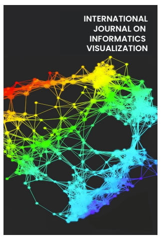
INTERNATIONAL JOURNAL
ON INFORMATICS VISUALIZATION

journal homepage : www.joiv.org/index.php/joiv



Enhancing Code Similarity with Augmented Data Filtering and Ensemble Strategies

Gyeongmin Kim [a], Minseok Kim [b], Jaechoon Jo [c,*]

[a] *Department of Computer Science and Engineering, Korea University, Seoul 02841, Republic of Korea*
[b] *Minds lab Inc., Seongnam 13493, Republic of Korea*
[c] *Division of Computer Engineering, Hanshin University, Osan 18101, Republic of Korea*
*Corresponding author: *jaechoon@hs.ac.kr*

***Abstract*— Although COVID-19 has severely affected the global economy, information technology (IT) employees managed to perform most of their work from home. Telecommuting and remote work have promoted a demand for IT services in various market sectors, including retail, entertainment, education, and healthcare. Consequently, computer and information experts are also in demand. However, producing IT, experts is difficult during a pandemic owing to limitations, such as the reduced enrollment of international students. Therefore, researching increasing software productivity is essential; this study proposes a code similarity determination model that utilizes augmented data filtering and ensemble strategies. This algorithm is the first automated development system for increasing software productivity that addresses the current situation—a worldwide shortage of software dramatically improves performance in various downstream natural language processing tasks (NLP). Unlike general-purpose pre-trained language models (PLMs), CodeBERT and GraphCodeBERT are PLMs that have learned both natural and programming languages. Hence, they are suitable as code similarity determination models. The data filtering process consists of three steps: (1) deduplication of data, (2) deletion of intersection, and (3) an exhaustive search. The best mating (BM) 25 and length normalization of BM25 (BM25L) algorithms were used to construct positive and negative pairs. The performance of the model was evaluated using the 5-fold cross-validation ensemble technique. Experiments demonstrate the effectiveness of the proposed method quantitatively. Moreover, we expect this method to be optimal for increasing software productivity in various NLP tasks.**

***Keywords*— Code similarity; language model; software productivity; CodeBERT; cross-validated ensemble.**

Manuscript received 19 Oct. 2021; revised 12 Jan. 2022; accepted 19 Apr. 2022. Date of publication 30 Sep. 2022.

I. Introduction

Pretrained language models (PLMs) such as Bidirectional Encoder Representations from Transformers (BERT) [1], Generative Pre-Training (GPT) [2], and eXtra Long NETwork (XLNET) [3], [4] have contributed to tremendous performance improvements in downstream tasks, such as machine reading comprehension, entity recognition, and relation extraction, of natural language processing (NLP) in recent years. These PLMs are based on the transformer architecture [5], and they are pretrained on a large unsupervised text corpus and then finetuned with training data in downstream tasks such as question answering [6], [7], named entity recognition [8] and relation extraction [9]. This pre-training and finetuning model approach in NLP has expedited pre-training development.

CodeBERT [10] and GraphCodeBERT [11] are language models trained in a programming language (PL) and natural language (NL). These are PLMs that have been optimized such as for code search and document generation. Both models were pretrained on the CodeSearchNet dataset containing functions in six PLs as well as NLs' documents.

In the pre-training process, the former model was trained on objective functions, including standard masked language modeling (MLM), and replaced token detection (RTD) [12]. The latter model was trained on code representations using the semantic structure of the code. These code-aware specific PLMs, which have learned code representations, differ from general PLMs that have not learned code as they can classify the similarity when inputting a PL. Moreover, these PLMs are considered more general.

Recently, a serious worldwide shortage of software developers and experts who can supply quality software has become evident [13], [14]. As the supply of software developers is limited, having a method prepared in advance for analyzing, developing, and maintaining software based on an automated method is essential for increasing software productivity. In addition, many studies are being conducted in this field worldwide. As a first step, an algorithm that determines whether two codes can produce the same results is crucial for enhancing software productivity.

