Code-to-Code Search Based on Deep Neural Network and Code Mutation

Yuji Fujiwara*, Norihiro Yoshida†, Eunjong Choi‡, and Katsuro Inoue*
*Osaka University, Japan, {y-fujiwr, inoue}@ist.osaka-u.ac.jp
†Nagoya University, Japan, yoshida@ertl.jp
‡Nara Institute of Science and Technology, Japan, choi@is.naist.jp

***Abstract*—Deep Neural Networks (DNNs) have been often used for the labeling of image files (e.g., object detection). Although they can be applied for the labeling of code fragment (i.e., code-to-code search) in software engineering, a large number of code fragments are required for each label in the learning process of DNNs. In this paper, we present an approach for code-to-code search based on a DNN model and code mutation for generating enough number of code fragments for each label. The preliminary experiment shows high precision and recall of the proposed approach.**

***Index Terms*—Code-to-Code Search, Deep Neural Network, Code Mutation**

I. Introduction

Oftentimes, developers are looking for code fragments that offer similar functionality than some other code fragments [1]. As an example, a developer may need to find implementations that could be more efficient and/or reliable than the one she/he has [1]. As another example, when a developer finds a code fragment from OSS projects or discussion platforms such as Stack Overflow[1], he/she may need to not only identify the license of the original code fragment but also find less vulnerable one. So far, several approaches have been proposed on code-to-code search to help developers to look for code fragments that offer similar functionality [1] [2] [3].

So far, Deep Neural Networks (DNNs) has been used for not only object detection in the computer vision field [4], [5] but also code clone detection in the software engineering field [6]–[9]. Our key insight is to use DNNs for labeling code fragments according to their functionality from source code. Once the labeling is successfully achieved by DNN, the resultant labeling is able to be used for a code-to-code search. This insight is gained from the DNN-based labeling of parts of images according to their pixel characteristics.

In this paper, we present an approach for code block search using a **Feed-Forward Neural Network (referred to as FFNN)**. Our approach is able to find similar code fragments corresponding to a query code fragment. Our code search approach is comprised of two steps, namely *STEP L (Learning)* and *STEP S (Search)*. In the *STEP L*, mutated source code fragments are generated from the original source code and these fragments are clustered based on original source code. Their code blocks are extracted and then feature vectors are generated from these code blocks. Unique labels are given to these feature vectors for each cluster. Finally, the feed-forward neural network model learns tuples consisting of feature vectors and the corresponding labels with supervised learning. In *STEP S*, the trained model calculates a label from feature vectors generated from query code fragment, and the original code blocks of the cluster corresponding to this label is gained as a search result. In the case study, we applied our approach into three OSS systems and then confirm the effectiveness of our approach.

[1]https://stackoverflow.com/

```
1   public static void BubbleSort()
2   {
3       int temp;
4       for (int j = 0; j < num.length - 1; j++) {
5           if (num[j] > num[j + 1]) {
6               temp = num[j];
7               num[j] = num[j + 1];
8               num[j + 1] = temp;
9           }
10      }
11  }
```

(a)

```
1   public static void BubbleSort()
2   {
3       int temp;
4       for (int j = 0; j < num.length - 1; j++) {
5           if (num[j] > num[j + 1]) {
6               ;
7               num[j] = num[j + 1];
8               num[j + 1] = temp;
9           }
10      }
11  }
```

(b)

Fig. 1. Example of the Mutation Operator mSDL

II. Background

A. Mutation Operators for Cloning

Roy and Cordy presented mutation operators for cloning which create new code clones by editing original code fragments [10]. They also defined 13 kinds of the mutation operators for generating different types of code clones. Figure 1 shows an example of applying the mutation operator *mSDL*, small deletions within a line. Figure 1(b) was created by deleting a part of a line of an original code fragment, shown in Figure 1(a).

B. Feed-Forward Neural Network

Feed-Forward Neural Network (FFNN) is widely used an artificial neural network, where the inner architecture is organized in a subsequent layer of neurons [11]. It consists of at least three layers, input, output, and hidden layer. The weight values are associated with the neurons, and these values are adjusted during a training process that compares input values with output values. If a vector similar to the trained input value is given to the FFNN model, a vector similar to the trained output value is outputted from the FFNN model.

