Multi-level Semantic Representation Model for code search

Donzhen Wen
sewen@mail.dlut.edu.cn
Dalian University of Technology
Dalian, China

Liang Yang
liang@dlut.edu.cn
Dalian University of Technology
Dalian, China

Yingying Zhang
zhangyingying@mail.dlut.edu.cn
Dalian University of Technology
Dalian, China

Yuan Lin
zhlin@dlut.edu.cn
Dalian University of Technology
Dalian, China

Kan Xu
xukan@dlut.edu.cn
Dalian University of Technology
Dalian, China

Hongfei Lin*
hflin@dlut.edu.cn
Dalian University of Technology
Dalian, China

ABSTRACT

With the huge amount of open-source software is publicly available today, code search has become more and more important to software development. Matching natural language(NL) between programming language(PL) cross the semantic gap is the key to the code search problem. In this paper, we introduce a word-based Multi-level Semantic Representation (MSR) model from the perspective of text representation to maximum semantic matching. Then we perform a series of experiments to find the significance of different parts in code snippets when modeling semantic relevance between natural language and programming language. The conclusion can be used to support further study on semantic matching modeling between PL and NL like the neural matching model.

KEYWORDS

text representation, source code search, software engineering

1 INTRODUCTION

The developer would reuse a huge amount of well designed and fully tested code snippets when developing a new software project. However, there are two main difficulties for the developer to find proper code snippets. First, it is hard to get access to these code snippets for those are distributed on different platforms, e.g., Github, personal blogs, online communities, and so on. Second, the developer's needs are always expressed in natural language form. How to model the relevance between natural language and programming language is our main challenge.

As Figure 1 shows, a code snippet contains a different domain for code search. Some parts like method names and code comments give more natural language contributions than code body, whereas source code body gives runtime functional information in instruction form. How the different parts of the code snippet contribute to the retrieval task is our main focus.

In this paper, we mainly focus on how natural language words contribute to the search task. The main contributions of this paper are as follows:

- We propose a word-based Multi-level Semantic Representation (MSR) model to jointly model natural language and programming language.
- Based on the MSR model, we combine learning to rank model to find the contribution of different parts of code snippets.

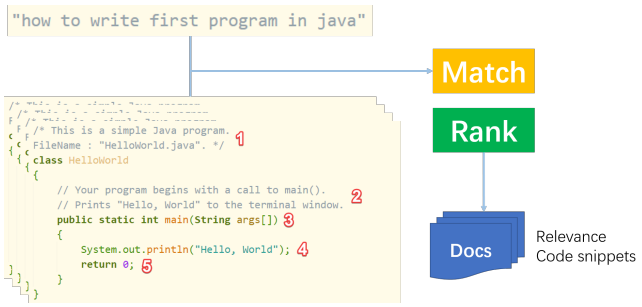

Figure 1: Code search framework. While sub-figure on the left side has shown different parts for a code snippet.

- By leveraging the sequential encoding ability of the neural model, we verify the natural language words' contribution to the code search task.

2 RELATED WORK

A code search task can be seen as a text-based information retrieval task. Early in 1993 Chang et al.[4] proposed a keyword-based code search system SMART, which enhances the keyword search effect by processing source code identifiers and extracting code element information. They gave developers the ability to access source code by matching full words in a software project. While other tools such as Debian code search, Google code search tool, StackOverFlow and Github enhance the code search system by adding regex technique support.

On the basis of the full-match based text match, the semantic-based text match has been proposed for a better understanding of the developer's intent. Lv et al.[16] proposed a code search technique that can recognize potential APIs that a user query refers to. Bajracharya et al.[3] proposed the Structural Semantic Indexing model to construct the connection between words and source code entities in natural language. Steven [18] built a semantic search engine by filtering out irrelevant code snippets according to description and test, their search engine can provide users with more accurate results. Lucia et al.[1]use the vector space model (VSM) to

https://codesearch.debian.net
https://github.com/google/codesearch
https://stackoverflow.com/
https://github.com/

trace the source code document basing on the source code identifier processing. Clarke et al.[7] proposed a text-based code search framework, which improved the source code pre-processing and source code storage management, divided the source code into multiple domains, and combined the text retrieval method to retrieve the code.

Marcus et al.[17] proposed a concept located in the source code file combined with latent semantic analysis (LSI) technique. Arwan et al.[2] introduce the topic model to enhance the effect of the code search system by applying the latent Dirichlet distribution (LDA) model into the code search task. Jiang et al.[13] combine word-based semantic feature extraction with learn to rank algorithm which inspires us combining the different levels of semantic feature for a better understanding the relationship between natural language and programming language.

