Program Generation with Hybrid of Structural and Semantic Features Retrieval

Kang Yang

Department of Computer Science and Engineering
East China University of Science and Technology, Shanghai 200237, China
Shanghai Key Laboratory of Computer Software
Evaluating and Testing, Shanghai 201112, China
15921709583@163.com

Huiqun Yu*

Department of Computer Science and Engineering
East China University of Science and Technology, Shanghai 200237, China
Shanghai Engineering Research Center of Smart Energy, Shanghai, China
yhq@ecust.edu.cn

Guisheng Fan†

Department of Computer Science and Engineering
East China University of Science and Technology, Shanghai 200237, China
gsfan@ecust.edu.cn

Zijie Huang

Department of Computer Science and Engineering
East China University of Science and Technology, Shanghai 200237, China
hzjdev@foxmail.com

Ziyi Zhou

Department of Computer Science and Engineering
East China University of Science and Technology, Shanghai 200237, China
zhouziyi96@qq.com

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

Due to the accelerating code development and improving code maintainability, code generation has recently attracted more and more attention. In the model of generating program source code from natural language, the most effective method is to generate an intermediate architecture (such as Abstract Syntax Tree) combined with a deep learning model. However, these models have the following problems: 1. The data information is

*Corresponding authors
†Corresponding authors

underutilized and the correlation between samples is not considered. 2. Lack of the ability to memorize large and complex structures, so that complex codes cannot be generated correctly. To this end, we propose HRCODE model, a code generation architecture based on **H**ybrid of structural and semantic features **R**etrieval **CODE** model. We transform the natural language description into an intermediate structure with structural features. Then, the NL and the intermediate structure are embedded into a vector through weight mixing, and we use the similarity between each vector to retrieve the most relevant samples. Finally, the new input is brought into the PLBART model to generate code. Experiments show that HRCODE is at least 4.7% higher than the state-of-the-art models in the ACC indicator and at least 10.3% higher in the BLEU-4 indicator. We have released our code at https://github.com/jesokang/HRCODE.

Keywords: Code Generation; Deep Learning; Program Comprehension.

1. Introduction

Code generation is a extremely challenging task of converting natural language(NL) descriptions into codes[1][2][3]. The major challenge encountered in this task is that the input and output data are cross-language. Besides, the NL structure is more arbitrary, but the generated code needs a clear and executable structure output. Accordingly, it is difficult to construct an effective generative model for code generation. However, generating uniformly structured code through natural language description can accelerate the development efficiency of programmers and improve the maintainability of the code. Therefore, code generation is an important area and worthy of in-depth research.

In response, generating code from natural language has been actively studied. Researchers constructed traditional generative models[4][5][6] through static grammatical rules. With the rapid development of deep learning, more and more researchers use deep learning technology for program generation. Jia et al.[7] and Locascio et al.[8] use sequence-to-sequence model to convert the target code into a sequence of symbols. However, Sequential approaches cannot ensure that the generated code is syntactically and structurally correct. To solve this problem, researchers have proposed a method of transforming natural language description into the intermediate structure of the code, and then using the structure to generate the code. Yin et al.[9] and Rabinovich et al.[2] represent the code as Abstract Syntax Tree (AST), which is effective in improving accuracy. Li Dong[10] proposed a method of generating rough sketches, which masked fine-grained information (such as variable names). Then, the generated model fills in the missing details by combining the structure of the sketch. These methods are effective as it enforces the well-formedness of the output code. However, the number of nodes in the tree usually greatly exceeds the length described by NL. Therefore, structure-based methods usually cannot generate correct codes for low frequency words.

Afterwards, Hayati et al.[11] propose RECODE model which inspired by machine translation[12]. Retrieval and neural models[13][14] have proved to be successful in processing rare words, so the model effectively improves the accuracy through NL retrieval. However, RECODE model use edit distance to calculate the similarity

between samples, which is not appropriate to retrieve. Besides, the current retrieval methods do not fully utilize the existing information.

To address these issues, we propose a novel framework to generate better codes by using the hybrid retrieval method of NL and code intermediate structure. We use TRANX model[15] to transform NL utterances into formal meaning representations(MR), which is intermediate structure of code. We convert the NL and MR embeddings into vectors, and hybridize the two features to vector characterize each sample. Then, the hybrid vector will be used to retrieve the training samples. Finally, we bring the new input into the model to generate code. To evaluate the performance of our proposed model, we conduct experiments on real-world dataset. Compared with the six the state-of-the-art models, our model can improve from 3.7% to 10.5% at metric of accuracy and can promote BLEU-4 from 4.8% to 6.2%.

The main contributions of this paper are as follows:

- We propose a new framework. The model retrieval based approach HRCODE to improve the accuracy and BLEU-4 performance for code generation. Specifically, we introduce a new retrieval code strategy. Then, we generate the code by more efficient model input.
- We design a new retrieval strategy. We convert natural language and structural feature embeddings into hybrid feature vector representations, which can effectively retrieve more relevant codes.
- We evaluate our proposed model on real-world dataset. Experimental results show that HRCODE achieves the best performance compared with the state-of-the-art models.

The rest of this paper is organized as follows: Section 2 discusses related work. The proposed apporach is explained in Section 3. Experimental setup is introduced in Section 4. Experimental results and discussion are introduced in Section 5. Section 6 introduces threats to validity. Finally, conclusions and future work are presented in Section 7.

2. Related Work

2.1. *Code Generation*

Program generation is to automatically generate code according to the requirements given by the user. In a specific programming language, algorithms can replace programmers to complete simple or specific code, such as sql statements[16][17]. Traditional program generation methods[18][19][20] require programmers to design logic specifications, so that the machine can automatically generate source code based on these logic specifications. However, these methods not only have higher requirements for programmers, but the generated code is often single in function and cannot complete complex tasks. To solve this problem, researchers use data-driven deep learning methods to generate code. Ling et al.[1] proposed a method based on a network of latent predictors to generate source code in Python or Java. This

method can achieve an average BLEU Score of 0.776 on data sets such as Django. To the best of our knowledge, it is the first program to generate source code in a general-purpose programming language. Yin and Neubig[9] proposed a syntactic neural model, which can transform NL descriptions into intermediate framework Actions to generate AST, the model uses AST to generate code finally. Later, they proposed the TRANX[15] model, which has better generalization and extended Python to other programming languages. Coarse-to-Fine model[10] also uses the intermediate architecture. Li Dong et al. generate a rough sketch through given input sentence, which masks low-level information (such as variable names and parameters). Then, Coarse-to-Fine model fills in the missing details by considering the natural language input and the sketch itself. Rabinovich et al.[2] use multiple decoders in code generation, where different types of nodes in the AST are generated by different decoders. Pengcheng Yin et al.[21] proposed STRUCTVAE model, which is a variational automatic coding model for semi-supervised semantic analysis. Hayati et al.[11] established a method called ReCode. The RECODE model can effectively use the nearest neighbors to generate code, and solve the problem of rare words by retrieving the most relevant samples successfully. TRANX-R2L[22] is a modified version of the TRANX model, which can optimize the selector through reinforcement learning and solve the problem of non-differentiable selection of the expansion order.

