Source Code Plagiarism Detector

Tsung-Han Wu
National Taiwan University
b05902013@ntu.edu.tw

Yi Chou
National Taiwan University
b05902086@ntu.edu.tw

Yu-Chien Hsu
National Taiwan University
b05902021@ntu.edu.tw

Yu-Ting Tseng
National Taiwan University
b05902120@ntu.edu.tw

1 INTRODUCTION

Information retrieval is the activity of obtaining relevant information from a collection of those resources. Over the past 20 years, information retrieval system was widely used in many areas and plays an important roles in our everyday-life. Despite of traditional text-based retrieval system, there are more and more popular applications, such as image-to-image retrieval system or audio-based search engine.

However, as the advancement of information retrieval techniques, there are still few works focusing on source code retrieval. Source code retrieval problem is to find logically similar programs from a collection of source codes. The main difference from text retrieval is that the variance between codes are much larger than articles, such as variable naming, different function calls or code aligning.

One of the applications of source code retrieval is detecting code plagiarism. Some early works try to use tokenized codes to get semantic meaning [1]. Another similar studies use IR-based method to detect code plagiarism [2]. In recent years, may works detect plagiarism based via abstract syntax tree, such as [3].

In this project, we want to build an IR-based code plagiarism detector that uses a similarity function, Okapi BM25[4]. We first look at finding optimal compile options that can mitigate variance between codes. Then, we try to find out how to extract features from assembly codes and tune hyper-parameters to reach great performance.

2 METHODOLOGY

2.1 Feature Extraction

Before talking about IR models, we need to extract useful features that can represent the behavior of the program from source codes. Knowing that plaintext cannot resist variable renaming, different function calls or code aligning, we decide to compile these source codes to assembly codes and extract features from there.

2.1.1 *Optimal Compiler Options.*

To mitigate the problems listed above, we take a closer look on gcc compiler options. The easiest way is to add *-O2* (or *-O3*) options while compiling codes, which makes assembly codes more similar. However, one can still hack it by adding function calls, which will make assembly codes differ from the original one. To handle this problem, we set some compiler options to make all functions inline, unroll loops and remove all unused sections.

Furthermore, we add some options to make the assembly codes more cleaner by removing EBP,RBP before and after function frame and stack canary. The advantages of these compiler options can see in Section 3.3.

2.1.2 *Program Behavior.*

For source code retrieval tasks, we would like to retrieve codes with similar program behaviors. For example, we want to retrieve codes first go through a for loop to process an array, and then print out the value of the whole array.

We think that there are three critical issues that may play an important role on program behavior. First, we need to find out the basic block from the assembly codes. For example, one *call printf* follows one *mov* in assembly always means *printf* in C. Second, program behavior is a sequence of operations. Thus, we need to consider some instructions together rather than a single one. Third, the operand of the instructions may increase variance. The reason is that the operands may contains string name or address information, which is useless on our tasks.

For the first and second observation, we use N-gram to consider multiple operations at one time. As for the third observations, we discarded operands and use only instructions as our features. The result of our work can be seen at Section 3.3.

2.2 Okapi BM25 Model

The formulation of our Okapi model is as below :

$$Okapi(Q,D) = \sum_{t \in Q \cap D} IDF(t) \cdot \frac{(k_1+1)f_{d,t}}{f_{d,t}+K} \cdot \frac{(k_3+1)f_{d,t}}{k_3+f_{d,t}} \quad (1)$$

$$\text{where } K = k_1 \cdot ((1-b) + b \cdot \frac{|D_{terms}|}{avgD_{terms}})$$

$$IDF(t) = \log(\frac{N-f_t+0.5}{f_t+0.5})$$

Some hyper-parameters we use are as follows:

- Frequency normalization : $k_1 = k_3 = 1.2$
- Length Penalty : $b = 0.75$

3 EXPERIMENT

3.1 Source Code Collections

For training and testing, we collect a collections containing about 8000 codes written in the C programming language. The collections consist of actual student submissions to assignments.

We labeled all plagiarised codes as relevant. The ground truth for the collections was carefully checked by MOSS, the state-of-the-art code plagiarism detector, and eleven teaching assists. The training and testing query files are codes that are considered cheating in one assignment.

