The Use of Machine Semantic Analysis in Plagiarism Detection

Maxim Mozgovoy[*]
University of Joensuu, Finland
mmozgo@cs.joensuu.fi

Vitaly Tusov
St. Petersburg State University, Russia
tusovvitalij@mail.ru

Vitaly Klyuev
University of Aizu, Japan
vkluev@u-aizu.ac.jp

Abstract

Plagiarism detection systems are known for years in the university community. However, most of the existing detectors for the natural language texts use rather simple comparison methods that make the instances of plagiarism easy to hide. The software, designed for plagiarism detection in computer programs, utilizes far more advanced techniques. We propose a method, which adds functionalities similar to tokenization and tree matching, to the natural language texts-oriented detectors. This method requires noticeable work to be applied in practice, but also makes use of the existing software for parsing and word sense disambiguation.

Keywords

machine semantic analysis, plagiarism detection, string matching, plagiarism, computational linguistics.

1. Introduction

Plagiarism in universities remains in the scope of interest of researchers for years. Scientific investigations cover various aspects of plagiarism: its origins, pedagogical and ethical issues, plagiarism prevention and detection, people's attitude, legal affairs and honor codes, etc. From the point of view of practical computer science, one of the most interesting directions is plagiarism detection.

Numerous systems were developed in recent years to detect plagiarism in natural language texts as well as in computer programs. Here we propose a possible technique that can help to improve existing natural language-oriented plagiarism detection software. This technique can be roughly treated as an analogue of a well-known *tokenization* procedure in program code-oriented plagiarism detection systems. Furthermore, we discuss a way of utilizing language parsers to deal with the rephrasing of the sentences.

Our current studies are based on *semantic analyzer* for the Russian language [1], but other similar solutions can be used as well. The detection system, which is described here, is just a part of our research on natural language processing. We are also trying to use semantic analysis for information retrieval tasks and for machine translation.

2. Related Works

Most existing plagiarism detectors are specially designed to process either program source code or natural language texts. In the first case the system usually treats a submitted collection of documents as hermetic and performs a pairwise comparison between single submissions only. Such projects utilize advanced techniques to detect partial matches (RKS-GST [2], matching in the repository [3]), and regular changes of the code structure (tokenization [4], p-matching [5]). The systems, designed to find similarities in the natural language texts, mainly search the Internet for the possible matches. Generally, they do not use sophisticated comparison methods, aiming mostly at processing speed and wide coverage (e.g. the developers of *Turnitin* [6] system claim they maintain "a huge database of books and journals, and a database of the millions of papers already submitted").

"Hermetic" systems for plagiarism detection in the natural language texts exist as well, though they are little-known. We can mention, e.g. CopyCatch Gold [7], YAP3 [8], and WCopyfind [9]. As a rule, the detection software can find only partial exact matches: rephrasing and rewording can conceal the evidence of plagiarism. CopyCatch Gold reduces the effect of rewording by taking into account only

[*] The corresponding author.

hapax legomena words (those that appear only once in the text) during the comparison, but this technique is not very reliable.

3. Tokenization

Tokenization [4] is a well-known method that makes useless all kinds of renaming tricks in plagiarism in computer programs (such as variable renaming and changing the type of loop structure). Tokenization algorithms substitute the elements of program code with single tokens. For example, any identifier can be replaced by the token <IDT>, and every numerical value by the token <VALUE>. Now, if a program contains a line **a = b + 45;** this line will be replaced by the string **<IDT>=<IDT>+<VALUE>;** So trying to rename the variables will not help since every line of the form “identifier = identifier + value;” is translated to the same tokenized sequence (the aforementioned example is taken from [10]).

Tokenization can be treated as substitution of single elements of some class by the name of the class itself. E.g. 5, 11.5 and -32 are elements of the class <VALUE>. In the natural language texts we can use the same approach. For instance, the words *device* and *gadget* are interchangeable in many contexts. If we substitute these words by the name of their class <MECHANISM>, such rewording will be useless for the plagiarizer.