The above models all generate code through natural language description, and there is also a kind of program generation based on input and output examples. For example, deep learning[23][24] is used to simulate the execution trajectory of the program, thereby generating the statements and parameters required for the execution of the program.

2.2. *Code Retrieval*

Code retrieval is an important research problem in the field of software engineering, and its main task is to retrieve and reuse codes. Earlier, code retrieval focused on natural language descriptions, using similarities between natural languages to find the most relevant samples. Haiduc et al.[25] proposed the Refoqus model, which can automatically recommend code for a given query after training with samples. Hill et al.[26] uses the view of NL to search and maintain the source code, and the method integrates multiple feedback mechanisms into the search result view to solve the problem of irrelevance of the retrieved samples. Lu et al.[27] further processed NL information, they proposed a method to expand the query using synonyms generated by WordNet[28]. This method extracts key natural language phrases from NL, matches and ranks these phrases with samples. Experiments show that this method can quickly identify related program elements for invocation or quickly identify substitute words for query reconstruction. In order to solve the high time complexity of code retrieval or query language limitation, Keivanloo et al.[29] proposed a method of discovering working code examples, which can be adopted by

Internet-scale source code search engines. However, the model that only uses natural language terms for code search is not ideal because programming concepts do not always match their syntactic form in the vocabulary. Therefore, people consider generating intermediate architectures to match codes or include target codes in calculations. ANNE[30] is a technology that can discover the mapping between NL's syntax and programming language terms to find related codes. DeepCS[31] is a deep learning retrieval model that takes code into consideration. It embeds NL and code into a vector space of the same dimension through deep learning. Finally, DeepCS model retrieves related samples using the similarity of vectors. Iyer et al.[32] proposed CODE-NN model, which uses a long short-term memory (LSTM)[33] network and pays attention to generating a summary of the relevant code language. Chen et al.[34] proposed a neural framework BVAE, which uses the two-way mapping between source code and natural language to improve the retrieval model. BVAE aims to have two variational autoencoders (VAE)[35] to model bimodal data: C-VAE for source code and L-VAE for natural language.

3. Proposed Approach

In this section, we will introduces the process of the HRCODE in detail. The overview of the approach is shown in Fig. 1. HRCODE model can be divided into three parts. Firstly, we transform the NL data into a action representation which contains structural features. Then, the model embeds NL and action into fixed-dimensional vectors. We use the hybrid vector to retrieve the training data. Finally, the retrieved NL and code will be new input for generation model which we use PLBART[36] Pre-training model.

In the following, we introduce the process of HRCODE model in detail from the three aspects of MR structure, Retrieval Model, and Code generation.

3.1. *Problem Definition*

The goal of code generation is to generate code from natural language. Utterance is defined as u. We retrieval the max similarity score training sample x_i, y_i through structure and semantic retrieval, where $i \in s$. x_i represents the natural language of the sample, y_i is corresponding code. Therefore, the input sequence of the model becomes $u \oplus x_i \oplus y_i$, where $\oplus$ denote the sequence concatenation operation. More formally:

$$\exists i \in \{1, 2, 3, ...s\} : \arg\max_{x_i} Score\ (u, x_i) \tag{1}$$

$$code : Gen(u \oplus x_i \oplus y_i) \tag{2}$$

where Gen is code generation model, the s represents the number of training samples.

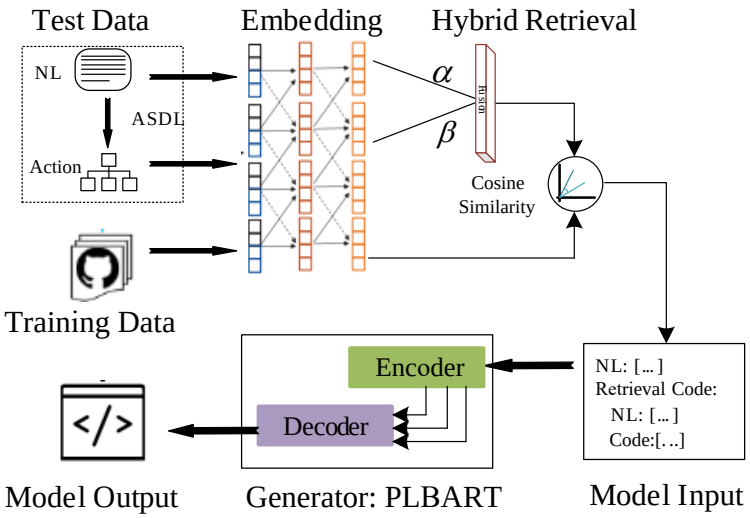

Fig. 1. The overview of HRCODE model

3.2. *Action Structure*

In order to obtain intermediate representations of NL, researchers transduce natural language into formal meaning representations (MRs). In HRCODE model, we use the Tranx model to transform NL into an Action set under Abstract Syntax Description Language(ASDL).

