Attempts on Finding Cross-Language Code Clones based on Text and AST Information

DAI Feng, Shigeru CHIBA

Currently, there are no concrete definitions of what is a code clone. It could refer to a pair of code pieces that have similar structures, or it could also refer to a pair of code pieces that implement similar functionality. Such ambiguity is especially significant under cross-language scenario, as different programming languages have different syntax and grammar. However, most existing works do not recognize the ambiguity, and evaluate their approaches on inappropriate datasets, making their approaches for cross-language code clone detection unreliable. In this paper, we develop several evaluation procedures and provide better datasets for robust evaluation of cross-language code clone detection task. To be specific, we focus on detecting method-level code clones instead of entire code files. We also develop our model based on two similarity measurements, and use non-dominated ranking system to select optimal code clone pairs. Our approach can provide better performance under our new evaluation procedures.

1 Introduction

Code clone detection is a valuable research topic, as it can help to refactor and improve the maintainability of large code bases. Duplicated code fragments can be difficult to maintain if there is an error in such code fragments. Nowadays, the number of programming languages is increasing, and a system can be implemented in multiple programming languages. Under such scenario, repetitions of source code functionality in different programming languages are unavoidable. Therefore, an effective cross-language code clone detection tool is desirable.

When we say code clones, we are normally talking about two fragments of source code, either in same language or different languages, that are syntactically or semantically similar to each other. However, there is no consensus about what is the concrete definition of a code clone in academia so far. [6] gives an explanation of four types of code clones. But they are more like an induction instead of a definition. This leads to a disadvantage of current literature tackling this problem.

However, when researchers try to expand the research to cross-language scenario, the disadvantage becomes more significant. As different programming languages have different syntax and grammar, it is difficult to say what is similar across two code pieces. Is a pair of code pieces sharing same structure a code clone? Or do they have to share same functionality to be called a code clone? It can be seen that there is not a clear objective about what to detect when it comes to detecting cross-language code clones. Unfortunately, most existing works do not recognize the problem, making their detection models unreliable.

There are two main reasons for the unreliability. First, existing approaches, whether machine learning based or non-machine learning based, base their evaluation on competitive programming datasets,

戴峰、千葉滋, 東京大学情報理工学系研究科, Graduate School of Information Science and Technology, The University of Tokyo.

such as Leetcode or AtCoder. This kind of datasets usually contains solutions to many individual algorithm problems. The solutions are implemented by different programmers, and do not necessarily share similar structures or even functionality. However, existing approaches treat such pairs of solutions as clones. This is inappropriate and unreliable. Second, most approaches calculate similarity score based on single similarity measurement such as text-similarity or AST-similarity. They either treat source code as simple natural language text and symbols, or do not consider different abstract syntax grammar for different languages. Furthermore, they naively apply machine learning models for the calculation.

In this paper, we present two main contributions for this topic. First, we come up with new evaluation procedures and develop new datasets to avoid the drawbacks of competitive programming datasets. Current competitive programming dataset uses entire code files, but we focus on method-level code pieces. Our datasets are more appropriate for code clone detection evaluation. Second, we design a new multi-modal model. We consider both text-based similarity and AST-based similarity, and train each machine learning model individually. The models have clear objectives about what is similar across two code pieces. After getting similarity scores, we use non-dominated ranking system to select optimal code clone pairs, so that each similarity measurement is equally considered. We believe our model works better under new evaluation procedures.

