Assessing and Improving Dataset and Evaluation Methodology in Deep Learning for Code Clone Detection

Haiyang Li*†‡, Qing Gao*†‡ and Shikun Zhang*†‡
*Key Laboratory of High Confidence Software Technologies (Peking University), MoE
†National Engineering Research Center of Software Engineering, Peking University, Beijing, China
‡School of Software and Microelectronics, Peking University, Beijing, China
{lihaiyang, gaoqing, zhangsk}@pku.edu.cn

Abstract—**Code clone detection is a task that identifies whether two code snippets are semantically identical. In recent years, deep learning models have shown high performance in detecting Type-3 and Type-4 code clones, and received increasing attention from the research community. However, compared with the attention given to the model design by the researchers, there is little research work on the quality of the datasets and the evaluation methodology (the way of dividing the dataset into training set and test set), which poses a challenge to the credibility of deep learning models.**

In this paper, we conduct experiments to evaluate the performance of the existing state-of-the-art models in multi-perspectives. At the same time, we release two new datasets for code clone detection, namely ConBigCloneBench and GoogleCodeJam2 based on the existing datasets BigCloneBench and GoogleCodeJam, respectively. Our experiments show that the performance of the same model decreases up to 0.5 F1 score (from 0.9 to 0.4) on different evaluation perspectives and datasets, and the performance of some models is only similar to the simple MLP model. We analyze reasons for the performance decline further, and provide suggestions for future research to improve the performance of deep learning models from multi-perspectives.

Index Terms—**code clone detection, datasets, evaluation method, neural networks**

I. Introduction

Code clone refers to the existence of two or more pieces of source code in the software, which are the same or have a different structure, but have the same functionality. In the process of software development and maintenance, developers often use "copy and paste" to increase the speed of software development, leading to code clones. However, studies have shown that a large number of code clones will have a negative impact on software systems [1]–[4] . For example, reusing code fragments may contain unknown bugs, which may lead to software bug propagation. In addition, without good management of code clones present in the system, the size of software will continue to grow, resulting in code redundancy and increased maintenance costs. Therefore, the detection of code clones in software systems is important in the field of software engineering.

Code clones can be divided into four types according to the degree of similarity of code pairs [5]: Type-1 clones refer to two code snippets that differ only in comments, spaces and blanks; Type-2 clones refer to two code snippets that differ in variable names and constant values in addition to Type-1 clones; Type-3 clones refer to two code snippets that contain differences such as statements added or deleted in addition to Type-1 and Type-2 clones; Type-4 clones refer to two code snippets that differ in syntax but share the same functionality.

Approaches based on text and token similarity [6]–[8] can detect Type-1 and Type-2 clones efficiently, but is less accurate in detecting Type-3 and Type-4 clones. In recent years, researchers have proposed a large number of deep learning models [9]–[12] to detect Type-3 and Type-4 clones and achieved high performance, which improve the F1 score of Type-3 and Type-4 clones from about 0.1 to 0.9 according to the survey of Lei et al. [13]

Despite recent advances in applying deep learning models on code clone detection, researchers have shown that dataset issues can pose challenges to the credibility of deep learning models. Yu et al. [14] found that the variable names in BigCloneBench, a dataset commonly used in code clone detection tasks, have a significant impact on the performance of clone detection models, while Jens Krinke et al. [15] pointed out that previous research work had wrong assumption in using BigCloneBench when constructing clone and non-clone pairs, which resulted in unreliable model results. Besides the dataset quality issues, BigCloneBench and GoogleCodeJam [11], another dataset used for clone detection tasks, also suffer from significant data imbalance between different functionalities. An unbalanced dataset can seriously reduce the number of clone pairs of the functionalities containing few samples in the training and test sets, which in turn reduces the generalization of the model. Meanwhile, the lack of sufficiently diverse clone pairs in the test data can reduce the credibility of the evaluation results.

In addition to dataset issues, research in other software engineering tasks with deep learning (e.g., code summarization

This work is supported by the National Key Research and Development Program under Grant No. 2021YFB3101802. Qing Gao is the corresponding author.

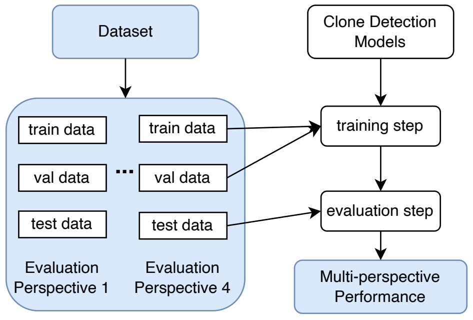


Fig. 1. The overall framework used in our work. this paper focus on the blue part: research on dataset construction and evaluation methodologies

[16], [17], defect detection [18]) has shown that evaluation methodology (the way of dividing the dataset into training set and test set) can also significantly impact the performance of deep learning models. However, there is currently no research in code clone detection studying the impact of different evaluation methodologies on model performance, which poses challenges to the credibility of model's performance in different practical scenarios.

To address the above issues, this paper conducts experimental evaluation on deep learning models for code clone detection in two aspects: dataset quality and model evaluation methodology. In terms of datasets, we released two code clone detection datasets namely ConBigCloneBench and GoogleCodeJam2. The new datasets corrected the misuse in previous research work [9]–[12] with higher data quality, and the data distribution is more balanced than that in the original datasets. In terms of evaluation methodology, We conduct experiments from multi-perspectives for state-of-the-art clone detection models and analyze the practical application scenarios of different perspectives. **Our experiments find a decrease of up to 0.5 F1 score for some models when the model performance is evaluated from different perspectives and datasets.** To investigate the reason of performance fluctuation under different perspectives, we extend the study for code abstraction method [14] by combining it with the multi-perspective code evaluation methodology, and evaluate the performance impact of code abstraction on the model under different evaluation perspectives. To the best of our knowledge, this is the first work to consider the impact of code abstraction on model performance from different perspectives. The framework is shown in Figure 1, where the dataset quality and evaluation methodologies is concerned.

The contributions of this paper are as follows:

1) We release two datasets for clone detection, ConBigCloneBench and GoogleCodeJam2 based on BigCloneBench and GoogleCodeJam with more balanced distribution and higher quality (Section II).
2) We conduct experiments from multi-perspectives for state-of-the-art clone detection deep learning models, and analyze the performance of existing models under the multi-perspective evaluation methodology (Section III, Section V-A and Section V-B).
3) We investigate the performance impact of the code abstraction approach on the model for the first time and provide suggestions for subsequent research work with the experimental results (Section V-C).

II. Data Preparation

There are mainly two kinds of datasets used in the code clone detection task, one is the datasets derived from actual projects represented by BigCloneBench [19], and the other is derived from programming competitions represented by GoogleCodeJam [11] and OJClone [20].

