UNIVERSITATEA POLITEHNICA BUCUREŞTI
FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

LUCRARE DE LICENŢĂ

AuthentiCOP: Sistem pentru detecţie automată a plagiatului în corpusuri specializate

Coordonatori ştiinţifici:
Conf. dr. ing. Răzvan Rughiniş
Ing. Traian Rebedea

Absolvent:
Filip Cristian Buruiană

BUCUREŞTI

2012

UNIVERSITY POLITECHNICA OF BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE DEPARTMENT

DIPLOMA PROJECT

AuthentiCOP: Automatic Plagiarism Detection System for Specialized Corpora

Thesis supervisors:
Conf. dr. eng. Răzvan Rughiniş
Eng. Traian Rebedea

Filip Cristian Buruiană

BUCHAREST

2012

Table of Contents

Plagiarism in academic writing is considered one of the worst breaches of professional conduct and automatic plagiarism detection has become an important use case for Natural Language Processing research. Due to the ambiguity of natural language, the automatic detection is an open problem to date, although several commercial solutions with encouraging results exist on the market. However, as the task gained increasing interest in the academic field, intensive research was carried out and several programs were tested and implemented, some of them with very good results. This thesis focuses on designing and implementing AuthentiCop, a system for detecting plagiarism instances in Computer Science academic writings. We evaluate and compare several variations of well-known algorithms, while trying personal approaches as well. Also, a scalable and maintainable software architecture is created, such that this work can be continued and improved continuously. Finally, our results are compared to those obtained by teams across the world who participated in the PAN plagiarism detection competition.

Keywords: plagiarism detection, Information Retrieval, Natural Language Processing, PAN competition

1. Introduction

According to dictionary.com, plagiarism "is an act or instance of using or closely imitating the language and thoughts of another author without authorization and the representation of that author's work as one's own, as by not crediting the original author". The word "plagiarism" is derived from the Latin word "plagiare", which means "to steal".

Plagiarism is a growing concern in the academic community. As an investigation at Duke University points out, "about 40% of the surveyed students confessed to copying sentences without citing the original source, while 11% reported almost verbatim copying of material", according to [1], which cites Robert Bliwise's work, "A matter of honor". Other statistics showing the prevalence of this issue are presented in [2], based on a research conducted by Coverdale and Henning in "An analysis of cheating behavior during training by medical students": "29% of a group of US medical students surveyed admitted falsifying references and 17% had submitted material copied from previous year's papers".

Even though certain uses of vocabulary, incoherent text or common spelling mistakes are a good signal to indicate plagiarism, it is basically impossible for a reviewer to detect highly obfuscated passages, which are based on paraphrasing or reuse of structure and ideas. This problem has become even more difficult to tackle nowadays, because of the multitude of written materials available, which increases constantly mainly due to the Internet. This is why automatic detection tools are gaining further interest, being able to offer valuable hints in case of plagiarized works. According to Maurer et al [3], plagiarism can not only be intentional, but also accidental (due to lack of understanding citation or improper quotation) or unintentional (due to the vastness of materials available, which increases the chances that some idea appears in someone else's work), which makes the plagiarism detection task a complex one, even with sufficient computing power available.

The first approaches to identify plagiarism and collusion automatically can be traced back in the 1970s, when tools were created to check programming assignments handed in by students [4]. Plagiarism in free text is harder to detect, because of the ambiguity of natural language, but caught the attention of scientific research more recently. There are even international competitions on plagiarism detection, such as the annual International Workshop on Uncovering Plagiarism, Authorship, and Social Software Misuse (PAN) [5], where teams across the world submit their works and are evaluated under a given framework.

Because of the prevalent nature of this problem, the need for an automatic plagiarism detection system for University Politehnica of Bucharest is essential for maintaining academic integrity and a high value for graduation theses. Based on the latest research in the field, the system described in this work, AuthentiCOP, is able to provide professors and students with an online platform for helping them to minimize the plagiarism cases. However, this tool is not supposed to replace the human factor, the ultimate decision having to be taken only after a manually assessment of the results.

1.1 Overview

Plagiarism detection divides into two major problem classes: external plagiarism detection and intrinsic plagiarism detection [5]. External plagiarism refers to the case in which a reference collection is available. This is similar to intra-corpal detection, described in [6], because attention is restricted to a given corpus of documents. On the other side, intrinsic detection aims at detecting plagiarism based on methods such as stylometry, which look for variations and inconsistencies in the author's writing style [7]. These methods can be used to detect plagiarism either when no reference is given or when the size of the corpus is too restrictive to be indexed and queried, as in the case of World Wide Web. However, when dealing with the Web, the alternative is to use the information

already existent in search engines to find similar documents to a given query, approach known as Document Retrieval.

A detection engine can be either general or aimed towards specialized corpora, i.e. only consider works in the medical or Computer Science field. In the latter case, the problem presents some interesting particularities which can be exploited, such as using a terminology known in advance, automatic term extraction [8] or detecting features to enable text classification, based on a fixed number of predefined categories [9].

It is even possible to deal with plagiarism in a multilingual setting, as described in [10]. In this case, methods from Cross-Language Information Retrieval can be used, or machine translation can be employed, using public services such as Google Translate or BabelFish.

Our system is aimed at Computer Science works written in English, comparing them to a collection of already available documents. In this respect, given the classification above, AuthentiCOP falls under the external detection category.

The need for a new plagiarism detection system at Politehnica University may not seem so obvious, given that several commercial detection engines are available today, both for free text and for source code. Turnitin [11] is an example of a leading academic plagiarism detector, used by teachers and students to avoid plagiarism and to ensure academic integrity. It is a well known web service for plagiarism detection in academic writings, having a database of over 220 million archived student papers, according to their website. The ANTIPLAG system [12] has been developed since 2008, and from 2010 was implemented as an official project by Slovak Centre of Scientific and Technical Information under the auspices of the Ministry of Education of the Slovak Republic. This system ranked first in the PAN 2011 competition, obtaining the best score for all four main parameters. While most of the engines offer plagiarism prevention services, by detecting unoriginal content, there are also some services which, in addition to detection, educate students about the negative consequences in case of academic misconduct. SafeAssign is such a service [13].

Even with so many products available on the market, some options are too expensive to consider, while the cheaper ones produce unsatisfactory results. Also, as presented above, taking into account any particularities of the problem may benefit the final results, as it is expected, for example, that a plagiarism detection engine targeted only at Computer Science papers would produce at least the same results as a general engine, in terms of quality. The system we create can be very flexible to the new requirements that appear, and although we do not consider all particularities for now, we leave behind a system that can be easily maintained and improved in-house.

1.2 Information Retrieval in Plagiarism Detection

There are several well-known techniques from Information Retrieval that can be applied successfully to the plagiarism detection task. All these techniques are presented briefly in the next sections.

1.2.1 Vector Space Model and Weighting Schemes

The first thing that needs to be considered is a standard, formal representation of free text. The Vector Space Model (VSM) is widely used for this purpose. In this model, documents are represented as vectors, where the size of each vector is equal to the cardinality of the vocabulary, i.e. the number of distinct terms in all considered documents:

$$d = (w_1\ w_2\ \dots\ w_n) \quad (1)$$

In Equation (1), n is the cardinality of the vocabulary and $w_1\ w_2\ \dots\ w_n$ are the non-negative term weights of the document d. It can be seen that any word order is lost. This is why the model is also referred to as *bag of words*. Each dimension in the vector corresponds to a term. If a document contains that term, then the corresponding weight in the representation is different than 0. The procedure which attributes weights to terms is known as weighting scheme.

One of the best known schemes is "term frequency - inverse document frequency" (tf-idf). In this scheme, the following equations hold:

$$tf_{t,d} = \sum_{i=1}^{|d|} \begin{cases} 1, & if\ d_i = t \\ 0, & otherwise \end{cases} \quad (2)$$

$$idf_t = log \frac{|D|}{|d \in D : t \in d|} \quad (3)$$

$tf_{t,d}$ is the term frequency of term t in document d, idf_t is the inverse document frequency of t, and D is the collection of all documents (the corpus). Then, the weight of t in d would be $tf_{t,d} \cdot idf_t$. For example, let's suppose that we have a document in which the word *computer* appears 4 times. If we have a collection of 10 000 documents and the word *computer* appears only in 100, then the weight would be 4 x log(10000 / 100) = 8, if we consider the 10-base logarithm. The reasoning behind this is that rare words would be considered more important than frequent words, and that the more a term appears in a document, the more important it is. For instance, the match of the word *stylometry* in two different texts is more likely to show that the texts are related than if the word *computer* matched. Similarly, two texts are more closely related if the word *graphics* matched 20 times instead of only one time. One thing to note is that there are words that carry no information. These words, also known as stop-words, are the most common, and are words such as *the*, *and* or *are*. In the tf-idf scheme, the weight of stop-words is 0, and they are basically ignored.

The tf-idf scheme has many variations, as presented in [14]. For example, $tf_{t,d}$ can be a binary value (1 for terms present in document and 0 for all other terms), or the number of appearances in the document (as presented in Equation 2), or even:

$$tf_{t,d} = 0.5 + 0.5\ \frac{tf}{\max(tf)} \quad (4)$$

which is the augmented normalized term frequency (tf factor normalized by maximum tf in the vector, and further normalized to lie between 0.5 and 1). Also, in [14], the SMART notation is introduced, to refer to the multiple variants proposed.

1.2.2 Similarity Measures

Having a formal way of representing text, the next question is how to compare two representations. A similarity function is a function $f: V\ x\ V \rightarrow \mathcal{R}$, where V is the set of all possible vector representations. Thus, the function receives two vectors and returns a real number which quantifies how related are the two vectors. In Information Retrieval, these measures can be used by search engines to rank matching documents by their relevance to a given query. The higher the similarity, the more relevant the document. There are many similarity measures which can be used, the most known being enumerated below.

$$\text{overlap}(A,B) = \sum_{i=1}^{n} \begin{cases} A_i \oplus B_i & if\ A_i\ x\ B_i \neq 0 \\ 0 & otherwise \end{cases} \quad (5)$$

$$\text{jaccard}(A,B) = \frac{\sum_{i=1}^{n} C_i}{\sum_{i=1}^{n}(A_i \oplus B_i)},\ with\ C_i = \begin{cases} A_i \oplus B_i & if\ A_i\ x\ B_i \neq 0 \\ 0 & otherwise \end{cases} \quad (6)$$

$$\cos ine(A,B) = \frac{\sum_{i=1}^{n}(A_i\ x\ B_i)}{\sqrt{\sum_{i=1}^{n} {A_i}^2}\ x\ \sqrt{\sum_{i=1}^{n} {B_i}^2}} \quad (7)$$

$\oplus$ replaces any binary operator (addition, subtraction, product, min, max, etc). The overlap similarity described in (5) is a measure of intersection. The Jaccard similarity in (6) is the measure of intersection over the measure of reunion. The cosine similarity, presented in (7), is the cosine of the angle between vectors A and B (because it is the dot product over the norms of the two vectors), and is non-negative because all the terms in the equation are non-negative. Moreover, the Jaccard and cosine similarities are normalized, i.e. they produce real values between 0 and 1, with 0 meaning that the passages have no term in common, and 1 meaning that the passages are identical under the bag of words model.

There are also other similarity measures, such as the Hamming distance (also known as symmetric difference) or the editing distance (also known as Levenshtein distance). The editing distance for example is the minimum number of operations needed to transform the first text into the second, where an operation can be an insertion, deletion or substitution of a term. Because this is computationally expensive, it is generally not used, or if it is, is used in a post-processing phase.

1.2.3 Stemming

Using similarity measures presented in the previous section together with the Vector Space Model has the advantage of being simple and efficient. However, certain correlations between words, even at the most basic level, can be easily missed. For example, under the VSM representation, words like *computer* and *computers* appear to be different. The same undesired effect will be achieved when changing tenses (i.e. *write* vs. *wrote*), changing grammatical persons (i.e. *compute* vs. *computes*) or changing part of speech (i.e. *beauty* vs. *beautiful*). Although these words may carry different information depending on the context they are in, in most cases these variations are a result of paraphrasing, which, in case of plagiarism detection, need to be identified as such.