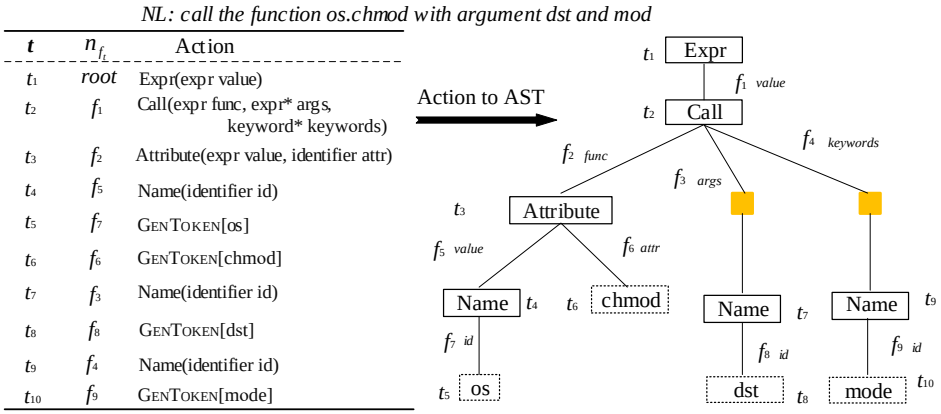

Fig. 2. An example of NL-Action-AST.

Tranx model is a transition system in essence, it parses each utterance into a for-

mal MR which represented as general-purpose programming languages. As shown in Figure 2, this transition system can map the NL utterance into an Abstract Syntax Tree(AST) using a sequence of tree-construction actions. In the action table, these tree-construction actions contain rich structural features. The generation process derives the AST from the beginning of a single root node, and traversals according to the depth-first order of AST. The model will select one of the following three types of actions to expand the frontier field n_{f_t} of the derivation:

APPLYCONSTR[c] actions apply a constructor c to the opening composite frontier field which has the same type as c, populating the opening node using the fields in c. If the frontier field has sequential cardinality, the action appends the constructor to the list of constructors held by the field.

REDUCE actions represent the completion of the optional (?) or multiple ($*$) cardinalities generation in APPLYCONSTR[c].

GENTOKEN[v] actions is used to construct the leaf nodes of AST. In Figure 2, the field f_8 has type identifier and the value is *chmod*.

The probability of an APPLYCONSTR[c] action with embedding a_c is:

$$p = (a_t = APPLYCONSTR[c]|a_{<t}, x) = softmax(a_c^t W \tilde{s_t}) \tag{3}$$

REDUCE is treated as a special APPLYCONSTR action. The $\tilde{s_t}$ is the attentional vector defined as in Luong et al[37].

For GENTOKEN actions, model employs a hybrid approach of generation and copying, allowing for out-of-vocabulary variable names and literals in NL to be directly copied to the derivation.

$$\begin{aligned} p =& (a_t = GENTOKEN[v]|a_{<t}, x) \\ &= p(gen|a_t, x)p(v|gen, a_t, x) + p(copy|a_t, x)p(v|copy, a_t, x) \end{aligned} \tag{4}$$

The probability of $p(gen|\cdot)$ and $p(copy|\cdot)$ are calculated by softmax($W\tilde{s_t}$). The probability of generating v from the vocabulary set.

Although, we can choose the generated AST as the structure retrieval part, the AST would lost important structural information[11]. For example, in figure 2, [Attribute(expr value, identifier attr)] action will only show *Attribute* node in AST. Therefore, we choose actions rather than AST nodes as the structural feature of the retrieval model.

3.3. *Retrieval Part*

In previous work[11], the model used the edit distance of NL to calculate the similarity between the training data and the test data. Then, using the similarity score retrieve relevant samples. There are two shortcomings: 1.The representation of the sample is rough. 2.Insufficient information for retrieval.

To obtain more effective data representation, we use FastText pre-trained model[a] as our embedding model. By vectorizing the tokens of the samples, the model can

[a]https://dl.fbaipublicfiles.com/fasttext/vectors-wiki/wiki.en.vec

obtain the correlation between the tokens better and improve the accuracy of retrieving related data.

For each utterance u, we transform the u into actions by method of Section 3.2. The NL description contains semantics and rare tokens information, and the actions have the structural characteristics of the data. Embedding these semantic and structural features into hybrid vectors for sample representation. More formally:

$$\begin{gathered} V = \alpha * embed(u) + \beta * embed(Actions) \\ \alpha + \beta = 1 \end{gathered} \tag{5}$$

where $embed()$ is embedding model, α and β are the hybrid weight. After obtaining the vector representation of each sample, we use the cosine similarity for the measurement, which is defined as:

$$\exists i \in \{1, 2, 3, ...s\} : \arg\max_{c_i} cos(c_i, d) = \frac{c_i^T d}{\|c_i\| \, \|d\|} \tag{6}$$

The hybrid vector d is test data, c_i is the hybrid vector of train data, s is the number of train samples. The weight of α and β are hyperparameters, we will discuss them in the experimental part.

3.4. *Code generation*

In generation part, we use PLBART model to generate the code. The PLBART has the same architecture as $BART_{base}$ which uses the sequence-to-sequence Transformer with 6 layers of encoder and 6 layers of decoder. The only difference is that BLBART model includes an extra layer of normalization on top of the encoder and decoder, because this process can stabilize training with FP16 precision[38].

The PLBART is a pre-trained on lots of unlabeled instances, each instance is corrupted using the noise function f and PLBART predict the original instance x from $f(x)$. In this task, formally, PLBART is trained to maximize $\pounds_\theta$:

$$\pounds_\theta = \sum_{i=1}^{m} logP(x_i | f(x_i); \theta) \tag{7}$$

where m is the number of pre-trained dataset. The noise function f can make the model better learn the syntax and semantics features of the language.

The input to the PLBART model is a text sequence, we spliced the NL description and the retrieved sample NL description and code as the sequence input of the model. In the input sequence, we add the special identifiers $< nlsep >$ and $< codeseq >$ to distinguish them. Finally, the model input is [NL $< nlsep >$ NL $< codeseq >$ code].

4. Experimental Setup

This article aims to study the following five research questions.

- **RQ1**: Can our model outperform the state-of-the-art models in code generation of the django dataset?
- **RQ2**: Compared with the existing retrieval and non-retrieval models, how effective is the generation of HRCODE?
- **RQ3**: Which part has more influence on the effect of the retrieval model?
- **RQ4**: What is the effectiveness of parameters on the model?
- **RQ5**: Does the result different from the ground truth must be the wrong code?

4.1. *Data Set*

The DJANGO dataset is a collection of lines of code from the Django Web framework in the real world. DJANGO contains a wide range of real-world use cases, such as string operations, IO operations, and exception handling. The natural language description examples of this dataset are diverse, and each line of code is manually annotated with natural language description. The Table 1 shows that the data contains 18805 examples of data in total, 16000 examples are used for training, 1000 examples are used for verification, and the rest are used for testing. The upper half of Table 2 is the grammatical data statistics with w/o unary closure, and the other part refers to statistics of w/ unary closure.