The rest of the paper is organized as follow. In section 2, we will introduce the motivation of our research. In section 3, we will introduce the overall structure of our model and each component of our model. In section 4, we will introduce the evaluation procedures and how we organize the datasets. In section 5, we will show some existing literature in the field.

```java
public static List<List<Record>> groupRecordsByUsers(List<Record>
   records){
    Collections.sort(records, (r1, r2) -> r1.getTime() - r2.
       getTime());
    HashMap<Integer, List<Record>> grouped = new HashMap<>();
    for (Record r: records){
        if (!grouped.containsKey(r.getUserId())){
            grouped.put(r.getUserId(), new ArrayList<Record>());
        }
        grouped.get(r.getUserId()).add(r);
    }
    List<List<Record>> result = new ArrayList<>();
    for (List<Record> value: grouped.values()){
        result.add(value);
    }
    Collections.sort(result, (r1, r2) ->
        r2.get(r2.size() - 1).getTime() - r1.get(r1.size() - 1).
           getTime();
    );
    return result;
}
```

図 1 Java code of groupRecordsByUsers()

2 Motivation

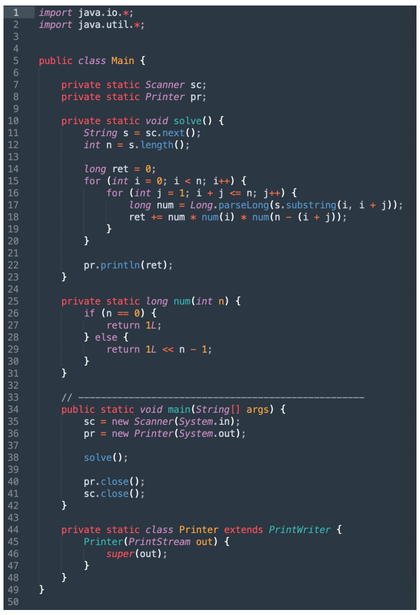
One of the most common scenarios that code clones happen is that programmers use multiple programming languages in a single application project. For example, programmers may use JavaScript for Web client and switch to Java for mobile client. We present a simple case which is often encountered in real development. In the development of a shopping record management system, we want to add a feature to group shopping records by users' id, and sort the groups by the time of most recent record in each group. Moreover, in each group, all the records are sorted by the time of each record. That is, we want a list of groups as a result. All the records in the same group are from the same user. Both the groups and the records in each group are sorted by time. Since this feature is not part of the initial design, we only have an API that returns a list of unsorted records from different users in the form of an HTTP GET endpoint. As the API is restricted, we need to implement the feature from the front end side. We show the implementation in JavaScript and Java in Fig.2 and Fig.1. JavaScript code is for Web clients and Java code is for Android clients. In the code, the logic is exactly the same even there are some

```javascript
function groupRecordsByUsers(records){
    const sortedRecords = records.sort((a, b) => a.time - b.time);
    const grouped = {};
    for (const record of sortedRecords){
        if (!grouped[record.userId]){
            grouped[record.userId] = [];
        }
        grouped[record.userId].push(record);
    }
    const groups = [];
    for (const key in grouped){
        groups.push(grouped[key]);
    }
    return groups.sort((a, b) => b[b.length-1].time - a[a.length-1].time);
}
```

図 2 JavaScript code of groupRecordsByUsers()

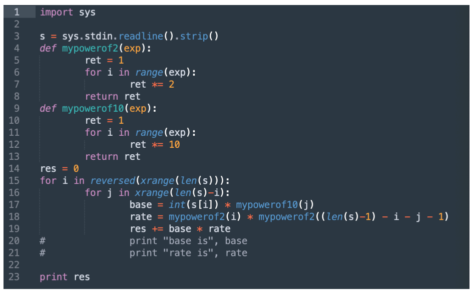
semantic differences. We first sort all the records by time in the list of records. Then we initialize a list of groups and assign each record to its corresponding group. Finally we sort the groups by each group's most recent record time. This is almost an ideal example for a cross-language code clone. The structure is similar and the functionality is same. In practice, there might be some slight changes in both structure and functionality, but a valid clone must consider both structural similarity and functionality. Currently, existing evaluation datasets contain a lot of invalid pairs of code according to this criteria.

