Code Clone Detection Using Representation Learning

Fnu Jirigesi
Department of Informatics
University of California, Irvine
Irvine, CA, United States
fjiriges@uci.edu

Andrew Truelove
Department of Informatics
University of California, Irvine
Irvine, CA, United States
truelova@uci.edu

Faraz Yazdani
Department of Informatics
University of California, Irvine
Irvine, CA, United States
fyazdan1@uci.edu

Abstract

We seek to improve the performance of code clone detection over currently employed approaches. Almost all of the current approaches for clone detection depend on a set of predefined features of code, ranging from simple syntactic ones to more complex features computed using static analysis. All of the current approaches show very poor performance on more complex types of code clones, however. We investigate whether using representation learning of code snippets can help develop a much more efficient and objective feature vector for codes, which contain the semantic information required for detection of more complex code clones. Feeding possibly long code snippets to a neural network, in a way they can be processed efficiently and understood completely, can be very tricky. For this project, we developed an encoder meant to generate latent representations conditioned on code snippets for the purpose of detecting whether two code snippets are clones. Our results suggest that this approach has considerable potential in detecting code clones.

1 Background

Code clones are segments of code that are highly similar to another piece of code, typically because one piece was duplicated from the other [5]. When a piece of software contains significant amounts of duplicated sections of code, the software is likely going to suffer from higher maintenance costs [5]. Additionally, when code clones are changed inconsistently, these changes are likely to lead to faults [5]. As such, code clone detection is an active area of research.

There exists a commonly accepted taxonomy for types of code clones, consisting of four types [9]. Type-1 clones are "exact copies of code without any modification except formatting, such as white space and comments" [1]. Type-2 clones are "copies of code with only variations in formatting and parameters (i.e., variables, type, function identifiers, or literals)" [1]. Type-3 clones are "copies of code with modifications including added, deleted, or modified statements" [1]. Type-4 clones are "code fragments that perform a similar task (similar semantic) but with different implementations" [1]. Among the four types of clones, type-1 to type-3 clones are more or less syntax-based, while type-4 clones are semantic-based [1]. More specifically, there is a spectrum of subtypes for type-3 clones that have varying levels of complexity; some of the more complex type-3 clones can look fairly similar to type-4 clones [10]. In this project, we mainly focus on type-1 to type- 3 code clone detection. Our project will also include these more complex type-3 clones.

For this project, we aim to see if representation learning through the use of a sequence-to-sequence model can be used to effectively identify code clones. While machine learning approaches have been used to detect code clones, to our knowledge, the use of representation learning with a sequence-to-sequence model has not been attempted before.

2 Related Work

2.1 Classical Approaches

Code clone detection techniques begin by creating code representations before measuring similarity, and these techniques are classified based on their source code representation. Text-based techniques apply minor transformations on the source code, and they directly calculate the similarity based on the text [4]. These text-based clone detection techniques even have poor performance on type 1 and type 2 clones. Token-based techniques lexically analyze the code and produce a vector of tokens and then compare the sub-sequences to detect the clone fragments [7]. Matching the sub-sequence of tokens improves the clone detection power but also increases the false positive rate. Tree-based techniques measure the similarity of subtrees in tree-like syntactic representations [2]. There are two popular trees used in representation code: parse trees and Abstract Syntax Trees. The trees are converted into vectors where each vector represents a code pattern. This approach can lead to improvements in clone detection. Graph-based techniques use static program analysis to transform code into a program dependence graph (PCG), and representation of data and control dependencies [3].

2.2 Machine Learning in Code Clone Detection

Some researchers have focused on utilizing machine learning techniques for code clone detection. Tsunoda et al. assessed the differences in clone detection methods implemented in fault-prone module prediction, and they used logistic regression to build the prediction model [15]. Svajlenko et al. utilized the concept of CloneWorks, and built a clone detection tool by using a Jaccard similarity metric for a near type-3 clone detection [14]. Sudhamani et al. proposed a code clone detection approach using K-means clustering of similar code fragments for type-1 to type-3 clones [12].

