: A deep language model for software code. (2016) arXiv preprint arXiv:1608.02715

Devlin, J., Chang, MW., Lee, K., et al.: Bert: Pre-training of deep bidirectional transformers for language understanding. (2018) arXiv preprint arXiv:1810.04805

Feng, Z., Guo, D., Tang, D., et al.: Codebert: A pre-trained model for programming and natural languages. In: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings, pp 1536–1547 (2020)

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

Hellendoorn, VJ., Devanbu, P.: Are deep neural networks the best choice for modeling source code? In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp 763–773 (2017)

Hindle, A., Barr, ET., Su, Z., et al.: On the naturalness of software. In: Glinz M, Murphy GC, Pezzè M (eds) 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland. IEEE Computer Society, pp. 837–847, (2012) https://doi.org/10.1109/ICSE.2012.6227135,

Ho, TK.: Random decision forests. In: Proceedings of 3rd international conference on document analysis and recognition, IEEE, pp. 278–282 (1995)

Hou, D., Pletcher, DM.: 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 (2010)

Kamath, A., Jia, R., Liang, P.: Selective question answering under domain shift. In: Jurafsky D, Chai J, Schluter N, et al (eds) Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020. Association for Computational Linguistics, pp. 5684–5696, (2020) https://doi.org/10.18653/v1/2020.acl-main.503,

Karampatsis, RM., Babii, H., Robbes, R., et al.: Big code!= big vocabulary: Open-vocabulary models for source code. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), IEEE, pp. 1073–1085 (2020)

Kim, S., Zhao, J., Tian, Y., et al.: Code prediction by feeding trees to transformers. (2020) arXiv preprint arXiv:2003.13848

Kim, S., Zhao, J., Tian, Y., et al.: Code prediction by feeding trees to transformers. In: 43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021. IEEE, pp. 150–162, (2021) https://doi.org/10.1109/ICSE43902.2021.00026,

Kuang, K., Xiong, R., Cui, P., et al.: Stable prediction with model misspecification and agnostic distribution shift. In: The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applicationsof Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020. AAAI Press, pp. 4485–4492, (2020) https://aaai.org/ojs/index.php/AAAI/article/view/5876

Li, J., Wang, Y., Lyu, MR., et al.: Code completion with neural attention and pointer networks. In: Lang J (ed) Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden. ijcai.org, pp. 4159–4165, (2018) https://doi.org/10.24963/ijcai.2018/578,

Liu, C., Wang, X., Shin, R., et al.: Neural code completion. (2017) https://openreview.net/forum?id=rJbPBt9lg

Liu, F., Li, G., Wei, B., et al.: 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 (2020a)

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

Nguyen, TT., Nguyen, AT., Nguyen, HA., et al.: A statistical semantic language model for source code. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 532–542 (2013)

Pedregosa, F., Varoquaux, G., Gramfort, A., et al.: Scikit-learn: machine learning in python. J. Mach. Learn. Res. **12**, 2825–2830 (2011)

Radford, A., Wu, J., Child, R., et al.: Language models are unsupervised multitask learners. OpenAI Blog **1**(8), 9 (2019)

Raychev, V., Vechev, MT., Yahav, E.: Code completion with statistical language models. In: PLDI. ACM, pp. 419–428 (2014)

Robbes, R., Lanza, M.: Improving code completion with program history. Autom. Softw. Eng. **17**(2), 181–212 (2010). https://doi.org/10.1007/s10515-010-0064-x

Roos, P.: Fast and precise statistical code completion. In: ICSE (2). IEEE Computer Society, pp. 757–759 (2015)

Saenko, K., Kulis, B., Fritz, M., et al.: Adapting visual category models to new domains. In: Daniilidis K, Maragos P, Paragios N (eds) Computer Vision - ECCV 2010, 11th European Conference on Computer Vision, Heraklion, Crete, Greece, September 5-11, 2010, Proceedings, Part IV, Lecture Notes in Computer Science, vol 6314. Springer, pp. 213–226, (2010) https://doi.org/10.1007/978-3-642-15561-1_16,

Salton, G., Ross, R., Kelleher, J.: Attentive language models. In: Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 1: Long Papers), pp. 441–450 (2017)

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

Vashishth, S., Yadav, P., Bhandari, M., et al.: Confidence-based graph convolutional networks for semi-supervised learning. In: The 22nd International Conference on Artificial Intelligence and Statistics, PMLR, pp. 1792–1801 (2019)

Vaswani, A., Shazeer, N., Parmar, N., et al.: Attention is all you need. (2017) arXiv preprint arXiv:1706.03762

Vinyals, O., Fortunato, M., Jaitly, N.: Pointer networks. (2015) arXiv preprint arXiv:1506.03134

Wang, J., Lan, C., Liu, C., et al.: Generalizing to unseen domains: A survey on domain generalization. In: Zhou Z (ed) Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021. ijcai.org, pp. 4627–4635, (2021) https://doi.org/10.24963/ijcai.2021/628,

Wang, Y., Li, H.: Code completion by modeling flattened abstract syntax trees as graphs. In: Proceedings of the AAAI Conference on Artificial Intelligence, pp. 14,015–14,023 (2021)

Yang, K., Yu, H., Fan, G., et al.: A graph sequence neural architecture for code completion with semantic structure features. J. Softw. Evol. Process **34**(1), e2414 (2022)

Yang, X., Song, Q., Wang, Y.: A weighted support vector machine for data classification. Int. J. Pattern Recogn. Artif. Intell. **21**(05), 961–976 (2007)

Authors and Affiliations

Zejun Wang[1,2] · Fang Liu[3] · Yiyang Hao[4] · Zhi Jin[1,2]

Zejun Wang
zejunwang@pku.edu.cn

Fang Liu
fangliu@buaa.edu.cn

Yiyang Hao
haoyiyang@nnthink.com

1 School of Computer Science, Peking University, Beijing, China

2 Key Lab of High Confidence Software Technology, MoE (Peking University), Beijing, China

3 State Key Laboratory of Software Development Environment, Beihang University, Beijing, China

4 Silicon Heart Tech Co., Beijing, China