Our research heavily involves the use of deep bimodal architectures trained on both PL and NL. Specifically, we used CodeBERT, GraphCodeBERT, CodeBERT-MLM (developed by Microsoft Research), and CodeBERTaPy. CodeBERT and CodeBERT-MLM were among the first research to create models that capture the semantic connection between NL and PL, such as Python, Java, JavaScript, etc. In the pre-training step, the input data consists of two sequences concatenated with a special token i.e. $[CLS]\ w_1, w_2, \ldots, w_n\ [SEP]\ c_1, c_2, \ldots, c_m\ [EOS]$ where segment $[w_1, w_2, \ldots, w_n]$ denotes NL text and the segment $[c_1, c_2, \ldots, c_n]$ denotes a certain PL. This input is then passed through a standard BERT architecture, and the outputs of the model are the contextual vector representation of each token for both the language and the code component and the [CLS] token embedding containing the pooled sequence information. The pre-training objective functions include MLM and RTD, as we mentioned above. Both bimodal and unimodal data were used to train CodeBERT, and bimodal data was obtained from GitHub repositories where a code is paired with its description. CodeBERT [10] was then finetuned for downstream tasks such as code search and code-to-text generation, and it achieved state-of-the-art performances in both tasks. GraphCodeBERT [11] followed CodeBERT, and it is a graph-based pre-trained model based on Transformer [5] for PL. It is one of the first works that leverages code structure to learn code representation to improve code understanding dramatically. The pre-training tasks for GraphCodeBERT are unique in that they include not only masked language modeling but also data flow edge prediction and the variable-alignment across source code and data flow to align representation between source code and data flow.

Furthermore, the CodeSearchNet challenge [15] launched in 2020 actively encourages research in retrieving relevant code given an NL query. The organizers provide a corpus consisting of 99 NL queries with approximately 4000 expert relevance annotations of likely results. It also consists of automatically generated NL queries for 2 million code functions. A leaderboard was run, encouraging competition and advancement in this area of research. Another notable project that made use of deep learning for code understanding is the AlphaCode project [16] initiated by Deep Mind. AlphaCode aims to train deep learning models that understand code for code generation, specifically generating code solutions in various PLs to competitive coding problems from CodeForces. They approached the problem as a sequence-to-sequence task, where given a problem statement in NL, a solution code string was generated. Thus, the researchers of AlphaCode made use of an encoder-decoder Transformer architecture [17] to solve the problem. This project also made its training dataset public, which came in very handy for us to use as an additional source of training data when fine-tuning our CodeBERT models for detecting code similarity.

Despite the advancements in code and natural language understanding, there are not many existing works of literature on utilizing these architectures to classify code performing similar tasks directly. Instead, widely used approaches are text-based detection [18], token-based detection [19], which extracts a sequence of tokens using compiler-style source code transformation that is subsequently used for similarity matching, and tree-based detection [20], where the code is transformed to abstract syntax trees and are later used in tree sub matching algorithms for similarity identification and metrics based detection [21] that extracts a number of metrics from the source code fragments then comparing these metrics for detecting similarity. Another research is about using flow charts [22] to detect code similarity, but it is applied for cross-language source code detection (i.e., detecting similarity even when source codes are written in different PLs). The aforementioned techniques do not rely on deep learning, but recently relatively simple deep learning-based code embedding techniques such as Code2Vec [23] were explored to embed codes for similar or clone code detection.

For most code search systems being used in practice, it appears that simpler methods that do not involve deep neural networks, such as a cross encoder or a bi-encoder structure for classifying similarity, seem to be preferred. Our work shows that fine-tuning deep pre-trained language models that understand code performs efficiently on the task of code similarity detection and thus provides an incentive for using deep pre-trained models for future code similarity detection systems.

In this study, we propose a language model system that can determine code similarity using an automated method. The three-step strategic data filtering process for effectively training a language model can eliminate duplication between training and testing. In addition, positive and negative pairs were generated using the best matching (BM25) [24] and length normalization of BM25 (BM25L) [25] algorithms of the Okapi system with proven performance. Moreover, the cross-validation ensemble method [26] enabled an accurate and effective performance in the model training process in which data imbalance existed.

The contributions are as follows:

- Three data filtering strategies for removing duplication in the training dataset are introduced. In addition, positive and negative pairs are obtained using the BM25 and BM25L algorithms.
- CodeBERT and graphcodebert, pretrained on code representations, are finetuned. Then, the cross-validation ensemble, verified when inferencing, is applied to extract a more effective performance.
- The effectiveness of the proposed method is verified using quantitative experiments for single and ensemble models.