In Fig.3 and Fig.4, we give an example of clone in existing competitive programming dataset. We believe this pair of code is inappropriate to be called a clone for following reasons. First, they both contain multiple methods, and each of them implement different functionality. Second, they are different in structure judged by human expert. Existing dataset contains a lot of such pairs.



```java
import java.io.*;
import java.util.*;


public class Main {

    private static Scanner sc;
    private static Printer pr;

    private static void solve() {
        String s = sc.next();
        int n = s.length();

        long ret = 0;
        for (int i = 0; i < n; i++) {
            for (int j = 1; i + j <= n; j++) {
                long num = Long.parseLong(s.substring(i, i + j));
                ret += num * num(i) * num(n - (i + j));
            }
        }

        pr.println(ret);
    }

    private static long num(int n) {
        if (n == 0) {
            return 1L;
        } else {
            return 1L << n - 1;
        }
    }

    // -------------------------------------------------------
    public static void main(String[] args) {
        sc = new Scanner(System.in);
        pr = new Printer(System.out);

        solve();

        pr.close();
        sc.close();
    }

    private static class Printer extends PrintWriter {
        Printer(PrintStream out) {
            super(out);
        }
    }
}

```

図 3 Java code of a clone in existing dataset



```python
import sys

s = sys.stdin.readline().strip()
def mypowerof2(exp):
        ret = 1
        for i in range(exp):
                ret *= 2
        return ret
def mypowerof10(exp):
        ret = 1
        for i in range(exp):
                ret *= 10
        return ret
res = 0
for i in reversed(xrange(len(s))):
        for j in xrange(len(s)-i):
                base = int(s[i]) * mypowerof10(j)
                rate = mypowerof2(i) * mypowerof2((len(s)-1) - i - j - 1)
                res += base * rate
#               print "base is", base
#               print "rate is", rate

print res
```

図 4 Python code of a clone in existing dataset

3 Model Detail

Existing models are not capable of finding clones mentioned in Sec.2. This is because they have a unclear definition of code clone and consider single similarity measurement. To solve this problem, we propose our model with multiple similarity measurements. To be specific, we use text-based similarity and AST path-based similarity. In this section, we will introduce the structure of our model. We currently divide our model into two sub-parts, each of which is related to one similarity measurement. We use a non-dominated ranking system to choose the code pair with the highest similarity score. Since non-dominated ranking system supports multiple objectives, our model can be extendable for more similarity measurements. The overview of our model is illustrated in Fig.5. Some important parts are introduced as follow.

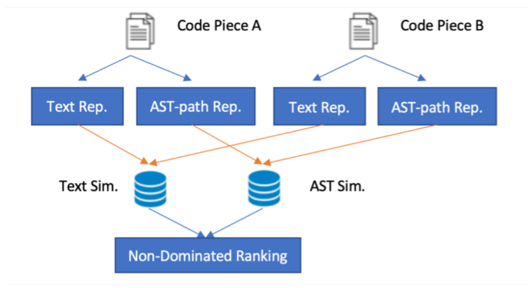
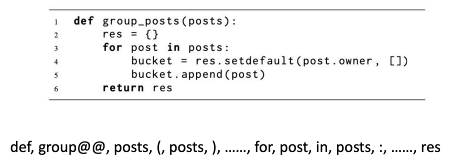


図 5　**Model structure overview**



```
def group_posts(posts):
    res = {}
    for post in posts:
        bucket = res.setdefault(post.owner, [])
        bucket.append(post)
    return res
```

def, group@@, posts, (, posts,), ……, for, post, in, posts, :, ……, res

図 6　**How to get sub-tokens of source code**

3. 1　Text-based similarity

Text-based similarity is most frequently used. It basically works as what a lexical analyzer does. It treats source code as a sequence of sub-tokens. The sub-tokens are achieved by adopting byte-pair encoding algorithm. The advantage of BPE algorithms is that it can avoid rare tokens, and reduce the size of vocabulary significantly. We use Transformer as base model and train the model in an unsupervised way. The common tokens such as keywords and symbols can serve as anchor points in vector space. Therefore, the higher similarity that final representation of source code pieces share, the more they are similar in text manner. Fig.6 gives an example to get sub-tokens of source code. Afterwards, we use Transformer as base to train a masked language model of source code and get vector representation of code pieces. The similarity score of two code pieces is calculated by cosine similarity equation.