This problem can be partially solved by using stemming. A stem is part of a word, usually its base form. For example, the stem for *computer* is *computer*, for *trying* is *try* and for *previously* is *previous*. The stem does not need to be a valid word in the dictionary. For instance, the stem for *beauty* is *beauti*. This allows other words, such as *beautiful*, to come under the same stem.

The easiest approach to obtain these stems is based on a dictionary, containing all the words together with their stems. Other approaches can use Porter's stemming algorithm for English language words or free libraries such as Apache SnowBall [15]. After reading a document, all words will be stemmed, to enhance the probability of catching paraphrasing.

1.3 Related Work

Plagiarism detection techniques have drawn the attention of researchers in the last decades. The methods used range from very simplistic and ad-hoc, to more advanced techniques, based on machine learning or mathematical algorithms, such as Singular Value Decomposition (SVD) or Principal Component Analysis (PCA).

As stated earlier, the first attempts were aimed at detecting cheating in programming assignments. Moss (Measure of Software Similarity) is a system widely used for this purpose, as it can analyze code in many programming languages, and is based on Winnowing [16]. Although it is a system for determining software plagiarism, the fingerprinting method it uses can be applied to the free text problem as well. As Moss, YAP3 also detects similar computer program texts, its underlying algorithm being Running-Karp-Rabin Greedy-String-Tiling (or RKS-GST).

In the 90s, the focus was on copy detection in Digital Libraries. COPS [17] and SCAM [18] are two systems dealing with this issue. Both systems are similar in terms that they both use a sort of hashing on text chunks. What is more, SCAM uses an inverted index, together with a ranking system for ordering suspicious documents according to relevance. Developed during the late 90s for the same purpose as COPS and SCAM, the CHECK system [19] uses ORACLE database management system for indexing features, called structural characteristics (SC). Then, weighted cosine similarity is computed on a set of extracted keywords.

In the last decade, the focus moved from copy detection to detecting plagiarism in academic works. In [20], two different approaches are presented and compared. The first method is based on Patricia trees, a data structure very similar to a trie, that allows fast string comparison, functionality also offered by other data structures (i.e. suffix trees). The second method produces better results and uses shingles, which are distinct n-grams, together with Jaccard similarity. The Ferret system [21] has a similar implementation, being a fast plagiarism detector searching for matches of three-word sequences or trigrams. It is based on the empirical assumption that texts have usually a comparatively low level of matching trigrams, assumption confirmed by the research carried out on the TV news corpus, the Federalist papers and the Wall Street Journal corpora. Because the metrics is computed without any knowledge of the linguistic rules, according to the classification described in [6], the system is superficial.

A structural system is proposed in [22], based on synonym replacement, Latent Semantic Analysis (LSA) and topic detection. In [23], synsets (group of words considered semantically equivalent) are used, together with fuzzy similarity, to detect the resemblance of sentences. Similar consecutive sentences are then joined to form the final plagiarized passages. The synsets from WordNet [24] are used. A complex approach based on machine learning is described in [25], dealing with Document Retrieval (DR). Having a large collection of documents and a query, the problem is to select the documents that are the most relevant to the query. For each functional unit, such as a page or a paragraph, a set of features is computed, obtained after applying Principal Component Analysis (PCA) on the word histogram. Then a Multilayer Self Organizing Map is trained to return the closest document for the detected features.

Plagiarism detection is also related to methods in other fields. As mentioned in [26], sequence alignment heuristics can be useful when comparing in detail two documents, thus the connections to bioinformatics and image processing. Smith-Waterman, used in finding local alignments within biological sequences, is also employed in plagiarism detection [27].

The continuous expansion of the Internet, which resulted in more and more materials available in the "on-line" world, established a new requirement for modern systems: detect plagiarism against the documents on the web. DOCODE-Lite [28] is such a system, using tf-idf weighting and randomized term extraction to generate queries which will be given in parallel to multiple search engines. A unified scoring function is then used to merge the results and retrieve the

top documents, for which an in-depth analysis is carried out. An even harder problem is when original passages, eventually taken from the Web, are translated into another language. Cross Language Plagiarism Detection uses method from CLIR (Cross Language Information Retrieval), such as CL-ASA or CL-ESA [10], together with machine translation, to handle this type of detection.

Without a formal measure to rank the quality of the produced results, the multitude of algorithms presented above would have been very hard to compare. The PAN workshop [29] is an annual workshop dedicated to Natural Language Processing topics, such as plagiarism and authorship identification, Wikipedia vandalism and social software misuse. The workshop competition provides with an invaluable dataset and benchmark for the algorithms. In 2011, out of 30 registered groups around the world, 11 submitted runs for the PAN competition.

1.4 Our Contribution

In developing our plagiarism detection system, we have evaluated the best ranking solutions submitted to the 2011 PAN edition and we have built on the approaches published on this occasion, mainly focusing on FastDocode [30], Encoplot [31][32] and their applications. Apart from these algorithms and other personal approaches, we also experimented with All-Pairs [33] and ppjoin [34] for candidate selection, and tried several variations of some of the algorithms presented so far.

For an enhanced user interaction with these algorithms, we implemented an online platform allowing users to upload documents to be compared with a collection of documents. Detected similar passages are displayed in a web interface for further manual introspection. A limitation of the existing approach is that it limits the searching to a predefined corpus and does not query the whole Web. However, we designed an architecture such that any additional features can be integrated easily. A repository containing the source code, together with other resources (dictionaries, stem words, word frequencies, libraries, etc), is left for further development.

1.5 A Method of Evaluating Results

We evaluate our results in terms of precision and recall, which are also widely used in Information Retrieval. The precision is intuitively defined as the total length of correctly identified cases over the total length of all identified cases. The recall is defined as the total length of correctly identified cases over the total length of all correct plagiarism cases. Although these two measures can be analyzed separately, they are combined into an overall performance score, F-measure (or F1-score), defined as:

$$F = \frac{2}{\frac{1}{precision}+\frac{1}{recall}} = 2 * \frac{precision*recall}{precision+recall} \quad (8)$$

It can be observed that the F-measure is the harmonic mean of precision and recall. Another measure, granularity, is introduced in [5] to quantify the contiguity of the detections and to encourage plagiarized passages to be reported as one instance instead of two disjoint passages. Granularity is then added to the final score:

$$PlagDet = \frac{F}{log_2(1+granularity)} \quad (9)$$

All the submitted programs at the PAN competition were ranked by the PlagDet score presented in Equation (9). In order to evaluate the performance of our algorithms, we tested them on the 2011 corpus, computing the PlagDet score to determine what rank we would have obtained in the competition.

As presented in [35], the PAN corpus contains over 22 000 real documents, taken from Project Gutenberg [36]. The vast majority of the documents is in English, but there are also documents in German and Spanish. The plagiarism cases have different degree of obfuscation, from none to high, and the length of these cases ranges from 50 to 5000 words. There are both manually and artificially plagiarized passages. Because of this wide distribution of parameters, a software producing good results on this corpus is likely to produce good results when given real data.

2. System Architecture

Developing a plagiarism detection system raises both technical and algorithmic challenges. On one hand, such a system will most likely have modules written in many programming languages that need to be interconnected. In our case, we use C++, Java, PHP, JavaScript and Bash scripts. Also, we need to integrate multiple frameworks, each performing a specific task (i.e. converting documents to raw text or running semantic analysis). On the other hand, the system needs to use a multitude of algorithms, from topic detection to local alignment algorithms. Also, there are size issues: a corpus can contain thousands of documents, each containing tens of thousands of words.

In this section we focus on building an architecture that is scalable, easily maintainable and that can be distributed easily if more computer power is added.

2.1 Overview

Whenever creating a plagiarism detection system, there are common approaches which involve some general steps that need to be performed, as suggested in [10][28][37]. These steps are presented in Figure 1.

The evaluated document is uploaded using a graphical interface, which in our case, is web-based. Afterwards, the document is transferred to a converter, which obtains the raw text starting from a PDF, DOC, DOCX, or an HTML file. Because of the multitude of formats supported, we use Apache Tika [38], a project of the Apache Software Foundation, which extracts meta-data and structured text content from various documents using existing parser libraries. Also, at this step, it is usual to do all the necessary preprocessing, i.e. stemming and stop-words removal.

In an ideal scenario, the uploaded document should be compared both against an existing corpus and the Web. It is possible to compare the document against the Web using simulation of human behavior. First, a set of related topics is determined, and then the first documents returned by search engines when queried against the topics are fetched and analyzed for plagiarism. Multiple search engines can be queried (i.e. Google and Bing for Web pages, scholar.google.com for scientific papers and journals) and a unified relevance scoring function can be computed, as suggested in [28]. However, at this stage of development, AuthentiCOP only uses a corpus, and doesn't include any Web detection features. The corpus contains a collection of theses from previous years and all Computer Science articles from the English Wikipedia. Because there are thousands of documents in the corpus, a candidate selection phase is introduced, which will return the documents that the current text is most likely to plagiarize from.

The candidate documents from both the corpus and the web are passed to a detailed analysis phase aiming to find the exact plagiarized passages. There can also be a post-processing

phase, where results from the detailed phase are checked in even more detail, using semantic analysis and local alignment techniques.

The results are displayed in a web interface. The evaluated document and the source documents, if any, are displayed side by side, and the plagiarized passages are marked with distinct colours.

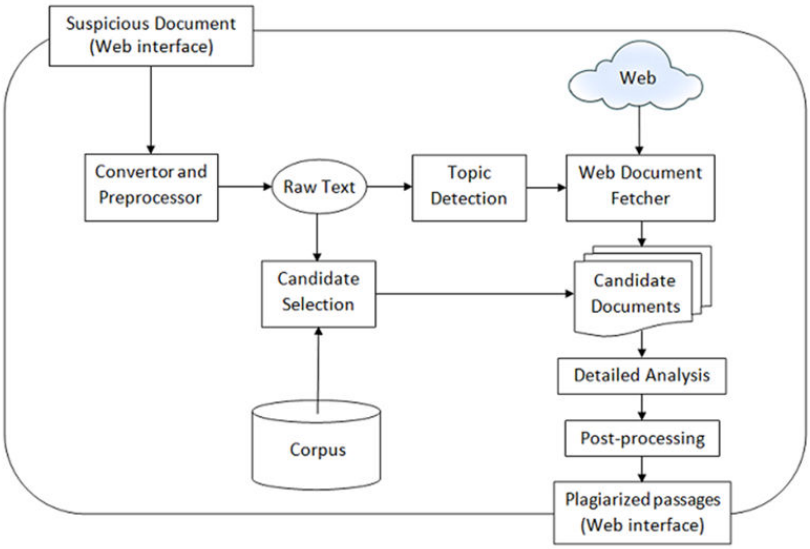

Fig. 1. Stages in a general plagiarism detection system.

2.2 Architectural Problems

The first thing we took into consideration when building AuthentiCOP was a separation between the components the user interacts with, such as the panel which displays plagiarized passages side by side or the form used for uploading documents, and the internal processes of deciding which passages are plagiarized. In this respect, we make a clear distinction between these two parts: all the user interaction comes under the *frontend* part, while the actual processing of the documents and the decision making are the responsibility of the *backend*.

AuthentiCOP is a browser-based application, the service being accessible from a web location. The user can upload the documents that need to be checked directly from a web interface and then he can see the results of the analysis in a separate panel. For this purpose, PHP is the scripting language used on the server-side to render a requested page, while JavaScript is added to improve the client-side experience. According to the frontend-backend separation, this code falls under the frontend part. After the user uploads his documents, there are several stages that need to be completed on the backend part (candidate selection, detailed analysis, etc), as presented in the previous section.

Algorithms detecting plagiarism detection are both IO and CPU intensive. These algorithms first need to read the corpus, which contains thousands of documents residing on a hard-disk or even on a distributed file system, and after this they need a considerable processing power to find the matches. For this reason, we considered that PHP is not suitable for this task, due to the fact that it is too slow, not only because it is interpreted but also for other language-specific reasons, such as its dynamic typing, which makes the variable types to be checked at run-time instead of compile-time. The approach to solve the performance issues is to implement the algorithms in a fast

compiled language, such as C++. The alternative for speeding up would be to first write the code in PHP and then automatically translate this code to C++, using a dedicated tool, such as HipHop for PHP [39]. This is an open source project developed by Facebook and, according to [39], the company sees about a 50% reduction in CPU usage when serving equal amounts of Web traffic when compared to Apache and PHP. Even though this approach wouldn't be too hard to put into practice, we decided to implement the backend in C++ directly, to take advantage of language specific features, like mapping files in memory or using templates to bring some computations from run-time to compile-time.

