A Novel Framework to Detect Source Code Plagiarism: Now, Students Have to Work for Real!

Boris Lesner, Romain Brixtel, Cyril Bazin, Guillaume Bagan

A Novel Framework to Detect Source Code Plagiarism: Now, Students Have to Work for Real!

Boris Lesner Romain Brixtel Cyril Bazin Guillaume Bagan
{blesner, rbrixtel, cbazin, gbagan}@info.unicaen.fr
GREYC (CNRS UMR 6072) 6, Avenue Maréchal Juin F14032 Caen CEDEX - France

ABSTRACT

Our work focuses on detecting plagiarism within a source code corpus. The case study is to help a human corrector to find out plagiarism within source code written by Computer Science students. Like other approaches, we use the notion of similarity distance. However, in this work we introduce segmentation to split documents into smaller parts and propose a document-wise distance based on the cost of permuting segments to transform one document to another. Our framework is laid out as a pipeline, where each stage can be parameterized to build up a plagirism detector fitting user needs. The approach makes no assumption about the programming language being analyzed. Furthermore, it provides a synthetical report of the results to ease the decision making process, as we consider that only a human user has final word on wether it is plagiarism or not. We tested our framework on hundreds of real source files, involving many programming languages, allowing us to discover previously undetected frauds.

Categories and Subject Descriptors

K.3.2 [**Computer and Education**]: Computer and Information Science Education

Keywords

Source Code Plagiarism, Similarity Measure, Segmentation

1. INTRODUCTION

As computer science teachers we sometimes have to deal with unethical students who copy other's work (ie. source code) for their projects. Tracking this plagiarism is a time-consuming task since it requires to compare each pair of documents containing hundreds or even thousands of lines. We propose a semi-automated framework, which highlights the most suspicious pairs of documents within the corpus for being manually checked afterwards. To gain generality we want this method to be programming-language independent, it should not have previous knoweldge about the language being processed. The only restriction we make is that the corpus should only contain source code documents written in the same language.

2. A MODEL OF PLAGIARISM

In source code, we define plagiarism as the application of successive *transformations* applied on an original document. A transformation preserves the program function but not it's appearance. We aim to handle four kinds of transformations: renaming, code reordering, adding/removing uninterpreted text and replacing code sections or intructions by equivalent ones. The more transformations are done, the less two documents are plagiarized. We aimed to build a framework to help teachers finding plagiarised documents into a corpus of source-code files. This framework pinpoints *suspects* pairs of documents to be manually checked by the corrector, greatly reducing the number of pairs to check.

3. A BOTTOM-UP APPROACH

Our method is based on six major stages, forming the detection pipeline : (1) pre-filtering, (2) segmentation and similarity measurement stage, (3) segment matching, (4) post-filtering, (5) document-wise distance evaluation, (6) and corpus analysis presentation. It works in a bottom-up fashion, the first stages operating at the character level and the subsequent ones becoming more and more abstract, by operating at the string, document and finally corpus level. Algorithm 1 shows how different stages are coordinated.

Pre-filtering : Pre-filtering gives robustness to *renaming* by replacing each alphanumerical string (eg. variable name or keywords) by a single symbol.

Algorithm 1: Main algorithm

Input: $\mathcal{D}$: a set of documents
begin
  /* Prefilter the documents */
  $\mathcal{D}' \leftarrow \{PreFilter(d) \mid d \in \mathcal{D}\}$
  /* Segment filtered documents */
  foreach $d'_i \in \mathcal{D}'$ **do** $S_i \leftarrow Seg(d'_i)$
  /* Compute the distance between each pair of documents */
  foreach $d'_i, d'_j \in \mathcal{D}'$, $i > j$ **do**
    $\mathcal{M}_{(i,j)} \leftarrow$ DocumentDistance (S_i, S_j)
  /* Return a human-readable result */
  return Display $(\mathcal{M})$
end

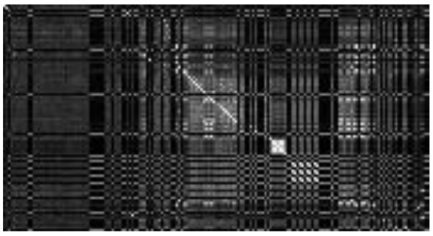

Figure 1: Distance matrix of 2 plagiarized documents. Light points means small distance