Table 1. Dataset statistics for DJANGO

Dataset	DJANGO
Train	16000
Development	1000
Test	1805
Avg. tokens in NL	14.3
Avg. characters in code	41.1
Avg. size of AST	17.2

Table 2. Statistics for associated grammars

Statistics of Grammar	DJANGO
# productions	222
# node types	96
terminal vocabulary size	6733
# productions	237
# node types	92
Avg. # actions per example	16.4

4.2. *Baseline Models*

We compare our approach with six deep learning based code generation models from previous work. The six baselines can be divided into two types: retrieval model and non-retrieval model.

YN17(2017): This work is a data-driven syntax-based neural network model. It can capture the strong underlying syntax structure of specific programming language, thereby transforming natural language into an abstract syntax tree. Finally,

the model converts the AST into target code. Yin et al. were the first to use Django data in code generation tasks.

STRUCTVAE(2018): This work is a variational auto-encoding model for semi-supervised code generation. It can learn from limited amount of parallel data and readily available unlabeled natural language utterances. The STRUCTVAE model provides a method for training VAE with structured latent variables, and contributes to the entire generative model research.

RECODE(2018): RECODE model is a method based on n-gram action subtree retrieval. it can explicitly reference most relevant code examples in the existing data to enhance the model generate code. The RECODE model increases the probability that the retrieved n-gram action subtree will predict the action in the code.

TRANX(2018): Inspired by YN17 model, Yin et al. further proposed the TRANX model. This work is a transformation-based neural semantic parser that can map natural language utterances into formal meaning representations. Compared with the previous work, the TRANX model has higher accuracy and greater scalability.

COARSE2FINE(2018): In this work, Li Dong et al. proposed a structure-aware neural architecture Coarse2Fine model. The model transforms the input natural sentences into rough sketches, ignoring fine-grained information temporarily. Then, the model fills in the missing details by considering natural language input and the sketch itself. The Coarse2Fine model can achieve great performance in the structural models.

TRANX-R2L(2021): The model mentions that traversal generation is not suitable for processing all multi-branch nodes. So they proposed new method to equip the Seq2Tree model with a context-based branch selector, which can dynamically determine the multi-branch nodes. In particular, the TRANX-R2L model solves the problem that the choice of expansion order is not differentiable through reinforcement learning to optimize the selector.

4.3. *Evaluation Mertics*

We evaluate the quality of code generation using two automatic metrics: exact-match accuracy and BLEU-4.

For exact-match accuracy, we use the evaluation method of the previous work[9][15]. We convert the predicted code and ground truth into AST for comparison. When the two data exactly match, we think the the prediction correct, and vice versa. It is calculated as:

$$Acc = \frac{1}{n}\sum_{i=1}^{n} acc_i \tag{8}$$

For BLEU-4 metric, we follow[9][11], and use token-level BLEU-4[39] with as a secondary metric, defined as the averaged BLEU scores over all test data. It is

calculated as

$$BP = \begin{cases} 1 & if \quad c > r \\ exp(1 - \frac{r}{c}) & if \quad c \leq r \end{cases} \tag{9}$$

$$BLEU = BP * exp(\sum_{n=1}^{4} w_n log p_n) \tag{10}$$

where p_n is the co-occurrence rate of length n subsequences between candidate and reference, c is the length of the candidate, r is the effective reference sentence length, and BP refers to brevity penalty.

4.4. *Parameter Setting*

We use the official codebase of PLBART (Ahmad et al., 2021). The model learning rate is 5×10^{-5}, the max epoch is 10 and the patience set 3. We tune the batch size in [4, 8, 16]. The range of hyperparameters α and β is [0,1], and the sum of α and β is equal to 1, which we will discuss in the experimental results.

5. Experimental Results and Discussion

5.1. *Overall Performance*

For **RQ1**, Table 3 illustrates the overall performance of our combined model compared to baselines. We can see from the table that our model is obviously outperforms all baselines on accuracy and BLEU-4. Comparing to the non-Retrieval based baselines, that is, YN17, STRUCTVAE, TRANX, COARSE2FINE and TRANX-R2L, our model can achieve at least 4.7% performance accuracy improvement. It indicates that retrieval model can effectively improve the accuracy of code generation. Besides, we also compared with the retrieval model RECODE, HRCODE model can improve 10.3% on accuracy metric. This shows that the retrieval model that combines semantic and structural features is more effective. We will compare the results generated by the HRCODE model and these two types of models in next section.

For the second metric BLEU-4, our model also achieved outstanding results. However, in code generation tasks, the accuracy indicator can clearly show the difference in the generated code between models. Therefore, we did not reproduce the results of those models that did not give BLEU-4 values in their work.

5.2. *Generation Analysis*

For **RQ2**, we select three test samples to analyze the generated results. These case studies are from RECODE work, and the three examples are the 1700th, 876th, and 101st in the test data. We chose the retrieval model (RECODE) and non-retrieval

Table 3. Overall Performance

Model	ACC	BLEU-4
YN17	71.6	84.5
STRUCTVAE	73.7	-
RECODE	72.8	84.7
TRANX	73.7	85.9
COARSE2FINE	77.4	-
TRANX-R2L	**78.4**	**-**
HRCODE	**83.10**	**91.3**

model (YN17, TRANX) to compare the results generated by the HRCODE model. It can be seen from Table 4 that the code generation result of the YN17 model is the worst, and the generation results of the three samples have deviations from the ground truth. For the TRANX and RECODE models, the code of the first example was successfully generated, but the other two have obvious errors. Overall, the HRCODE model has the best output, and the model successfully generated the first two examples. For the third example, comparing with other models, we successfully predicted the built-in function *str()*. As for the generation error of the function name *SafeString*, we think that is a problem with the input data. Since the input of Example 3 is "return an instance of SafeText", it is obviously impossible to generate the value of *SafeString*, and the *SafeText* function name is successfully predicted for all four models. Therefore, the HRCODE model can be generated correctly for the above three examples, which is obviously better than the existing retrieval and non-retrieval models.

In order to facilitate the observation of the effect of HRCODE generation, we provide the results of all test data generation[b].

