Normalization based stop-word approach to source code plagiarism detection

Saimadhav Heblikar
PES Institute of Technology
Bangalore, India
saimadhavheblikar@gmail.com

Poorva Sharma
PES Institute of Technology
Bangalore, India
poorvasharma0615@gmail.com

Manogna Munnangi
PES Institute of Technology
Bangalore, India
manogna08@gmail.com

Channa Bankapur
PES University
Bangalore, India
channabankapur@pes.edu

ABSTRACT

This paper is a report of PES Institute of Technology's participation in the Cross Language Detection of Source Code Reuse (CL-SOCO) task at FIRE 2015 [1]. We approach this task as text document plagiarism task, without considering formal programming language grammatical structure. We use normalization of commonly used identifiers to detect pair of programs which have the same objective. We also find that entirely removing these normalized operations improves the system.

CCS Concepts

•Information systems → Similarity measures; Clustering and classification;

Keywords

Source code reuse, Plagiarism detection

1. INTRODUCTION

Vast amounts of software code has become easily available on the Internet. Sites like Stackoverflow make available solutions to common problems. In such a scenario, software developers are tempted to copy and paste code from one place to another. This could cause the owners of the software legal, ethical, licensing and maintenance problems in the long run. Software plagiarism also affects competitive programming competitions like ACM ICPC. The sheer scale of available resources to plagiarize from and the possible number of plagiarized documents makes this a source code plagiarism detection a daunting task.

Plagiarism detection in software source code is different from text plagiarism detection task. One of the popular approaches to text plagiarism detection is bag-of-words model[4]. However, this is not useful in a software source code context as a small set of programming constructs are bound to be reused repeatedly, whilst doing altogether different things.

There currently exist tools like MOSS [2] and JPLAG [3] which try to solve this problem. MOSS stands for "Measure Of Software Similarity" and is a system for detecting similarity in software. JPLAG is a system for detecting software similarity considering text features as well as programming language features. Both JPLag and MOSS are used in academic environments.

2. TASK DESCRIPTION

Cross language Source code reuse(CL-SOCO) track of FIRE 2015 deals with the detection of plagiarism in software source code. The cross language aspect deals with detecting plagiarism from C to Java sources.

The training set given to us consisted of 599 C and 599 Java files. These files were numbered from 001.C to 599.C and 001.java to 599.java The files with the same number represented a plagiarized case. That is, 012.c and 012.java represents a reuse case, while 012.c and 021.java don't. Since some of these files were generated using a tool, they contained parse errors. This was true for both the C and Java data-set.

The test set given to us consisted of 79 C files and 79 Java files. These files were numbered from 300.c to 378.c and 000.java to 078.java.

Both the training and test corpus are available at [1].

It is important to note that we do not have to mention the direction of reuse. That is, whether the reuse was from C to Java or from Java to C.

3. CURRENT WORK

In this section, we describe the state of research in the field of plagiarism detection in general, and source code plagiarism detection in particular.

3.1 Bag-of-words-model

In this model, the document is represented as a bag-of-words. In practice, it is a multi-set. It disregards order or grammar, but accounts for multiplicity. Bag of words is shown to work well for the text plagiarism detection task [4]. It's performance is not satisfactory for the programming language plagiarism detection task [5]. The reason being similar programming constructs are bound to be a very high number of times in programs. However, these programs may be doing entirely different things.

3.2 NLP Techniques

Common Natural Language Processing techniques like word n-grams are used to detect similarity between documents. Some works also consider using features of the text like number of white-spaces, average indentation, and other stylistic features for evaluation. A popular tool is XPLAG [6].

3.3 Longest common sub-string

Tools like JPLAG[3] make use of the Longest common substring (LCS) approach. This is a pair wise approach. The similarity between a pair is decided by the length of the longest common substring.

4. SYSTEM DESCRIPTION

We build upon the existing work described in Section 3.1. We work on the bag-of-words model, modifying it to support term weighting. We then use word 1-grams as features. Our approach is from XPLAG[6] in the sense that we do not consider any other NLP techniques which were described in Section 3.2.

In this section we describe our approach. We present three iterative runs, each built upon and improving over its predecessor. Only the preprocessing stage varies for each run. The first run is the baseline run. The second run is the normalization run. The third run builds upon the second, and removes any normalized operation or identifier. We call this third run as using the removal of stopwords, from the normalized operations or identifiers.

We divide the workflow into four stages :

4.1 Preprocessing

The preprocessing stage is divided into 2 parts. The first part is same for all approaches and is described below

In the first part of preprocessing, more than one continuous whitespace are converted to a single whitespace. The code is then converted to lower case. Any accents in the text are stripped. The source code is then passed to a lexer. The lexer removes lexemes like +, -, *, / etc. The output of the lexer is a stream of tokens.

Subsection, 4.1.1 to 4.1.3 provides a detailed description of approach to the second part of preprocessing stage for each run.

4.1.1 Baseline

In this stage, no work is done. That is, there is no transformation of the tokenized stream obtained from part one of preprocessing. This approach serves as a baseline.