Table 1: Performance under different compile options

Features	MAP@10
Raw Asm	0.611
Asm opt (optimal options)	**0.711**
Asm opt w/o loop unroll, lnline func	0.621

3.2 Evaluation

In our collection, one queries may have multiple relevant codes (about 10 relevant), including codes that are written by the same students and plagiarism code pairs. Thus, we think that MAP@10 might be a proper evaluation metric in this project. The formulations are as below :

$$MAP@10 = \frac{1}{|Q|} \sum_{q \in Q} AveP(q)@10 \tag{2}$$

where Q is the set of all queries and q means a single query. The definition of $AveP(q)$ is as follows :

$$AveP(q)@10 = \frac{1}{|min(|R(q), 10|)|} \sum_{k=1}^{10} (P(k)\dot{r}el(k)) \tag{3}$$

where $P(k)$ is the precision of the top k ranking in the upload result (Precision). $rel(k)$ indicates whether the k-th in the upload result is relates. The value is 1 if labeled as plagiarism and 0 if not. $R(q)$ represents the plagiarism code set of the test query code q.

3.3 Results

3.3.1 *Different Compiler Options.*

We study the impact of different compiler options on performance. In this experiment, we use all single word in assembly code as terms in Okapi model.

The result in Table 1 shows that our assumption makes sense. Inline function call and Loop unrolling plays an important role on this task. Furthermore, using features after removing unimportant information, such as EBP/RBP or stack canary, can slightly improve the performance.

3.3.2 *Different N-gram model.*

After studying on compiler options, we take a closer look on the representation of program behavior. We take uni-gram to 10-gram instructions as features and feed into our Okapi BM25 model. The result in 2 shows that using N-gram instructions as terms get similar scores with single word in whole assembly codes.

The reason why uni-gram get bad performance is that it lost too much information. While using bi-gram or more grams as terms can really represent program behavior. In addition, we ensemble the model from tri-gram to 5-gram and grid search for best parameters. The result shows that it outperforms any single model and has about 8% performance gain compared to the original one.

3.3.3 *Ability to tell plagiarism code and irrelevant code.*

From an application point of view, code plagiarism detector should have ability to judge two codes are plagiarism or not, not just return the top 10 relevant source codes. In other words, we need to set a threshold and return only scores higher than that.

Table 2: Performance under different N-gram model

Features	MAP@10
whole Asm	0.711
instruction uni-gram	0.514
instruction bi-gram	0.729
instruction tri-gram	0.792
instruction 5-gram	0.709
instruction 10-gram	0.725
Ensemble 3-5 gram	**0.798**

To achieve this, we standardize our raw similarity scores and found that 9 standard deviation over the average in across all pairs will be a proper threshold. In our investigation, about 70 % of plagiarism code pairs' similarity scores are higher than the threshold. In contrast, over 95 % non-plagiarism code pairs' scores are lower than the threshold.

4 CONCLUSION

In this project, we proposed an IR-based method to detect code plagiarism on C language. Compared to MOSS or other plagiarism detector, our method can run much more faster (The entire process only takes fewer than one hour) and reach acceptable performance. We think that IR-based plagiarism detector on compilable programming languages is possible.

REFERENCES

[1] Steven Burrows, S. M. M. Tahaghoghi, and Justin Zobel. [n. d.]. Efficient plagiarism detection for large code repositories. *Software: Practice and Experience* 37, 2 ([n. d.]), 151–175. https://doi.org/10.1002/spe.750 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.750

[2] Victor Ciesielski, Nelson Wu, and Seyed M. M. Tahaghoghi. 2008. Evolving similarity functions for code plagiarism detection. In *GECCO*.

[3] Deqiang Fu, Yanyan Xu, Haoran Yu, and Boyang Yang. 2017. Wastk: A weighted abstract syntax tree kernel method for source code plagiarism detection. *Scientific Programming* 2017 (2017).

[4] Stephen E. Robertson, Steve Walker, and Micheline Beaulieu. 2000. Experimentation as a way of life: Okapi at TREC. *Information processing & management* 36, 1 (2000), 95–108.