II. Materials and Method

In this section, we describe the following three crucial procedures (data preprocessing process, three filtering strategies, forming positive, and negative pairs), and finally, a cross-validated ensemble method for achieving performance from the ensembled results of a single model. Fig. 1 shows the overall schematic of the process.

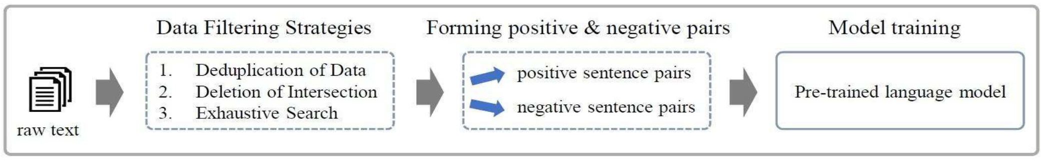


Fig. 1 Scheme of the model's training process by applying the proposed technique from the text input sequence

A. Data Preprocessing

Before the code data is fed as an input sequence of the model, cleaning the raw text must be performed because of the characteristic of the code. The code may be written in an unorganized manner by the coder or contain text information that is irrelevant to compilation. Especially, the existing typo information in the code is unnecessary, and it can act as a fatal factor in the model training procedure. First, unnecessary new lines and spaces are removed. Comments (sequences marked with '#') written to help understand the code are also somewhat unnecessary for our training process. Hence, they are removed along with white spaces.

B. Data Filtering Strategies

Duplication of data may occur when using different datasets as training and test data. Therefore, a model trained with these duplicated features cannot perform an accurate evaluation. Before enhancing the model with additional data, we observed data duplication between the additional training data and the test data. Hence, filtering was performed with three strategies. Algorithm 1 describes the three-step data filtering strategies in Fig. 1 more effectively.

1) Deduplication of Data: First, the data corresponding to A and B, and the TEST codes are loaded. The first filtering strategy filters overlapped sequence data using a hash table (HT). This method records all values of A in the HT and then checks whether these values exist in B; this is the most common and widely used method. For the first filtering of Algorithm 1 (1~6 lines), (1) the sequence input to the DEDUPLICATION function and initialized HT creates a new HT from the A, and (2) the new HT is compared with B to create *First_filtered_codes* (6 lines). Most duplicate data are filtered out in this process.

2) Deletion of Intersection: Second, the purpose of the *Second Filtering* is to filter out data not completely filtered out by the HT owing to reasons such as trailing space. This includes spaces such as white spaces ' ' in the right or left edge of sentences, tabs '\t', In the second filtering of Algorithm 1 (7~10 lines), (1) the SIMPLIFY function concatenates all newlines existing in the character strings of the code. (2) All spaces and newlines before and after the character strings are removed. (3) Filtering is performed once more by taking the intersection of these filtered sequences of the test code and using *first_filtered_codes* to generate *second_filtered_codes*.

3) Exhaustive Search: Most of the duplicated data are removed by the second filtering process. However, a method that completely eliminates duplication is required. Therefore, an exhaustive search is performed, and s and *TEST_codes* are mutually compared in the third filtering of Algorithm 1 to remove the few remaining duplicate data traces (11~20 lines). Now, the *third_filtered_codes*, which are used as the training data, are finally generated.

C. Forming positive and negative pairs

Then, the dataset that has been trimmed with the previous three preprocessing strategies should be composed of positive and negative pairs for the training process. One type of training data is positive pairs which can determine whether the two codes are similar, and the other is negative pairs which can determine whether the two codes are not similar. For negative samples, ranks are assigned after sorting the data in descending order based on the confidence score. BM25 algorithm of the Okapi system, which is based on probabilistic retrieval research, is a ranking function utilized by search engines to rank matching sentences according to their relevance to a given query. For length normalization of BM25, BM25L, a newer variant to boost scores of very long documents and more effective than BM25, was proposed [25]. The Okapi BM25 scores a document D with respect to query sentence Q containing keywords $k_1, \dots, k_n$ as follows:

$$score(D,Q) = \sum_{i=1}^{n} IDF(k_i) \cdot \frac{f(k_i,D)\cdot(m_1+1)}{f(k_i,D)+m_1\cdot(b\cdot\frac{|D|}{adl}+1-b)}$$

where $f(k_1, D)$ is the term frequency of k_1 in D, $|D|$ is the length of the words, and adl is average document length. m_1 *and* b are hyperparameters.