4. Word Classes

The above described technique can be quite easily implemented by having a dictionary that matches every word of a natural language with the corresponding class. Our current version of the system (for Russian) includes more than 1600 classes that form a hierarchy. The small extraction from this tree is shown in Fig. 1.

For example, a class <PHYSICAL-OBJECT> has a subclass <ALIVE>, having, in its turn, a subclass <ANIMAL> that includes classes <ANIMAL-WILD> and <ANIMAL-DOMESTIC>. Undoubtedly, the problem of classification is very nontrivial, and no “best classification” can be invented. We created only one possible hierarchy that (according to our studies) satisfactorily reflects general knowledge about human environment. A variation of a specially created hierarchy might be helpful in order to process documents, dedicated to some narrow fields. For example, it is usually reasonable to consider *gadget* and *device* as direct successors of the class <MECHANISM>, but for the technical texts a more detailed classification of mechanisms will produce better results.

The use of subclasses can help to tune the tokenizer. For example, we might want to find more plagiarisms by widening the generalizations, e.g. it is possible to substitute the word *fox* with the more general class <ANIMAL> instead of <ANIMAL-WILD>.

Figure 1. A Fragment of Concept Classes Tree

```
something
├─noun
│ ├─general-concepts
│ │ ├─event
│ │ │ ├─situation
│ │ │ …
│ │ …
│ ├─physical-object
│ │ ├─alive
│ │ │ ├─animal-wild
│ │ │ ├─animal-domestic
│ │ │ …
│ │ ├─money
│ │ │ ├─banknote
│ │ │ ├─payments
│ │ │ …
│ │ …
│ …
…
```

The obvious difficulty concerns polysemantic words and homonyms. For instance, the system should select the correct class for the word *table* from the two alternatives — <FURNITURE> and <DRAWING/TABLE>. We may suggest to use any tool for word sense disambiguation (WSD), referenced in [11]. Our software relies on the results, provided by the semantic analyzer that performs WSD as well.

5. Fast Plagiarism Detection Algorithm

To obtain working software, we took a system [3], and substituted the tokenization module with the natural language version. The corresponding author is a member of the team that created the aforementioned system.

The system is intended for hermetic, many-to-many comparison of all files of the submitted collection of documents that contain Java listings. Most hermetic detection programs perform naïve pairwise file-to-file comparison, which results in $O(f(n)N^2)$ complexity, where N is the number of files in the collection and $f(n)$ is the time to make the comparison between one pair of files of length n. Our software tries to decrease the algorithmic complexity while preserving almost the same quality of detection.

The system firstly creates a suffix array from the tokenized collection of files. A suffix array is a lexicographically sorted array of all suffixes of a given string. It allows us to quickly find a file (or files), containing any given substring. A binary search is utilized to achieve this.

To find all collection files that are similar to a given query file, the system executes Alg. 1. It tries to find the substrings of the tokenized query file, Q[1..q], in the suffix array, where q is the number of tokens. Matching substrings are recorded and each match contributes to the similarity score. The algorithm takes contiguous non-overlapping token substrings of length γ from the query file and searches all the matching substrings from the index. These matches are recorded into a 'repository'. This phase also includes a sanity check as overlapping matches are not allowed.

Algorithm 1. Search a File in a Collection

```
p = 1 // the first token of Q
WHILE $p \leq q - \gamma + 1$
  find Q[p...p + $\gamma$ - 1] from the suffix array
  IF Q[p...p + $\gamma$ - 1] was found
    UpdateRepository
    p = p + $\gamma$
  ELSE
    p = p + 1
FOR EVERY file $F_i$ in the collection
  Similarity(Q, $F_i$) = MatchedTokens($F_i$)/q
```

In Alg. 2, the system encounters two types of collisions. The first one appears when more than one match is found in the same file. If several matches that are found correspond to the same indexed file, these matches are extended to Γ tokens, $\Gamma \geq \gamma$, such that only one of the original matches survives for each indexed file. Therefore, for each file in the index, the algorithm finds all matching substrings that are longer than other matching substrings and whose lengths are at least γ tokens.