As the neural model has achieved huge success in natural language processing, some techniques were introduced to represent programming language. Jiang et al.[15] proposed the Word2API model, which uses word vectors to represent code text. Hongyu Li et al.[19] researched neural code search. They train the word vector to perform a document level representation of the code segment and calculate the vector cosine similarity between code snippets and natural language queries. Gu et al.[11] proposed a new code search framework by encoding token sequences to get high-dimensional representations of the code snippet. Chen et al.[5] introduce variational Auto Encoders(VAEs) to code search and code summarization tasks. The bert model proposed in 2018 by Devlin et al.[9] completely change the paradigm of word embedding and natural language workflow. Then Kanade et al.[14] present the first attempt at pre-training a BERT contextual embedding of source code. Their results show that the fine-tuned models outperform the baseline LSTM models supported by Word2Vec embeddings, and Transformers trained from scratch. Basing on Clark et el.[6] Zhang et al.[10] propose a new variant of Bert-like source code pre-train model.

For better understanding the role which natural language words play in the source code search task, we propose three research questions for semantic modeling research:

- **RQ1:** How to model the semantic correlation between the programming language and natural language?
- **RQ2:** How different parts of code snippet contribute to code search task?
- **RQ3:** The advantage and disadvantage of word-based semantic modeling.

3 METHODOLOGY

3.1 Multi-level Semantic Representation

Our main framework for the code search task is shown in Figure 2. The source code fragment and the user query text are first re-encoded by the text cleaning step. The code text and user query after cleaning are processed into the MSR model by code element extraction and text normalization, and the text features, implicit semantic features, deep semantic features, and other related features are extracted. Finally a learning-to-rank model can give code snippet sorting by their semantic relevance based on our representation model.

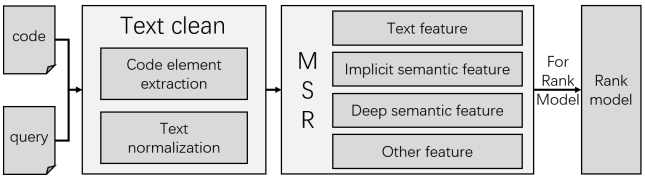

Figure 2: Multi-level semantic feature construction process.

A specific definition of multilevel semantic features can be seen in Table 1. We classify the semantic representation feature into three levels: text feature, Latent semantic feature, and deep semantic feature, and they are named as T-series, S-series, and D-series. A more detailed description of the multilevel semantic relevance feature can be seen in Table 2.

T1 mainly describes the editing distance between the query and the code text. The code text and the query text are expressed in the form of Unigram, and the editing distance of the two is calculated as the interactive feature in combination with the text sequence represented by the Unigram.

T2 analyzes the co-occurrence index between the texts. The code text and query text are represented as a collection of words by combining 1,2,3-grams and BOW models. Then, a co-occurrence of words between two ends of texts is calculated by combining the Jaccard index and the Dice index. In addition, the Jaccard index can be modified by introducing the word weight by the TFIDF model, and the weighted Jaccard index information is calculated.

T3 mainly calculates the vector distance between the texts. The vectorization method selects the 1,2,3-gram and TFIDF method to convert the code text and query text into vectors. The vector interaction part uses the cosine distance, the Euclidean distance, the Chebyshev distance and the Manhattan distance to calculate the distance feature between the texts.

S1 performs implicit semantic analysis (LSA) based on text vectorization and combines singular value decomposition (SVD) and non-negative matrix factorization (NMF) based on TFIDF text vectorization to perform latent semantics. After analysis, the similarity between the code text and the query text is calculated by combining the cosine distance and Euclidean distance formula.

S2 calculates the similarity feature between the code text and the query text from the subject angle. The vectorization method is still based on the text processing of 1,2,3-gram, and the text vectorization is performed by TFIDF, and then the theme vector is extracted by combining LDA. After obtaining the subject vector of the code text and the query text, the similarity feature is calculated by the pre-selected distance and the Euclidean distance calculation formula.