$$IDF(k_i) = \ln\left(\frac{N-n(k_i)+0.5}{n(k_i)+0.5}\right)+1$$

The inverse document frequency (IDF) means how common or rare a word is in the total document set. $IDF(k_i)$ is consisted of IDF weight of the query term k_i and $n(k_i)$ is the number of documents containing k_i, which is calculated as above. Using these algorithms, the volume of the original dataset was dramatically expanded, and we experimented with demonstrating the improved efficiency of the code similarity evaluation model.

```
Algorithm 1 Efficient Data Filtering Strategies
A = DATA1, B = DATA2
Load TEST_codes
  procedure DEDUPLICATION(sequence, HT)
  → Quick check if i is in HT
  1:   Read All A, B values
  2:   for i = 0 to len(A[i]) do
  3:       HT.add(A[i])
  → Build HT
  4:   for i = 0 to len(A[i]) do
  5:       if B[i] not in HT then
  6:           First_filtered_codes.append(B[i])
  → First Filtering
  procedure SIMPLIFY(code)
  7:   return ''.join(code.split('\n')).rstrip(' ').strip()
```

```
procedure DELETE INTERSECTION(First_filtered_codes)
8:    for i = 0 to len(First_filtered_codes[i]) do
9:        if SIMPLIFY(First_filtered_codes[i]) not in INTERSECTION(TEST_codes) then
10:           Second_filtered_codes.append(First_filtered_codes[i])
→ Second Filtering
procedure EXHAUSTIVE SEARCH(Second_filtered_codes)

11:   for i = 0 to len (Second_filtered_codes[i]) do
12:       USE_TOKEN = True
13:       if Second_filtered_codes[i] in TEST_codes then
14:           continue
15:       else
16:           for s = 0 to TEST_codes do len(s) > 0 and len(Second_filtered_codes[i]) > and ((Second_filtered_codes[i] in s)
                  or (s in Second_filtered_codes[i]))
17:               USE_TOKEN = False
18:           if USE_TOKEN == True then
19:               Third_filtered_codes.append(Second_filtered_codes[i])
→ Third Filtering
20:   Return Third_filtered_codes
```

D. *Cross-Validation Ensemble*

The training process of the language models is divided into training, validating, and testing datasets, respectively. For evaluation of our model, any data imbalance-related issues can not be allowed; if there are, an accurate measure is impossible. In Fig. 2, the 5-fold cross-validation can train each model in the training step, alternating each set 5 times. We finally divided the overall training dataset by 8:2, finetuned the model with each train & valid features, and evaluated it on the test dataset.

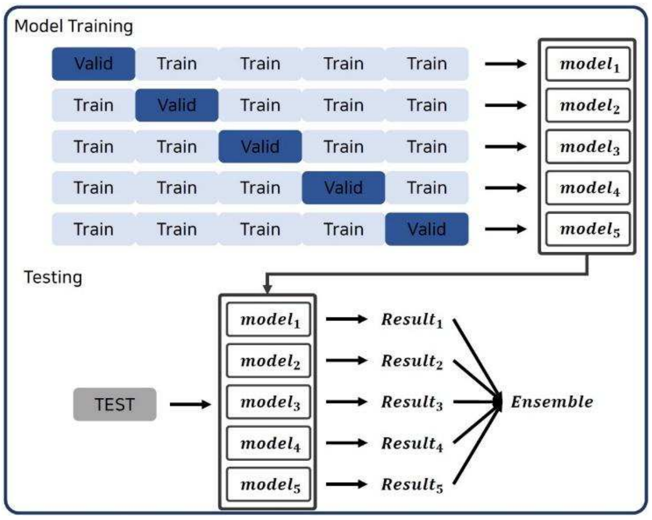

Model Training
Valid Train Train Train Train → $model_1$
Train Valid Train Train Train → $model_2$
Train Train Valid Train Train → $model_3$
Train Train Train Valid Train → $model_4$
Train Train Train Train Valid → $model_5$
Testing
TEST
$Result_1$
$Result_2$
$Result_3$
$Result_4$
$Result_5$
Ensemble

Fig. 2 A method of 5-fold cross-validation. The models were ensembled by voting from each $model_{\{n=1,2,3,4,5\}}$ and resulting in these models' training and testing

Consequently, a more accurate performance evaluation can be performed. We utilized the method of evaluating the final performance by recording the checkpoint that indicates the best performance for each fold and creating an ensemble of the determinations made by the 5 models generated in this method in the testing process.