3. 2　Generic AST design

Different programming languages have different abstract syntax grammars. The node definition of different syntax trees is different. This will lead to fewer anchor points, and cause lower efficiency for training. The solution we propose is to design a generic abstract grammar for multiple languages. While designing, we make a trade-off between the level of adaption for different languages, and the simplicity of grammar size. The designing logic here is we preserve common node for both languages, and preserve some key language-specific nodes. The common nodes include control structures, operators, variables and class and function definition. For some examples of key nodes, we include `Switch`, `Do-While` node for Java, and `List Comprehension` node for Python. There are more such kind of language-specific key nodes for both languages. Therefore, generic ASTs obtained from two languages are not exactly same. To emphasize, we keep key language-specific nodes so that important information would not be lost when transferring an AST to a generic AST.

3. 3　AST path-based similarity

AST-based similarity gives more consideration to code structure. Given the AST of a code snippet, instead of using depth-first search or Breadth-first search algorithm to represent a tree, we consider all pairwise paths between terminals, and represent them as sequences of terminal and non-terminal nodes. This idea is originally proposed by an existing work called code2vec [1]. The idea is shown in Fig.7. In an abstract syntax tree, leaves are identifiers in source code. Consider the three Java methods in Fig.8. These methods share a similar syntactic structure: they all have a single parameter named target, iterate over a field named elements, and have an if condition inside the loop body. Therefore, if we extract such a path from the AST, this path can capture the main functionality of the method. The method iterates over a field called elements, and for each of its values it checks an if condition; if the condition is true, the method returns something. From this example, we can see that AST paths are very representative to indicate the syntactic structure of a piece of source code.

In detail, we represent each of the path nodes and leaf values of a path context as real-valued vector representation, or known as embeddings. Then, the leaf value vectors and node vectors of each context are concatenated to a single vector that represents that path-context. The base model here we use is bi-directional LSTM. Since there are multiple paths in an AST, after getting path context representation, we uniformly sample K paths from all the paths where K is a hyper parameter, and calculate the representation of K paths. Then we average K representations as the final representation of the code piece. Still, the higher similarity that code representation share, the more they are similar in AST manner.

3. 4 Non-dominated ranking

Non-dominated ranking is an algorithm that orders results with multiple objectives without aggregation. In this algorithm, a result `s` is said to dominate another result `t`, if `s` is no worse than `t` in any objective and is better than `t` in at least one objective. Otherwise, there is a tie. In case of a tie, we select the result that has the dominant objective closest to the optimal value, which is this case, the one more closer to the highest cosine similarity score 1. The detail of non-dominated ranking is displayed in Fig.9.

4 Evaluation Procedure

In this section, we will introduce our new evaluation procedures. As mentioned before, a valid clone must consider both structural similarity and functionality. Therefore, we work on method-level code clone detection and organize our datasets based on this criteria. We design two evaluation procedures for a complete and robust evaluation and we organize datasets for each evaluation procedure individually. Our evaluation procedures are better in evaluating performance of cross-language code clone detectors.

4. 1 Competitive programming evaluation

Most current works are using competitive programming code for evaluation. They recognize solutions to a same problem as clones. The drawback is that these solutions are implemented by different programmers, and do not necessarily contain similar structures. Furthermore, different implementations may vary a lot. Some implements are long and include many functions, while others tend to be short and well-encapsulated in one single function. Thus methods in the dataset may not implement similar functionality. Therefore, evaluations based on current competitive programming dataset are not reliable.