D1 leveraging word embedding technique to get the document representation. The vectorized representation of the entire document would be constructed by the word vector of each word. The word vector pre-training process is performed by the word2vector and the glove method on cleaned code corpus. Suppose that the total number of words in a piece of text is N, the dimension of the word vector obtained by pre-training is P, and the j_th dimension of the i_th word in the vocabulary is recorded as $w_{i,j}$. The corresponding document indicates that the component corresponding to the

Table 1: Multilevel semantic representation

ID	Category	Feature set name	Description
T1	Text feature	Edit distance	Edit distance
T2		Co-occurrence index	Word co-occurrence
T3		Vector distance	TF-IDF vector distance
S1	Latent semantic feature	Latent semantic	LSI vector distance
S2		Topic feature	Topic vector distance
D1	Deep semantic feature	Word2vec	Word vector for source code
D2		Weighted Word2vec	Weighted word vector for source code

Table 2: Multilevel semantic interaction

ID	Representation model	Interaction model
T1	Unigram	Levenshtein
T2	1,2,3-gram+bow+tfidf	Jaccard, Dice, etc.
T3	1,2,3-gram+tfidf	Cosine, Euler, etc.
S1	1,2,3-gram+tfidf+svd	Cosine, Euler, etc.
S2	1,2,3-gram+tfidf+lda	Cosine, Euler, etc.
D1	Unigram+word2vec/glove	Cosine, Euler, etc.
D2	1,2,3-gram+tfidf+word2vec	Cosine, Euler, etc.

j_th dimension in the vector D is recorded as d_j. The formula for vector representation of documents using word vectors is:

$$D = [d_j | j : 0 \rightarrow P - 1] \tag{1}$$

$$d_j = \sum_{i=0}^{N-1} w_{i,j} \tag{2}$$

D2 introduces word weight information on the basis of D1 to integrate the document. In D1, there is no difference in the word vectors of all words. There is no information that can take into account the importance of the different words in the text. Many large but unimportant words (such as partial adverbs, auxiliary words, articles, etc.) correspond to the word vector; Additionally, there will be a huge impact on the overall document expression.

$$D_p = [d_j | j : 0 \rightarrow P - 1] \tag{3}$$

$$d_j = \sum_{i=0}^{N-1} t_{q,i} * w_{i,j} \tag{4}$$

In D2 method, the TFIDF representation of the document is introduced on the basis of 1,2,3-gram, and the result is corrected by combining the TFIDF weight information of the word in the document when the word vector is added. The word vector and the document vector definition are the same as D1. Here, the TFIDF weight of the i_{th} word in the q_{th} document in the document is $t_(q, i))$, and the weighted word vector is used to perform the document vectorization representation formula corresponding to Equation 3 and Equation 4.

4 EVALUATION

Here we uses NDCG@K and MAP to evaluate the code search task. For the dataset we use ROSF[13]. ROSF contains 35 most frequently asked development questions as natural language queries. Each query has about one hundred labeled relevance Android code snippets which relevance score from 1 to 4. While score 1 and 2 will be considered irrelevant and 3,4 mean relevant . Besides the labeled data, ROSF contains about half a million unlabeled Android code snippets which will be used for LDA and word2vec language model training.

5 EXPERIMENTS

In this part, we focus on the three research questions mentioned above. Parameters for the MSR model are shown in Table **??**. Here, the feature size of TFIDF is set to 5000 since a larger vocabulary size will consume too much memory resources with few performance gain. For another reason, we do statistics on the total number of tokens in our code search dataset then choose the first 95 % tokens to make a vocabulary table. Then we get about 5000 different tokens. On this basis, the topic model and the latent semantic analysis model dimensions are selected in 200 dimensions. Finally, we train word embedding on 50, 150, 300, and 500 dimensions for deep semantic representation.

Table 3: Hyperparameter of feature extraction object

Feature	Type	Parameter
TFIDF	VSM feature	1-gram; Feature size 5000
LDA	Topic feature	Dim 200
PCA	Latent semantic feature	Dim 200
NMF		Dim 200
Word2vec	Deep semantics feature	Dim: 50,150,300,500
Glove		Dim: 50,150,300,500

5.1 Experiment Settings

In order to find out the importance of different parts in code snippets when modeling semantic relevance between natural language query, we divide the code snippet into four parts: method name, API sequence, code token texts, code comment texts. For the code method name, we do nothing but using the raw code method name as target texts. For API sequence, we use the java parser tools to get the parse tree of the whole code snippet. As for those snippets, which can not be parsed, we use a regex pattern for extracting the API call sequence. For code token text, we combine the API sequence, method input parameter, and method output parameter