Writing the frontend in one language (PHP) and the backend in another (C++) raises the problem of communication between the components. Also, for some parts, like converting PDF files to raw text, we would like to use existing tools, that need to be integrated with the rest of the architecture even though they might be written in different languages, such as Java.

The cross-language communication and in-depth analysis for each component are presented in the next sections.

2.3 Main Components

When building the architecture, we focused on scalability and easy maintenance. For example, the present work only detects plagiarism when having an available corpus and does not deal with the entire web, but adding this facility (which may include topic detection and querying search engines) should be integrated with minimum, if with no effort, with the rest of the components. Also, if having enough computing power, the computations can be easily distributed, as we will show below.

The frontend and the backend, which are written in different programming languages, communicate with the help of an intermediate component, which is the Broker. The Broker is in fact a server listening on a specific port and, depending on the request, may choose to initiate new processes. The Broker is written in C++ and communicates with the frontend using sockets. The interactions between all these components is presented in Figure 2.

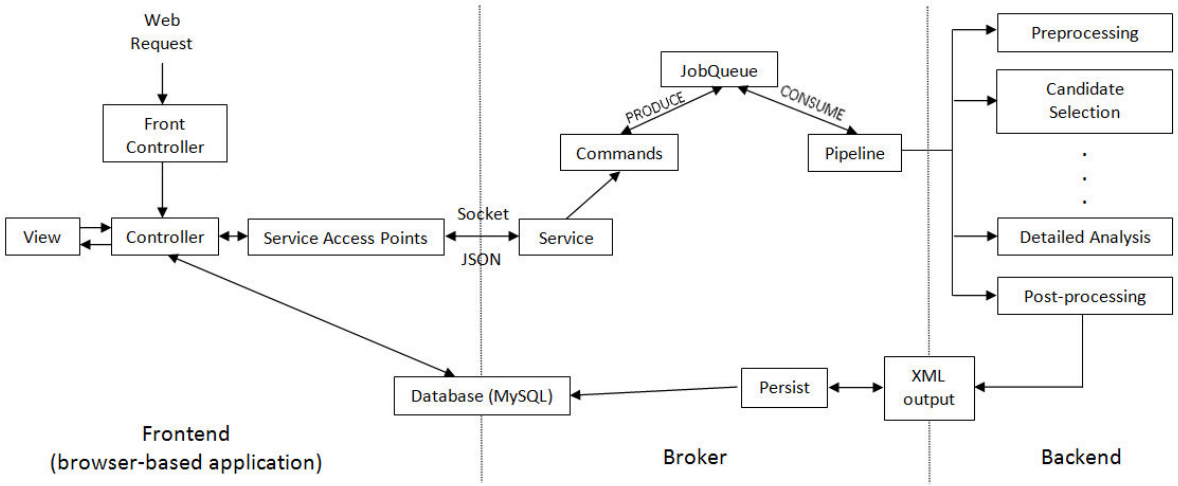

Fig. 2. Interaction between the main components.

The separation of the frontend, the Broker and the backend, not only has the advantage of readability and easy maintenance, but also allows different components to run on different machines. For example, the web server might be hosted on a shared machine which does not have enough computing power to run the backend algorithms. In this case, the Broker and the backend may run on a different machine, the only requirement being the point-to-point connectivity between the two machines. To do this kind of setup on different machines, the only thing that needs to be changed is a line in a configuration file, which holds the machines' IPs or DNS names. For testing purposes, during development, all these names are localhost, as all the components run on the same machine.

2.4 Frontend Architecture

The component the user interacts with, as it can be seen from the Figure 2 above, is the Frontend. The application is web-based. We use Apache as a web server, PHP and JavaScript as the main languages for the code, and MySQL for database support. We also use other technologies (i.e. AJAX) and libraries (i.e. jquery) on the frontend to create a smooth user interaction.

The frontend follows the Model-View-Controller design pattern. In fact, because the browser application is not too complex, containing only a few different views and forms and having a simple database schema, with only a few tables, the model part is ignored, the SQL queries and data manipulation being performed directly in the controller. Following this design pattern, there is a clear separation between the rendering part of the page and the logic of the application (managing uploaded files, reading files from disk, communicating with the Broker, etc). What is more, the structure is simple, without unnecessary complications. For example, we avoided using a PHP framework, such as Zend or CodeIgniter, mostly because they were too complex for our needs, containing many unnecessary libraries and features that were beyond the scope of this project, i.e. filters, a layer of abstraction over several database types, support for internalization, roles (Access Control Lists), etc. Instead, we implemented the pattern from scratch.

The URLs are rewritten using Apache's rewrite module, every request being redirected to the index page. An excerpt from the .htaccess configuration file, which makes this redirection possible, is presented below.

```
RewriteCond %{SCRIPT_FILENAME} !-f
RewriteRule (.*) index.php?page=$1  [QSA,L]
```

Fig. 3. htaccess excerpt, redirecting every request to the index page

Thus, the index page acts as a Front Controller in well established PHP frameworks, being an entry point for the application. The initial request is preserved in the page attribute, which is then split to determine both the controller to forward the request to, and the proper method of the controller to call. Initial GET arguments are appended to the URL because of the QSA flag (query string append). Besides forwarding the request to another controller, the index script performs common tasks, such as loading the necessary dependencies, using require_once, and establishing the connection with the database and with the Broker.

2.4.1 Controllers

The frontend component responsible with the business logic and data manipulation is the controller. A controller is in fact a PHP file containing one or more functions, every function being either an action or a helper. For convenience, the web application is procedural.

An action is a method that will be executed as a result of a Web request. By convention, action functions have their name in the following format: $CONTROLLER_$ACTION, where $CONTROLLER is the name of the controller (the name of the current file, without the PHP extension) and $ACTION is the name of the method to be called. For example, the URL http://$SERVER/analyze/paired ($SERVER being the server name of the server hosting the application) would be rewritten to http://$SERVER/index.php?page=analyze/paired. The index script will determine that it has to forward the request to the controller named *analyze*, calling its method named *paired*. A controller can contain more than one action, so logically grouped actions can go under the same controller. For example, the *analyze* controller may contain two actions: one for showing all the documents the analyzed file might have plagiarized from, and one for displaying a side-by-side comparison of the plagiarized passages for one source document. If no action is specified (if the URL is only http://$SERVER/analyze, for example), the default action is loaded, which is the index action. Every method that has its name starting with underscore is private, meaning that it cannot be accessed from the URL and serves only as a helper to actions.

The role of the controller is to prepare the data that will be displayed in the view. This data will be placed in a global variable named $view, which is only an associative array. The controller can fetch and prepare multiple fields (i.e. a list of possible source documents, the path on the disk of the suspicious file, error or warning messages), and unrelated fields can come under different keys. When rendering the page, the view will look for the necessary keys, i.e. it will get $view['error'] to check if there was an error on the controller side.

The data fetched by the controllers can come from two different sources: from the database and from the backend, through the Broker. The data stored in the database may refer to already completed actions, while the data from the Broker may give information about actions that are in progress. For example, every document that is sent for analyzing is placed in a waiting queue. This queue is stored in the Broker's RAM memory, and when the processing of the document has been completed, the results are stored in the database, for further reference. In this way, the user can see the whole history of uploaded documents, but can query the status of in-progress actions as well. Moreover, if the Broker happens to crash, the results for already processed documents are not lost.

To query MySQL databases, we use the API offered by PHP. The communication with the Broker is done through sockets, the messages being represented in the JSON format for convenience. To interpret these messages, json_encode and json_decode are used.

2.4.2 Views

The views are used only to render pages, having the data to be displayed already fetched by the controller in the $view associative array. CSS is used to add styles to elements. We ensured that all the new versions of the major browsers (Chrome, Firefox and Internet Explorer) are compatible and that the application works properly in all of them.

The user uploads his documents using a form. By clicking on the Browse button, he can select the file to be analyzed from his local hard disk, using the window that appears. Below the uploading form there is a monitor for the result queue, which contains the status of all submitted documents: already processed documents for which results are available, documents that are

currently being processed and documents that are in a waiting-state, until there is processing power available. To add interactivity, the monitor automatically refreshes itself, without reloading the whole page. AJAX is used for this purpose, and in the registered callback called when receiving the response, a new request is scheduled after a timeout of 2 seconds. The timeout is added using the jquery JavaScript library. The panel with all the functionalities described above is presented in Figure 4 below.

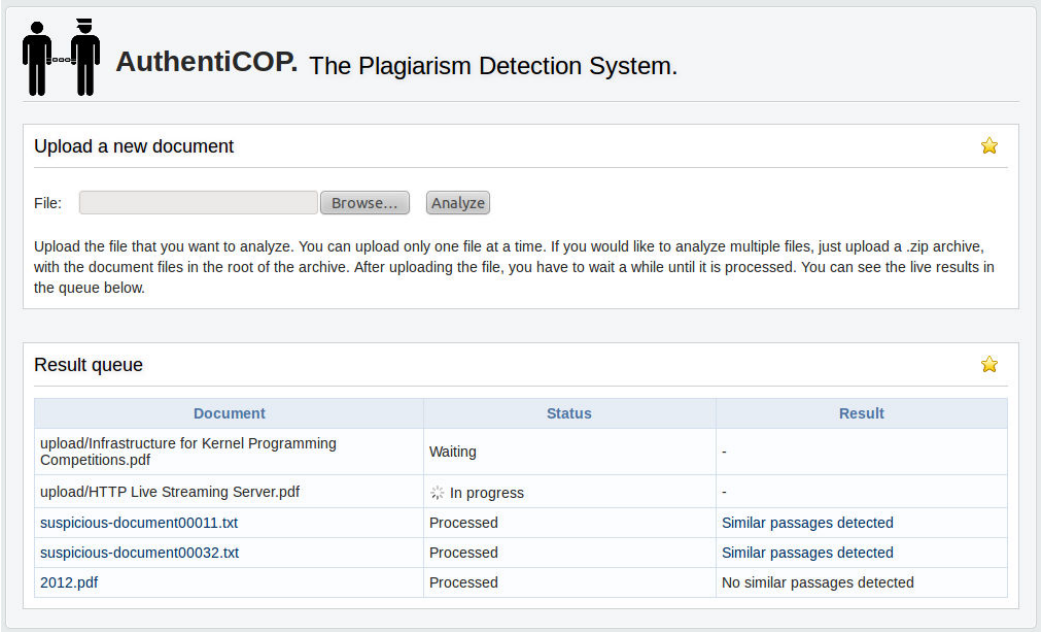

Fig. 4. The upload form for new documents and the result queue.

After a document processing has finished, if any similar passages from other documents were detected, a more detailed analysis is available, as shown in Figure 5.

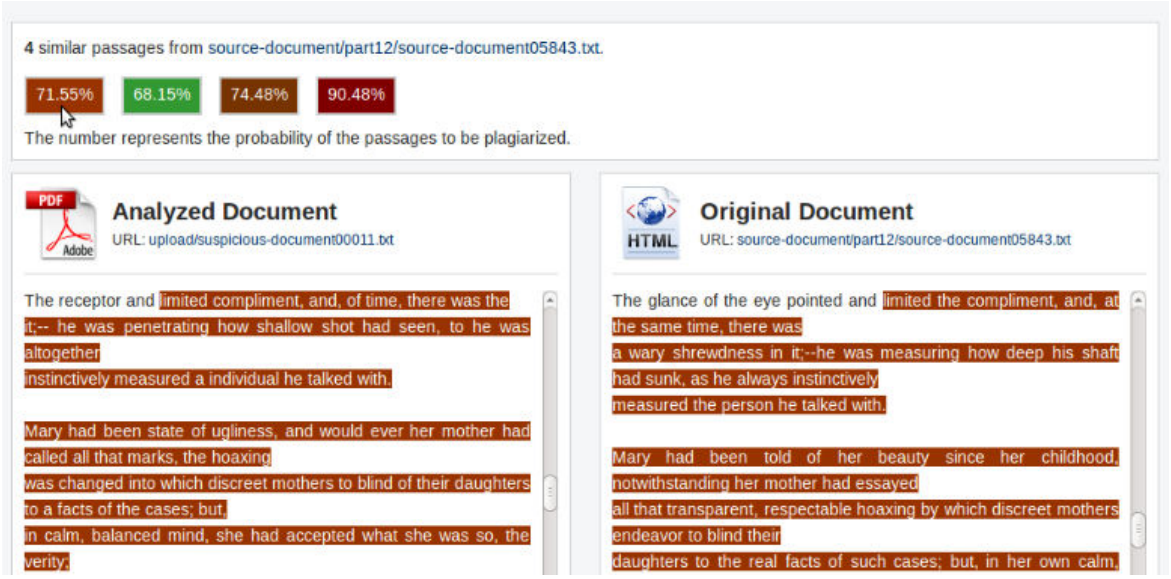

Fig. 5. Side-by-side comparison for two documents.