4.1.2 Normalization

The input to this approach is the output from baseline approach. In this stage, we study the language usage features from the training data. We obtain frequency statistics about the most commonly used identifiers in both the languages C and Java. This was sorted based on frequency in non-increasing order. This list was pruned to consider those in the top-n positions of the list. We also considered keywords as identifiers.

We then manually mapped similar identifiers/functions to new operation identifiers or op-codes. For example, printf is used as output function in C. System.out.println is used as a output function in Java. Both these functions perform similar operations. Therefore, we replace all occurrences of printf and System.out.println with op1. Refer to Table 1 for a full list of such replacements. The output from the stage is fed to the vectorizer.

Table 1: Normalization List

Op-Code	Identifiers assigned to Op-Code
op1	len,strlen,length,size
op2	stdio,stdlib,system
op3	size,sizeof
op4	struct,typedef,class,object
op5	string,str,StringFunctions
op6	list,iter
op7	new,malloc
op8	argv,argValue
op9	rand
op10	argc,args
op11	pthis,pthread
op12	print, fprintf, printf, sprintf, println System.out.println, System.out.print, System.out.printf, puts, putchar,fputs
op13	array,charAt
op14	ret,return
op15	file,fd
op16	int,integer
op17	char,character
op18	bool,Boolean,boolean
op19	float,Float
op20	scanf,scanner,gets,getch,getchar

4.1.3 Removing stopwords

The input to this approach is the output from preprocessing of normalization approach. All op-codes which were generated in the previous stage are removed. This can also be seen as using a stop-word list consisting of op-codes generated earlier.

4.2 Vectorizer

This stage receives as input a set of tokenized documents. Each document is converted to a vector. The feature chosen to create the vector is word 1-gram and 2-gram. The weighting factor used is term frequency-inverse document frequency (tf-idf).

$$tf(t,d) = 0.5 + \frac{0.5 * f(t,d)}{maxf(t,d) : t \in d} \quad (1)$$

We know that term frequency(tf) increases proportionally to the number of times it appears in a document, but is offset by its frequency in the corpus. We require the offset weights because certain set of programming language constructs are used a very high number of times, almost always in programs which do very different things. Inverse document frequency(idf) serves this purpose. The output of this stage is a set of tf-idf vectors, each vector representing a document.

We group the output into training set and a testing set. The training set is passed to the similarity phase. The testing set is passed to the deciding phase.

Table 2: Comparison of mean vs. median for deciding threshold

	Mean	Median
Threshold(similarity value)	0.644	0.776
False positives	139	88
F1	0.450	0.324
Precision	0.738	0.775

4.3 Similarity Phase

The input to this stage is a set of vectors representing the documents in the training set. The set of vectors corresponding to the training set was divided into sets corresponding to C files and Java files. A cross product was taken between these two sets. This cross product set represents comparing every C file with every Java file. A cross product was taken between these two sets. This cross product set represents every possible (C, Java) pair from training data.

$$similarity = \cos(\theta) = \frac{A.B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i * B_i}{\sqrt{\sum_{i=1}^{n}(A_i)^2} * \sqrt{\sum_{i=1}^{n}(B_i)^2}} \tag{2}$$

As mentioned in Section 2, we know that a (C, Java) file pair is plagiarized if and only if they have the same file number. Any other case they are not plagiarized. We use the above statistic to record the mean and median cosine similarity values for all plagiarized cases and all non-plagiarized cases.

4.4 Deciding Phase

The input to this stage is a set of vectors from the testing data and similarity statistics like mean or median for the plagiarized cases from the similarity phase. The set of vectors corresponding to the test data was divided into sets corresponding to C files and Java files. A cross product is taken between these two sets. This cross product set represents every possible (C, Java) pair from test data. For every pair in the cross product set, cosine similarity is computed. The deciding factor to say that a pair is plagiarized was done by choosing the mean/median obtained from the similarity phase as threshold. Anything above this threshold was considered as plagiarized. For the 3 runs, we chose mean value from the previous phase as a threshold. The reason for choosing mean over median is given below.

4.4.1 *Choosing threshold for deciding*

The statistics in Table 2 are results from baseline approach. The input documents were the training set itself. There were 599x599 (C, Java pairs). The number of plagiarized cases were 599. We measured the mean and median values of cosine similarity for all the plagiarized cases.

We see that using mean as threshold gives us slightly lower precision, but a far better recall, and therefore a better F1 value. We see that number of false positives is higher using mean as threshold, but the difference between using mean or median as threshold is tending to 0(0.00014) when expressed as percentage.

Table 3: Results for the cross language collection

Preprocessing approach	F1	Precision	Recall
Baseline	0.683	1.000	0.519
Normalization	0.697	1.000	0.534
Removing stopwords	0.740	1.000	0.603

5. RESULTS AND ANALYSIS

Precision is the fraction of retrieved instances that are relevant, while **recall** is the fraction of relevant instances that are retrieved.

F1 score is the harmonic mean of precision and recall. Since it takes both precision and recall into equal consideration, it's an overall measure of relevance.