Table 4: Evaluation on different code search domain for MSR

Metric	NDCG@10					P@10				
Ranking method	**Method**	**API**	**Token**	**Comment**	**ALL**	**Method**	**API**	**Token**	**Comment**	**ALL**
ROSF	—-	—-	—-	—-	0.4370	—-	—-	—-	—-	0.5250
MSR+LR	0.4144	0.4306	0.4313	0.4470	0.4106	0.4400	0.4800	0.4350	0.5050	0.4950
MSR+RF	0.4167	0.4428	0.4462	0.4223	0.4865	0.4150	0.4700	0.5150	0.5100	0.4900
MSR+GNB	0.3909	0.4472	0.3974	0.4340	0.4357	0.4600	0.4350	0.4300	0.5100	0.4300
MSR+LGB	0.4000	0.4490	0.4285	0.4143	0.4517	0.4350	0.5250	0.6050	0.4950	0.4950
MSR+AdaRank	0.379	0.3642	0.3778	0.3949	0.4188	0.3900	0.3750	0.3450	0.4550	0.5200
MSR+LambdaMart	0.3871	0.4517	0.4631	0.4432	0.4616	0.4450	0.4700	0.5400	0.5450	0.4800
MSR+ListNet	0.3647	0.3644	0.3653	0.4355	0.3681	0.3850	0.4550	0.280	0.4900	0.4550
MSR+MART	0.3853	0.4327	0.4815	0.4429	0.4549	0.4400	0.4950	0.5250	0.5300	0.5650

to get the raw token text. Then we wash out stop words (keywords in java language) from token text and split apart the identifier that in the form of the Camel case. Finally, we can get the code comment texts by concatenating all comment text in the code snippet.

In our experiment, we combine MSR with multiple learning to rank algorithm which implemented by Dang V.[8] in Ranklib. For model comparison, we re-implement the ROSF[13] model which combines text vector distance, topic vector distance with a linear model for code search. We also combine MSR with classification algorithms like logistic regression, random forest classifier, Gaussian Bayes classifier, and lightgbm classifier for comparison. f Finally we use lightgbm to output feature importance. Lightgbm is an efficient implementation of the GBDT algorithm, introduced by Microsoft Corporation. Guided by labels, lightgbm can output the importance score of the input features. We evaluate the validity of different semantic levels in MSR model by the scores given by the algorithm.

5.2 Results

Table 4 shows the evaluation of MSR method on different code search domains in NDCG and P@10 metrics comparing with ROSF[13] at the first line. Full set of word-based semantic features are used in each search domain when notation ALL means full feature set on all four search domain. Here we notice that scores for ROSF are lower than the original paper shows. Two reasons are responsible for this phenomenon: First, the original paper uses a linear model for ranking. We do not know how they implement this linear model, so we use a logistic regression model to replace them. Second, the original paper uses a two-stage method to evaluate their model: they recall the relevance code snippets for a huge code base then re-rank the candidate snippets. In comparison, we directly apply the ROSF feature extraction and re-ranking method on test data.

Table 4 compares eight ranking models on MSR method from where we can see some general conclusions on roles that different search domains playing part in the search task. Compared to the scores in other domains, the method name domain seems always lower. While Gu et al.[12] report the importance of API sequence in the search task we achieve a relatively lower score in the ranking process. According to our analysis, it is mainly a compounded identifier that causes a lower score when ranking on these two domains. The word-based semantic feature will suffer a lot from word mismatch problem. In MSR we use 5000 size vocabulary for TF-IDF where compounded identifier in method name and API

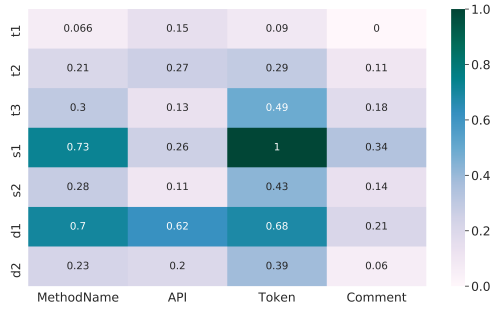

Figure 3: Multi semantic representation feature importance in different code search domain

Table 5: Feature importance among different semantic series

	NDCG@10			
Ranking method	T-series	S-series	D-series	ALL
LR	0.459	0.4074	0.4438	0.4105
RF	0.4512	0.4528	0.4094	0.4211
GNB	0.432	0.4372	0.4385	0.4357
LGB	0.4421	0.4389	0.4164	0.4517
AdaRank	0.4188	0.3586	0.3527	0.4188
LambdaMart	0.4739	0.4489	0.4772	0.4616
Listnet	0.3612	0.4025	0.3802	0.3681
Mart	0.448	0.4992	0.439	0.4549

sequence do not include in this. Token and comment show relatively high scores in eight ranking method while the token and comment forms are closer to natural language words.

Figure 3 shows feature importance among different semantic feature sets and different search domains. Table 5 shows the ranking score at a single series of features. Combine Figure 3 with Table5 we can see the latent semantic feature and deep semantic feature achieve the highest two importance among other features.