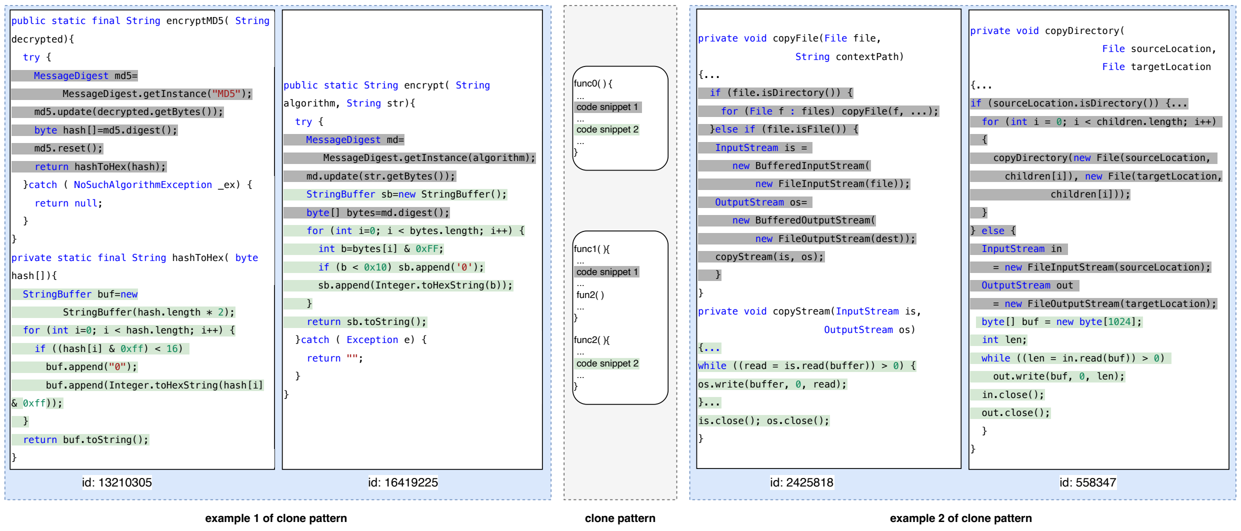
However, existing datasets for code clone detection suffer from a serious imbalance in the distribution of samples among different functionalities, For instance, if a dataset contains code with three functionalities, with a sample size ratio of 0.8:0.1:0.1, then the ratio of clone pairs sampled by these three functionalities, according to the principle that code snippets with the same functionality are clone pairs, would be 0.64:0.01:0.01. Consequently, the dataset's imbalance problem is amplified at the square level when collecting clone pairs, leading to very few clone pairs of functionalities with a small proportion of samples in the training and test data. This hinders the model's ability to learn all kinds of code and reduces its generalization ability. Additionally, the clone pairs in the test set only comprise a small number of large-scale functionality in the dataset, leading to an inaccurate evaluation of the model's performance. Although oversampling and weighted loss functions can mitigate this, they may lead to overfitting problems, and the other two existing datasets OJClone [20] and CodeSearchNET [21] with relatively balanced distribution have a large gap with the codes in real projects. For example, the number of unique variable names, frequency of function calls and code length in OJClone and CodeNET are 50% different from BigCloneBench (from real projects), while GoogleCodeJam is very close. Therefore, it is important to construct datasets with a balanced distribution that is close to real projects.

To address the limitations of these existing datasets, this paper introduces two new datasets, ConBigCloneBench and GoogleCodeJam2, which are based on BigCloneBench and GoogleCodeJam, respectively[1]. The detailed dataset construction methods are described below.

A. Confidence BigCloneBench

BigCloneBench is a dataset published by Roy et al. [19], [22] for code clone detection, which is one of the most frequently used datasets for deep learning models. The code snippets extracted from the IJDataset [23] contains 43 functionalities and about 1 million code clone pairs. For each functionality f in this dataset, they search the Internet with a designed string pattern to collect three sets: $S_{example}$ denotes the example code snippets that implement the target functionality f collected from community like StackOverflow;

[1]Datasets and codes: https://github.com/lihy11/MultiPerspectiveCloneEval



```
public static final String encryptMD5( String
decrypted){
  try {
    MessageDigest md5=
           MessageDigest.getInstance("MD5");
    md5.update(decrypted.getBytes());
    byte hash[]=md5.digest();
    md5.reset();
    return hashToHex(hash);
  }catch ( NoSuchAlgorithmException _ex) {
    return null;
  }
}
private static final String hashToHex( byte
hash[]){
  StringBuffer buf=new
           StringBuffer(hash.length * 2);
  for (int i=0; i < hash.length; i++) {
    if ((hash[i] & 0xff) < 16)
      buf.append("0");
      buf.append(Integer.toHexString(hash[i]
& 0xff));
  }
  return buf.toString();
}
```

id: 13210305

```
public static String encrypt( String
algorithm, String str){
  try {
    MessageDigest md=
      MessageDigest.getInstance(algorithm);
    md.update(str.getBytes());
    StringBuffer sb=new StringBuffer();
    byte[] bytes=md.digest();
    for (int i=0; i < bytes.length; i++) {
      int b=bytes[i] & 0xFF;
      if (b < 0x10) sb.append('0');
      sb.append(Integer.toHexString(b));
    }
    return sb.toString();
  }catch ( Exception e) {
    return "";
  }
}
```

id: 16419225

```
private void copyFile(File file,
                     String contextPath)
{...
  if (file.isDirectory()) {
    for (File f : files) copyFile(f, ...);
  }else if (file.isFile()) {
   InputStream is =
      new BufferedInputStream(
          new FileInputStream(file));
   OutputStream os=
      new BufferedOutputStream(
          new FileOutputStream(dest));
   copyStream(is, os);
   }
}
private void copyStream(InputStream is,
                        OutputStream os)
{...
while ((read = is.read(buffer)) > 0) {
os.write(buffer, 0, read);
}...
is.close(); os.close();
}
```

id: 2425818

```
private void copyDirectory(
                      File sourceLocation,
                      File targetLocation
{...
if (sourceLocation.isDirectory()) {...
  for (int i = 0; i < children.length; i++)
  {
    copyDirectory(new File(sourceLocation,
      children[i]), new File(targetLocation,
              children[i]));
  }
} else {
  InputStream in
    = new FileInputStream(sourceLocation);
  OutputStream out
    = new FileOutputStream(targetLocation);
  byte[] buf = new byte[1024];
  int len;
  while ((len = in.read(buf)) > 0)
    out.write(buf, 0, len);
  in.close();
  out.close();
  }
}
```

id: 558347

Fig. 2. Code clone pattern and its examples in BigCloneBench. The left and right sides in the figure show two pairs of code clone samples in BigCloneBench, which belong to different functionalities. These samples show that BigCloneBench is a multi-function dataset (described in Section II-A), and the clone pattern (as shown in the middle of the figure) is not related to the functionality (described in Section III-C).

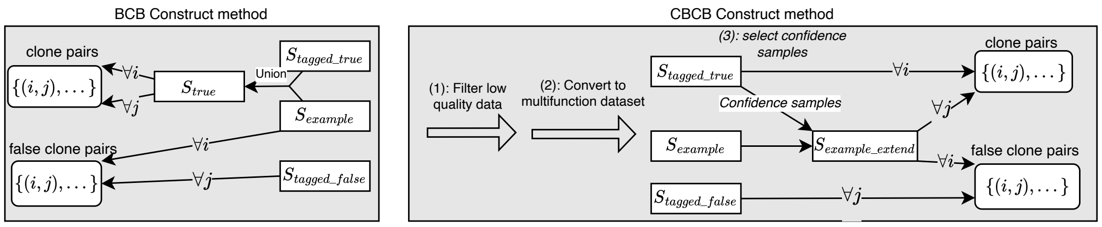


Fig. 3. The Construction of BigCloneBench (BCB) and ConBigCloneBench (CBCB)

S_{tagged_true} denotes code snippets marked as positive that implement the target functionality f collected from IJDataset; S_{tagged_false}: marked as negative code snippets of the functionality f collected from IJDataset. Let S_{true} be the union of $S_{example}$ and S_{tagged_true} which is the code snippets set that implements the functionality f. The clone pairs are obtained by pairwise matching of code snippets in S_{true}, totaling $N(S_{true})*(N(S_{true})-1)/2$ pairs, and the false-clone pairs are obtained by by pairwise matching of code snippets in $S_{example}$ and S_{tagged_false}, totaling $N(S_{example})*N(S_{tagged\_false})/2$ pairs.