First, a list of original documents is displayed. Clicking on any of these documents will load all the similar passages from that document, together with the original text. AJAX is used to avoid reloading the whole page when a new document is clicked. A similarity score for each passage is also displayed. The higher the score, the more likely the passages to be plagiarized. Pairs of similar passages are colored distinctly, the color code being related to the score (i.e. paragraphs with the similarity above a threshold may be colored in dark red, while paragraphs below the threshold may be colored in light red). To navigate quickly through the paragraphs, clicking on a box scrolls to the corresponding passages both in the suspicious and in the source text. This is achieved using the jquery scrollTop method of a DOM element.

2.5 The Broker

The component which acts as a mediator between the Frontend and the Backend is the Broker. The Broker is a server which listens for new connections from the frontend and creates new backend processes. It is written in C++ and uses helper libraries for certain tasks, such as JsonCpp [40], RapidXML [41], pthread API or MySQL C API [42].

Besides being a mediator, the Broker must ensure that computing resources are allocated properly. For example, there might be simultaneous requests from the frontend, but at any time, on a single-core machine, there should be at most one document which is analyzed (at most one backend process running), because otherwise the time spent with context switching would negatively impact performance. Also, the presence of the Broker makes it easier to distribute the computations, the backend becoming responsible only for implementing the serial algorithms used in plagiarism detection. The tasks, i.e. the documents that need to be processed, can be given to different machines, using a round-robin approach, each machine performing a single task at a time. This separation allows the backend modules to be very specific and only algorithm-oriented, without dealing with synchronization problems, distribution, communication with the frontend or with the database, etc.

The Broker starts a new server socket, listening for connections on a port specified in a configuration file (the default value is 4552). Whenever there is a new connection request, a new thread is created, the further communication taking place in the context of the newly created thread. This works similar to the Apache web server, which creates a new thread for every new connection. The messages are sent through sockets and are encoded in the JSON (JavaScript Object Notation) format, which allows adding new messages without any additional parsing effort. Because C++ doesn't offer native support for interpreting the JSON format, an external library is used: JsonCpp [40], which is built from sources using the Scons build system. A message example is presented in Figure 6.

```
{
   "command" => "getQ",
   "current"     => "upload/HTTP Live Streaming Server.pdf",
   "waiting"     => {"upload/Kernel Programming.doc", "Search Engine Optimization.docx"}
}
```

Fig. 6. Example of a JSON message between the frontend and the Broker

As it can be seen from the message above, the format can contain simple entries, objects and arrays. There is a list of predefined message types that the Broker recognizes. One command is used to request the processing of a new document, already uploaded on the server via the web interface, using the move_uploaded_file PHP function. The only argument of the command is the

path on the server on which the considered document locates. All documents that are not already processed are placed in a queue residing in the Broker's RAM memory. To print these documents to the user, a similar message to the one presented in Figure 6 is used.

The waiting queue in which documents are placed is acted upon using the producer-consumer model. The pseudocode of this model is presented in Figure 7 below.

```
e = ProduceElement()                 | decrease_semaphore(sem)
mutex_lock(mutex)                    | mutex_lock(mutex)
queue.push(e)                        | e = queue.pop()
mutex_unlock(mutex)                  | mutex_unlock(mutex)
increase_semaphore(sem)              | ConsumeElement(e)

PRODUCER                             CONSUMER
```

Fig. 7. Producer - Consumer pseudocode.

Modifying the queue requires obtaining a lock. The consumer, instead of polling, waits at a semaphore until a producer inserts a job in the queue and signals the semaphore. Connection threads are created whenever the frontend connects to the Broker, which happens at the beginning of a new browser request. The subsequent communication takes place in the created connection thread. When adding a new document for processing, the thread issues a new JSON command and becomes a producer. There is only one consumer, represented by a special thread created when the Broker starts. From this thread, backend processes will be created. By waiting on a semaphore, the consumer thread enters in a suspended state and it does not consume processing cycles when there is no task to perform. The module inside the Broker responsible with implementing the producer-consumer strategy is called JobQueue.

2.5.1 The Pipeline

As mentioned in the Overview section, the plagiarism detection task is not atomic - it is a combination of more phases, such as preprocessing, space reduction, detailed analysis, etc., which succeed one another, in sequential order. In this respect, these steps are represented as a pipeline.

```
{  "stages" : [
        // Step 1. Converting PDF to raw text
        {
            "title" : "Converting to text",
            "cmd" : "java -jar tika-app-1.1.jar -t '$ANALYZED_URL' > '$ANALYZED_PATH'",
        },
        // Step 2. Candidate selection phase
        {
            "title" : "Extracting candidate documents",
            "cmd" : "./candidates '$ANALYZED_PATH' '$SOURCE_PATH' candidates_file"
        },
        // Step 3. Detailed analysis phase
        {
            "title" : "Performing detailed analysis",
            "cmd" : "./detailed candidates_file '$RESULTS_PATH' "
        }
]}
```

Fig. 8. Example of a possible pipeline configuration file.

Every step of the pipeline is in fact a distinct process, created by the Broker using the *system* function call. The *system* function is called in the context of the Consumer special thread described in the previous section. The pipeline is easily configurable, being described by a file in the JSON format, which looks similar to the one presented in Figure 8. Again, JsonCpp is used to parse this file.

Every pipeline process is supposed, though this is not mandatory, to read the data produced by the previous step and to write data for the next step. For example, the candidates binary will read the raw text produced by the previous stage and will write all possible candidates in a file, which will be read in the next stage of the pipeline. It should be observed that communication between different stages is done through files only.

Every command might require access to specific information such as the path to the analyzed file or other configurable directives such as the path to the corpus. These are given as arguments directly in the command line, each backend process having the responsibility to read the parameters immediately after loading. To avoid hard-coding these parameters, each command is run in a given context. A context is a set of variables, each variable having a well-known pre-determined significance. The commands may contain these variables as parameters, and the Broker will replace them with their value. The variables are quoted in the configuration file presented in Figure 8 because the file paths can contains spaces. The list of all variables, together with their significance, is presented in Table 1.

Table 1

Variables in the pipelined processes

Variable name	Description
$ANALYZED_PATH	Path on the file system to the raw text file of the analyzed document
$ANALYZED_URL	Path on the file system to the analyzed document (PDF, DOC, etc.)
$SOURCE_PATH	The path to the corpus
$RESULTS_PATH	The directory in which result files will be stored

The presented approach, in which the stages are completely separated, has the advantage of simplicity. It is very easy to integrate other frameworks, even if they are written in different languages. For example, to convert files from the PDF format to raw text, we use Apache Tika [38], which is written in Java and is integrated as a command line utility in our application. Integrating it is as simple as writing "java -jar tika-app-1.1.jar -t '$ANALYZED_URL' > '$ANALYZED_PATH'" in the pipeline configuration file. Also, the application can be extended easily with other modules. Topic detection can be integrated with no extra effort as a new stage. Replacing one algorithm with another is equally easy. Thus, the pipeline abstraction is very well fitted in our architecture.

2.5.2 The Persistor

The pipeline produces XML files which contain the result of the analysis. There is a component in the Broker, called the Persistor, which parse the result files, using the RapidXML library [41], and writes the data into the database, using the MySQL C API [42]. The results might have been written directly into the database, but the XML files are necessary to evaluate performances on the PAN corpus, because the official evaluation script of the competition, written in Python, receives XML input. Furthermore, storing results in XML files is not enough, because there is no easy and fast way to query them from the frontend code, in PHP. Thus, the writing in the database is a necessary step. Also, this is also good from a security point of view because the data is replicated and is less prone to be lost.

2.6 The Database

As stated in the previous section, even though XML files with the results are written on the disk after every uploaded file is analyzed, the application needs database support. Otherwise, it would have been hard to manipulate the already existing data, because each query would have meant to take the files one by one, parse them and then combine the results in order to simulate table joins. To avoid this overhead, we use a database system, based on a MySQL implementation. The database schema is simple, containing only a few tables and relationships, and is presented in Figure 9. The schema is created with MySQL Workbench.

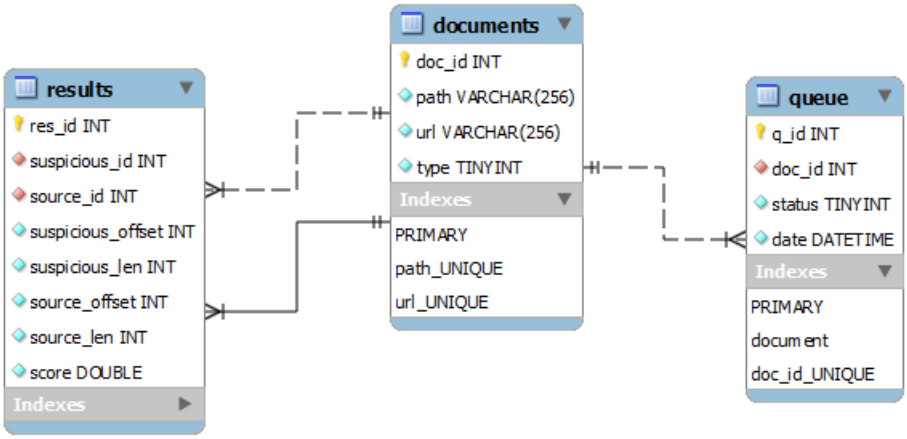

Fig. 9. AuthentiCOP Database Schema.

The DOCUMENTS table stores all the documents known by the system, a given document appearing at most once in the database. For space reasons, only documents that appear either in the RESULTS or in the QUEUE table are stored. This means that in case of source documents contained in the corpus, they will not be inserted in the table until there is some suspicious document that plagiarized from them. In case of suspicious documents, they will be written to the table only when they are completely analyzed and inserted in the queue. The URL refers to the location on the file system on which the original document resides (in PDF, DOC or other format for local files, or a Web address for source documents, when the system will be enhanced with Web detection features), while the path refers to the location on the file system of the file transformed to raw text. Thus, when displaying the documents, the path field is used, so the user sees only raw text in the comparison phase. However, the user has the possibility to view the original document, by downloading it from the server's file system, in which case he is pointed to the URL field. The type field is necessary in order to be able to distinguish between source and suspicious documents.

The QUEUE table stores information about the already processed documents. Documents that are in progress or documents that are not already scheduled are kept in the Broker's memory instead. The RESULTS table holds all the passages that have been identified as suspicious, together with a similarity score. The higher the score, the more likely the passages to be plagiarized.

The database is accessed both from the frontend (PHP code) and from the Broker (C++ code). The frontend code will mainly read from the database (with the controllers as the initiators of the data fetching process), while the Broker will parse the XML file produced by the pipeline and will write in the database. By using transactions, there is no need to deal with synchronizations from our code, as the database system will inherently handle these situations. A PHP extension is used to access the database from PHP code (mysqli), while to access it from C++, the MySQL C API is used.

There are also some subtleties regarding queries. For example, in case the same document is sent for processing more than once, it should only be reprocessed if the corpus or the algorithms changed from the last run. This can be implemented by using a hash value, like MD5, for each document, together with a timestamp, to keep track of the new changes. However, we do not implement this in the current version. What is more, we assume that the documents that need to be processed have different names, so we do not use hash values. We use an "INSERT ... ON DUPLICATE KEY UPDATE" strategy, which will insert the document if it is not already in the table, and will update the fields (i.e. timestamp) of the document with the same name, if there is one.