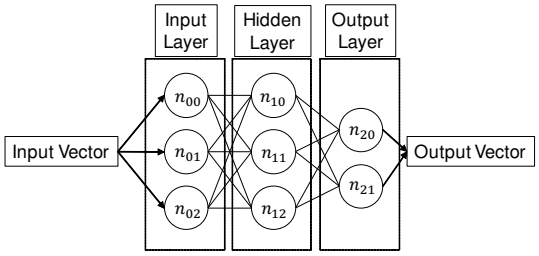

Fig. 2. Example of a Feed-Forward Neural Network (FFNN) Model

Figure 2 shows an example of a three-layered FFNN model composed of 8 neurons $n_{00}, n_{01}, \ldots, n_{21}$. In this figure, the input of FFNN is a 3-dimensional vector and the output is a 2-dimensional vector.

III. Proposed Approach

In this paper, we present an approach for code-to-code search based on deep neural networks and code mutation. Our approach searches syntactically similar code fragments to input code fragments based on deep neural networks because deep-learning based approaches are good at the mapping of two elements. In addition, by using code fragments that were incapable of being searched for learning the model they are enabled to be searched and it is possible to reduce omission of code search and erroneous code search. Our approach is comprised of two steps; *STEP L (Learning)* and *STEP S (Search)*.

A. Definition of Terms

Code Block: A code fragment that satisfies one of the following two conditions:

Condition 1. A mthod body

Condition 2. A code fragment between a pair of brackets of if, else, for, while, do-while, or switch blocks

Similarity: Similarity between two normalized code blocks $t1$, $t2$ is defined as follows:

$$Similarity(t1, t2) = \frac{2 * |t1 \cap t2|}{|t1| + |t2|}$$

where $t1 \cap t2$ is the intersection between two code blocks $t1$ and $t2$ and $|t|$ is the number of lines of t.

Similar Code Block: A pair of code blocks is syntactically identical or similar code blocks (i.e. a pair of code blocks whose $Similarity$ value is greater than 0).

Similar Code Block Set: an equivalence class of similar code blocks

Negative Data: When searching similar code to a given source code, no search results might be obtained. In this study, negative data for the "No code search results" are defined as vectors that are generated from code blocks that satisfy the following condition:

Condition 3. Code blocks whose label of feature vectors are 0 meaning "No code search results".

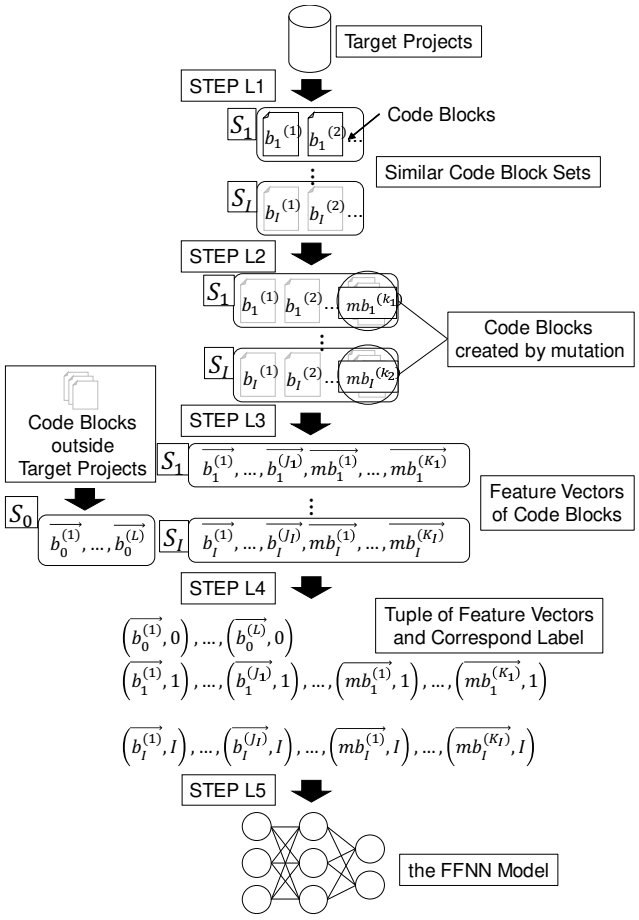

Fig. 3. Overview of the STEP L

Positive Data: In this study, positive data are defined as vectors that are generated from code blocks that satisfy the following condition:

Condition 4. Code blocks whose labels of the feature vectors are other than 0 in the target projects and similar code block to them created by applying mutation operators described in Section II-A.