Algorithm 2. Update the Repository

```
Let S be the set of matches of Q[p...p+$\gamma$-1]
IF some elems of S are found in the same file
  leave only the longest one
FOR every string M from the remaining list S
  IF M doesn't intersect with repository elems
    insert M to the repository
  ELSE IF M is longer than conflicting elems
    remove all conflicting repository elements
    insert M to the repository
```

The second type of collision is the reverse of the first problem: we should forbid the situation when two different places in the input file correspond to the same place in some collection file. To resolve collisions we use 'longest wins' heuristics. We sum the lengths of all the previous matches that intersect with the current one, and if the current match is longer, we use it to replace the intersecting previous matches.

The complexity of Algorithm 1 is highly dependent on the value of the γ parameter. Line 3 of Algorithm 1 takes $O(\gamma + \log n)$ average time, where is n the total number of tokens in the collection (assuming atomic token comparisons). If we make the simplifying assumption that two randomly picked tokens match each other (independently) with fixed probability p, then on average we obtain np^γ matches for substrings of length γ. If Q was found, we call Algorithm 2. Its total complexity is, on average, at most $O((q/\gamma \cdot np^\gamma)^2)$. To keep the total average complexity of Algorithm 1 to at most $O(q(\gamma + \log n))$, it is enough that $\gamma = \Omega(\log_{1/p} n)$. This results in $O(q \log n)$ total average time. Since we require that $\gamma = \Omega(\log n)$, and may adjust γ to tune the quality of the detection results, we state the time bound as $O(q\gamma)$. Finally, the scores for each file can be computed in O(N) time. To summarize, the total average complexity of Algorithm 1 can be made $O(q(\gamma + \log n) + N) = O(q\gamma + N)$. The $O(\gamma + \log n)$ factors can be easily reduced to O(1) (worst case) using suffix trees with suffix links, instead of suffix arrays. This would result in O(q + N) total time.

This analysis does not include tokenization, but it is a linear process (both for Java files and for the natural language texts), and the number of tokens depends linearly on the file length.

6. Tree Matching

Any sentence of the given text can be automatically represented in the form of the tree, which reflects the structure of the sentence. The principles of organization of such parse trees still serve as the subject of wide discussions. Most automatic English parsers use Chomsky-styled Penn Treebank grammars [12], based on the traditional linguistic approach to the syntax analysis. For example, the phrase *the monkey ate the banana* will be parsed by such software as shown in Fig. 2.

Figure 2. Parsing the Sentence

```
     ┌──────SENTENCE─────┐
 SUBJECT   └─VERB     OBJECT
 ├─ARTICLE   └─ate    ├─ARTICLE
 │ └─the              │ └─the
 └─NOUN               └─NOUN
   └─monkey             └─banana
```

Our semantic analyzer also builds a parse tree for any given sentence, but it is not based on Chomsky grammars (they are not well-suitable for the Russian language, because the order of the parts of a sentence in Russian is not fixed). The semantic analyzer treats the sentence as a control structure, having a functional nature. More specifically, it considers the sentence as a superposition of *words-functions* that depend on *words-arguments*.

The parse tree for the same phrase *the monkey ate the banana* in this model will look like this:

```
    ┌──────ate─────┐
 the_monkey    the_banana
```

Here the word *ate* is considered as a computable function of two arguments: the_monkey and the_banana.

Having ready-made parse trees (of any kind), we can invoke a tree matching procedure. This technique is described in [13] for the case of plagiarism detection in program code.

Initially the algorithm builds a flowchart-styled parse tree for each file to be analyzed. Then for each pair of files, the algorithm performs a rough "abstract comparison", when only types of the parse tree elements (like ASSIGNMENT, LOOP, BRANCHING) are taken into account. This is done recursively for the each level of tree nodes.

If the similarity percentage becomes lower than some threshold at some step, the trees are immediately treated as not similar.

If the abstract comparison indicates enough similarity, a special low-level "micro comparison" procedure is invoked. At this point each node represents an individual statement. Thus, each tree node turns into a separate subtree that has to be compared with the corresponding subtree taken from another file.