2.7 Backend Architecture

As mentioned before, the final backend application will put together many interdependent components. There are algorithms (such as those used for candidate selection or detailed-analysis), data structures (such as inverted indexes or dictionaries), text preprocessing components, and so on, all these components being shared between different stages of the pipeline. Some components will be integrated as they are, others will be implemented from scratch, and others will need to be partially changed to suit our needs.

We would like to structure all these components in such a way as to minimize the effort of integration. Trying another variant of one of the algorithms or adding a new stage, like topic detection, should be done with minimum effort.

Also, to have the code maintainable and to have a clear view of the role of each component, we would like to split functionally different components in different directories. For example, we should have algorithms separated from data structures. This separation can go even further and should be done at any level. For example we would like to have candidate selection algorithms separated from classification algorithms, and so on. Thus, we obtain a hierarchy of directories, which looks similar to the Figure 10 below.

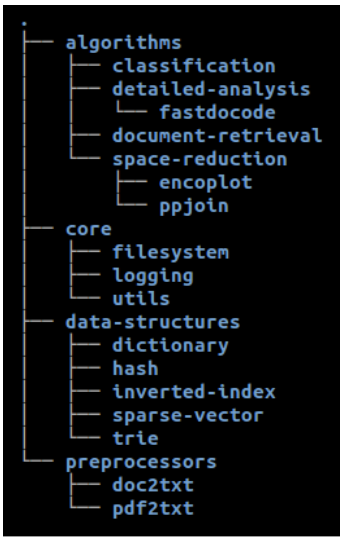

Fig. 10. An example of possible hierarchy of backend modules.

In this way, the application becomes structured. We can consider every directory in the hierarchy as being a module, each having a very precise specification. All the source files for a module will be found in the corresponding directory. However, this representation raises a problem, as we will see below.

Each module depends on others to perform specific tasks. For example, algorithms used in the space-reduction phase will rely on an inverted index for data manipulation. In C++, this dependency means including the header file for the inverted index in the space-reduction algorithm source code, and linking objects together. To optimize the compilation process, we rely on a Makefile, which contains all the dependencies. The Makefile should be changed whenever a new module is added, so it can grow quite rapidly and it can become hard to manage. Also, the problem with separating components in different directories is that we will have complicated relative include paths (i.e. #include "../../algorithms/space-reduction/encoplot/encoplot.h"). To solve these problems automatically, we created a lightweight build system, whose details are explained in the next section.

2.7.1 The Build System

The build system we created generates the Makefile and includes the necessary headers automatically, according to existing dependencies. From its perspective, each directory in the file-system hierarchy is considered a *module*. Every directory contains a configuration file named *module_init*, which describes the module by specifying its type (object file or executable), the needed dependencies and command line arguments to be used by the compiler. An example of a module_init file is presented in Figure 11 below.

```
module_type('EXE')

require_module('core/utils')
require_module('core/lexer)
require_module('data-structures/inverted-index')
require_module('algorithms/space-reduction/encoplot')

compiler_args('-Wall -O2')
```

Fig. 11. Possible module_init file for a module.

Each line in the file describing the module, if not blank or a comment, must have the general syntax STATEMENT('ARGUMENT'). STATEMENT can be any of the following: *module_type*, *require_module* or *compiler_args*. Each statement receives only a single string parameter.

The modules can be either objects, their purpose being to offer functionality to other components (similar to a library), or executables, which enables them to be used as stages in the pipeline. The argument of the *module_type* statement can thus be either OBJ or EXE, covering one of these two cases.

Dependencies are described using the require_module statement. The relationship is transitive: if A depends on B and B depends on C, then, in order to use A, C has to be built first. Because of this property, it is possible to have a scenario in which there are cyclic dependencies (i.e. A depends on B, B depends on C and C depends on A), which of course should be avoided. For this reason, the build system ensures that the dependencies form a DAG (Directed Acyclic Graph), and if not, it prints an error message and aborts the building process. The DAG is used to create the Makefile.

Header files can be included automatically in the source code by rewriting the source files, but this approach is hard to implement, because there are particular cases to deal with. For example, the user might add some initial dependencies, then create the build using the build system and after that add another dependency. The initial dependencies should not be added twice, while the newly added dependency should be identified as such. To simplify this, every module must

include only a single header file, named deps.h. This is the only header file that need to be included, with respect to the modules (there might be other includes from the Standard Template Library which are not covered by the deps file). For each module, the build system will automatically create the deps file in the corresponding path, and will populate it with all the necessary includes.

The tool helped us to manage the Makefile, which rapidly grew in size and complexity, and provided a neat way of avoiding complicated relative paths in the include statements. As expected, this resulted in increased productivity.

2.8 Deployment

Deploying AuthentiCOP requires several steps: the build system has to be compiled in order to build the backend; the backend and the broker have to be built separately; all the executable files from the pipeline and all the static resources need to be in the same top-level directory (because of the way resources are loaded in the backend); the updated frontend files have to be copied to the document root of the web server; all configuration files need to be put in the same place to access them easier, etc. All these steps are performed automatically by an install script, which calls the necessary Makefiles and creates symlinks to the needed executables, resources and configuration files so that they all appear to be in the same directory. As with the build system, the install script increases productivity and completes the list of tools which are essential when implementing a complex software product.

3. Algorithms for Plagiarism Detection

The architecture presented in the previous section allows us to implement the algorithms in a complete separate environment, focusing only on the particularities of the problem without dealing with synchronization, integration, parallelism or other engineering-related tasks. We inspired our work from existing systems, but we experimented with multiple variations and also tried personal approaches. The next sections present the details of the algorithms we tried and the iterations we gone through in creating the system.

3.1 The Candidate Selection Phase

According to Potthast [26], "ideally, the suspicious document would be compared to all other documents available, but in practice, because of the efforts involved, one has to limit comparisons to a reasonable number of <<candidate documents>>. Hence, these candidates must be chosen carefully in order to maximize the likelihood of finding the true originals, if there are any". This type of reduction does not only apply to the plagiarism detection problem, but also to most tasks that use brute-force, where heuristics are required to prune the search space.

We tried several approaches for candidate selection, testing them on the PAN 2011 corpus. Having the golden standard of candidates, extracted from the solution files provided on the competition website [43], we were able to rank the used methods. The evaluation measures we considered are those used to assess the binary classification performance: the number of true positives (correctly identified pairs, or TP), false positives (incorrectly identified pairs, or FP) and false negatives (missed correct pairs, or FN). In artificial intelligence, these values are arranged in a 2x2 matrix, called confusion matrix, or table of confusion. In this matrix, positive predictive value

(equivalent to precision in Information Retrieval) and sensitivity or true positive rate (equivalent to recall) can be computed, using the Equation 10 below.

$$precision = \frac{TP}{TP+FP}, recall = \frac{TP}{TP+FN} \quad (10)$$

For absolute ranking, a unified score has to be used, combining true positives, false positives and false negatives. One of the most well-known scores is the F-measure (or F1-score), which is the harmonic mean of precision and recall. The formula for this measure is given in Equation 8. Other unified score is Matthews correlation coefficient (MCC), first introduced in [44]:

$$MCC = \frac{TP * TN - FP * FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \quad (11)$$

For candidate selection, there is usually an order of magnitude in difference between TN (true negatives, or the number of incorrect pairs that were identified as being incorrect) and the other variables involved in Equation 11, because each document is likely to plagiarize from only a few sources, not from hundreds of documents. This is why we prefer the F-measure for absolute ranking, which only involves TP, FP and FN. As a matter of fact, this measure is also used in the PAN competition, but only for evaluating the final results, and not for the intermediate results given by the candidate selection phase.

To reduce the search space, we mainly inspired our work from the FastDocode system [30], All-Pairs [33] and ppjoin [34]. The latter two are not explicitly used in any other plagiarism detection system to our knowledge, but we considered that they can be adapted to our task with good results. For the algorithms presented above, we experimented with multiple parameter values, the results being presented in the next sections.

3.1.1 Initial Assumption

Our initial approach was to find some circumstances in which there is a separation between plagiarized and non-plagiarized passages, i.e. if there is a threshold θ for which most of the pairs with similarity larger than θ are plagiarized and most of the pairs with lower similarity are not.

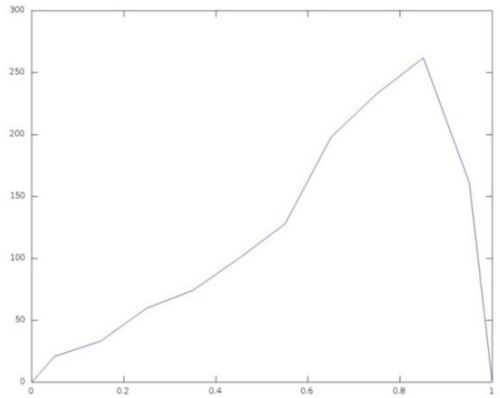

Fig. 12. Distribution of scores for lowly obfuscated passages.

For this purpose, we experimented with the cosine similarity and tf-idf weighting scheme on the PAN 2011 corpus. Also, before computing the cosine similarity, we performed pre-processing,

removing stop words and applying stemming on all the remaining words. Figure 12 above shows the distribution of scores for a sample of about 1000 lowly obfuscated plagiarized passages. The peak of the graph is for θ around 0.8.

Figure 13 also shows the distribution of scores for a sample of about 1000 plagiarized passages, but this time the plagiarism cases are highly obfuscated. The peak decreases to 0.3, which is natural, because obfuscation implies, among others, replacing words with their synonyms and paraphrasing.

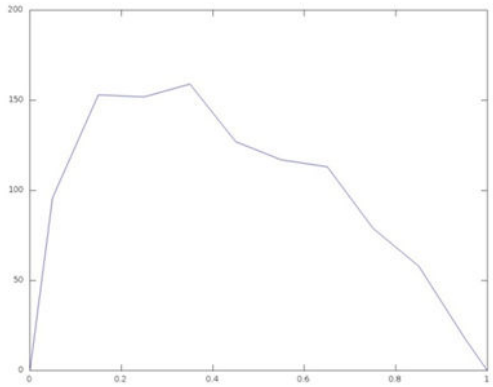

Fig. 13. Distribution of scores for highly obfuscated passages.

However, the distribution for the best matching passages from 1000 random pairs is much smaller and is displayed in Figure 14. In this case, the peak is below 0.1.

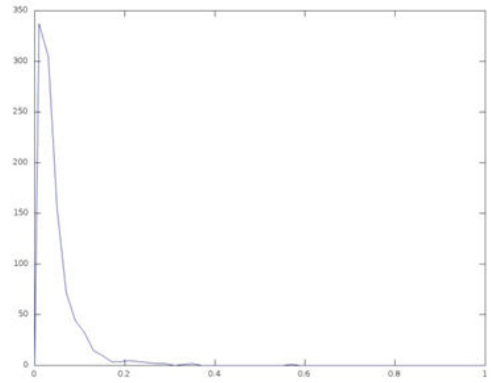

Fig. 14. Distribution of scores for the most similar passages in 1000 random document pairs.

With this empirical observation, we concluded that the method of splitting is effective. For this reason, we decided to focus on an efficient way of finding pairs of passages that are highly similar under the cosine similarity with tf-idf weighting.

3.1.2 The Similarity Search Problem

Having a large collection of vectors, the similarity search problem aims to find all pairs of vectors whose similarity score, determined by using a similarity measure, is above a given threshold. On the assumption presented in section 3.1.1, that similar passages have a high score when compared under the cosine similarity with tf-idf weights, the plagiarism detection task can reduced

to splitting the documents in passages and then employ a similarity search to find highly similar pairs. For the PAN corpus, which contains around 22000 documents, assuming that each document will be split in 100 paragraphs, there will be more than 2 million entries to be analyzed. A brute-force approach running in quadratic time will compare 4000 billion pairs, where each comparison requires at least to iterate the words in the two paragraphs, to find the matches, which means an increased latency. This makes the brute-force unusable, and other methods have to be found.