The BigCloneBench used in previous work has other 3 issues in addition to distribution problem:

1) The BigCloneBench dataset contains some untrustworthy data marked by only one person [15], which has a negative impact on the authenticity of the dataset;
2) We found that previous work used BigCloneBench as a single-function clone dataset. But 30% of the functionalities are implemented by multiple functions. Figure 2 shows two example of multi-function code snippets in BigCloneBench, and on the left side of both examples, code snippets are split into two functions to complete a target functionality. If we consider BigCloneBench as a single function dataset like previous work [9], [11], [12], both examples will be non-clone pair;
3) Code snippets in S_{tagged_true} are not necessarily clones of each other according to Krinke's observation. Because these code snippets may implement more functionalities than the samples, these data are mainly contained in the Weakly Type-3/Type-4 clones [15].

In addition, previous work [9], [11], [12] did not use the proper method for constructing clone and non-clone pairs in BigCloneBench but introduced the wrong assumption that "code snippets between different categories are false-clone pairs", because similar functionalities like delete folder and delete files may share same code snippets [15]. This reduced the reliability of model's performance.

To address the problems above, we construct a new trusted multi-function dataset Confidence-BigCloneBench (ConBigCloneBench or CBCB), based on the original data of BigCloneBench as shown in Figure 3. In order to improve the usability of the datasets for future research, we have provided processed code data and clone pair data that is easy to use. The main steps of constructing the dataset are as follows:

1) Remove all data marked by single person in BigCloneBench, only keep the data which two markers agree on. we use this step to increase the credibility of the tagged code snippets and resolve issue (1).

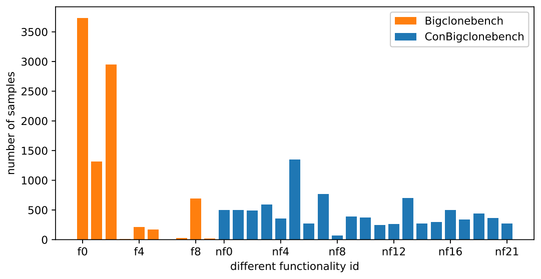


Fig. 4. The sample distribution between each functionality of the BigCloneBench and ConBigCloneBench, where f0-f9 refers to functionality id in the original dataset and nf0-nf21 refers to functionality id in the new dataset. Total count is 9323 for ConBigCloneBench, and 9313 for BigCloneBench.

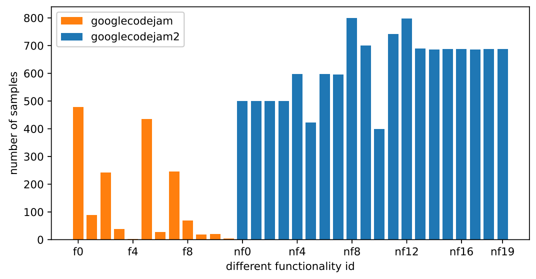


Fig. 5. The sample distribution of GoogleCodeJam and GoogleCodeJam2 between each functionality, where f0-f12 refers to functionality id in the original dataset and nf0-nf20 refers to functionality id in the new dataset. Total count is 12460 for GoogleCodeJam2 and 1665 for GoogleCodeJam

2) Use function call graph to analyze all called functions by the entry function, and expand the single function code snippet to a multi-function code snippet. we use this step to make BigCloneBench a multi-function dataset and resolve issue (2).
3) Select the functions whose text similarity[2] is greater than a threshold[3] θ in S_{tagged_true}, merge with $S_{example}$ and remove them from S_{tagged_true} at the same time as shown in Figure 3 (3) . We use this step to remove the false positive clone pairs introduced by pairwise matching samples in S_{tagged_true} and keep enough clone pairs at the same time.
4) Randomly remove the categories with too many code snippets to keep the number relatively even among different functionalities. We keep each functionalities size to 500 code snippets.
5) Reconstruct the clone pairs and false-clone pairs as show in Figure 3. we use this step to reconstruct training and test data without the wrong assumption ”code snippets between different categories are false-clone pairs” previous work introduced.

Figure 4 shows the distribution of the number of functionalities in BigCloneBench used in previous studies and ConBigCloneBench . ConBigCloneBench has a more balanced distribution of samples, with an average of 444.0 samples per functionality and a variance of 256.6. On the other hand, BigCloneBench has an average of 913.3 samples per functionality and a higher variance of 1288.0. ConBigCloneBench has totally 9323 samples, which is bigger than BigCloneBench, which is 9133.

B. GoogleCodeJam2

The GoogleCodeJam dataset is a code clone detection dataset collected from programming competitions. The solutions of each problem are Type-4 clones of each other, and the solutions of different problems are negative samples of each other. The GoogleCodeJam dataset used in previous work is seriously imbalanced among functionalities (as shown in Figure 5). To fix this problem, we recollected the competition data between year 2013 to 2017, including problem information and contestant information, and then selected each question containing more than 500 solutions. We finally obtained a dataset with 12460 functions in 20 functionalities.

As depicted in Figure 5, the sample distribution of GoogleCodeJam2 is more balanced compared to the previous GoogleCodeJam dataset, with an average of 623.0 samples per category and a variance of 114.8, compared to an average of 138.8 samples per category and a variance of 163.4 in the previous dataset. This demonstrates an improvement in the sample distribution and a large increase in the total number of samples. GoogleCodeJam2 has totally 12460 samples, which is bigger than GoogleCodeJam, which is 9133.

III. Evaluation Methodology

Deep learning approaches usually split the dataset into training and test sets used in the training and evaluation phases respectively. The evaluation methodology described in this paper particularly refers to the split method used to obtain the training and test sets. Previous work usually use the “random-view” evaluation method, which randomly splits all code snippets in the dataset into training and test sets according to a pre-defined ratio.

Using random-view alone is not comprehensive for model evaluation. Yu et al. [14] show that there are significant differences in the variable names between different functionalities in the BigCloneBench dataset, which reminds us that random splitting may lead to the model paying too much attention to the variable names and code structure information of the code. This problem invalidates the model for new functionality types (including variable names and code structures). Therefore, we conduct an experiment in multi-perspectives for code clone detection models.

The details of the multi-perspective evaluation methodology are described below, including a total of four perspectives. For brevity, we use test set to represent both the validation set and test set in this Section.

[2]We use line similarity introduced by BigCloneBench.
[3]We select θ as 0.2 which can filter all false clone pairs resulted by pairwise matching samples in S_{tagged_true} as previous work found [15].