III. Results and Discussion

This section describes the quantitative experimental method and experimental results that prove the effectiveness of our proposed method.

A. *Data*

The experiment was conducted by randomly extracting only 50% of the CodeNet data, owing to resource problems, presented at a code similarity competition, which was a NLP AI contest held in Korea in May 2022. The problems were numbered, and there were two solution codes (1 and 2) for solving each problem.

TABLE I
Volume and Count of Data for Competition and CodeNet. Only 50% of CodeNet was used owing to a resource issue. The positive and negative pairs created in this process are combined and used as the total training data.

DATA		**Competition**	**CodeNet**
Count		45,000	120,000
Volume		185.5MB	493.25MB
Positive & Negative Pairs	Total Count	Train: 5,445,594 Valid: 63,242	Train: 10,316,052 Valid: 117,168
	Total Volume	Train: 3.52GB Valid: 55.1MB	Train: 7.34GB Valid: 117.4MB

Table 1 shows the count and volume of the competition and CodeNet data used in the experiment. Positive and negative pairs were created to increase the volume of the data. In addition, the validation loss was calculated every 1000 steps by classifying the data as training and validation data. When two codes are input, the accuracy is used to measure the performance of the binary classification task, which determines whether the two codes solve the same problem.

E. *Experiment Setup*

The experiment environment is as follows. The graphcodebert and codebert-mlm models were used for the proposed method. Both models were trained on a large dataset used for researching the probing capability of code-based PLMs. These PLMs can learn the general-purpose representation as well as code search and documentation generation tasks. Moreover, these PLMs are language models, and studies have been recently conducted on these models for downstream tasks related to code as in Table 2.

TABLE II
Experimental Results of the Language Model Trained on the Korean Numerical Reasoning Data

Models	**Competition Only**		**Competition + CodeNet**	
	BM25	**BM25L**	**BM25**	**BM25L**
	Accuracy			
graphcodebert	95.7206	96.3735	96.4251	97.3456
codebert-mlm	94.9471	95.8456	95.6421	96.8657
Cross validated ensemble	96.0823	97.0988	97.6552	98.8785

Meanwhile, Table 3 lists the hyperparameters for finetuning the two models using the proposed method. These models were trained using an A100 GPU and by setting the learning rate to 2e-5, epsilon to 1e-5, the optimizer to AdamW, max epochs to 2, batch size to 32, and max sequence length to 512.

TABLE III
HYPERPARAMETER SETTINGS FOR THE FINETUNING PROCESS WITH GRAPHCODEBERT AND CODEBERT-MLM.

Hyperparameters	**Models**
Learning rate	2e-5
Epsilon	1e-5
Optimizer	AdamW
Max epochs	2
Batch size	32
Max sequence length	512

F. Results

Table 3 shows the quantitative results of the code similarity evaluation model. The first column shows the single models used in the experiment, graphcodebert and codebert-mlm, and cross-validation ensemble of the two models. In addition, the competition-only and competition + CodeNet data were used as the training data. The similarity was measured by applying the BM25 and BM25L similarity algorithms of the Okapi system to each dataset to generate the positive and negative pairs.

The results show a consistent performance improvement when the BM25L algorithm was applied, unlike with the BM25 algorithm. In addition, the graphcodebert model (the first row of Models) outperformed the codebert-mlm model (the second row of Models). Moreover, the cross-validation ensemble of the two models achieved a better performance than the two single models. In particular, the cross-validation ensemble model finetuned with the competition + CodeNet dataset achieved the highest score with a performance of 98.8785.

IV. CONCLUSION

In this study, we proposed an effective code similarity representation model. Our approach of data filtering strategies: deduplication, delete intersection, and exhaustive search, can eliminate data redundancy and contributes to efficient model training. Subsequently, the positive and negative pairs were formed by applying the BM25 and BM25L algorithms to the filtered dataset. Then, the model was finetuned with the 5-fold cross-validation method. Finally, the best performance was achieved by performing a final inference on the test dataset by the five models, each of which was trained with the cross-validated ensemble method. The performance improvement was demonstrated quantitatively by comparing our approach with conventional methods. In addition, the ensemble was proven effective on both single and augmented data. Our approach to improving the productivity of this software is crucial as the impact of labor-supply shortages in IT can negatively affect both productivity and innovation, and we believe our methods will be leveraged in various NLP tasks.