In [34], an open-source algorithm for performing fast similarity search is presented, based on indexing and optimization strategies. All demonstrations and pseudocodes use the Jaccard similarity, but the algorithm can be extended, though not trivially, to use other measures. Among other improvements, such as the positional filtering principle, the paper presents index reduction techniques, which need the entries to be sorted by their length. Because of this restriction, both the source and the suspicious documents need to be preprocessed together. In our case, we wanted to build an interactive system, which means that the documents to be analyzed are not known in advance, leaving the preprocessing only for the source documents. This is why we decided not to use ppjoin and focus to All-Pairs instead, presented in the next section.

3.1.3 The "All-Pairs" Algorithm

In [33], another algorithm for performing fast similarity search is presented. It addition to being fast, it also has the advantage that it does not require extensive parameter tuning and that it can be used in the interactive problem, in which the suspicious documents are not known in advance. At the core of the algorithm there is a simple data structure called inverted index, which is also used on a large scale in search engines. The data structure holds for each word the documents containing it, together with the positions on which it appears. An example is presented in Figure 15.

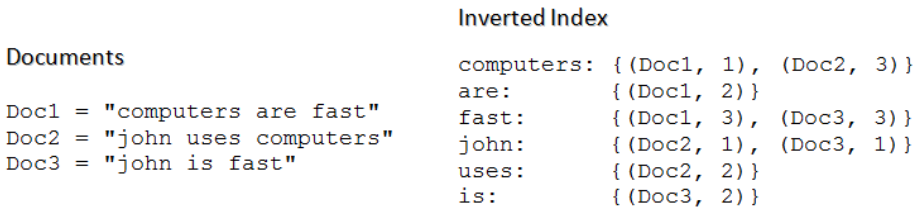

Fig. 15. An inverted index example.

Given a passage, all paragraphs that share at least a common term can be returned by querying the inverted index. For example, all documents containing the term *computer* can be determined by looking at the inverted list of the term, without having to iterate through all the documents again. Computing the score for the retrieved passages directly is still a major improvement over the brute force solution, because in general a passage will share common words with only a few other passages. This only holds if the stop words are removed (otherwise, it is likely for almost every passage to share words like *the* or *is*). Thus, the removal of the stop words not only improves the quality of the results but also reduces the size of the index drastically, improving the speed of the algorithm.

There are other simple, yet effective, optimizations that can be implemented to improve the speed. These optimizations mainly exploit the threshold. For example, let's suppose we have the vector x which is queried against the index, and that the vector y is returned. If the two vectors are normalized (i.e. their Euclidian norm is equal to 1), then their cosine similarity is equal to their dot product. We can estimate an upper bound of the dot product, presented in Equation 12.

$$min\begin{cases}(\max_{i=1\dots N} x_i) * (\sum_{i=1}^{N} y_i) \\ (\sum_{i=1}^{N} x_i) * (\max_{i=1\dots N} y_i)\end{cases} \quad (12)$$

This is trivial to prove: $\sum_{i=1}^{N} x_i * y_i$ is less that the expression in which the elements in x or y are replaced with the maximum value in the vector. If the upper bound in Equation 12 is lower than the required threshold, then the pairs can be discarded directly, without any further computation. This optimization is known as the *size filtering principle*. The values required, like the maximum or the sum of the weights, can be preprocessed, and so the upper bound can be computed in constant time.

Another useful optimization is also based on establishing an upper bound, but after iterating the first i tokens in x (also known as a *prefix*). Let's define $maxDot_i$ as the maximum dot product of the current prefix in x with any other passages, and:

$$maxRemaining_i = \sum_{k=i}^{N}(x_k * \max_{y \epsilon Collection} y_k) \quad (12)$$

$maxRemaining_i$ is the maximum dot product that can be obtained with the suffix of x starting from position i, and can be achieved when there is a vector y in the collection such that $y_k = \max_{z \epsilon Collection} z_k$, for every $k \geq i$. In other words, the maximum is achieved when every element on position k is matched with the highest weighted term on that position. If $maxDot_i + maxRemaining_{i+1}$ is lower than the required threshold, then x can be safely discarded.

The above principle brings the possibility of another optimization. Because the elements in $maxRemaining$ decrease, the tokens can be sorted by their idf, which means that the most rare elements come first. Thus, $maxDot_i$ will increase slower, making $maxDot_i + maxRemaining_{i+1}$ to come below the threshold faster.

We implemented the algorithm as it is described in [33], with all the optimizations presented above. Because the index was too big to be kept in memory, we needed a specialized fast inverted index on disk. For this purpose, we used Apache Lucene.

3.1.1.1 Apache Lucene

Apache Lucene is an open-source, high-performance text search engine [45]. The unit of search and index in Lucene's taxonomy is the *Document*, but it is not related in any way to a real document in the corpus. In fact, it can hold only a passage, a sentence or even a single token. Every Document comprises one or more name-value pairs, called *Fields*.

The index is a collection of *Document* objects, created using an IndexWriter. The changes are buffered in memory and periodically flushed to disk. After the index is created completely, an IndexReader is used to create the query and to retrieve the results. These results are ranked, using a scoring formula, which is based on the dot product of weighted vectors. Other components may be boosting factors or vector norms. Because of its ranking system, it would have been possible to use Lucene in a standalone phase. Though, we preferred to use it only as an inverted index and have full control of the implementation details and scoring mechanisms.

Because our backend is written in C++, to avoid the overhead of cross language communication, we used CLucene [46], which is the C++ API variant of Lucene. Because it is compiled into machine code, CLucene is faster than Lucene. The drawback is that it is not up to date. For example, starting with the 2.9 version, Lucene introduced numeric fields. This feature is not

implemented in CLucene, and because most of our data is numeric (i.e. vector lengths, maximum weight for suffixes, etc), we had to store them as strings.

Indexing the source files in the PAN 2011 corpus, including the necessary data for the All-Pairs algorithm, resulted in an index of about 12 GB.

3.1.1.2 Results of the All Pairs Similarity Search

The results of the All Pairs algorithm in our plagiarism detection task were not promising. We noticed that for about every 70 candidate pairs returned, only one was a true positive, the other ones being incorrectly signaled (false positives). This ratio can be explained by the fact that many documents that deal with the same subject are likely to use the same vocabulary, scoring high under a bag of words model, no matter of the similarity function used. Our initial assumption that there is a separation threshold for plagiarized and non-plagiarized passages under the bag of words model was wrong, but it couldn't have been proved until the algorithm was fully implemented. Still, the false negatives will be low, meaning there will be few missed correct pairs, which is a good sign. However, the algorithm needs to be more restrictive, because too many pairs are passed on to the next and more time consuming stage. This is why the algorithms based on n-grams presented in the following sections score better in the candidate selection phase, as they also use positional information. Even though they miss a large number of correct pairs, they still perform better than the current approach.

3.1.3 FastDocode

As shown before, the algorithms that rely only on the bag of words model do not perform well in the candidate selection phase. This is why we experimented with FastDocode, which uses positional information. The algorithm was initially presented in the PAN 2010 competition. The approach used in [30] to select the candidates is based on n-grams: if two documents have at least two word 4-grams coincidences close enough as to be in the same paragraph, the documents are given to the next phase. Otherwise the pair is discarded.

To reduce the total number of computations, some preprocessing steps are employed. First, the algorithm removes stop words. Then, the documents are split into chunks, each being represented as a set of n-grams. For each chunk, a fingerprint is computed, the items that do not belong to the fingerprint being discarded. A fingerprint is a sample of n-grams chosen using a selection strategy, such as random selection. A better approach is to employ a sorting technique on the n-grams. The n-grams can be sorted either in lexicographical order, or by their total count in a given corpus, such as Google's scanned books, which contains hundreds of thousands of books. The frequencies of n-grams in this corpus are publicly available using Ngram Viewer [47]. After sorting, the first (or the last) K entries can be selected, where K is the sample size desired. In this respect, this method is similar to Winnowing [16]. In the Winnowing algorithm, a sequence of hashes is split into a sequence of overlapping windows. Then, a representative hash value is chosen for each window, namely the minimum value. Winnowing is used in the MOSS plagiarism detection service with good results, but only deals with exact or very similar copies. For this reason, in order to be able to improve the probability of detecting pairs that are not exact matches, FastDocode selects a larger number of n-grams in each chunk, being different also in the sense that the chunks (equivalent to windows in Winnowing) do not overlap. After the preprocessing step, every pair of documents is then compared, analyzing only their corresponding fingerprints.

In our implementation, the sorted list of n-grams for a file (no matter if a source or a suspicious document) is called *profile*. What is more, each n-gram has its tokens sorted. This way, for example, *john meets marry* and *marry meets john* become the same, but the drawback of losing the semantic meaning does not exceed the advantage of catching inversions, which are frequent in plagiarism cases. The responsibility of the preprocessing step becomes to compute the profiles and store them on the disk, so that they can be accessed later, without any further computations. A sorting strategy is implemented, as described before, keeping only the first entries in each chunk.

If there are M suspicious documents and N source documents, there are O(M x N) documents comparisons. The profiles cannot be kept in memory because of their prohibitive total size. A naive approach would be to read the files from disk, every time they need to be evaluated, as shown in Figure 16.

FastDocode-0

```
for (i = 0; i < M; i++) {
        suspiciousProfile = read_profile(SuspiciousDocument[i]);
        for (j = 0; j < N; j++) {
                sourceProfile = read_profile(SourceDocument[j]);
                compare_profiles(suspiciousProfile, sourceProfile);
        }
}
```

Fig. 16. A naive approach of iterating the profiles.

Although very simple, this method is very inefficient, requiring O(M x N) file readings from the disk, which are very expensive compared to memory access. To improve the performance, an intermediate approach is used, by introducing a "caching" system: at each moment, only $O(\sqrt{M} + \sqrt{N})$ profiles are kept in memory. Now, each source file will be reloaded $N\sqrt{M}$ times, and each suspicious document will be reloaded M times, which is a major improvement over the initial solution. The pseudocode is presented in Figure 17.

FastDocode-1

```
for (i = 0; i < M; i += √M) {
        for (k = i; k < i + √M; k++) {
                suspiciousProfile = read_profile(SuspiciousDocument[k]);
                put_profile_in_cache(SuspiciousDocument[k], suspiciousProfile);
        }
        for (j = 0; j < N; j += √N) {
                for (l = j; l < j + √N; l++) {
                        sourceProfile = read_profile(SourceDocument[l]);
                        put_profile_in_cache(SourceDocument[l], sourceProfile);
                }
                for (k = i; k < i + √M; k++)
                        for (l = j; l < j + √N; l++) {
                                suspiciousProfile = get_profile_from_cache(SuspiciousDocument[k]);
                                sourceProfile = get_profile_from_cache(SourceDocument[l]);
                                compare_profiles(suspiciousProfile, sourceProfile);
                        }
        }
}
```

Fig. 17. The pseudocode for caching $O(\sqrt{M} + \sqrt{N})$ profiles at a time.

Having two profiles, a decision has to be made whether or not the corresponding documents should be passed to the next stage. As stated in the beginning of the section, we look for n-gram coincidences. As the profiles have already the n-grams sorted, intersection can be determined in linear time, using a merge algorithm. However, this can be further improved if using sublinear merge. Let $suspicious$ be the array of n-grams corresponding to the fingerprint of the suspicious document, and $source$ the array corresponding to the fingerprint of the source document. Also, for simplicity, let's assume that each fingerprint has distinct elements. In the simple merge algorithm, in every step there are two positions i and j to be compared. If $suspicious_i$ is less than $source_j$, then i is advanced one position. If $source_j$ is less than $suspicious_i$, then j is advanced one position. If the two values are equal, both i and j are advanced and the intersection count increases by one. In sublinear merge, instead of increasing the pointers i and j only by one, we extend as much as possible, using powers of 2. For example, if $suspicious_i$ is less than $source_j$, we first increase i by 1, and compare $suspicious_{i+1}$ to $source_j$. Then, we do the same for $suspicious_{i+2^k}$, until we find a k for which for $suspicious_{i+2^k} \geq source_j$. In this case, we stop and increase i by 2^{k-1}. Although in the worst case scenario, in which the two profiles are identical, this also has linear time, in practice the total execution time is reduced by half. The pseudocode is presented in Figure 18.

```
compare_profiles(suspicious, source)
    i = j = count = 0;
    while (i < suspicious.length() && j < source.length()) {
            if (suspicious[i] < source[j])
                    i = jumpToNext(suspicious, i, source[j]);
            else if (source[j] < suspicious[i])
                    j = jumpToNext(source, j, suspicious[i]);
            else // the elements are equal
                    i++, j++, count++;
    }
    return count;

jumpToNext(vec, i, value)
    for (k = 1; i + k < vec.length() && vec[i + k] < value; last = i + k, k = 2 * k);
    return last;
```

Fig. 18. Sublinear merge pseudocode for profile intersection

We implemented the FastDocode candidate selection method, using the caching system and the sublinear merge described above to increase the speed of the algorithm. We experimented with different values of the parameters, observing the fluctuation of the results when changing the n-gram size, the segment size (the size of each chunk, in number of n-grams) and the retention rate (the ratio between the number of selected n-grams in each chunk and the size of each chunk).