B. Deep Learning Using a Feed-Forward Neural Network

This section describes *STEP L*, a step for learning models using deep learning. In this step, positive and negative data are generated from the target projects and then an FFNN model is trained by this data. Figure 3 shows an overview of *STEP L*. This step consists of the following five steps:

STEP L1. At first, code blocks are extracted from the target projects and then similar code block sets $S_i (1 \leq i \leq I)$ are created by using *CCFinder* [12], a token-based code clone detection tool and a block clone detection tool by Yokoi et al. [13]. Similar code block sets S_i contains similar code block ${b_i}^{(j)} (1 \leq j \leq J)$. Positive data are generated from these code block sets. Next, code blocks are extracted from projects which are not included in the target projects. Negative data are generated from these code blocks.

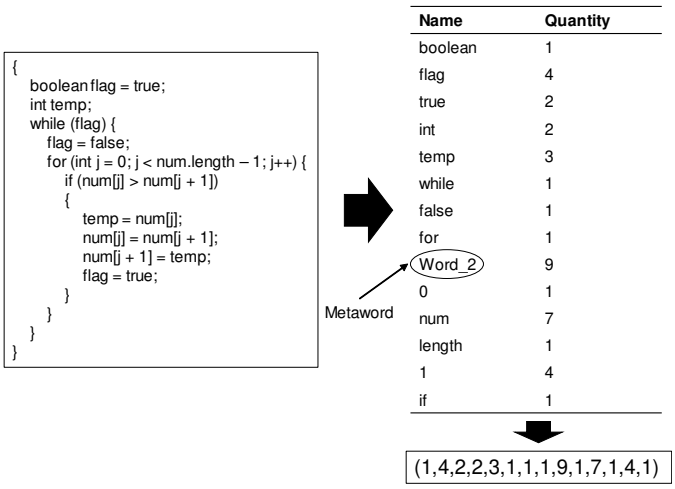

Fig. 4. Example of applying BoW to Source Code

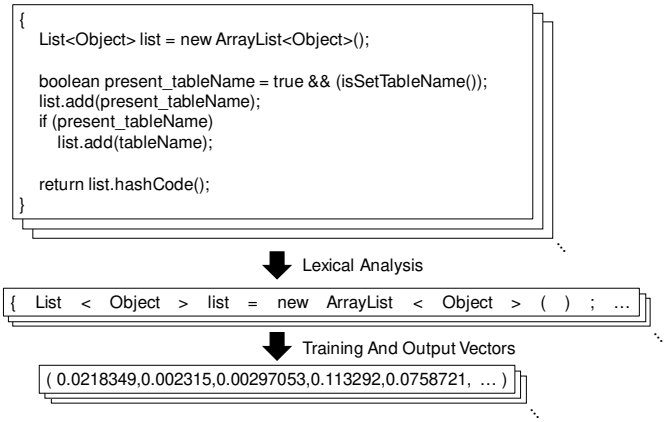

Fig. 5. Example of Applying Doc2Vec to Source Code

STEP L2. Using the mutation operators described in Section II-A, similar code blocks ${mb_i}^{(k)}(1 \leq k \leq K)$ to code blocks ${b_i}^{(j)}$ included in similar code block sets S_i are generated. At this time, mutated code blocks ${mb_i}^{(k)}(1 \leq k \leq K)$ are contained to the similar code block sets S_i.

STEP L3. Feature vectors of positive and negative data are generated from code blocks ${b_i}^{(j)}$ and ${mb_i}^{(k)}$. The approach generating feature vectors is described in Section III-B1.

STEP L4. Label i is given to positive data generated from code blocks belonging to similar code block sets S_i. Moreover, label 0 meaning "No code is included in the target project" are given to negative data generated from code blocks ${b_0}^{(l)}$.