We make our own dataset based on current competitive programming dataset manually. We only keep implementations that have similar lengths and have only one function. In this way we can at least make sure that these source code implements same functionality. We also manually select the implementations that share similar structure, such as common for loops and conditional statements.

Using implementations to same problems as ground truth, we can evaluate how our model can perform when trying to recognize whether two pieces of source code are clones or not. The cleaning procedures help to avoid the ambiguity mentioned before.

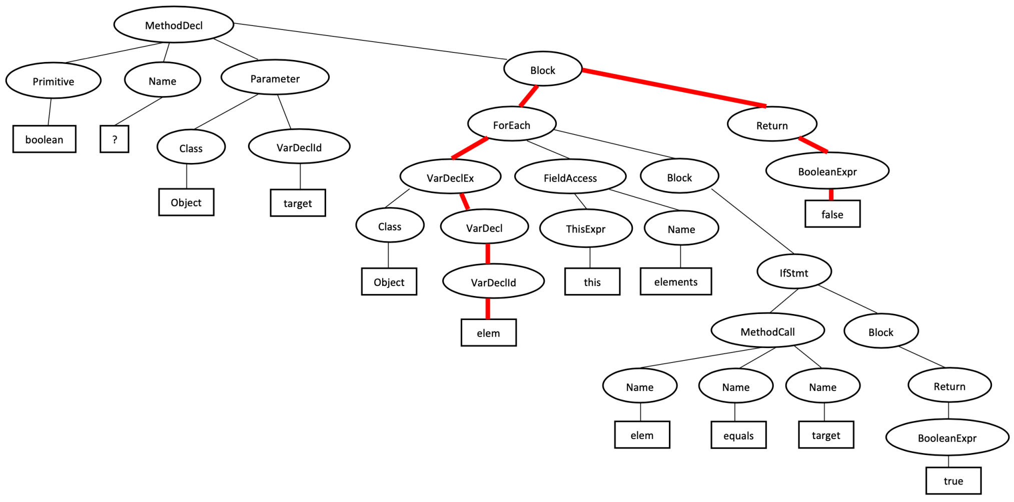
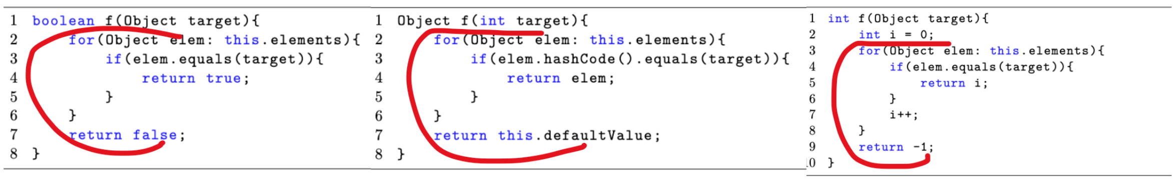


図 7 **Paths in AST**



```java
boolean f(Object target){
    for(Object elem: this.elements){
        if(elem.equals(target)){
            return true;
        }
    }
    return false;
}
```

```java
Object f(int target){
    for(Object elem: this.elements){
        if(elem.hashCode().equals(target)){
            return elem;
        }
    }
    return this.defaultValue;
}
```

```java
int f(Object target){
    int i = 0;
    for(Object elem: this.elements){
        if(elem.equals(target)){
            return i;
        }
        i++;
    }
    return -1;
}
```

図 8 **How paths are representative in source code**

scenario	d_A	d_B	d_C	winner
1	$\mathbf{s} > t$	$\mathbf{s} > t$	$\mathbf{s} > t$	s
2	$s = t$	$s = t$	$\mathbf{s} > t$	s
3	$s = t$	$s < \mathbf{t}$	$\mathbf{s} > t$	tie
4	$s < \mathbf{t}$	$s < \mathbf{t}$	$\mathbf{s} > t$	tie

図 9 **Non dominated ranking**

4. 2 Auto translation evaluation

In this procedure, we use auto-translated source code for evaluation. Auto translation tools can translate source code from one programming language to another. Current translation tools are mainly using pre-defined rules. When encountering some patterns in source language, tools generate target source code following certain templates. These tools cannot generate perfect target code since there is no correspondence between all features in two programming languages, but the generated code is structurally similar and implements same functionally.