Other researchers have attempted to use deep learning to create code clone detection models. Li et al. introduced learning-based detection techniques by leveraging a simple classifier feed forward neural network conditioned on hand-engineered features. [6]. White et al. use recurrent networks for automatic feature extraction from code. [16]. Sheneamer et al. use neural networks to encode pairs of functions to a single vector used for classification. [11]. To our best knowledge, no researchers have attempted to use generative models to detect code clones.

3 Methodology

3.1 Data Preparation

For our training data, we used Java code snippets collected from the BigCloneBench data set [13]. The data set provides a body of Java functions and corresponding cloned functions that we can easily query from. We used Spoon [8] to generate the AST trees for the collected Java source code. We then used preorder traversal on the AST tree nodes to generate sequences of tokens for each function. We only kept the 2,000 most frequent tokens, and replaced the rest with a specific UNKNOWN token. Also, for training and test purposes, we only considered functions with maximum sequence length of 2,500 tokens. As such, we ended up with 12,000 functions and 5.5 million clone pairs. Each training and test batch consists of 32 code pairs, 8 for type-1 clones, 8 for type-2 clones, 8 for type-3 clones, and 8 for non-clone pairs.

3.2 Model and Training

The model we used for our project is a sequence-to-sequence model augmented with a feed-forward neural network meant to determine whether two given functions are clones. Fig. 1 depicts a diagram of the model. The encoder in our model is a GRU-RNN which is used to produce a representation of

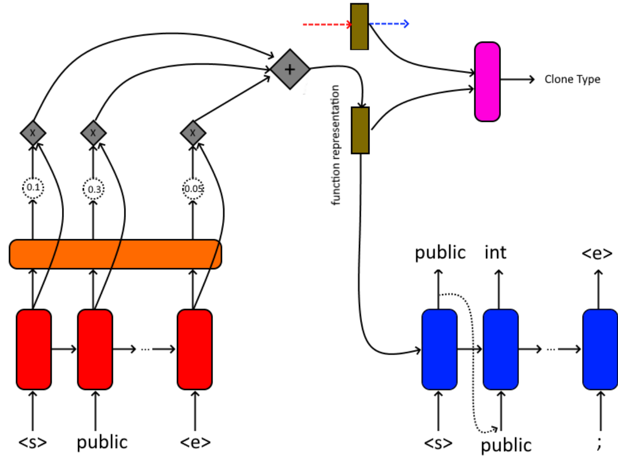

Figure 1: Our Sequence-to-Sequence Model. The red and blue units are unrolled units for the encoder and decoder respectively. The orange unit shows a Bahdanau-style attention layer (without the query), and the pink unit shows the feed forward classifier. In the decoder, when training, the input at timestep t is the function's token at time t, while in inference, it is the output of the decoder at timestep $t-1$.

a given function's code. On top of the encoder there is an attention layer. For every time step of the RNN, the attention layer takes all the outputs of the time step and generates a single context vector based on the particular attention weights. This context vector acts as the representation of the input Java function. The decoder, also a GRU-RNN, takes these representations as input and produces function code. We also used a 2-layer feed-forward neural network to determine the clone type of two functions given their representations. The encoder uses an embedding with size of 64, and both the encoder and the decoder use unit size of 1024. The training loss on a function pair consists of 2 reconstruction loss of the form of categorical cross entropy for each of the functions, and a weighted classification loss for the classifier network, also a categorical cross entropy loss. We trained the entire model in an end-to-end fashion.

4 Results

4.1 Clone Identification

Fig. 2 in the Appendices section shows the confusion matrix for the classification task. This shows not only that the model generally understands the notion of two functions being code clones, but that it can distinguish between different types of clones. As one expects most of the mistakes happen between non-clone pairs and type-3 clones, which are the most syntactically different clone types.