Now we can answer the three research questions. For RQ1, we propose a word-based Multi-level semantic representation model by combining text features, latent semantic features, and deep semantic features. For RQ2, according to Figure 3 we can see different

search domains have different adaptability with multi-level semantic representation. For RQ3, Table 4 shows that word based semantic model suffers from a compounded identifier. In further research we can model compounded identifiers by dividing them into affixes.

6 CONCLUSION

In this paper, we propose a word-based Multi-level semantic representation model to jointly modeling natural language and programming language. Starts with three research questions we investigate how different levels of semantic construct the complete representation of code text. The high contribution of the latent semantic feature shows that synonyms words play an important part in semantic modeling between NL and PL where the deep semantic feature also confirms this. Through experiments it can be seen that processed code token texts give a high contribution to code search tasks where identifiers that compounded by multiple natural language words give less contribution to the task. This inspired us to use the affix embedding for neural models in our future work.

REFERENCES

[1] G. Antoniol, G. Canfora, A. de Lucia, and G. Casazza. 2000. Information Retrieval Models for Recovering Traceability Links Between Code and Documentation. In *Proceedings of the International Conference on Software Maintenance (ICSM'00) (ICSM '00)*. IEEE Computer Society, Washington, DC, USA, 40–. http://dl.acm.org/citation.cfm?id=850948.853428

[2] A. Arwan, S. Rochimah, and R. J. Akbar. 2015. Source code retrieval on StackOverflow using LDA. In *2015 3rd International Conference on Information and Communication Technology (ICoICT)*. 295–299. https://doi.org/10.1109/ICoICT.2015.7231439

[3] Sushil K. Bajracharya, Joel Ossher, and Cristina V. Lopes. 2010. Leveraging Usage Similarity for Effective Retrieval of Examples in Code Repositories. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*. ACM, New York, NY, USA, 157–166. https://doi.org/10.1145/1882291.1882316

[4] Yuk Fung Chang and Caroline M. Eastman. 1993. An information retrieval system for reusable software. *Information Processing Management* 29, 5 (1993), 601 – 614. https://doi.org/10.1016/0306-4573(93)90082-O

[5] Qingying Chen and Minghui Zhou. 2018. 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)*. ACM, New York, NY, USA, 826–831. https://doi.org/10.1145/3238147.3240471

[6] Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. 2020. ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators. In *ICLR*.

[7] Charles Clarke, Anthony Cox, and Susan Sim. 1999. Searching Program Source Code with a Structured Text Retrieval System (Poster Abstract). In *Proceedings of the 22Nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '99)*. ACM, New York, NY, USA, 307–308. https://doi.org/10.1145/312624.312739

[8] V. Dang. [n.d.]. *The Lemur Project-Wiki-RankLib*. Lemur Project. http://sourceforge.net/p/lemur/wiki/RankLib.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805

[10] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:cs.CL/2002.08155

[11] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep Code Search. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 933–944. https://doi.org/10.1145/3180155.3180167

[12] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 631–642.

[13] H. Jiang, L. Nie, Z. Sun, Z. Ren, W. Kong, T. Zhang, and X. Luo. 2019. ROSF: Leveraging Information Retrieval and Supervised Learning for Recommending Code Snippets. *IEEE Transactions on Services Computing* 12, 1 (Jan 2019), 34–46. https://doi.org/10.1109/TSC.2016.2592909

[14] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2019. Pre-trained Contextual Embedding of Source Code. arXiv:cs.SE/2001.00059

[15] X. Li, H. Jiang, Y. Kamei, and X. Chen. 2018. Bridging Semantic Gaps between Natural Languages and APIs with Word Embedding. *IEEE Transactions on Software Engineering* (2018), 1–1. https://doi.org/10.1109/TSE.2018.2876006

[16] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. CodeHow: Effective Code Search Based on API Understanding and Extended Boolean Model (E). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE '15)*. IEEE Computer Society, Washington, DC, USA, 260–270. https://doi.org/10.1109/ASE.2015.42

[17] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan I. Maletic. 2004. An Information Retrieval Approach to Concept Location in Source Code. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE '04)*. IEEE Computer Society, Washington, DC, USA, 214–223. http://dl.acm.org/citation.cfm?id=1038267.1039053

[18] Steven P. Reiss. 2009. Semantics-based Code Search. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 243–253. https://doi.org/10.1109/ICSE.2009.5070525

[19] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on Source Code: A Neural Code Search. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2018)*. ACM, New York, NY, USA, 31–41. https://doi.org/10.1145/3211346.3211353