5.3. *Ablation Study*

For **RQ3**, we discuss the influence of NL and Action on the results of the retrieval model. We conduct an ablation study to check the effectiveness of our proposed method.

Full-model is retrieval model that takes into account the feature of NL and action. The model weights α and β, we choose $\alpha = 0.8$, $\beta = 0.2$, the discussion of the parameters will be introduced in the next section.

NL-model refers to retrieval process that only considers NL, and the model does not contain structural features.

[b]www

Table 4. Django examples on correct code and predicted

Example 1 (1700)	**Model**
"if offset is lesser than integer 0, sign is set to '-', otherwise sign is '+' ."	**Input**
sign = offset < 0 or '-'	**YN17**
sign = '-' if offset < 0 else '+'	**TRANX**
sign = '-' if offset < 0 else '+'	**RECODE**
sign = '-' if offset < 0 else '+'	**HRCODE**
sign = '-' if offset < 0 else '+'	**Gold**
Example 2 (876)	**Model**
"evaluate the function timesince with d, now and reversed set to boolean true as arguments, return the result."	**Input**
return reversed(d, reversed=now)	**YN17**
return timesince(d, now, reversed= reversed)	**TRANX**
return timesince(d, now, reversed=now)	**RECODE**
return timesince(d, now, reversed=True)	**HRCODE**
return timesince(d, now, reversed=True)	**Gold**
Example 3 (101)	**Model**
"return an instance of SafeText , created with an argument s converted into a string ."	**Input**
return SafeText(bool(s))	**YN17**
return SafeText(s)	**TRANX**
return SafeText(s)	**RECODE**
return SafeText(str(s))	**HRCODE**
return SafeString(str(s))	**Gold**

Act-model refers to retrieval process that only considers actions, and the model does not contain semantic features.

It can be seen from Figure 3 and 4 that the experimental result of full-model achieve the best score at epoch 3, while the Act-model that only uses action information has the worst result. We think there are two reasons: 1. Semantic features are more effective than structural features. The retrieval data takes the form of sequence as input, and the words in the semantic information are more recognizable in the embedded vector space. Therefore, NL-model can retrieve more effective

samples from the training data. 2. Action structure information is limited by the TRANX model. Since actions are generated by TRANX, for the wrong samples generated by TRANX, inaccurate action information will cause the wrong samples to be retrieved. Therefore, the accuracy of Act-model is limited by the TRANX model.

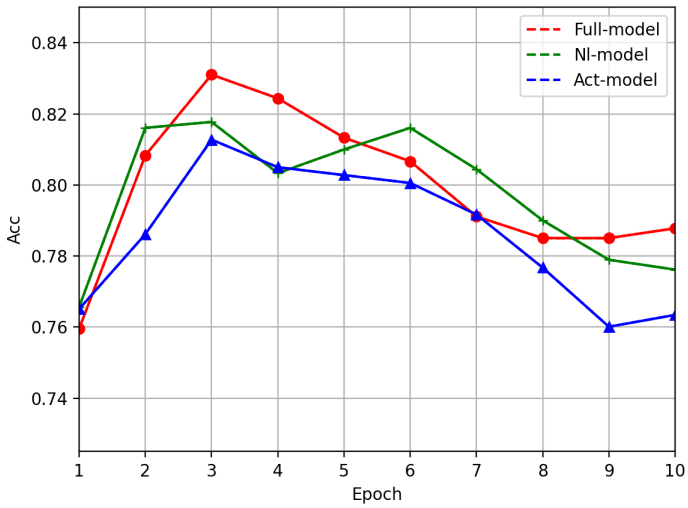

Fig. 3. Acc score for three models

Finally, we can know that the retrieval model with mixed semantics and structural features can effectively improve the accuracy of code generation.

5.4. *Parameter Influence*

For **RQ4**, we discuss the effects of hyperparameters α and β on the model, then we discuss the results of experiments with different batch-sizes.

HRCODE model combines structural and semantic features for retrieval, and we use weights α and β to control the proportion of feature fusion. As shown in Table 5, when the weights of action and NL are 0.8 and 0.2, the accuracy and BLEU-4 of the model both reach the maximum.

Then, we discuss the generation accuracy of retrieval models for different batch-sizes, as shown in Figure 5 and 6. Under the condition of hyperparameter α=0.8 and β=0.2, our model chooses three different batch-sizes of 4, 8 and 16. It can be seen from the figure that the overall effect of the model is better when batch-size is 8 at epoch 3.

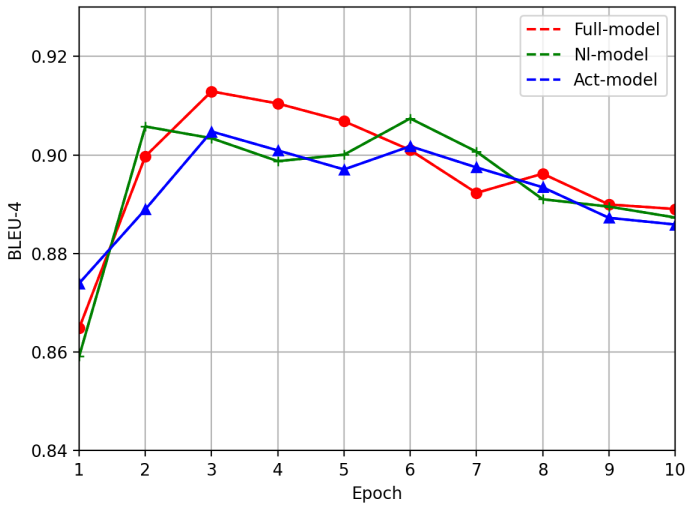

Fig. 4. BLEU-4 score for three models

Table 5. The effects of hyperparameters α and β on the model

a	b	E-1	E-2	E-3	E-4	E-5	E-6	E-7	E-8	E-9	E-10
0	1.0	76.51	78.61	**81.27**	80.50	80.28	80.06	79.17	77.67	76.01	76.34
0.1	0.9	75.51	79.83	**82.05**	80.89	81.05	80.50	80.22	79.22	78.23	77.12
0.2	0.8	77.67	81.00	81.27	**82.66**	81.61	81.22	79.72	78.89	77.34	75.78
0.3	0.7	77.29	80.61	81.33	**82.11**	80.00	80.22	79.78	79.50	79.22	77.79
0.4	0.6	77.67	81.50	**82.88**	82.60	80.11	80.22	79.78	78.84	76.79	76.57
0.5	0.5	77.67	81.50	81.00	**81.61**	80.31	81.61	81.33	79.61	78.06	78.12
0.6	0.4	77.17	81.11	81.50	**82.71**	81.16	81.05	80.00	78.73	77.51	78.12
0.7	0.3	77.73	**82.22**	82.10	81.88	81.50	81.39	80.72	80.11	78.95	76.95
0.8	0.2	75.96	80.83	**83.10**	82.44	81.33	80.66	79.11	78.50	78.50	78.78
0.9	0.1	77.62	81.11	**82.44**	81.50	81.55	81.83	81.05	79.94	78.56	77.62
1.0	0	76.57	80.61	**81.78**	80.33	81.00	81.61	80.44	79.00	77.90	77.62