As we can see in the results in Table 3, precision of all the three runs is 1. This means all the retrieved results are relevant i.e., there are no false positives.

Next, we compare the different approaches.

Comparison between baseline and normalization

The reuse cases of <345.c and 033.java>, <351.c and 043.java> and <368.c and 061.java> are there in normalization but not in baseline. Our reasoning about this is as follows - Since after preprocessing, there is no further transformation of the tokens obtained from the lexer in baseline, certain tokens in C and Java, although have the same meaning (perform similar actions), might not have been considered the same. This reduces the cosine similarity between the files under consideration.

For example, the statements

```
strcpy(url, ‘‘ ’’) (in 345.c)
```

and

```
url = ‘‘’’ (in 033.java)
```

do the same thing, but they are not considered the same by the lexer. The reuse case of <312.c and 005.java> is there in baseline but not in normalization. Since only the top-n positions of the frequency statistics regarding commonly used identifiers/keywords in C and Java were considered for the normalization, it would have missed out on certain identifiers/keywords that perform the same actions in both C and Java.

Comparison between baseline and stop-word removal

The reuse case of <337.c and 034.java> is there in baseline but not in stop word removal, reasons being similar to above mentioned (we know that output of preprocessing of normalization is fed to stop word removal preprocessing stage).

The reuse cases of **<321.c and 049.java>, <331.c and 065.java>**, <345.c and 033.java>, **<351.c and 043.java>**, <359.c and 029.java>, **<368.c and 051.java>, <373.c and 058.java>, <374.c and 006.java>, <375.c and 042.java> and <376.c and 074.java>** are all there in stop-word removal but not in baseline. The possible reason is since the op-codes are removed and direct mapping between the identifiers or keywords of both the languages is done, there is a higher possibility in matching similar constructs and identifiers.

Comparison between normalization and stop-word removal

The reuse cases in bold letters above along with <312.c and 005.java> are all there in stop-word removal but not in normalization run.

The reuse case of <337.c and 034.java> is there in normalization but not in stop-word removal. This may be because in normalization, there are certain op-codes for similar identifiers. In stop-word removal, we remove the op-codes. Suppose there are many number of identifiers in both the languages put together that have similar meaning, it might become difficult/inaccurate to keep track of them without using op-codes. (If we are using op-codes, all of them will have a single op-code).

6. FUTURE SCOPE AND CONCLUSION

6.1 Improving recall

In order to improve the recall, we may do the following: Use a combination of the methods used for normalization and stop-word removal. In some cases, where a large number of similar identifiers are there, op-code can be used. In others, direct mapping can be used. More number of identifiers can be grouped under one op-code. For example, as and when you encounter an identifier that has the same meaning as the identifiers in an already defined set, it can be added to the set. The value of 'n' in deciding the top-n by frequency identifiers can be varied.

6.2 Improving the normalization and stop-word removal procedure

We mention certain aspects which were lacking in our system and possibly suggestions on how it can be improved in the future. This suggestions are keeping in mind how the system can be made generic, that is, to support as many language pairs as possible.

6.2.1 Same language normalization automation

Most programming languages provide many functions or identifiers to do the same thing. For example, C provides printf, fprintf and puts to output a string. We use contextual information to automate the process of detecting such functions or identifiers. Once contextually similar pairs are detected, they may be assigned op-codes if they are indeed doing similar things.

6.2.2 Cross language normalization automation

The approach of Section 6.2.1 may be used to decide whether it would be feasible to automate the process of generating op-codes across languages for similar operation.

7. REFERENCES

[1] Flores, E. and Rosso, P. and Moreno, L. and Villatoro-Tello, E.: PAN@FIRE 2015: Overview of CL-SOCO Track on the Detection of Cross-Language SOurce COde Re-use. In Proceedings of the Seventh Forum for Information Retrieval Evaluation (FIRE 2015), Gandhinagar, India, 4-6 December (2015)

[2] Plagiarism Detection: 2015.
https://theory.stanford.edu/ aiken/moss/. Accessed: 2015-10- 18.

[3] Prechelt, L. and Malpohl, G. and Philippsen, M. Finding plagiarisms among a set of programs with JPlag. Journal of Universal Computer Science, 8(11):1016-1038, 2002.

[4] Barrón-Cedeño, A. and Rosso, P. On Automatic Plagiarism Detection Based on n-Grams Comparison. In Proceedings of the 31th European Conference on IR Research on Advances in Information Retrieval (ECIR '09), Mohand Boughanem, Catherine Berrut, Josiane Mothe, and Chantal Soule-Dupuy (Eds.). Springer-Verlag, Berlin, Heidelberg, 696-700, 2009. DOI=http://dx.doi.org/10.1007/978-3-642-00958-7_69')

[5] Ganguly, D., Jones, G.: DCU@FIRE-2014: An Information Retrieval Approach for Source Code Plagiarism Detection.

[6] Arwin, C., and Tahaghoghi, SMM. "Plagiarism detection across programming languages." Proceedings of the 29th Australasian Computer Science Conference-Volume 48. Australian Computer Society, Inc., 2006.