Note that the "abstract comparison" is a step when tokenized sequences are compared, so for our purposes (plagiarism detection over tokenized texts) we can skip the next "micro comparison" procedure. This technique seems to be the most advanced way of comparing structured documents, but our results in this direction are still very preliminary for any kind of evaluation.

On the other hand, it is already clear that the tree matching can help to reveal rewording. If we treat the children of every tree node as an unordered collection of nodes, e.g. the phrases *the monkey ate the banana* and *the banana was eaten by the monkey* will be very close after the tokenization.

7. Evaluation

The evaluation part is a very problematic issue for any kind of plagiarism detection system. It is especially hard for the software that searches the Internet for the possible occurrences of plagiarism, but even papers on "hermetic" systems usually just show the positive sides of the proposed approach [4, 13]. However, we can examine the reports that are produced by different plagiarism detection software when used on the same dataset.

The original system [3] was evaluated by using such "jury" method. The programs utilized for the analysis include MOSS [14], JPlag [2] and Sherlock [4]. Every system printed a report about the same real collection, consisting of 220 undergraduate students' Java programs (varying in size from 2 KB to 50 KB; the median length is 15 KB). Although the 'opinions' of all the tested systems are different for many of the files, most files are either detected or rejected by the majority of systems. This simple approach (to consider only detection or rejection) allows us to organize a 'voting' experiment. Let S_i be the number of 'jury' systems (MOSS, JPlag and Sherlock), which marked file *i* as suspicious. If $S_i \geq 2$, we should expect our system to mark this file as well. If $S_i < 2$, the file should, in general, remain unmarked. For the test set consisting of 155 files marked by at least one program, our system agreed with the 'jury' in 115 cases (and, correspondingly, disagreed in 40 cases). This result is more conformist than the results obtained when the same experiment was run on the other 3 tested systems. Each system was tested while the other three acted as jury.

For the evaluation of the new system we used a collection of 350 documents taken from the NEWSru.com news server. Each document had an informative title and was assigned to one of the following categories: *In Russia*, *In the World*, *Economics*, *Religion*, *Criminal*, *Sport*, and *Culture*. The size of the articles varies from 450 bytes to 19 KB with the median size of about 2 KB. The typical article consists of 8-12 small paragraphs that are made of strict narrative sentences and quotations.

This selection was based on the assumption that the newsreels often publish different documents on the same topic (though we do not expect direct plagiarism in this case), so the possibility to find similar files is quite high. Since we do not know about any other plagiarism detection systems that use natural language processing techniques, the results were analyzed manually.

The system found 20 relevant pairs of similar documents (with at least 4% degree of similarity). The typical examples include:

- A pair of documents about the solar eclipse on 20th of March. The first tells about the countries where this phenomenon is observed; the second is dedicated purely to the observation of the eclipse in Russia.
- A pair of documents on the weather conditions in Europe. The first is about floods in the EU; the second contains some weather predictions for the EU (including subsequent floods).
- A pair of documents on rumors about Russian military assistance to the Iraqi government in March of 2003. The first outlines the position of Moscow; the second states the reaction of Washington.

After tokenization the similarity degrees of the same file pairs increased (in most cases) by a factor of 1.5 or (in few cases) remained the same. Meanwhile, four additional false pairs were detected (but with very low similarity ratios that did not exceed 4-5%).

There are typical situations encountered in the experiment that noticeably affected the detection process after tokenization. They include:

- Changes to grammar cases in Russian. The phrases *in one of resorts* and *of one of resorts* are not matched at due to the changes of the endings of the words. After tokenization they become almost identical.
- The use of distinct words of the same classes in the same contexts in different documents. The phrases *the residence in Greece* and *the residence in Athens* do not match, but do match after the tokenization (*Greece* and *Athens* are translated to the same class <PLACE>).

The latter case is related to many mismatches as well. For example, the phrases *Vladimir Putin claimed* and *George Bush claimed* are treated as the same sequence <NAME><NAME><SPEAK> after the tokenization. It may be argued, though, that the system of classes we used was not specially designed for plagiarism/similarity detection procedures. A more advanced hierarchy may include a careful taxonomy that minimizes such collisions.