5.5. *Error Analysis*

For **RQ5**, we discuss whether the generated code is different from the ground truth should be consider as an error. The second discussion is about the types of codes

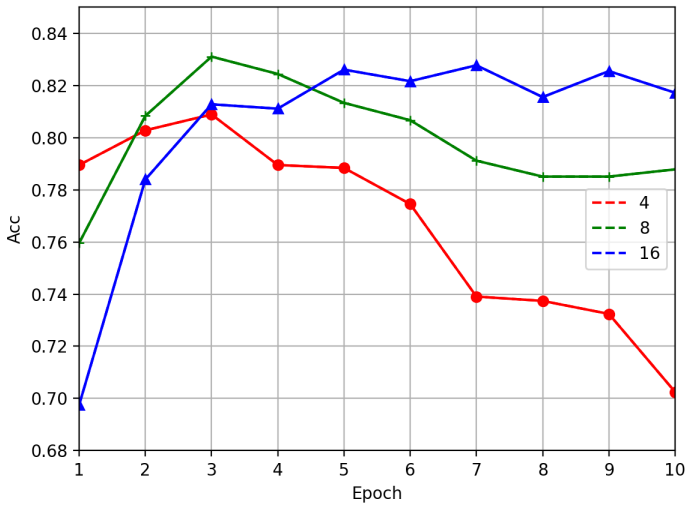

Fig. 5. Acc score for different batch size models

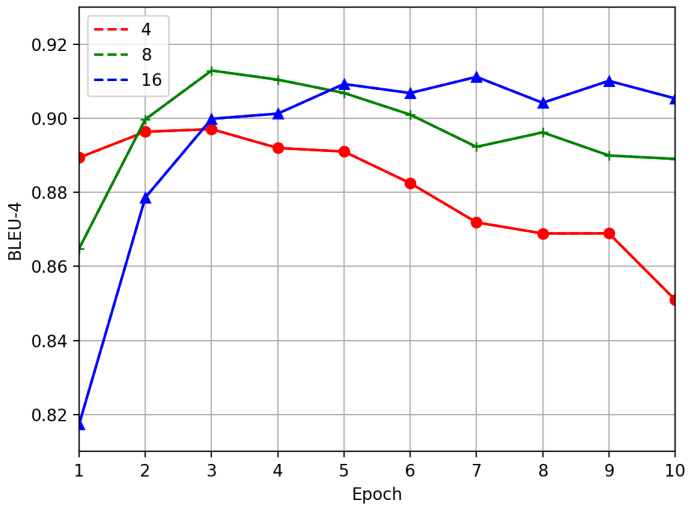

Fig. 6. BELU-4 score for different batch size models

that these error codes can be divided into.

Since we used exact-match accuracy as metric, if generation code has a little

difference with ground truth in semantic information during the evaluation process, it will be recognized as an error code. But the function of generating code and reference code is the same, such as Example 4 and Example 5. Therefore, the exact-match accuracy metric does not fully reflect the results of the model. In order to better analyze the information of the error codes generated by HRCODE model, we have classified them, as shown in the Table 6.

Table 6. Error Types

Type	**Same Function**	**Minor Error**	**Structure correct**	**Complete Irrelevant**	**Totally**
Number	12	124	111	58	305

Same Function means that the generated code is different from the semantic information of the target code, but the result of the code output is the same.

Minor Error represents that there are no more than two semantic information errors between the generated code and the target code, such as the value error of the list in Example 6.

Structure correct means that there are many semantic errors between the generated code and the target code, but the code structure consistency with ground truth.

Complete Irrelevant indicates that the generated code has a big difference with target code. Such as Example 10.

It can be seen from Table 6 that the total number of model code generation errors is 305, and the number of completely wrong samples is 58. In other words, the code that can generate the correct architecture or even the same function accounts for 80.98%.

In Table 7, for examples 6, 7 and 9, we can see that the model is more sensitive to the words in the NL description and can effectively use Local words. However, code generation is a context-sensitive task, so the HRCODE model cannot generate the "o" in example 6 and the "KeyError" in example 9, which appeared before.

6. Threats to Validity

We discuss the threats to the construct, internal, and external validity of our study in this section.

Construct validity. The experiment uses the django data set, which is collected by Kamei et al. from the software library. However, the total sample data is less, and the NL description in the data set may also be problematic. Therefore, the quality of the data set generated by the code has always been a challenge in the field of software defect prediction. Another threat to model performance is

Table 7. Error Examples

Same Function	**Example 4 (7)**	
	pos and last are integer 0.	**Input**
	pos , last = 0	**HRCODE**
	pos = last = 0	**Gold**
	Example 5 (18)	
	"for every i in range of integers from 0 to length of result, not included,"	**Input**
	for i in range (0 , len (result)) :	**HRCODE**
	for i in range (len (result)) :	**Gold**
Minor Error	**Example 6 (565)**	
	"append value under the 0 key of the opt_dict dictionary to code_list."	**Input**
	code_list . append (opt_dict [0])	**HRCODE**
	code_list . append (opt_dict [o])	**Gold**
	Example 7 (875)	
	"define the function timeuntil with 2 arguments, d and now defaulting to none."	**Input**
	def timeuntil (d , now = none) :	**HRCODE**
	def timeuntil (d , now = None) :	**Gold**
Structure Correct	**Example 8 (299)**	
	"return next element of the iterable it."	**Input**
	return next (self)	**HRCODE**
	return it . next ( )	**Gold**
	Example 9 (300)	
	"substitute klass.__dict__ for next."	**Input**
	next = klass . __dict__	**HRCODE**
	next = advance_iterator	**Gold**
Complete Error	**Example 10 (137)**	
	"instantiate class X, get its length."	**Input**
	super (X , self) . length	**HRCODE**
	len (X ( ))	**Gold**
	Example 11 (372)	
	"append i_args to args, append new_args with the result."	**Input**
	args . append (new_args)	**HRCODE**
	new_args . append (args [:] + i_args)	**Gold**

whether the evaluation indicators used in the experiment are suitable for real software development scenarios. We use exact-match accuracy and BLEU-4 to evaluate the performance of predictive models, which have been widely used in previous code generation studies[9]. However, the matching accuracy may ignore the same function code, and it does not fully reflect the performance of the model.