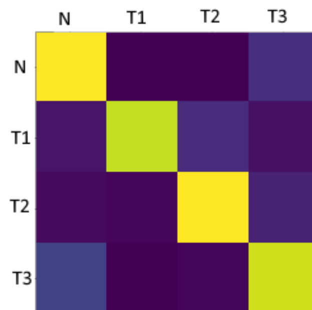

Figure 2: Confusion matrix for determining whether 2 functions are not clones (N), type-1 clones (T1), type-2 clones (T2) or type-3 clones (T3).

If the type of code clone is ignored, the code clone recall on test data is 91.6% and the accuracy is 95.5%. It is important to note that the test data contains same number of code clone types, therefore the result is not biased towards a specific code clone type. To see examples of correctly and wrongly detected code clones, see the Appendices section.

4.2 Reconstruction Accuracy

We also computed the reconstruction accuracy of the function vector sequences using the BLEU metric. On average, the BLEU score for reconstruction is 0.69, which shows the decoder is not ignored in the training process and is involved and has helped the model for learning the task of classifying clone types by providing constructive gradients so that the encoder generates better representations that are easier for the classifier to classify.

4.3 Function Representations

Intuitively, the encoder would represent functions with similar syntax and/or semantics to adjacent points. To test this, we calculated the Euclidean distance between representations of function pairs. Table 1 shows the mean of this distance for each pair type. These results suggest that the encoder learns to represent both semantics and syntax of the code snippets in an intuitive regime.

Additionally a two-sided t-test revealed that between all pairs of clone types besides Type 1 and Type 2, these distances are different in a statistically significant manner. The fact that the difference of distances between clone types 1 and 2 is not significant also suggests that the encoder probably attends more towards the functionality (semantics) of the code rather than its syntax.

Clone Type	Mean Distance
Type 1	0.39
Type 2	0.38
Type 3	4.79
Non-Clone	5.28

Table 1: The Euclidean distance between representations of pairs of functions

5 Conclusions

We investigated whether representation learning was a useful approach in detecting type-1, type-2, and type-3 code clones. We created a sequence-to-sequence model with a feed-forward neural network that took in a pair of code segments and classified the code clone type (or whether the pair had no code clones). Due to the long length of sequences and the time needed to train the model on our hardware, out results may not wholly reflect the full potential of the model. Our results, however, do seem to suggest that this technique has promise and that it is worth exploring further.

References

[1] V. Arammongkolvichai, R. Koschke, C. Ragkhitwetsagul, M. Choetkiertikul, and T. Sunetnanta. Improving clone detection precision using machine learning techniques. In *2019 10th International Workshop on Empirical Software Engineering in Practice (IWESEP)*, pages 31–315. IEEE, 2019.

[2] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377. IEEE, 1998.

[3] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*, pages 321–330, 2008.

[4] J. H. Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering-Volume 1*, pages 171–183. IBM Press, 1993.

[5] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *2009 IEEE 31st International Conference on Software Engineering*, pages 485–495. IEEE, 2009.

[6] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder. Cclearner: A deep learning-based clone detection approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 249–260. IEEE, 2017.

[7] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering*, 32(3):176–192, 2006.

[8] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46:1155–1179, 2015.

[9] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009.

[10] 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*, pages 354–365, 2018.

[11] A. Sheneamer and J. Kalita. Semantic clone detection using machine learning. In *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 1024–1028. IEEE, 2016.

[12] M. Sudhamani and L. Rangarajan. Code clone detection based on order and content of control statements. In *2016 2nd International Conference on Contemporary Computing and Informatics (IC3I)*, pages 59–64. IEEE, 2016.

[13] 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*, pages 476–480. IEEE, 2014.

[14] J. Svajlenko and C. K. Roy. Fast and flexible large-scale clone detection with cloneworks. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 27–30. IEEE, 2017.

[15] M. Tsunoda, Y. Kamei, and A. Sawada. Assessing the differences of clone detection methods used in the fault-prone module prediction. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 3, pages 15–16. IEEE, 2016.

[16] 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)*, pages 87–98. IEEE, 2016.