STEP L5. Using supervised learning, the FFNN model is generated by using positive and negative data and given labels.

1) Generation of Feature Vectors: In the case study, BoW (Bag of Words), the simplest approach to generating feature vectors, or Doc2Vec [14], the approach based on deep learning, are applied to *STEP L3*.

BoW: Feature vectors are generated based on reserved keywords and identifiers extracted from positive data for training. At this time, identifiers comprised of up to two letters are considered as metaword. If feature vectors are generated from the given source code, extracted reserved keywords and identifiers are included in positive data for training. Figure 4 shows the example of the generation of a feature vector using BoW. Word_2 represents metawords. Variables x and y correspond to metawords.

Doc2Vec: Doc2Vec [14] is an approach for generating feature vectors based on unsupervised learning, and its effectiveness has been demonstrated for tasks targeting natural language. We use gensim[2] to use Doc2Vec. Figure 5 shows an example of generation of a feature vector using Doc2Vec.

[2]https://radimrehurek.com/gensim/

First, lexical analysis of positive and negative data for training is performed using ANTLR[3]. Next, the tokens of each code block are placed on one line with a space between tokens. Finally, the Doc2Vec model is trained by these lines. If one line with a space between tokens of the input code block is given to this model, embedding vector of the input code block are obtained.

C. Searching Similar Code Blocks Using Trained Feed-Forward Neural Network Model

This section explains *STEP S*, a step for searching similar code block. In the *STEP S*, similar code search is performed using the FFNN model trained in *STEP L5*. This step consists of three steps. Figure 6 shows an overview of *STEP S*.

STEP S1. After parsing the input code fragment, its feature vector is generated in the same method as *STEP L3*.

STEP S2. By inputting feature vector generated in *STEP S1* to the model trained in *STEP L5*, the label of the input code fragment is suggested.

STEP S3. Code blocks ${b_x}^{(j)}(1 \leq j \leq J)$ belonging to similar code block sets S_x corresponding to label x suggested in *STEP S2* are output. In the case of $x = 0$, there is no code search result because the trained model suggests that the input is similar to negative data for training.

IV. Case Study

This section describes a case study about our approach. This case study shows the performance of our approach in precision, recall, and F-measure using benchmark made of three target OSS: HBase 2.0[4], OpenSSL 0.9.1$\cdots$1.1.0[5], and FreeBSD 11.1.0[6]. To use cross-version similar code blocks caused by updates and so on. For evaluation, multiple versions of OpenSSL are used.

[3]http://www.antlr.org/
[4]https://hbase.apache.org/
[5]https://www.openssl.org/
[6]https://www.freebsd.org/

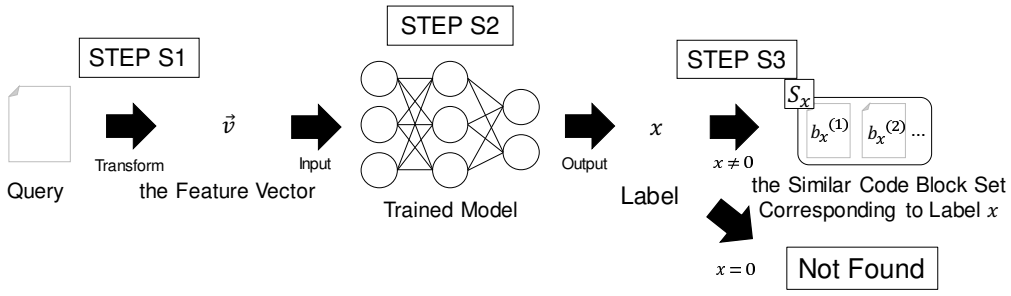

Fig. 6. Overview of the STEP S

A. Hyper Parameter

In this case study, the Feed-Forward Neural Network (FFNN) model is implemented using deep learning framework Chainer3.4.0[7]. The network consists of three layers: an input layer, a hidden layer, and an output layer. The dimension of input is the dimension of the input feature vector and output is the number of kinds of similar code block sets. The number of unit of the hidden layer has been empirically decided to 200. The output layer is used softmax function.