Table 2

Results for the FastDocode candidate selection algorithm on the PAN 2011 corpus

Experiment	n-gram size	Segment size	Retention rate	TP	FP	FN	Running Time (h)	Final score
E1	3	150	10%	5413	44522	11469	~ 1	0.162
E2	4	150	10%	4913	10297	11969	~ 2	0.306
E3	4	150	30%	7633	35169	9249	~ 4.5	0.256
E4	5	150	20%	5194	6256	11688	~ 3	0.367

On average, about one third of the pairs are correctly identified, the other two thirds being missed. Also, it is worth mentioning that a large number of identified pairs are false positives. The exact results, in number of pairs, together with the running time, are presented in Table 2.

3.1.4 Personal Approach

As it can be seen from the FN (false negatives) column in Table 2, a large number of correct pairs are missed in all experiments. Missing pairs in the early stages means that they can never be identified as plagiarized in the later stages. This is why it is preferable to miss as few pairs as possible in the candidate selection phase, even though the number of false positives is high. If there are many false positives, a post processing technique can be employed for pruning the incorrect pairs, while, eventually, the majority of correct pairs is still preserved.

The initial analyzed method, based on All-Pairs Similarity Search, identified most correct pairs, but it had the drawback of identifying too many incorrect pairs, which happened mainly because any positional information of the terms was lost. On the other side, FastDocode identified fewer incorrect pairs, but missed many correct ones, due to the fact that the positional requirement was too restrictive. We tried to balance these two perspectives, using the bag of words model together with a more relaxed positional condition.

After removing the stop-words and stemming all the remaining words, we split the documents in passages of 128 words. In each passage, we sorted the words by their idf in descending order, and kept only the first 32. The reason behind this was to reduce the size of the text that needs to be indexed, while still preserving the words that are harder to be replaced. Then, we indexed the passages from the source documents using CLucene. The total size of the resulted index is 1.1 GB. Each passage of the suspicious documents is then analyzed in turn. The index is queried, and the most similar passages are retrieved. Lucene's native scoring mechanism (based on a variation of cos similarity with tf-idf weights) is used for ranking. Every passage returned by Lucene is compared with the initial passage in terms of the number of inversions. This is because in order for a text to be considered plagiarized, the ideas have to be presented in almost the same order as in the initial text. If each word introduces a concept, then a change in the order of ideas is quantified by the number of inversions between all the terms.

We tested this approach on the first 1000 documents from the PAN corpus. The results are presented in Table 3 below.

Table 3

Comparison of the proposed method with FastDocode

Method	TP	FP	FN	Precision	Recall	Final score
Proposed Method	685	494	761	0.581	0.474	0.522
FastDocode#E3	634	4097	812	0.134	0.438	0.205
FastDocode#E4	424	815	1022	0.342	0.293	0.316

It can be observed that this algorithm produces significantly better results in terms of both precision and recall. However, the running time for the selected sample was about 5 hours, which means that the algorithm is an order of magnitude slower than FastDocode. Because of this, we decided to use FastDocode in the final application for the candidate selection phase.

3.2 The Detailed Analysis Phase

For each pair of documents resulted from the candidate selection phase, we need to determine all passages from the suspicious document that are plagiarized from passages of the original document. Each passage is uniquely identified by its byte offset in the document and its byte length. Thus, for each pair, the detailed analysis phase means identifying a set of quadruples $(offset_{suspicious}\ length_{suspicious}\ offset_{source}\ length_{source})$, where each quadruple must be classified as a plagiarism instance by a general binary relation $\mathcal{R}$. The relation decides if two passages are plagiarized based on a similarity measure and on a threshold: if the measure is above the threshold, then the passages are plagiarized.

3.2.1 The Sequence Alignment Problem

According to [26], the detailed analysis phase is connected to bioinformatics and image processing. This connection is based on the fact that two documents that are being compared can be represented visually using a similarity matrix, known as a dot plot, which is also used in comparing biological sequences. In the dot plot, every element is a point colored in white, black or nuances of gray. The color of the element at $(x\ y)$ is determined by comparing the x^{th} term in the first document with the y^{th} term in the second document. If the terms are equal, the corresponding point will be colored in black. Otherwise, the point will be colored in a nuance of gray that will represent how closely connected the two terms are, i.e. if the words are synonyms the color can be dark gray, if the words belong to the same semantic field the color can be light grey. If there is no correlation between terms, the color will be white. A dot plot is displayed in Figure 19.

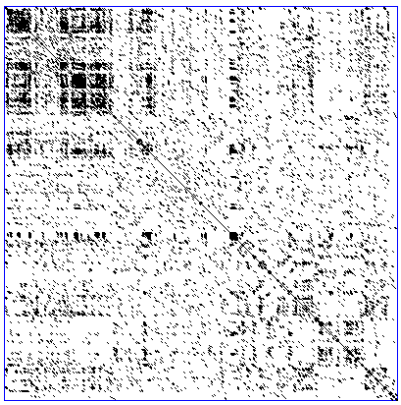

Fig. 19. A DNA dot plot (image taken from Wikipedia).

Having the dot plot, an estimate of how similar two passages are can be determined by looking at the rectangular region corresponding to the passages. If the region is predominantly dark or if it contains many dark points along a line, then the two passages are similar.

In the sequence alignment problem, a query is compared against a target string, with the purpose of finding a sequence in the target that most closely matches the query. In the dot plot representation, this simply becomes finding a gray line (known as a diagonal run) in the matrix. As the brute-force approach is too computationally expensive, heuristics, such as FASTA [48] or BLAST [49], are employed to determine approximate solutions. In FASTA, the 10 best regions of similarity are localized and combined, and then a Dynamic Programming is run around the best scoring diagonal. BLAST is more complex, using a series of steps to converge to the final solution. In one of these steps, the segment pairs are extended to the left and to the right until a high-scoring segment

pair is obtained. ENCOPLOT [31][32] is another algorithm based on the dot plot, but instead of considering the whole matrix, it uses a modified merge-sort algorithm to detect only one dot for every two equal items. Then, a clustering heuristic, based on a Monte Carlo optimization loop, is applied on the resulted dots.

3.2.2 Fast Detection of Similar Passages

The dot plot is an intuitive method of representing the problem, but it has the disadvantage of being too computationally intensive. Although there are heuristics that give approximate solutions for finding the dark areas, we focused our effort on detecting the passages that satisfy a given relation $\mathcal{R}$, which is, in our case, cosine similarity with tf-idf weights. Even though we showed that this method alone is not effective when dealing with candidate selection, this method is well-suited for the detailed analysis phase, when we can assume that most of the pairs are likely to contain similar passages. These similar passages still score higher than passages that are not plagiarized. We ran the method on the golden standard of candidates from the PAN 2011 competition and, without any parameter tuning, we obtained a recall of 0.337 and a precision of 0.760, which would have ranked us on the 3rd place, very close to the 2nd place.

The brute-force approach for finding all quadruples that satisfy the relation $\mathcal{R}$ is prohibitive, as it means generating all possible subsequences and then testing their score, which has at least linear time because the sequences need to be iterated over, giving a total complexity of $O(N^5)$. This can be refined to $O(N^4 \log N)$, because when adding a new element the score does not need to be computed from scratch. A binary search can be performed instead, in order to find if the newest inserted element in the second passage is found in the first one. Another approach, similar to what BLAST uses, is presented in Oberreuter et al [30]. The idea is to start from every possible pair of equal terms, and then extend to the left and to the right as long as the percentage of similar words is not less than a threshold value. This method is faster than the brute force approach, but still has prohibitive running time.

Our idea is to avoid trying every possible starting point. For this, we split the suspicious document in chunks, each chunk having the size a power of 2, ranging from 32 to 2048. The offset of a chunk of size x is divisible by x and chunks of different sizes may overlap. For each of these chunks we check if there is a similar chunk of the same size in the source document (i.e. with the cosine similarity bigger than a threshold). If there is, we add the chunk to a candidate set. After all chunks are considered, the intervals from the candidate set which intersect are concatenated, to obtain compact intervals. These compact intervals represent the *kernels*: namely, passages from the suspicious document that have a similar correspondent in the source document. The pseudocode is presented in Figure 20 below.

```
findKernels(suspiciousDoc, sourceDoc, threshold)
for (step = 32; step <= 2048; step = step * 2) {
        foreach (i j) s. t. suspiciousDoc[i] == sourceDoc[j]
                score[i / step][j / step] += idf[suspiciousDoc[i]] * idf[sourceDoc[j]]
        foreach (x y) s. t. score[x][y] > threshold {
                suspiciousIntervals.insert([x, x+step-1])
                sourceIntervals.insert([y, y+step-1])
        }
}
concatenateIntersected(suspiciousIntervals)
concatenateIntersected(sourceIntervals)
```

Fig. 20. Pseudocode for finding the kernels

The first foreach loop, which iterates over pairs of equal terms, can be implemented using an algorithm similar to linear merging, the difference being that when two equal elements are encountered, the whole intervals of equal elements are detected and are iterated over in quadratic time. For example, if the arrays of tokens are $x = (3\ 9\ 20\ 20\ 21)$ and $y = (20\ 20\ 84)$, when $x[3] = y[1]$ is encountered, the intervals $x[3 \ldots 4]$ and $y[1 \ldots 2]$ are detected as containing the value 20, and all the pairs $(x[3], y[1]), (x[3], y[2]), (x[4], y[1]), (x[4], y[2])$ add up to the final result. Because the stop words were removed, the intervals will contain only a few items, so, in practice, the running time is close to linear.

For each kernel, we can determine in O(NlogN) the passage in the source document that most closely matches it. At each step, we advance the passage in the source document by one element. This means that exactly one element will be removed from the window, while a new one will be added. We can do a binary search to see if these two elements were in the suspicious kernel.

Having a pair of passages that score high under $\mathcal{R}$, we can now further refine the boundaries using the idea from [30], trying to either extend or shrink the intervals, using a hill-climbing approach and maintaining the best solution on the way. We perform extensions / shrinks until the global optimum no longer improves for a given number of steps. For instance, we may choose the extend both intervals to the right with the same number of elements. Or we may choose to extend the suspicious interval to the right and the source interval to the left. We may even choose to extend only one interval, or to extend both intervals, but with different number of elements. Because these strategies are important for obtaining a high accuracy, we use 100 variations. To be able to perform these changes fast enough, we employ binary searching techniques. Also, it is important to note that shrinking should not be performed if the length of the intervals comes below a fixed minimum. Otherwise we would end up with intervals containing only one element, which of course has the perfect score, if the element is common in both intervals.

As mentioned in the beginning of the paragraph, this method performed well on the golden standard of candidates from the PAN competition, so we decided to use it in the final application. Also, it is important to note that the running time for 10000 pairs is around 15 hours, meaning that about 650 pairs can be analyzed in an hour.

3.3 The Post-Processing Phase

After the detailed analysis, a set of pairs of passages are returned. An additional step can be employed on these pairs, known as knowledge-based post-processing (as it is called in [26]). This involves, among others, filtering false detections based on more complex approaches, such as taking into account structural characteristics of the language (i.e. parts of speech, synonyms, related terms). Also, in this phase, particularities of the corpus can be exploited. For instance, when dealing with a specialized corpus, i.e. containing works only from Computer Science, a specialized semantic space can be constructed, as described in the next section. The following methods are not included in our final implementation, but they are presented here for future reference.