A. *Random-View*

Random-view is the most commonly used method in previous work. This method regards all code snippets in the dataset as homogeneous samples and randomly divides them into training and test sets, as shown in Figure 6. This evaluation methodology does not consider the impact of functionality implemented by the code snippets or its coding style on the performance of deep learning models.

This view corresponds to the scenarios where the training set and the test set obey a uniform distribution. In this case, the code functionality and coding style in the training set are similar to those in the test set.

B. *Cross-project-view*

The cross-project-view split method was used by Roy et al. [22] in the BigCloneBench dataset. This view is to group all code snippets according to the projects they belong to, and divide the dataset by projects. Code snippets belonging to the same project will only appear in one of the training set or test set, as shown in Figure 6. The intuition behind this view is that different projects may have different coding styles and habits, such as usage habits for different functions with similar functionalities, variable naming habits, etc.

The actual application scenario of this view is that the training set and the test set belong to different projects, but the functionalities between the projects are similar. In this case, the cross-project-view evaluation methodology can well evaluate the model's ability to handle different coding styles when training set and test set are similar in functionality.

C. *Cross-Functionality-View*

The cross-functionality-view evaluation is not commonly seen in previous work. The method is to group all code snippets according to the functionalities, and then divide the dataset by functionality. The code that implements a certain functionality only appears in one of the training set or test set, as shown in Figure 6.

The intuition behind the method is that the functionalities in the dataset may not cover all functionalities in practice. As software technology evolves, code implementing new functionalities is introduced. For instance, before deep learning became popular, most of the functionalities related to deep learning models in PyTorch, a framework for deep learning, were uncommon. Hence, we need to evaluate whether the model can learn the clone pattern on limited functionalities and generalize to more. Figure 2 displays two pairs of cloning codes, which belong to the "encryption algorithm" and "copy folder" functionality in BigCloneBench, respectively. The examples have different code structures and function calls, but the clone pattern can both be summarized as splitting the code that implements a specific functionality from one function into two functions. If our training data only contains the "encryption algorithm" on the left side of Figure 2, then cross-functionality-view can evaluate whether the existing model can predict the code clone on the right side of Figure 2 based on the training data.

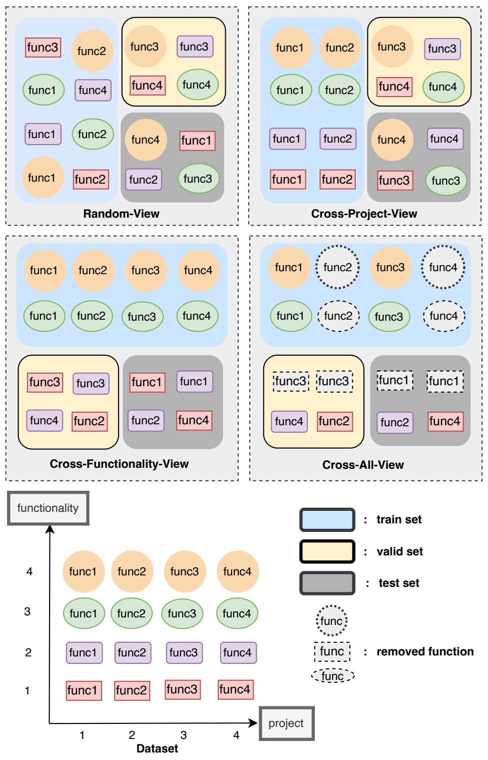


Fig. 6. Evaluation methodologies. The figure shows four training and test set construction methods. The blue rectangle represents the training dataset, the yellow rectangle represents the valid dataset, and the gray rectangle represents the testing dataset. One sample in the dataset (tagged as fun1-fun4) represents one code snippet, the samples in the same shape mean that they share the same functionality, and the samples with the same code snippet name mean that they belong to the same project.

D. *Cross-All-View*

In the previous subsections, we tried to divide the evaluation methods into cross-project-view and cross-functionality-view, but the two perspectives are not orthogonal. The more practical situation is that the data in the test set contains both projects and functionalities that are not in the training set. In order to evaluate the performance of the model in a real application scenario more effectively, we combine cross-project-view with cross-functionality-view as cross-all-view.

Take Figure 2 as an example, in the cross-functionality-view, we randomly selected the code snippets of functionality 3 and 4 (represented by orange and green) as training set, while the code of the other two functionalities is randomly split into test set and validation set. Then we randomly group code snippets according to projects, splitting the four projects in a 1:1 ratio. Project 1 and 3 (represented by func1 and func3) are grouped into training set, while the other two projects are grouped into test set and validation set. Finally, we remove the code snippets whose grouped sets are contradiction. In this example, we removed code snippets (represented by dotted gray samples) belonging to project 2 and 4 in training set, as well as the code snippets belonging to project 1 and 3 in test set and validation set.

IV. Experimental Studies

In this section we first introduce three state-of-the-art task-specific deep learning models and two pre-trained models used in our experiments and then designed the simple MLP model as an evaluation baseline. We conduct experiments to evaluate how the performance of existing work change under mult-perspective evaluation methodology. We also design experiments to analyze the impact of code abstraction methods (details in Section 4.3) on model performance.

We conduct experiments to answer the following research questions:

RQ1: How does different datasets affect the performance of different models?

RQ2: How does a multi-perspective evaluation methodology affect model's performance?

RQ3: How the code abstraction approach affects the performance of the model in different perspectives?

We use RQ1 to answer the performance of existing deep learning models on different datasets. We use RQ2 to answer the performance of existing deep learning models in different real-world application scenarios and to analyze the advantages and limitations of existing models. We use RQ3 to analyze the impact of the code abstraction approach on the performance of the models under different evaluation perspectives and to provide suggestions for subsequent research work.

A. *State-of-the-art approaches*

We selected three task-specific deep-learning models and two pre-trained models for experiments, including ASTNN [9], FA-AST [11], and TBCCD [12], GraphCodeBERT and UnixCoder. In addition, to evaluate the performance of the deep learning models, we also designed a very simple bag-of-words based MLP model (described in section 4.3) as a baseline.

1) ASTNN: ASTNN [9] is a code representation method based on AST. The method split the AST into subtree sequence, then use RNN to encode the sequence to obtain the final vector representation of the code.
2) FA-AST: FA-AST [11] is a code representation method based on AST. It enhances the representation of the AST by adding edges of control flow and data flow between the nodes of the AST, and then uses a Graph Matching Network (GMN) to learn the representation of code snippets.
3) TBCCD: TBCCD [12] is based on tree-based convolutional neural network, which combines a technique called "position-aware character embedding" (PACE) to reduce the impact of unseen words on the model.
4) GraphCodeBERT: GraphCodeBERT [24] is a pre-trained code model based on the BERT architecture. This model introduces the prediction of data flow edges and alignment of data flow variables as pre-training tasks, achieving the best performance on multiple downstream tasks. The model is publicly available[4].
5) UnixCoder: UnixCoder [25] is a unified cross-modal pre-trained model for programming languages, transforming Abstract Syntax Trees (AST) into a sequence. The model is publicly available[5].

B. *MLP Model*

To measure the effectiveness of existing deep learning models under multi-perspective evaluation methodology, we design a very simple MLP model. Specifically, we first parse the code segments into AST and then count all the words in the AST to construct a word list of the entire dataset, representing each code segment as a one-hot bag-of-words vector. A three-layer fully connected neural network with sigmoid activation is then used to learn the vector representation of that code segment, using cosine distance to measure whether two code segments are cloned pairs.