Segmentation and similarity measure : For this second stage, each document is divided into segments. We try to be robust against *code reordering* by detecting similar segments of two documents. Intuitively, the segmentation determine what will be a "unit" of code. For example, we may want to work at the line level or the function level. A *segment* is a contiguous subset of a document, a *segmentation function Seg* partitions a document d into a sequence of segments $Seg(d) = (s_1, \ldots, s_m)$. A distance function $Dist(s_1, s_2)$ between two segments is a real number in $[0, 1]$ satisfying the usual distance properties. For two given segmentations $S_1 = (s_1^1, \ldots, s_m^1)$ and $S_2 = (s_1^2, \ldots, s_n^2)$ and a distance function *Dist*, we obtain a $m \times n$ *distance matrix* $\mathcal{M}$ where $\mathcal{M}_{(i,j)} = Dist(s_i^1, s_j^2)$. When reasonably fine segmentation is choosen, contiguous and similar sections of documents make diagonals appear in the distance matrices, revealing possible plagiarism, these diagonals are often difficult to see because of the noise (see figure 1). It's important to see that dissimilar documents have no diagonals in their segments distance matrix. Many distance functions can be used on segments, like Hamming distance or Levenshtein [3] (aka. edit distance) counting the number of operations (inserts, deletes, or replaces) to transform one segment into another. An also interesting one is information distance (we refer to [1] for details). In practice, we use a line-by-line segmenter with edit distance.

Segment matching and Post-filtering : At this point, we have a distance matrix $\mathcal{M}$ for a pair of segmentations S_1 and S_2. From such a matrix, we want to find a distance between documents themselves. To that end, we look for a maximal matching of minimal distance within $\mathcal{M}$. A matching is a set of pairs $C \subset S_1 \times S_2$, such that each segment of S_1 and S_2 appears in at most one pair of C. A matching C is *maximal* iff all the segments of the smallest segmentation are in C, and therefore $|C| = \min(|S_1|, |S_2|)$. The *distance* of a matching C is defined as : $\sum_{(s_i^1, s_j^2) \in C} \mathcal{M}_{(i,j)}$. This stage aims to make the method robust to uninterpreted text modifications: if some comments are added, the size of the document will be greater than the original one, making the segments corresponding to this new text unlikely to be matched with the original segments. We use the Munkres algorithm [2] to perform the matching and obtain a $m \times n$ *matching matrix* $\mathcal{H}$ such that $\mathcal{H}_{(i,j)} = \mathcal{M}_{(i,j)}$ if $(s_i^1, s_j^2) \in C$ and $\mathcal{H}_{(i,j)} = 1$ otherwise. This algorithm performs in $O(\max(m, n)^3)$ time. The resulting matrix is then filtered with an identity convolution matrix to enhance the diagonals and remove the isolated matches.

Document-wise distance evaluation : From a filtered matching matrix $\mathcal{M}$ between two segmentations, we sum and normalize the matrix components with value less than 1, giving a distance in the range $[0, 1]$ (last line of algorithm 2).

Algorithm 2: DocumentDistance

Input: S_1, S_2: two sets of segments (one per document)
Data: *Dist*: a segment distance function
begin
  `/* Build the segments distance matrix M */`
  foreach $(s_i, s_j) \in S_1 \times S_2$ **do**
    $\mathcal{M}_{(i,j)} \leftarrow Dist(s_i, s_j)$
  `/* Find max matching with min distance */`
  $\mathcal{M}' \leftarrow Matcher(\mathcal{M})$
  `/* Post-filter the matching matrix */`
  $\mathcal{M}'' \leftarrow PostFilter(\mathcal{M}')$
  `/* Return the document-wise distance */`
  return $1 - \frac{1}{\min(|S_1|, |S_2|)} \sum_{i,j} 1 - \mathcal{M}''_{(i,j)}$
end

Corpus analysis presentation : At this point we assume we have the document-wise distance for every pair of documents of the corpus. Thus, we can put these distances in a spreadsheet where each cell contains a distance and is emphasized by a color. The cell color represents the similarity between documents wrt. the average corpus similarity, to this end we choosed *nested means* to classify the pairs of documents into 8 or 16 classes, each one colored on a scale going from green (legitimate documents) to red (probable plagiarism). We then used a hierarchical classification algorithm to produce a binary tree with documents on the leaves. The order on the leaves induced by the depth-first traversal of this tree was applied on the row and columns of the final distance matrix to group similar documents into neighbouring cells.

4. DISCUSSION AND FUTURE WORK

Many experiments were conducted on students source code, with some corpora with over 100 files and in many different programming languages such as C, Python, Haskell, Bash, PHP and Java. In most cases, plagiarism was found in suspected documents, but we do not determined if all the unsuspected ones were original work. For all tests, the results were given within minutes of computation time on a 2Ghz desktop workstation. In many corpora, the framework showed promising results : we were able to discover previously undetected frauds, even when the students were given a common codebase. It has been interfaced with the student homework repository of the University computer science department, so every teacher can now use it on his own corpora. This work is still in progress, we are working on an interactive corpus summary to ease the corrector work allowing him to explore the corpus at documents and segments levels.

5. REFERENCES

[1] R. Cilibrasi and P. Vitanyi. Clustering by compression. *IEEE Transactions on Information theory*, 51(4):1523–1545, 2005.

[2] H. Kuhn. The hungarian method for the assignment problem. *Naval Res. Logist. Quart.*, 2:83–97, 1955.

[3] Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10:707–710, 1966.