B. Procedure of the Case Study

The case study consists of STEP E (Experiment) 1, 2 and 3.

STEP E1. Dataset for training and evaluation using each target OSS are created.

STEP E2. The FFNN model is trained by code blocks of the dataset for training.

STEP E3. Precision, recall, and F-score are calculated using the dataset for evaluation.

C. Procedure of Creating Dataset

In the STEP E1, the dataset for training and evaluation are created to follow the following steps.

STEP E1 (1). Similar code block sets are extracted from target OSS and sets belonged by 100 or more code blocks are selected in order to prepare many similar code blocks for use in training and evaluation of the model.

STEP E1 (2). New similar code blocks are created by 20% of similar code blocks in the selected sets being applied mutation operators to and feature vectors generated from them are added to positive data for training. The corresponding label is given to each feature vector the following Section III-B *STEP L4*.

STEP E1 (3). 30000 feature vectors generated from code blocks that are syntactically dissimilar to the positive data created in STEP E1(2) are added to negative data for training. The source code used at this time is acquired from BigCloneBench [15] for HBase dataset, from FreeBSD for OpenSSL, and OpenSSL for FreeBSD.

[7]https://chainer.org/

STEP E1 (4). Feature vectors generated from code blocks extracted from each target OSS and 80% of similar code blocks not used in STEP E1(2) are added to the dataset for evaluation. If a feature vector is similar to a positive data for training, the corresponding label is given to it. If not, label 0 is given.

Table I shows the number of positive data and negative data in the dataset created using each target OSS. There is bias in the dataset for evaluation because of using all code blocks included in each target OSS and multiple versions of OpenSSL. Also, to investigate the relationship between the degree of syntactic difference between code blocks for training and input code fragments and search performance, the similarity between dataset for training and evaluation is changed for each target OSS. Syntactically identical code blocks whose the similarity is between 0.9 and 1.0 are extracted from Hbase, syntactically mostly similar code blocks whose similarity is between 0.7 and 1.0 are extracted from OpenSSL, and semantically similar code blocks whose similarity is between 0.1 and 0.2 are extracted from FreeBSD.

D. Criteria of Search Performance

In this experiment, precision, recall, and F-measure are used for evaluation of search performance. The following shows these criteria.

Precision: This is a criterion about accuracy. In this experiment, precision is defined as the ratio of the number of correct results for labels of positive data for evaluation to the number of results that the model calculated labels of positive data.

Recall: This is a criterion about comprehensiveness. In this experiment, recall is defined as the ratio of the number of correct results for labels of positive data for evaluation to positive data for evaluation.

TABLE I
DATASET

	for Training		for Evaluation	
OSS	Positive Data	Negative Data	Positive Data	Negative data
Hbase	28,822	30,000	740	12,688
OpenSSL	36,772	30,000	281	99,719
FreeBSD	27,852	30,000	747	8,177

Name	Expected label	Output label
A	0	0
B	0	1
C	1	1
D	2	2
E	3	3
F	4	3

$$precision = \frac{3}{5} = 0.6$$

$$recall = \frac{3}{4} = 0.75$$

$$F\_value = \frac{2 \times \frac{3}{5} \times \frac{3}{4}}{\frac{3}{5} + \frac{3}{4}} = 0.67$$

Fig. 7. Example of Calculating Search Performance

F-measure: This is a synthetic criterion of precision and recall. F-score is a harmonic mean of precision and recall.

Figure 7 shows an example of calculating search performance. In this example, the model output the label of the similar code block set for five blocks B to F as search results, but the correct output is for three blocks C to E, so precision is $3/5$. Also, among code blocks given as input, code blocks similar to four blocks C to F are learned by the model, but the correct output is for three blocks C to E, so recall is $3/4$. F-score is a harmonic mean of precision and recall, so it is $2/3$.

E. Result of the Experiment

Table II shows precision, recall, and F-score. Recall in experiments using Hbase and OpenSSL is 1.000, so the model learning code blocks in advance can search syntactically similar code blocks with high accuracy.