It should be noted that in the simplest case the tokenization can be considered as a variation of stemming technique that is widely used in information retrieval. Although, for some languages (including Russian) stemming procedures are not simple, since they have to deal with many non-trivial grammatical issues.

The use of tokenization results in the immediate increase of the number of matches. Tokenization of level 0 (stemming), level 1 (with the terminal classes in the hierarchy) and level 2 (with the direct ancestors of the terminal classes) makes sense, but the use of tokenization of higher levels results in many false matches, since the classes become too general. For example, tokenization of level 3 substitutes the word *cat* with the class name <ALIVE>. The same class corresponds to all alive objects, such as *worm*, *chairman* or *wife*.

The system also did not detect several file pairs that could be treated as similar under certain conditions. For example, some documents can have only a few common substrings, but most human readers consider them as similar. The manual analysis shows that the number of such pairs in our collection is less than five, and the corresponding files are not originated from the same source, i.e. they do not contain instances of plagiarism.

8. Conclusion

Plagiarism detection for text in natural languages is a challenge. Most natural language processing tools, such as parsers and taggers remain unused by the authors of plagiarism detection systems. Also these tools are language dependent and designed for English.

Our approach gives a possible solution to make a language independent system to determine plagiarism in collections of the texts. The key idea behind it is the use of hierarchies of concepts and the functional style of representing the sentences. To implement the system, we adopted algorithm [3], applied the tokenization technique and the tree matching procedure. We utilized the concept hierarchy for the Russian language. Our tests showed the promising results which include intelligent tokenization and high speed processing of the text data (O(q + N) is required to test a query file of size q against a collection of N files). The tree matching procedure is still very experimental, but we believe that it can significantly improve the quality of plagiarism detection. Our solution is scalable (see [3]), so it is suitable for large essays banks.

References

[1] V.A. Tusov, *Computer Semantics of the Russian Language (in Russian)*, S.-Petersburg University Press, S.-Petersburg, 2004.

[2] L. Prechelt, G. Malpohl, and M. Philippsen, *JPlag: Finding Plagiarisms among a Set of Programs*, Technical report, Fakultät für Informatik, Universität Karlsruhe, Germany, 2000.

[3] M. Mozgovoy, K. Fredriksson, D. White, M. Joy, and E. Sutinen, "Fast Plagiarism Detection System", *Lecture Notes in Computer Science*, vol. 3772, 2005, pp. 267-270.

[4] M.S. Joy, M. Luck, "Plagiarism in Programming Assignments", *IEEE Transactions on Education*, vol. 42(2), 1999, pp. 129-133.

[5] B.S. Baker, "Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance", *SIAM Journal on Computing*, vol. 26(5), 1997, pp. 1343-1362.

[6] Turnitin: www.turnitin.com

[7] CopyCatch Gold: www.copycatchgold.com

[8] M.J. Wise, "YAP3: Improved Detection of Similarities in Computer Program and Other Texts", *Proceedings of SIGCSE '96*, 1996, pp. 130-134.

[9] WCopyfind: plagiarism.phys.virginia.edu

[10] M. Mozgovoy, "Desktop Tools for Offline Plagiarism Detection in Computer Programs", *Informatics in Education*, vol. 5(1), 2006, pp. 97-112.

[11] Ph. Edmonds, A. Kilgarriff (Eds.), *Journal of Natural Language Engineering (Special Issue Based On Senseval-2)*, vol. 9(1), 2003.

[12] M.P. Marcus, B. Santorini, M.A. Marcinkiewicz, "Bulding a large annotated corpus of English: the Penn Treebank", *Computational Linguistics*, vol. 19, 1993, pp. 313-330.

[13] B. Belkhouche, A. Nix, J. Hassell, "Plagiarism Detection in Software Designs", *Proceedings of the 42nd Annual Southeast Regional Conference*, 2004, pp. 207-211.

[14] S. Schleimer, D. S. Wilkerson, A. Aiken, "Winnowing: Local Algorithms for Document Fingerprinting", *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, 2003, pp. 76-85.