C. *Code abstraction methodology*

Yu et al. [14] experimentally demonstrated the effect of code variable names on model performance and proposed the dataset absBigCloneBench with abstract variable names. However, the code abstraction approach may have a diverse impact on the performance of the model, so we explore the effect of different code abstraction methods on model performance under a multi-perspective evaluation methodology. We took a total of four code abstraction methods. The details are as follows:

1) Level-0: Only the initial "package name" and "comment" of the code segment are removed. This level corresponds to unprocessed source code.
2) Level-1: On the basis of Level-0, replace all custom variable names in the code with "var1, var2...", in addition, replace all custom class names with "Type0, Type1...". This level corresponds to simple code obfuscation, that is, replacing user-defined information that does not affect the running of the program, which is common in Type-2 clones.
3) Level-2: On the basis of Level-1, replace all the object names in the code with "Obj1, Obj2, Obj3...". This level corresponds to further code obfuscation, replacing the type information in the code, which is common in Type-3 clones.
4) Level-3: On the basis of Level-2, extract all function calls from the whole dataset and rename them as "fun0, fun1...". Then replace the function calls in each code segment with the corresponding names. This level corresponds to the case of malicious code obfuscation, which is also a common scenario for code clone detection, i.e., actively replacing the custom function name.

D. *Experiment settings*

The construction of the training set and test set for each dataset in our experiments is as follows: for the ConBigCloneBench dataset, the random-view evaluation randomly

[4]https://huggingface.co/microsoft/graphcodebert-base

[5]https://huggingface.co/microsoft/unixcoder-base

TABLE I
F1 SCORE ON GOOGLECODEJAM2 AND CONBIGCLONEBENCH

	GoogleCodeJam2							ConBigCloneBench						
Model	r-view	cp-view		cf-view		ca-view		r-view	cp-view		cf-view		ca-view	
	F1	F1	Δ	F1	Δ	F1	Δ	F1	F1	Δ	F1	Δ	F1	Δ
ASTNN	0.913	0.908	0.005	0.401	0.512	0.393	0.52	0.924	0.912	0.012	0.677	0.247	0.607	0.317
TBCCD	0.832	0.813	0.019	0.522	0.31	0.496	0.336	0.905	0.886	0.019	0.816	0.089	0.808	0.097
FA-AST	0.842	0.84	0.002	0.517	0.325	0.505	0.337	0.917	0.899	0.018	0.733	0.184	0.769	0.148
GraphCodeBERT	0.876	0.712	0.164	0.285	0.591	0.262	0.614	0.931	0.921	0.010	0.675	0.246	0.564	0.367
UnixCoder	0.468	0.514	-0.048	0.405	0.063	0.413	0.055	0.943	0.940	0.003	0.778	0.162	0.789	0.154
MLP	0.449	0.557	-0.108	0.401	0.048	0.384	0.065	0.892	0.844	0.048	0.767	0.125	0.737	0.155

split the code segments according to 3:1:1 ratio, and then reconstructs the clone pair and non-clone pair data according to the original clone relationships. Cross-project-view evaluation splits all the projects according to 3:1:1 ratio, and then reconstructs the data according to the code segments contained in each set of projects. In cross-fun-view evaluation, we randomly select 14 functionalities as the training set, and all code segments of the remaining 7 functionalities are splitted into the validation set and the test set randomly. In the cross-all-view evaluation, we remove the overlapping projects between the training and testing sets in the cross-functionality-view for ConBigCloneBench as described in Section 3.4.

For the GoogleCodeJam2 dataset, random-view and cross-project-view split is the same as ConBigCloneBench. In cross-functionality-view evaluation, we select 12 functionalities randomly as the training set, and the code segments in the remaining 8 functionalities are randomly divided into the validation set and the test set. In the cross-all-view evaluation, we remove overlapping projects between the training and testing sets in the cross-functionality-view for GoogleCodeJam2 as described in Section 3.4.

For all models, we use the parameter configurations given in the original paper, stop training when the performance of the validation set drops three times continuously, and select the best results from the test set. For the MLP model, we set the hidden-dim as 100, out-dim=50, cosine-similarity-threshold=0, activation function is sigmoid. For the two pre-trained model, we set code length as 1000 for GoogleCodeJam2 and 500 for ConBigCloneBench which can cover 90% of codes without losing information. We set lr=2e-5 and the batchsize=6. We control the total amount of training data under each evaluation perspective to 400k pairs of clones, and testing data to 250k pairs of clones.

V. RESULTS AND ANALYSIS

In this section we answer the questions and analyze the experimental results.

A. RQ1:How does different datasets affect the performance of different models

1) Finding 1: The performance of the model under different data sets fluctuated greatly: We found that the performance of all deep-learning models on different datasets fluctuated greatly. As shown in the data in Table I, the performance of all models on ConBigCloneBench is much higher than that on GoogleCodejam2, and the pre-trained model is particularly prominent. The two pre-trained models can achieve the best performance on ConBigCloneBench, but the worst performance on GoogleCodeJam2. We think there are two main reasons: (1) The code functionalities in ConBigCloneBench are relatively simple, while the code functionalities in GoogleCodeJam2 are more complicated, so GoogleCodeJam2 is harder for deep-learning models to capture the semantic information. (2) The code in ConBigCloneBench is public available from github, so there may be data leakage problems for the pre-trained models training on CodeSearchNet [21]. **Therefore, we believe that ConBigCloneBench is not enough for evaluating deep-learning models, especially the pre-trained models. Future research should combine GoogleCodeJam2 to conduct more comprehensive experiments.**

B. RQ2: How does a multi-perspective evaluation methodology affect model's performance

Table I show the performance of each model under each perspective on abstraction Level 1. We answer this question to analyze the performance of existing state-of-the-art models under various perspectives and point out the strengths and weaknesses of the models.

1) Finding 2: Cross-view performance of each model is significantly lower than random-view: From the data in Table I, it can be found that the cross-view performance of each model is significantly lower than that of random-view. The cross-view here includes all perspectives except random-view. Among them, the gap between cross-project-view and random-view is the smallest, the average drop rate on GoogleCodeJam2 is 0.009, and the drop rate on ConBigCloneBench is 0.017. The gap between cross-all-view and random-view is the largest, the average drop rate on GoogleCodeJam2 is 0.398, and the average drop on ConBigCloneBench is 0.187.

These experimental results show that the performance of the model in each cross-view is significantly lower than that in random-view, and the drop can reach up to 0.5 F1 score. This means that previous work using random-view to evaluate the performance of the models is not accurate and comprehensive enough. The huge performance degradation of the models under cross-functionality-view and cross-all-view also means that the models did not accurately capture the judgment principle of code cloning during the learning process as suggested by random-view results, but may have paid too much attention to the code structure, variable names and other information. Therefore, the rules learned by the model on some code snippets do not work on code of different functionalities.