ACKNOWLEDGMENTS

Hanshin University Research Grant supported this work.

REFERENCES

[1] D. Jacob, et al. "Bert: Pre-training of deep bidirectional transformers for language understanding." *arXiv preprint arXiv:1810.04805* (2018).

[2] A. Radford, et al. "Language models are unsupervised multitask learners." *OpenAI blog* 1.8 (2019): 9.

[3] Y. Zhilin, et al. "Xlnet: Generalized autoregressive pre-training for language understanding." *Advances in neural information processing systems* 32 (2019).

[4] D. Zihang, et al. "Transformer-xl: Attentive language models beyond a fixed-length context." *arXiv preprint arXiv:1901.02860* (2019).

[5] A. Vaswani, et al. "Attention is all you need." *Advances in neural information processing systems* 30 (2017).

[6] G. Kim, et al. "AI Student: A Machine Reading Comprehension System for the Korean College Scholastic Ability Test." *Mathematics* 10.9 (2022): 1486.

[7] S. Lee, G. Kim, and H. Lim, "Verification of educational goal of reading area in Korean SAT through natural language processing techniques," *Journal of the Korea Convergence Society*, vol. 13, no. 1, pp. 81–88, Jan. 2022.

[8] G. Kim, et al. "Automatic extraction of named entities of cyber threats using a deep Bi-LSTM-CRF network." *International journal of machine learning and cybernetics* 11.10 (2020): 2341-2355.

[9] K. Kim, et al. "GREG: A global level relation extraction with knowledge graph embedding." Applied Sciences 10.3 (2020): 1181.

[10] Z. Feng, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

[11] G. Daya, et al. "Graphcodebert: Pre-training code representations with data flow." *International Conference on Learning Representations: ICLR 2021.*

[12] C. Kevin, et al. "Electra: Pre-training text encoders as discriminators rather than generators." *arXiv preprint arXiv:2003.10555* (2020).

[13] T. Breaux and J. Moritz. 2021. The 2021 software developer shortage is coming. Commun. ACM 64, 7 (July 2021), 39–41. https://doi.org/10.1145/3440753.

[14] P. Tambe, Xuan Ye, Peter Cappelli (2020) Paying to Program? Engineering Brand and High-Tech Wages. Management Science 66(7):3010-3028. https://doi.org/10.1287/mnsc.2019.3343.

[15] H. Hamel, et al. "Codesearchnet challenge: Evaluating the state of semantic code search." *arXiv preprint arXiv:1909.09436* (2019).

[16] L. Yujia, et al. "Competition-level code generation with alphacode." *arXiv preprint arXiv:2203.07814* (2022).

[17] Z. Feng, et al. "Flowchart-based cross-language source code similarity detection." *Scientific Programming* (2020).

[18] S. Ducasse, M. Rieger and S. Demeyer, "A language independent approach for detecting duplicated code," Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360), 1999, pp. 109-118, doi: 10.1109/ICSM.1999.792593.

[19] B. S. Baker, "On finding duplication and near-duplication in large software systems," Proceedings of 2nd Working Conference on Reverse Engineering, 1995, pp. 86-95, doi: 10.1109/WCRE.1995.514697.

[20] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna and L. Bier, "Clone detection using abstract syntax trees," Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272), 1998, pp. 368-377, doi: 10.1109/ICSM.1998.738528.

[21] M. Leblanc and Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," 1996 Proceedings of International Conference on Software Maintenance, 1996, pp. 244-253, doi: 10.1109/ICSM.1996.565012.

[22] P. Anupriya. *Code Clone Detection Using Code2Vec*. University of California, Irvine, 2020.

[23] A. Uri, et al. "code2vec: Learning distributed representations of code." *Proceedings of the ACM on Programming Languages* 3.POPL (2019): 1-29.

[24] R. Stephen, et al. "Okapi at TREC-3." *Nist Special Publication Sp* 109 (1995): 109.

[25] Y. Lv and C. Zhai. 2011. When documents are very long, BM25 fails! In Proceedings of the 34th international ACM SIGIR conference on research and development in Information Retrieval (SIGIR '11). Association for Computing Machinery, New York, NY, USA, 1103–1104. https://doi.org/10.1145/2009916.2010070.

[26] K. Yang, et al. "Cross-Validated Ensemble Methods in Natural Language Inference." *Annual Conference on Human and Language Technology*. Human and Language Technology, 2019.