Internal validity. The threat to internal validity is mainly the possible errors in the experimental code. In order to avoid errors in the code, we carefully refer to the code of Ahmad et al.[36] and Yin et al.[15]. In addition, we have adopted mature third-party libraries to improve the reliability of the code. Finally, we carefully checked all the experimental codes.

External validity. The threat to external validity is mainly regarded as a generalization of experimental results. The experiment only uses a data set generated by the code of an open source project, and the proposed method has high predictive performance on this data set. However, whether our method can be applied to other projects remains to be further studied. Therefore, more data sets are needed to validate the model.

7. Conclusion and Future Work

In this paper, we propose a code generation architecture HRCODE that integrates structural and semantic feature retrieval. The model converts NL descriptions into intermediate framework actions to obtain structural features. Then, we find the most relevant samples through a retrieval method that combines semantic and structural features to construct a new model input. The experimental results show that the retrieval model of multiple feature fusion is more effective. And the HRCODE model is at least 5 points higher in Acc than the current best method.

Future work will be divided into the following four parts:

(1) Consider more data sets. Since the data set with the same baselines is only Django and the experimental workload of this work is large, we only consider one data set. In future work, we will consider new data sets, such as wikisql.

(2) Choose more effective indicators. In section 6, the code generated by the model has the same function as the ground truth, so the matching accuracy does not reflect the overall effect of the model. Therefore, we consider that the execution accuracy will be used in future work to verify the effect of the model.

(3) Obtain more effective structural features. Since we use the tranx model to generate actions in the structural part, the structural features are limited to the previous model. So we will consider new methods of structural feature extraction in the next work.

(4) Experiments with different sample retrieval numbers. In this paper, we only select the most relevant samples without considering more search samples. This is because in the djang data set, each NL description corresponds to each line of code, so it is sufficient to retrieve the code information. In future work, we will consider the impact of more data sets on the structure of the experiment in multiple retrieval

samples.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (No. 61772200), Shanghai Natural Science Foundation (No.21ZR1416300).

References

[1] W. Ling, P. Blunsom, E. Grefenstette, K. M. Hermann, T. Kociský, F. Wang and A. W. Senior, Latent predictor networks for code generation, 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, 2016).

[2] M. Rabinovich, M. Stern and D. Klein, Abstract syntax networks for code generation and semantic parsing, in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, eds. R. Barzilay and M. Kan (Association for Computational Linguistics, 2017), pp. 1139–1149.

[3] P. Yin and G. Neubig, Reranking for neural semantic parsing, in *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, eds. A. Korhonen, D. R. Traum and L. Màrquez (Association for Computational Linguistics, 2019), pp. 4553–4559.

[4] F. A. Kraemer, Engineering android applications based on UML activities, in *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings*, eds. J. Whittle, T. Clark and T. Kühne *Lecture Notes in Computer Science* **6981**, (Springer, 2011), pp. 183–197.

[5] C. Quirk, R. J. Mooney and M. Galley, Language to code: Learning semantic parsers for if-this-then-that recipes, in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*, (The Association for Computer Linguistics, 2015), pp. 878–888.

[6] G. Sunyé, A. L. Guennec and J. Jézéquel, Using UML action semantics for model execution and transformation, *Inf. Syst.* **27**(6) (2002) 445–457.

[7] R. Jia and P. Liang, Data recombination for neural semantic parsing, 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, 2016).

[8] N. Locascio, K. Narasimhan, E. DeLeon, N. Kushman and R. Barzilay, Neural generation of regular expressions from natural language with minimal domain knowledge, in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016*, eds. J. Su, X. Carreras and K. Duh (The Association for Computational Linguistics, 2016), pp. 1918–1923.

[9] P. Yin and G. Neubig, A syntactic neural model for general-purpose code generation, in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, eds. R. Barzilay and M. Kan (Association for Computational Linguistics, 2017), pp. 440–450.

[10] L. Dong and M. Lapata, Coarse-to-fine decoding for neural semantic parsing, in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, eds. I. Gurevych and Y. Miyao (Association for Computational Linguistics, 2018), pp. 731–742.

[11] S. A. Hayati, R. Olivier, P. Avvaru, P. Yin, A. Tomasic and G. Neubig, Retrieval-based neural code generation, in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, eds. E. Riloff, D. Chiang, J. Hockenmaier and J. Tsujii (Association for Computational Linguistics, 2018), pp. 925–930.

[12] J. Zhang, M. Utiyama, E. Sumita, G. Neubig and S. Nakamura, Guiding neural machine translation with retrieved translation pieces, in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 1 (Long Papers)*, eds. M. A. Walker, H. Ji and A. Stent (Association for Computational Linguistics, 2018), pp. 1325–1335.

[13] J. Gu, Y. Wang, K. Cho and V. O. K. Li, Search engine guided neural machine translation, in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, eds. S. A. McIlraith and K. Q. Weinberger (AAAI Press, 2018), pp. 5133–5140.

[14] X. Li, J. Zhang and C. Zong, One sentence one model for neural machine translation, in *Proceedings of the Eleventh International Conference on Language Resources and Evaluation, LREC 2018, Miyazaki, Japan, May 7-12, 2018*, eds. N. Calzolari, K. Choukri, C. Cieri, T. Declerck, S. Goggi, K. Hasida, H. Isahara, B. Maegaard,

J. Mariani, H. Mazo, A. Moreno, J. Odijk, S. Piperidis and T. Tokunaga (European Language Resources Association (ELRA), 2018).