In addition, the decrease in cross-functionality-view and cross-all-view on the ConBigCloneBench dataset (0.247 maximum) is much smaller than that on the GoogleCodeJam2 dataset (0.325 minimum), which is probably related to the type of dataset. The code in the ConBigCloneBench dataset implements a simple function, such as copying files, and the negative samples usually contain completely different code functions. The gap between negative samples and positive samples is large. The GoogleCodeJam2 dataset is a competition dataset, and the functions implemented by its code segments are usually more complex than those in ConBigCloneBench. In GoogleCodeJam2, there are more similar structures and variables between different functionality code snippets, so it is more difficult to judge the functionality of the code.

2) Finding 3: The performance under cross-functionality-view of some models is similar to the MLP model: In order to evaluate the effectiveness of existing models under the multi-view evaluation method, we design a simple MLP model (described in Section IV-B) as a baseline model to evaluate the other deep learning models. We tested the performance of the MLP model in each evaluation perspective, and the experimental results are shown in Table I . The data results show that the performance of the existing state-of-the-art model in both random-view and cross-project-view far exceeds that of the MLP model, which means that the existing models do capture effective code structure information.

However, under cross-functionality-view and cross-all-view, the improvement of state-of-the-art compared with the MLP model is small, and some models even perform worse than MLP model. For example, ASTNN achieved a performance of 0.401 F1 score on GoogleCodeJam2 , just as the same as the MLP model. On ConBigCloneBench, the F1 score of ASTNN is 0.677, and the F1 score of FA-AST is 0.733, both of which are smaller than the 0.767 F1 score of the MLP model, The same is true for GraphCodeBERT.

The semantic information of the code, including control flow and data flow, is important for identifying the code functionality in ConBigCloneBench and the programming problem solved by the code in GoogleCodeJam2. The experimental results may indicate that the existing complex models are limited in improving the ability to capture program semantics, which are important to recognize the functionality of code snippets.

It should be noted that each model contains the information of the words fed to the MLP model, and also has a larger amount of parameters, which means that some models do not capture the semantic information of the program to improve the ability of clone detection. More seriously, they suffer from overfitting problems, where the large number of parameters causes the model to fit the training set too closely, resulting in worse performance than simpler models on the same test data. Therefore, the current state-of-the-art models mainly improve the ability to recognize variable names and code structures but is relatively insufficient in capturing code semantics and patterns.

3) Finding 4: Different code representations and model architectures show significant performance fluctuations under different evaluation perspectives.: We found that the performance of existing state-of-the-art models depends on the choice of evaluation perspective and dataset. As shown in Table I, in the perspective of random-view and cross-project-view, the performance of ASTNN and GraphCodeBERT is significantly better than other work, both in GoogleCodeJam2 and ConBigCloneBench. However, the performance of this work is very poor under cross-functionality-view and cross-all-view, and the performance is similar to that of the MLP model.

In contrast, TBCCD performed poorly under cross-project-view and random-view, while in cross-functionality-view and cross-all-view, it is significantly better than the other two methods, and in all cases it was significantly higher than the MLP model performance. The reason may be that the TBCCD method performs a well-designed embedding called PACE for the variable names in the dataset, which makes it perform better in cross-functionality-view, but at the same time, the performance of the model in random-view declined.

FA-AST can better balance the performance of multiple viewing angles. While ensuring that the performance of random-view does not drop too much, the performance of cross-view is significantly higher than that of ASTNN. This is because FA-AST not only uses AST to represent structural information of the program in code representation but also uses control flow and data flow to represent the semantic information of the program.

While UnixCoder only can achieve best performance on ConBigCloneBench. These results show that the existing models have their own suitable evaluation perspectives and data types due to the use of different code representations and deep learning models. Therefore, a multi-perspective evaluation method is important to compare various models more fairly.

C. RQ3: How the code abstraction approach affects the performance of the model in different perspectives

Tables II and Table III show the performance (F1 score) of each model under various evaluation perspectives when using different code abstraction methods. We answer this question to analyze how code abstraction methods and multi-view evaluation methods affect the performance of clone detection models and provide recommendations for subsequent research work based on experimental analysis.

1) Finding 5: As the abstraction level increases, the performance of each model under random-view shows a decreasing trend: We found that as the code abstraction level increases, the performance of each model under random-view decreases significantly, as shown in Table II. Among them, the decline of TBCCD and FA-AST is more pronounced, while the decline of ASTNN is minimal, whose worst performance is still better than TBCCD and FA-AST. This is because, in random-view, the performance of the model is greatly affected by the words

TABLE II
F1 SCORE OF DIFFERENT CODE ABSTRACTION LEVEL IN RANDOM-VIEW AND CROSS-PROJECT-VIEW

		GoogleCodeJam2					ConBigCloneBench				
		ASTNN	TBCCD	FAAST	GraphCodeBERT	UnixCoder	ASTNN	TBCCD	FAAST	GraphCodeBERT	UnixCoder
r-view	level0	**0.938**	**0.889**	**0.919**	**0.827**	**0.844**	**0.924**	**0.917**	**0.917**	**0.934**	**0.95**
	level1	0.913	0.832	0.842	0.684	0.468	0.924	0.905	0.917	0.931	0.943
	level2	0.911	0.844	0.827	0.61	0.484	0.923	0.893	0.907	0.922	0.941
	level3	0.902	0.8	0.847	0.323	0.432	0.917	0.876	0.91	0.911	0.923
cp-view	level0	**0.939**	**0.876**	**0.912**	**0.876**	**0.867**	**0.922**	**0.901**	0.875	**0.923**	**0.946**
	level1	0.908	0.813	0.84	0.712	0.514	0.912	0.886	**0.899**	0.921	0.94
	level2	0.911	0.82	0.862	0.701	0.339	0.905	0.82	0.889	0.91	0.93
	level3	0.904	0.782	0.838	0.487	0.450	0.894	0.798	0.888	0.903	0.913

TABLE III
F1 SCORE OF DIFFERENT CODE ABSTRACTION LEVEL IN CROSS-FUNCTIONALITY-VIEW AND CROSS-ALL-VIEW

		GoogleCodeJam2					ConBigCloneBench				
		ASTNN	TBCCD	FAAST	GraphCodeBERT	UnixCoder	ASTNN	TBCCD	FAAST	GraphCodeBERT	UnixCoder
cf-view	level0	0.412	0.492	0.467	**0.392**	0.402	0.505	0.823	0.677	0.706	0.767
	level1	0.401	**0.522**	0.517	0.285	0.405	**0.677**	0.816	0.733	0.675	**0.778**
	level2	0.426	0.521	**0.54**	0.273	**0.432**	0.583	0.82	**0.736**	**0.714**	0.751
	level3	**0.491**	0.497	0.515	0.247	0.377	0.577	**0.826**	0.688	0.682	0.773
	max Δ	0.079	0.03	0.073	-0.107	-0.025	0.172	0.003	0.059	0.008	0.009
ca-view	level0	**0.411**	0.475	0.515	**0.334**	**0.499**	0.637	0.808	**0.791**	**0.579**	**0.800**
	level1	0.393	0.496	0.505	0.262	0.413	0.607	0.808	0.769	0.564	0.789
	level2	0.429	**0.5**	0.524	0.237	0.471	0.63	0.812	0.76	0.578	0.709
	level3	0.403	0.473	**0.53**	0.245	0.403	**0.688**	**0.823**	0.753	0.61	0.787

in the code [14]. ASTNN only retains about 3,000 high-frequency words in the data for its vocabulary table during training, while TBCCD and FA-AST will ultimately retain all words in datasets which is about 50,000. Code abstraction will significantly reduce the vocabulary in the dataset from about 50,000 to 30,000. Therefore, ASTNN performed better than the other two models on the abstracted code.