Figure 8 and 9 show examples of code blocks which are successfully found. The code blocks in Figure 8 (b) and Figure 9 (b) are used as positive data for training. The trained model suggested that a feature vector generated from Figure 8 (a) and Figure 9 (a) is similar to one generated from Figure 8 (b) and Figure 9 (b). The bold texts in Figure 8 and Figure 9 indicate the differences between each of query code and a search result.

Figure 10 shows the example of an incorrect search result. These code blocks are dissimilar although the trained model suggested that these are similar. The differences in Figure 10 are possible to cause a case that the model suggested these are similar. The consideration for it is described by Section IV-F.

F. Discussion

At first, we consider the search performance of the proposed approach. Recall in experiments using Hbase and OpenSSL was 1.000, so if code blocks have high similarity to input, the model suggests correct search result. However, precision was slightly lower. About the cause of lower precision, we consider that the model tends to suggest a label based on local features. For example, when the model is trained by dataset including the code block which has the sentence "*List<Object> list = new Arraylist<Object>();*" and is included in the target project, the model often suggest that it is similar to code blocks including the sentence to create list type objects such as "*List<Object> list = new Arraylist<Object>();*" if these code blocks are syntactically dissimilar on the whole. In this experiment, the differences in Figure 10 are possible to cause an incorrect suggestion that Figure 10 (a) and (b) are similar. The result of the experiment using FreeBSD is worse than HBase and OpenSSL, so dataset including syntactically less similar code block and our approach are incompatible.

TABLE II
SEARCH ACCURACY

OSS	BoW			Doc2Vec		
	Precision	Recall	F-score	Precision	Recall	F-score
Hbase	0.924	1.000	0.960	0.830	1.000	0.907
OpenSSL	0.733	1.000	0.846	0.652	1.000	0.789
FreeBSD	0.497	0.822	0.620	0.519	0.529	0.524

```
List<Object> list = new ArrayList<Object>();

boolean present_message = true && (isSetMessage());
list.add(present_message);
if (present_message)
   list.add(message);

return list.hashCode()
```

(a) Input Query

```
List<Object> list = new ArrayList<Object>();

boolean present_tableName = true && (isSetTableName());
list.add(present_tableName);
if (present_tableName)
   list.add(tableName);

return list.hashCode();
```

(b) Search Result

Fig. 8. Example of Successful Search (Match)

G. Case Study of the Effect of Mutation

In this section, we describe a case study of the effect of the mutation. The model output probability of each label using softmax function. In this case study, the effect of mutation was described by investigating the relationship between the number of positive data for training and the probability of correct label which trained model output when similar code blocks to positive data for training was given to it.

1) Steps of the Case Study: The case study consists of STEP M (evaluation of Mutation) 1, 2, 3, and 4.

STEP M1. The function f which included in over 200 versions of OpenSSL and similar code blocks $f_n(n = 1, 2, \cdots, N)$ to exist in other versions is selected and using the mutation operators syntactically similar code blocks are generated from it. Feature vectors of them are generated using Doc2Vec and they are given label '1'.

STEP M2. Only a of feature vectors created in STEP M1 are added to positive data for training, 30000 negative data for training given label '0' are prepared, and the models M_a are trained by positive and negative data. The value of a is changed

```
{
int num,i;
char *p;
(omission)
for (;;)
        {
        (omission)
        if (num >= arg->count)
                {
                char **tmp_p;
                int tlen = arg->count + 20;
                tmp_p = (char **)OPENSSL_realloc(arg->data,
                        sizeof(char *)*tlen);
                if (tmp_p == NULL)
                        return 0;
                arg->data = tmp_p;
                arg->count = tlen;
                for (i = num; i < arg->count; i++)
                        arg->data[i] = NULL;
                }
        arg->data[num++]=p;
        (omission)
        }
*argc=num;
*argv=arg->data;
return(1);
}
```

(a) Input Query

```
{
int num,len,i;
char *p;
(omission)
for (;;)
        {
        (omission)
        if (num >= arg->count)
                {
                arg->count+=20;
                arg->data=(char **)OPENSSL_realloc(arg->data,
                        sizeof(char *)*arg->count);
                if (argc == 0) return(0);
                }
        arg->data[num++]=p;
        (omission)
        }
*argc=num;
*argv=arg->data;
return(1);
}
```

(b) Search Result

Fig. 9. Example of Successful Search (Similarity)

as $a = 1, 100, 1000, 2000, 3000, 4000, 5000, 7000, 10000$, so 9 models are created.