We utilize such kind of tools to generate a translated version of original source code. Afterwards, we mix the translated version with a pool of source code pieces to let our model calculates the similarity score between original source code and all pieces in the pool. We then select the piece with highest score to see if it is the translated version, which is ground truth.

We only use auto-translated code for evaluating the model and do not use such code for training. This is important because this evaluation is useful in that it selects the optimal candidate without knowing any knowledge about the candidates. It's closer to real scenarios where we try to find code clones among a large pool of source code pieces.

5 Related Work

Code clone detection is a popular research topic, especially single language code clone detection. For example, CCFinder in [3] and SourcererCC in [7] are token-based, and Deckard in [2] is AST-based. Recently, machine learning has also been adopted in this topic, as shown in [9] and [8].

Compared with single language code clone detection, cross-language code clone detection is less popular, but there are still some existing works. For example, [5] proposes to use tree-based skip-gram algorithm and LSTM-based neural network to detect clones in competitive programming dataset.

One of the most recent works is [4], which utilizes static and dynamic analysis instead of machine learning, and achieves the state-of-the-art performance. They also evaluate their approach on competitive programming dataset.

6 Conclusion and Future Work

In this paper, we show that current works in the field of cross-language code clone detection do not have a clear definition of code clones, and they evaluate their approaches on inappropriate datasets. To solve, we propose new procedures and datasets for better evaluation of detection tools. Moreover, we develop our own model and perform better under the circumstance of new evaluation procedures.

Currently, there are still some experiments undergoing, and results are not complete yet. We plan to finish the comparison between our model and state-of-the-art model to see final results. In future, we plan to have a deeper discussion about what is code clone and how to measure similarity between two pieces of source code in different languages.

Acknowledgements Senxi Li, Tetsuro Yamazaki and Soramichi Akiyama provide a lot of valuable suggestions in the discussion of this paper. We greatly appreciate their help.

参 考 文 献

[1] Alon, U., Zilberstein, M., Levy, O., and Yahav, E.: Code2Vec: Learning Distributed Representations of Code, *Proc. ACM Program. Lang.*, Vol. 3, No. POPL(2019), pp. 40:1–40:29.

[2] Jiang, L., Misherghi, G., Su, Z., and Glondu, S.: Deckard: Scalable and accurate tree-based detection of code clones, *29th International Conference on Software Engineering (ICSE'07)*, IEEE, 2007, pp. 96–105.

[3] Kamiya, T., Kusumoto, S., and Inoue, K.: CCFinder: a multilinguistic token-based code clone detection system for large scale source code, *IEEE Transactions on Software Engineering*, Vol. 28, No. 7(2002), pp. 654–670.

[4] Mathew, G. and Stolee, K. T.: Cross-Language Code Search Using Static and Dynamic Analyses, *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, Association for Computing Machinery, 2021, pp. 205–217.

[5] Perez, D. and Chiba, S.: Cross-language clone detection by learning over abstract syntax trees, *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, IEEE, 2019, pp. 518–528.

[6] Roy, C. K. and Cordy, J. R.: A survey on software clone detection research, *Queen' s School of Computing TR*, Vol. 541, No. 115(2007), pp. 64–68.

[7] Sajnani, H., Saini, V., Svajlenko, J., Roy, C. K., and Lopes, C. V.: Sourcerercc: Scaling code clone detection to big-code, *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 1157–1168.

[8] Wei, H. and Li, M.: Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code., *IJCAI*, 2017, pp. 3034–3040.

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