The above experimental results show that the high performance of the existing state-of-the-art model in random-view is related to words, and reducing the input vocabulary size of the model can effectively reduce the model's performance in random-view. This also shows that in the random-view scenario, especially the random-view scenario of complex code (such as the code in the GoogleCodeJam2 dataset), the code abstraction method will seriously affect the performance of the model, so we should choose the code abstraction method more carefully.

2) Finding 6: Code abstraction is effective for improving the performance under cross-functionality-view and cross-all-view for task-specific models.: Table III show the performance of models at different abstraction level. We found that the performance of the task-specific models under both cross-functionality-view and cross-all-view improve significantly as the code abstraction level increase. Δ shows that the maximum increase is 0.08 F1 score (from 0.412 to 0.491). This indicates that the difference between the identifiers and function names can weaken the model's generalization ability in cross-functionality-view and cross-all-view. This is effectively mitigated by increasing the abstraction level of the code. This reminds us that when applying the model to cross-functionality-view and cross-all-view scenarios, we need to choose the abstracted code to obtain the best performance.

However, code abstractions hardly improve the performance of pre-trained models. This is because the input of the pre-training model is word sequence. **Although the two pre-trained models we selected use the syntactic and semantic information for pre-training, they cannot recover the high-level structure and semantic information of the code sequence well at inference step, so the pre-trained models perform worse on abstracted code**.

3) Finding 7: Well-designed embeddings of variable names can effectively improve the performance of the model in cross-functionality-view: We found that TBCCD not only showed significantly better performance in the cross-functionality-view perspective, but also improved slightly compared to the other two models at different code abstraction Levels, shown as Δ in Table III. We think this may be due to TBCCD adding a word embedding method called PACE to the model design. This method can make the model better cope with words that have not been seen. This reminds us that a more effective embedding method for words is possibly an important research direction to improve the performance of the model in the cross-functionality-view scenario.

VI. RELATED WORK

A. Code clone detection

Code clones refers to two semantically identical code that differ syntactically to various degrees. Code clone detection is important in a lot of software engineering tasks, such as defect detection, code search, code refactoring, etc. According to different code representation methods, code clone detection methods can be divided into text-based, token-based [6]–[8], [26], syntax-based [9], [10], [12], [20], [27]–[31] and semantic-based [11], [32], [33], among which text-based and token-based approaches can detect Type-1 and Type-2 clones. CCAligner [6], CCFinder [7] and SourceCC [8] detect code clones by detecting the similarity of tokens between code segments. SEGA [26] further improves the matching speed of token subsequences between code segments, and improves the efficiency of clone detection algorithms in large-scale software systems. The above algorithms cannot detect Type-3 and Type-4 clones because losing syntax and semantic information.

The syntax-based approaches identity code clones by capturing the syntax structure information, and can detect Type-1, Type-2 and most of Type-3 code clones. TBCNN [20] and TBCCD [12] try to use tree-based convolutional neural network model for code clone detection. Zhang et al. [9], White et al. [28] and Lutz et al. [29] try to convert AST to sequence and use recurrent neural network models to detect code clones. CDLH [10] uses the Tree-LSTM model to capture code information in the abstract syntax tree, uses the hash function to predicts the code clones according to the vector distance.

Semantic-based approaches detect code clones by capturing the semantic information of code (the functionalities of code). This type of method needs to parse the source code into representations such as program dependency graphs and then use deep learning models to learn the program semantics in the data. Deepsim [33] parses the program's control flow graph and data flow graph into an adjacency matrix, and then uses a deep learning model to learn semantic information to predict code cloning. Fang et al. [32] proposed a cross-function clone detection method by constructing a cross-function control flow graph combining with AST. FA-AST [11] uses data flow and control flow information to enhance the representation ability of AST, and then uses a graph neural network to detect clones. Tailor [34] uses CPG and CPGNN to capture the semantic information of source code.

Code clone detection is also often used as one of the downstream tasks of the pre-trained code model [24], [25], [35], and the existing pre-training model has achieved excellent performance [36]–[38]. GraphCodeBERT [24] uses the data stream recovery task as a pre-training task to further enhance the performance of the pre-trained code model. UnixCoder [25] is a multi-language pre-trained code model that combines multiple model architectures.

The above approaches achieved good performance on the task of code clone detection [39], but the quality of the dataset and single evaluation method have brought certain challenges to the credibility of the results. Our experimental results show that existing clone detection approaches are indeed overestimated in some evaluation perspectives.

B. Evaluation methodology

To the best of our knowledge, our work is the first to focus on the evaluation method of code clone detection models. However, in other cross-fields of software engineering and deep learning, there have been several work that focus on the evaluation methods of deep learning models. In code summarization, Nie et al. [17] studied the performance of code in different development cycles on code summarization models, while LeClair et al. [16] explored the performance of code summarization models under cross-project-view. In neural language processing, Bender et al. [40] also pointed out that deep learning models will produce significant performance deviations in practical scenarios. Overall, these work indicate that the lack of evaluation method research poses a challenge to the credibility of the model's performance, and it is necessary to explore the evaluation method and corresponding practical application scenarios of the model.

C. Code clone dataset

With the rise of large-scale open-source code, researchers have also proposed several datasets for code clone detection for evaluation and training.

BigCloneBench is a code clone detection dataset proposed by Roy et al. [19] The code in this dataset is extracted from the project dataset IJDataset.

Yu et al. [14] proposed the dataset named AbsBigCloneBench to address the variable name issue in BigCloneBench. Mou et al. [20] proposed the OJClone dataset, which is extracted from student homework on website, and considered code that solves the same problem as clones.

Although previous work has focused on the quality of code clone datasets. Yu et al. [14] focused on the variable names in the dataset, and Krinke et al. [15] focused on the low-quality data in BigCloneBench. Different from previous work, we propose two new datasets with more uniform distribution and higher reliability.

VII. Conclusion

In this paper, we conduct experiments from multi-perspectives for code clone detection model evaluation, and release two new code clone detection datasets: ConBigCloneBench and GoogleCodeJam2, to address the problems in existing code clone detection datasets. We conduct experiments to analyze the performance of existing deep learning models under the multi-perspective evaluation methodology and design a simple MLP model to evaluate the effectiveness of the above models. Our experimental results show that the existing optimal models perform poorly in cross-functionality-view and cross-all-view, and some models are only close to the performance of MLP model. Meanwhile, the performance of deep-learning models fluctuate greatly on different datasets. Through the analysis of the experimental results and the characteristics of the model itself, we point out that the code semantic information used in FA-AST and the word embedding technology used in TBCCD can effectively improve the performance of the model from the perspective of partial evaluation.

We hope that our work can inspire follow-up research on the evaluation perspective of code clone detection models: researchers should choose the evaluation methodology more carefully instead of simply using random-view to evaluate the performance of the model, which improves the model reliability and application in real scenarios.

References

[1] C. K. Roy and J. R. Cordy, "A mutation/injection-based automatic framework for evaluating code clone detection tools," in *2009 International Conference on Software Testing, Verification, and Validation Workshops*, Apr 2009, p. 157–166.