STEP M3. Feature vectors are generated from code blocks $f_n (n = 1, 2, \cdots, N)$ similar to f, they are input to 9 models $M_1, M_{100}, \cdots, M_{10000}$.

STEP M4. The probability of label '1' $P_{M_a}(\vec{f_n}, 1)$ which the models M_n output and the relationship between the number of positive data a and the probability $P_{M_a}(\vec{f_n}, 1)$.

2) Result of Experiment: Table 11 and Figure III show result of the case study based on Section IV-G1. $average_a$ and min_a are defined by the following two formula. They are the average and minimum value of the probability of label '1' calculated by the model M_a when $\vec{f_n}(n = 1, 2, \cdots, N)$ are

```
ArrayList<Long> timestamps = new ArrayList<>(filterArguments.size());

for (int i = 0; i<filterArguments.size(); i++) {
    long timestamp =
        ParseFilter.convertByteArrayToLong(filterArguments.get(i));
    timestamps.add(timestamp);
  }
return new TimestampsFilter(timestamps);
```

(a) Input Query

```
List<Object> list = new ArrayList<Object>();

boolean present_tableName = true && (isSetTableName());
list.add(present_tableName);
if (present_tableName)
   list.add(tableName);

return list.hashCode()
```

(b) Search Result

Fig. 10. Example of Failed Search

input to M_a.

$$average_a = \frac{1}{N} \sum_{n=1}^{N} P_{M_a}(\vec{f_n}) \quad (1)$$

$$min_a = \min_{n=1,2,\cdots,N} \{P_{M_a}(\vec{f_n})\} \quad (2)$$

Larger the number of positive data for training a, more increase in $average_a$ and min_a.

3) Discussion: The approach that the training data are created using mutation is inspired by the problem of image classification such as MNIST which is an introduction to deep learning. In this problem, in order to increase the training data, operations such as enlargement, reduction, and movement are performed on the original image and similar images slightly different from the original image is often created and added to the training data. In our approach, this idea is applied to the source code using mutation.

In the case study of this section, the models using more than 2000 positive data for training output over 99% $average_a$, and the models using more than 7000 positive data for training output over 90% min_a. This result shows that increasing training data using mutation helps to advance training.

TABLE III
RELATIONSHIP TABLE BETWEEN THE NUMBER OF POSITIVE DATA AND JUDGMENT PROBABILITY

positive data	$average_a$	min_a
1	0.000	0.000
100	0.000	0.000
1000	0.000	0.000
2000	0.973	0.173
3000	0.992	0.675
4000	0.989	0.731
5000	0.998	0.871
7000	0.999	0.955
10000	0.999	0.978

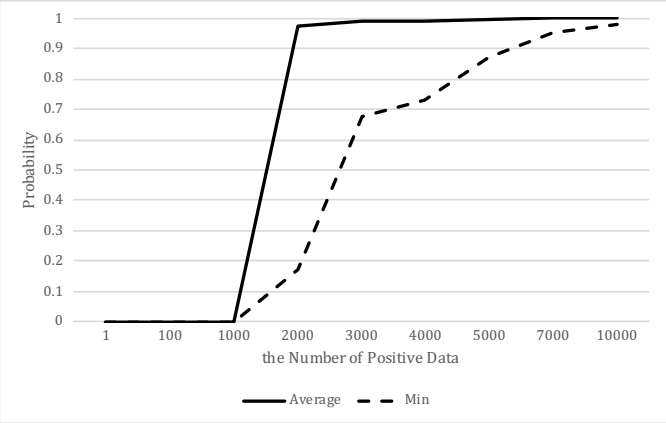

Fig. 11. Relationship between the Number of Positive Data and Judgment Probability

V. Related Work

Code search and code clone detection are similar tasks. Code search takes input code fragment as a query to check whether code clones of the query exist in the target projects. Therefore several code search engines such as Ichi Tracker [16] use code clone detection tools for code search. Kamiya et al. [12] developed a code clone detection tool namely CCFinder. This tool parses the source code, converts user-defined names to special characters, and then detects token strings that match with a length longer than the threshold as code clones. In this research, reusable source code in component and file units are detected based on the signature (class name, method signature, field name, etc.) of the class included in the component, the Jaccard coefficient between the files included in the component, or the longest common subsequence between the files. In this paper, we present a search approach of code fragments which are finer-grained than components and files based on deep learning.