[15] P. Yin and G. Neubig, TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation, in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, EMNLP 2018: System Demonstrations, Brussels, Belgium, October 31 - November 4, 2018*, eds. E. Blanco and W. Lu (Association for Computational Linguistics, 2018), pp. 7–12.

[16] V. Zhong, C. Xiong and R. Socher, Seq2sql: Generating structured queries from natural language using reinforcement learning, *CoRR* **abs/1709.00103** (2017).

[17] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang and D. R. Radev, Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task, in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, eds. E. Riloff, D. Chiang, J. Hockenmaier and J. Tsujii (Association for Computational Linguistics, 2018), pp. 3911–3921.

[18] R. J. Waldinger and R. C. T. Lee, PROW: A step toward automatic program writing, in *Proceedings of the 1st International Joint Conference on Artificial Intelligence, Washington, DC, USA, May 7-9, 1969*, eds. D. E. Walker and L. M. Norton (William Kaufmann, 1969), pp. 241–252.

[19] C. C. Green, Application of theorem proving to problem solving, in *Proceedings of the 1st International Joint Conference on Artificial Intelligence, Washington, DC, USA, May 7-9, 1969*, eds. D. E. Walker and L. M. Norton (William Kaufmann, 1969), pp. 219–240.

[20] Z. Manna and R. J. Waldinger, Toward automatic program synthesis, *Commun. ACM* **14**(3) (1971) 151–165.

[21] P. Yin, C. Zhou, J. He and G. Neubig, Structvae: Tree-structured latent variable models for semi-supervised semantic parsing, in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, eds. I. Gurevych and Y. Miyao (Association for Computational Linguistics, 2018), pp. 754–765.

[22] H. Jiang, C. Zhou, F. Meng, B. Zhang, J. Zhou, D. Huang, Q. Wu and J. Su, Exploring dynamic selection of branch expansion orders for code generation, in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, eds. C. Zong, F. Xia, W. Li and R. Navigli (Association for Computational Linguistics, 2021), pp. 5076–5085.

[23] S. E. Reed and N. de Freitas, Neural programmer-interpreters, in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May*

2-4, 2016, Conference Track Proceedings, eds. Y. Bengio and Y. LeCun (International Conference on Learning Representations, 2016).

[24] J. Cai, R. Shin and D. Song, Making neural programming architectures generalize via recursion, in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, (OpenReview.net, 2017).

[25] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. D. Lucia and T. Menzies, Automatic query reformulations for text retrieval in software engineering, in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, eds. D. Notkin, B. H. C. Cheng and K. Pohl (IEEE Computer Society, 2013), pp. 842–851.

[26] E. Hill, M. Roldan-Vega, J. A. Fails and G. Mallet, Nl-based query refinement and contextualized code search results: A user study, in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014*, eds. S. Demeyer, D. W. Binkley and F. Ricca (IEEE Computer Society, 2014), pp. 34–43.

[27] M. Lu, X. Sun, S. Wang, D. Lo and Y. Duan, Query expansion via wordnet for effective code search, in *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, eds. Y. Guéhéneuc, B. Adams and A. Serebrenik (IEEE Computer Society, 2015), pp. 545–549.

[28] G. A. Miller, Wordnet: A lexical database for english, *Commun. ACM* **38**(11) (1995) 39–41.

[29] I. Keivanloo, J. Rilling and Y. Zou, Spotting working code examples, in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, eds. P. Jalote, L. C. Briand and A. van der Hoek (ACM, 2014), pp. 664–675.

[30] V. Vinayakarao, A. Sarma, R. Purandare, S. Jain and S. Jain, ANNE: improving source code search using entity retrieval approach, in *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining, WSDM 2017, Cambridge, United Kingdom, February 6-10, 2017*, eds. M. de Rijke, M. Shokouhi, A. Tomkins and M. Zhang (ACM, 2017), pp. 211–220.

[31] X. Gu, H. Zhang and S. Kim, Deep code search, in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, eds. M. Chaudron, I. Crnkovic, M. Chechik and M. Harman (ACM, 2018), pp. 933–944.

[32] S. Iyer, I. Konstas, A. Cheung and L. Zettlemoyer, 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, 2016).

[33] P. Malhotra, L. Vig, G. Shroff and P. Agarwal, Long short term memory networks for anomaly detection in time series, in *23rd European Symposium on Artificial Neural Networks, ESANN 2015, Bruges, Belgium, April 22-24, 2015*, (European Symposium on Artificial Neural Networks, 2015).

[34] Q. Chen and M. Zhou, A neural framework for retrieval and summarization of source code, in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, eds. M. Huchard, C. Kästner and G. Fraser (ACM, 2018), pp. 826–831.

[35] S. R. Bowman, L. Vilnis, O. Vinyals, A. M. Dai, R. Józefowicz and S. Bengio, Generating sentences from a continuous space, in *Proceedings of the 20th SIGNLL Conference on Computational Natural Language Learning, CoNLL 2016, Berlin, Germany, August 11-12, 2016*, eds. Y. Goldberg and S. Riezler (ACL, 2016), pp. 10–21.

[36] W. U. Ahmad, S. Chakraborty, B. Ray and K. Chang, Unified pre-training for program understanding and generation, in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2021, Online, June 6-11, 2021*, eds. K. Toutanova, A. Rumshisky, L. Zettlemoyer, D. Hakkani-Tür, I. Beltagy, S. Bethard, R. Cotterell, T. Chakraborty and Y. Zhou (Association for Computational Linguistics, 2021), pp. 2655–2668.

[37] T. Luong, H. Pham and C. D. Manning, Effective approaches to attention-based neural machine translation, in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015*, eds. L. Màrquez, C. Callison-Burch, J. Su, D. Pighin and Y. Marton (The Association for Computational Linguistics, 2015), pp. 1412–1421.

[38] Y. Liu, J. Gu, N. Goyal, X. Li, S. Edunov, M. Ghazvininejad, M. Lewis and L. Zettlemoyer, Multilingual denoising pre-training for neural machine translation, *Trans. Assoc. Comput. Linguistics* **8** (2020) 726–742.

[39] K. Papineni, S. Roukos, T. Ward and W. Zhu, Bleu: a method for automatic evaluation of machine translation, in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*, (ACL, 2002), pp. 311–318.