[2] M. Mondal, M. S. Rahman, C. K. Roy, and K. A. Schneider, "Is cloned code really stable?" *Empirical Software Engineering*, vol. 23, no. 2, p. 693–770, Apr 2018.

[3] M. Mondal, C. K. Roy, and K. A. Schneider, "Does cloned code increase maintenance effort?" in *2017 IEEE 11th International Workshop on Software Clones (IWSC)*, Feb 2017, p. 1–7.

[4] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan, "An empirical study on inconsistent changes to code clones at release level," in *2009 16th Working Conference on Reverse Engineering*, Oct 2009, p. 85–94.

[5] C. Roy and J. Cordy, "A survey on software clone detection research," *School of Computing TR 2007-541*, 01 2007.

[6] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy, "Ccaligner: a token based large-gap clone detector," in *Proceedings of the 40th International Conference on Software Engineering*. Gothenburg Sweden: ACM, May 2018, p. 1066–1077. [Online]. Available: https://dl.acm.org/doi/10.1145/3180155.3180179

[7] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, p. 654–670, Jul 2002.

[8] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, May 2016, p. 1157–1168.

[9] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. Montreal, QC, Canada: IEEE, May 2019, p. 783–794. [Online]. Available: https://ieeexplore.ieee.org/document/8812062/

[10] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*. Melbourne, Australia: International Joint Conferences on Artificial Intelligence Organization, Aug 2017, p. 3034–3040. [Online]. Available: https://www.ijcai.org/proceedings/2017/423

[11] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. London, ON, Canada: IEEE, Feb 2020, p. 261–271. [Online]. Available: https://ieeexplore.ieee.org/document/9054857/

[12] H. Yu, W. Lam, L. Chen, G. Li, T. Xie, and Q. Wang, "Neural detection of semantic code clones via tree-based convolution," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 2019, p. 70–80.

[13] M. Lei, H. Li, J. Li, N. Aundhkar, and D.-K. Kim, "Deep learning application on code clone detection: A review of current knowledge," *Journal of Systems and Software*, vol. 184, p. 111141, Feb 2022.

[14] H. Yu, X. Hu, G. Li, Y. Li, Q. Wang, and T. Xie, "Assessing and improving an evaluation dataset for detecting semantic code clones via deep learning," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 4, pp. 62:1–62:25, Jul 2022.

[15] J. Krinke and C. Ragkhitwetsagul, "Bigclonebench considered harmful for machine learning," in *2022 IEEE 16th International Workshop on Software Clones (IWSC)*, Oct 2022, p. 1–7.

[16] A. LeClair and C. McMillan, "Recommendations for datasets for source code summarization," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun 2019, p. 3931–3937. [Online]. Available: https://aclanthology.org/N19-1394

[17] P. Nie, J. Zhang, J. J. Li, R. Mooney, and M. Gligoric, "Impact of evaluation methodologies on code summarization," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Dublin, Ireland: Association for Computational Linguistics, May 2022, p. 4936–4960. [Online]. Available: https://aclanthology.org/2022.acl-long.339

[18] S. Wang, T. Liu, J. Nam, and L. Tan, "Deep semantic feature learning for software defect prediction," *IEEE Transactions on Software Engineering*, vol. 46, no. 12, pp. 1267–1293, 2020.

[19] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *2014 IEEE International Conference on Software Maintenance and Evolution*. Victoria, BC, Canada: IEEE, Sep 2014, p. 476–480. [Online]. Available: http://ieeexplore.ieee.org/document/6976121/

[20] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Thirtieth AAAI conference on artificial intelligence*, 2016.

[21] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," no. arXiv:1909.09436, Jun 2020, arXiv:1909.09436 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1909.09436

[22] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with bigclonebench," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep 2015, p. 131–140.

[23] A. S. E. Group, "Ijadataset 2.0," Jan 2013.

[24] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," no. arXiv:2009.08366, Sep 2021, arXiv:2009.08366 [cs]. [Online]. Available: http://arxiv.org/abs/2009.08366

[25] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," no. arXiv:2203.03850, Mar 2022, arXiv:2203.03850 [cs]. [Online]. Available: http://arxiv.org/abs/2203.03850

[26] G. Li, Y. Wu, C. K. Roy, J. Sun, X. Peng, N. Zhan, B. Hu, and J. Ma, "Saga: Efficient and large-scale detection of near-miss clones with gpu acceleration," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. London, ON, Canada: IEEE, Feb 2020, p. 272–283. [Online]. Available: https://ieeexplore.ieee.org/document/9054832/

[27] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," Oct 2006, p. 253–262.

[28] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sep 2016, p. 87–98.

[29] L. Buch and A. Andrzejak, "Learning-based recursive aggregation of abstract syntax trees for code clone detection," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Hangzhou, China: IEEE, Feb 2019, p. 95–104. [Online]. Available: https://ieeexplore.ieee.org/document/8668039/

[30] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, "Oreo: detection of clones in the twilight zone," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Lake Buena Vista FL USA: ACM, Oct 2018, p. 354–365. [Online]. Available: https://dl.acm.org/doi/10.1145/3236024.3236026

[31] K. W. Nafi, T. S. Kar, B. Roy, C. K. Roy, and K. A. Schneider, "Clcdsa: Cross language code clone detection using syntactical features and api documentation," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. San Diego, CA, USA: IEEE, Nov 2019, p. 1026–1037. [Online]. Available: https://ieeexplore.ieee.org/document/8952189/

[32] C. Fang, Z. Liu, Y. Shi, J. Huang, and Q. Shi, "Functional code clone detection with syntax and semantics fusion learning," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, Jul 2020, p. 516–527. [Online]. Available: https://doi.org/10.1145/3395363.3397362

[33] G. Zhao and J. Huang, "Deepsim: deep learning code functional similarity," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Lake Buena Vista FL USA: ACM, Oct 2018, p. 141–151. [Online]. Available: https://dl.acm.org/doi/10.1145/3236024.3236068

[34] J. Liu, J. Zeng, X. Wang, and Z. Liang, "Learning graph-based code representations for source-level functional similarity detection."

[35] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov 2020, p. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139

[36] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang, "An extensive study on pre-trained models for program understanding and generation," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. Virtual South Korea: ACM, Jul 2022, p. 39–51. [Online]. Available: https://dl.acm.org/doi/10.1145/3533767.3534390

[37] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," no. arXiv:2107.03374, Jul 2021, arXiv:2107.03374 [cs]. [Online]. Available: http://arxiv.org/abs/2107.03374

[38] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," no. arXiv:2102.04664, Mar 2021, arXiv:2102.04664 [cs]. [Online]. Available: http://arxiv.org/abs/2102.04664

[39] J. K. Siow, S. Liu, X. Xie, G. Meng, and Y. Liu, "Learning program semantics with code representations: An empirical study," no. arXiv:2203.11790, Mar 2022, arXiv:2203.11790 [cs]. [Online]. Available: http://arxiv.org/abs/2203.11790

[40] E. M. Bender, T. Gebru, A. McMillan-Major, and S. Shmitchell, "On the dangers of stochastic parrots: Can language models be too big?" in *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, ser. FAccT '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 610–623. [Online]. Available: https://doi.org/10.1145/3442188.3445922