White et al. [8] proposed an approach of code clone detection that feature vectors are generated from abstract syntax tree of source code using recurrent neural networks (RNN) and autoencoder and the l2 norm of each vector are calculated. Gu et al. [17] proposed "API Learning", the approach of generating examples of API usage order from the query in natural language using RNN Encoder-Decoder model which are used for machine translation. Unlike these research, our research aims at searching similar code blocks based on deep learning and we use the Feed-Forward Neural Network which is a network with a simpler structure than RNN.

VI. Conclusion

In this paper, we propose a code search approach using a Feed-Forward Neural Network (FFNN). In the learning step, the dataset for training are generated from code blocks of target source code and syntactically similar source code to it created by mutation and using created dataset the FFNN model is trained using supervised learning. In the search step, similar code blocks corresponding to the label which the model calculated based on feature vectors of input code fragments are output. The case study shows that our approach enables to search syntactically similar source code with high accuracy. Future works are shown as follows.

- This approach only uses FFNN, so we are going to use other deep networks such as RNN.
- It is necessary to evaluate whether our approach can manage larger projects by increasing the number of similar code block sets for training.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Numbers JP25220003, JP18H04094 and JP16K16034.

References

[1] Kisub Kim, Dongsun Kim, Tegawendé F. Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. FaCoY: A code-to-code search engine. In *Proc. of ICSE 2018*, pages 946–957, 2018.

[2] Norihiro Yoshida, Takeshi Hattori, and Katsuro Inoue. Finding similar defects using synonymous identifier retrieval. In *Proc. of IWSC 2010*, pages 49–56, 2010.

[3] Yoshiki Higo, Yasushi Ueda, Shinji Kusumoto, and Katsuro Inoue. Simultaneous modification support based on code clone analysis. In *Proc. of APSEC 2007*, pages 262–269, 2007.

[4] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proc. of CVPR*, pages 779–788, 2016.

[5] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: Single shot multibox detector. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Proc. of ECCV 2016*, pages 21–37. Springer International Publishing, 2016.

[6] Gang Zhao and Jeff Huang. Deepsim: Deep learning code functional similarity. In *Proc. of ESEC/FSE 2018*, pages 141–151, 2018.

[7] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V. Lopes. Oreo: Detection of clones in the twilight zone. In *Proc. of ESEC/FSE 2018*, pages 354–365, 2018.

[8] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proc. of ASE 2016*, pages 87–98, 2016.

[9] Hui-Hui Wei and Ming Li. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *Proc. of IJCAI-17*, pages 3034–3040, 2017.

[10] Chanchal K Roy and James R Cordy. A mutation/injection-based automatic framework for evaluating code clone detection tools. In *Proc. of ICSTW 2009*, pages 157–166, 2009.

[11] Howard Demuth, Mark Beale, and Martin Hagan. *Neural network toolbox 6 User's Guide*. Mathworks, 1994.

[12] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[13] Kazuki Yokoi, Eunjong Choi, Norihiro Yoshida, and Katsuro Inoue. Investigating vector-based detection of code clones using bigclonebench. In *Proc. of APSEC 2018*, pages 699–700, 2018.

[14] Jey Han Lau and Timothy Baldwin. An empirical evaluation of doc2vec with practical insights into document embedding generation. *arXiv preprint arXiv:1607.05368*, 2016.

[15] Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In *Proc. of ICSME 2014*, pages 476–480, 2014.

[16] Katsuro Inoue, Yusuke Sasaki, Pei Xia, and Yuki Manabe. Where does this code come from and where does it go?-integrated code history tracker for open source systems. In *Proc. of ICSE 2012*, pages 331–341, 2012.

[17] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In *Proc. of FSE 2016*, pages 631–642, 2016.