Appendices

An example of correctly identified code clones. As one can see, these are essentially type 3 clones for copying a file.

```
public static boolean decodeFileToFile(String
        infile, String outfile) {
    boolean success = false;
    java.io.InputStream in = null;
    java.io.OutputStream out = null;
    try {
        in = new Base64.InputStream(new java.io.
                BufferedInputStream(new java.io.
                FileInputStream(infile)), Base64.
                DECODE);
        out = new java.io.BufferedOutputStream(
                new java.io.FileOutputStream(outfile
                ));
        byte[] buffer = new byte[65536];
        int read = -1;
        while ((read = in.read(buffer)) >= 0) {
            out.write(buffer, 0, read);
        }
        success = true;
    } catch (java.io.IOException exc) {
        exc.printStackTrace();
    } finally {
        try {
            in.close();
        } catch (Exception exc) {
        }
        try {
            out.close();
        } catch (Exception exc) {
        }
    }
    return success;
}
```

```
private static void copyFile(File src, File dest,
        int bufSize, boolean force) throws
        IOException {
    if (dest.exists()) {
        if (force) {
            dest.delete();
        } else {
            throw new IOException("Cannot␣
                    overwrite␣existing␣file:␣" +
                    dest.getName());
        }
    }
    byte[] buffer = new byte[bufSize];
    int read = 0;
    InputStream in = null;
    OutputStream out = null;
    try {
        in = new FileInputStream(src);
        out = new FileOutputStream(dest);
        while (true) {
            read = in.read(buffer);
            if (read == -1) {
                break;
            }
            out.write(buffer, 0, read);
        }
    } finally {
        if (in != null) {
            try {
                in.close();
            } finally {
                if (out != null) {
                    out.close();
                }
            }
        }
    }
}
```

Examples of undetected clones. This is unexpected since the two codes are exactly the same. The authors' guess is that because the code involves two dimensional array access, which is a less frequent pattern than many other patterns, the encoder has not learned to encode it correctly and therefore the classifier is unable to understand the meaning of the code using the representation.

```
public Matrix transpose() {
    Matrix X = new Matrix(n, m);
    double[][] C = X.getArray();
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            C[j][i] = A[i][j];
        }
    }
    return X;
}
```

```
public Matrix transpose() {
    Matrix X = new Matrix(n, m);
    double[][] C = X.getArray();
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            C[j][i] = A[i][j];
        }
    }
    return X;
}
```

Examples of two non-clone pairs that are detected as type 3 clones by the model. Interestingly, the function `email_get_public_hash` uses hashing, which is what the function `hash` does.

```java
private String hash(String message) {
    MessageDigest md = null;
    try {
        md = MessageDigest.getInstance("SHA1");
    } catch (NoSuchAlgorithmException e) {
        throw new AssertionError("Can't␣find␣the␣
              SHA1␣algorithm␣in␣the␣java.security␣
              package");
    }
    String saltString = String.valueOf(12345);
    md.update(saltString.getBytes());
    md.update(message.getBytes());
    byte[] digestBytes = md.digest();
    StringBuffer digestSB = new StringBuffer();
    for (int i = 0; i < digestBytes.length; i++)
         {
        int lowNibble = digestBytes[i] & 0x0f;
        int highNibble = (digestBytes[i] >> 4) &
              0x0f;
        digestSB.append(Integer.toHexString(
              highNibble));
        digestSB.append(Integer.toHexString(
              lowNibble));
    }
    String digestStr = digestSB.toString().trim()
         ;
    return digestStr;
}
```

```java
public static String email_get_public_hash(String
      email) {
    try {
        if (email != null) {
            email = email.trim().toLowerCase();
            CRC32 crc32 = new CRC32();
            crc32.reset();
            crc32.update(email.getBytes());
            MessageDigest md5 = MessageDigest.
                  getInstance("MD5");
            md5.reset();
            return crc32.getValue() + "␣" + new
                  String(md5.digest(email.getBytes
                  ()));
        }
    } catch (Exception e) {
    }
    return "";
}
```