3.3.1 Latent Semantic Analysis

Two passages can speak about the same subject, even if their vocabulary is not exactly the same. This can be done by paraphrasing and synonymy. However, even though the vocabulary differs, the used terms are correlated to each other, meaning that they are likely to be found in the same context. For example, "*heap data structure*" is correlated to "*binary trees*", because they refer

to related concepts and are both probable to be found in Computer Science texts dealing with hierarchical data structures.

Replacing terms with their most common synonyms, using a dictionary like WordNet for example, is not effective, because terms need to be replaced depending on the context in which they appear. Thus, in order to identify whether two passages are related to each other (i.e. they speak about the same subject), Latent Semantic Analysis, or LSA (also referred to as Latent Semantic Indexing in the context of Information Retrieval), can be used.

LSA performs SVD (Singular Value Decomposition) on the term-document matrix M, which means that M is written as a product of the form U∑V*, where U and V* are unitary matrices and ∑ is a diagonal matrix with the singular values on the diagonal. We can reduce the data to a lower dimensional space by keeping only the biggest k values on the diagonal of ∑, and their corresponding singular vectors from U and V*, and multiplying these new matrices. Performing this semantic analysis would make the detection system to be Structural rather than Superficial, considering the classification proposed in [6].

It is known that LSA is affected by polysemy (words having different meanings in different contexts). For example, the word *tree* has a different meaning in Computer Science texts than in biology documents. The correlation of *tree* and *trunk* might be greater than the correlation between *tree* and *graph*, which is not desirable. This can be avoided by choosing the relevant semantic space, i.e. a corpus constructed from texts relating only to the area of interest. In case of a software aimed at detecting plagiarism in Computer Science documents, LSA can be performed only on Computer Science texts, which can be taken, for instance, from Wikipedia.

To perform LSA, one can use Gensim [50], which is a free framework written in Python that already contains algorithms for semantic analysis such as LSA or Latent Dirichlet Allocation (LDA). Initially, a preprocessing step can be employed for the whole corpus. Then, each document that needs to be analyzed can be brought in the lower dimensional space, by performing a dimensionality reduction.

3.3.2 Smith-Waterman Algorithm

Most of the algorithms presented so far fall under the bag of words model, and therefore all positional information is lost. It is possible for related texts to use the same vocabulary without being plagiarized. For example, it is not desirable to mark as similar long passages in which the words are highly scrambled. Smith-Waterman is a dynamic programming algorithm similar to the longest common subsequence, the difference being that operations like deletions or insertions can have a cost higher than one. The algorithm is widely-used in finding good near-matches, or so-called local alignments, within biological sequences. In [27], a variation of this algorithm is presented for the plagiarism and collusion detection case.

3.4 The Selected Approach in the Final Application. Results.

Even though our personal algorithm presented in section 3.1.4 produces better results than FastDocode, it is about an order of magnitude slower, so we decided to use the latter. Thus, the 5-gram version of FastDocode is used in the candidate selection phase. Because it obtained good results when tested against the golden standard of candidates, the method based on cosine similarity described in section 3.2.2 is employed for the detailed analysis phase. Also, to improve the precision of the results, a post-processing phase implementing the Smith-Waterman algorithm is included.

The total running time on the entire PAN 2011 corpus is about 20 hours. With no parameter tuning, we obtained the results displayed in Table 4. This ranked our algorithm between the 5th and the 6th place in the competition, and obtained the 4th recall.

Table 4

Our results on the PAN 2011 corpus

Plagdet Score	Recall	Precision	Granularity
0.221929185094	0.202996955425	0.366482242839	1.26150173611

The official results for the PAN 2011 competition, which can be found at [43], are also presented in the Figure 21 below. As it can be observed, our final score is very close to that of the contestants ranked on the 4th and the 5th place. Using the more time consuming candidate selection algorithm or improving the precision by filtering out passages in the post processing phase could bring us a better result.

External Plagiarism Detection Performance

Rank	Plagdet	Recall	Precision	Granularity	Participant
1	0.5563430	0.3965569	0.9368736	1.0022487	J. Grman and R. Ravas SVOP Ltd., Slovakia
2	0.4153395	0.3376925	0.8119867	1.2167900	C. Grozea* and M. Popescu° *Fraunhofer Institute FIRST, Germany °University of Bucharest, Romania
3	0.3468605	0.2257937	0.9116530	1.0611984	G. Oberreuter, G. L'Huillier, S A. Ríos, and J.D. Velásquez Universidad de Chile, Chile
4	0.2467329	0.1500480	0.7106536	1.0058894	N. Cooke, L. Gillam, P. Wrobel, H. Cooke, and F. Al-Obaidli University of Surrey, United Kingdom
5	0.2340035	0.1612845	0.8512947	1.2328923	D.A. Rodríguez Torrejón*,° and J.M. Martín Ramos° *IES "José Caballero", Spain °Universidad de Huelva, Spain
6	0.1990889	0.1618067	0.4541152	1.2949292	S. Rao, P. Gupta, K. Singhal, and P. Majumder DA-IICT, India
7	0.1892155	0.1390201	0.4435687	1.1716516	Y. Palkovskii, A. Belov, I. Muzyka Zhytomyr State University and SkyLine Inc., Ukraine
8	0.0804139	0.0885330	0.2780244	2.1823870	R.M.A. Nawab, M. Stevenson, and P. Clough University of Sheffield, United Kingdom
9	0.0012063	0.0011829	0.0050052	2.0028818	A. Ghosh, P. Bhaskar, S. Pal, and S. Bandyopadhyay Jadavpur University, India

Fig. 21. The official results for the PAN 2011 competition

We also ran AuthentiCOP on previous years theses. The corpus contained both the theses and about 8700 Computer Science English articles taken from Wikipedia. Because our dictionaries (i.e. for stemming and word frequencies) were in English, we first selected the English theses. The command line tool named TextCat [51] was used for the language detection task. Out of 940 theses, 307 were selected for analysis. Some theses were detected to contain passages which were very similar to extracts from Wikipedia articles. Table 5 presents such examples. Only short examples are presented here for brevity. As it can be observed, the paragraphs were not copy-pasted. Actually, the students slightly altered the original content. Some phrases are changed, others are removed and new phrases are sometimes added. Other example which is not presented in the table was a changed version of Dijkstra's algorithm pseudocode. This example shows that AuthentiCOP is able to deal with plagiarism not only in free text, but also in source code. There were also similar passages belonging to different theses, which may be because the authors copied the text from the same source. An example is presented in Table 6.

Table 5

Example of detected plagiarized passage from Wikipedia

Text taken from Theses	Original Text (taken from an Wikipedia article)
MapReduce [25] is a framework for processing huge datasets on certain kinds of distributable problems using a large number of computers (nodes), collectively referred to as a cluster (if all nodes use the same hardware) or as a grid. ... In the "Map" step the master node takes the input, partitions it up into smaller sub-problems, and distributes those to worker nodes. A worker node may do this again in turn, leading to a multi-level tree structure. The worker node processes that smaller problem, and passes the answer back to its master node. In the "Reduce" step the master node then takes the answers to all the sub-problems and combines them in some way to get the output – the answer to the problem it was originally trying to solve. The advantage of MapReduce is that it allows for distributed processing of the map and reduction operations.	MapReduce is a framework for processing embarrassingly parallel problems across huge datasets using a large number of computers (nodes), collectively referred to as a cluster (if all nodes are on the same local network and use similar hardware) or a grid (if the nodes are shared across geographically and administratively distributed systems, and use more heterogenous hardware). "Map" step: The master node takes the input, divides it into smaller sub-problems, and distributes them to worker nodes. A worker node may do this again in turn, leading to a multi-level tree structure. The worker node processes the smaller problem, and passes the answer back to its master node. "Reduce" step: The master node then collects the answers to all the sub-problems and combines them in some way to form the output – the answer to the problem it was originally trying to solve. MapReduce allows for distributed processing of the map and reduction operations.
The Canny edge detector uses a filter based on the first derivative of a Gaussian, because it is susceptible to noise present on raw unprocessed image data, so to begin with, the raw image is convolved with a Gaussian filter. The result is a slightly blurred version of the original which is not affected by a single noisy pixel to any significant degree.	Because the Canny edge detector is susceptible to noise present in raw unprocessed image data, it uses a filter based on a Gaussian (bell curve), where the raw image is convolved with a Gaussian filter. The result is a slightly blurred version of the original which is not affected by a single noisy pixel to any significant degree.

Table 6

Example of detected plagiarized passages from two different theses

The WSDL defines services as collections of network endpoints, or ports. The abstract definition of ports and messages are separated from their concrete use or instance, allowing the reuse of these definitions (Figure 3.4). A port is defined by associating a network address with a reusable binding, and a collection of ports defines a service.	The WSDL specification provides an XML format for documents for this purpose. The abstract definition of ports and messages are separated from their concrete use or instance, allowing the reuse of these definitions. A port is defined by associating a network address with a reusable binding, while a collection of ports defines a service.

It is worth mentioning that there were some detected passages which couldn't be considered plagiarism cases. Some of these passages contained common bibliographic resources or similar Java code (i.e. common imports). In a few cases, two different theses cited the same source. In these cases, the ultimate decision has to be taken only after a manually assessment of the results.

4. Conclusions

In this work we focused not only on implementing plagiarism detection algorithms and evaluating alternatives, but also on building a complete application that would allow users the see the results in a web interface. There are several future directions for improving the frontend application. A very useful feature would be to allow teachers to add documents in the corpus. In addition to that, every submitted work should be added to the corpus for reference. Another problem is that in the current implementation, the users have to send the documents for analysis one by one, which is inconvenient. What is more, this means that whenever a document is sent, a new backend process is started, which is time consuming. The alternative would be to allow users to send multiple documents at a time (i.e. in a ZIP or RAR archive), and then do batch processing, thus avoiding the creation of new processes for each document and improving the overall speed.

On the detection side, the algorithms for the candidate selection and for the detailed analysis phase need to be further benchmarked for various combinations of threshold parameters. In this respect, an important aspect that might improve the quality of the results is to let the system learn from users' feedback. Thus, after manual introspection, teachers can mark results as correct or incorrect, which can trigger parameter updates. Besides parameter tuning, we noticed that a significant performance improvement can be achieved if the algorithms combine positional information with the bag of words model. Also, we observed that when dealing with real word data, post processing is necessary, because otherwise there would be too many passages to be analyzed manually. Last but not least, there is a need to improve the semantic analysis component and the stemming support.

Despite the challenges posed by the task, initial results indicate that satisfactory detection results can be obtained on specialized corpora. Continuing the current work and improving the existent approaches can finally lead to a solution which can be used by University Politechnica of Bucharest to check if the theses submitted by students are plagiarism free.

B I B L I O G R A P H Y

[1] M. Mozgovoy, Desktop tools for Offline Plagiarism Detection in Computer Programs, Informatics in Education, 2006, Vol. 5, No. 1, pp. 97-112

[2] J. Carroll and J. Appleton, Plagiarism: A Good Practice Guide, May 2001, Oxford Brookes University, May 2011

[3] H. Maurer, F. Kappe, B. Zaka, Plagiarism - A Survey, Journal of Universal Computer Science, vol. 12, no. 8, August 2006, pp. 1050-1084

[4] P. Clough, Old and new challenges in automatic plagiarism detection, Department of Information Studies, University of Sheffield, February 2003

[5] M. Potthast, B. Stein, A. Eiselt, A. Barrón-Cedeño, P. Rosso, Overview of the 1st International Competition on Plagiarism Detection, Workshop on Uncovering Plagiarism, Authorship, and Social Software Misuse (PAN 09)

[6] T. Lancaster, F. Culwin, Classification of Plagiarism Detection Engines. E-journal ITALICS, vol. 4 issue 2, 2005

[7] S. Meyer zu Eissen and B. Stein, Intrinsic Plagiarism Detection, Springer, Proceedings of the 28th European Conference on IR Research, ECIR 2006, London, pp. 565-569

[8] E. Milios, Y. Zhang, B. He and L. Dong, Automatic Term Extraction and Document Similarity Search, Pacific Association for Computational Linguistics, 2003

[9] T. Joachims, Text Categorization with Support Vector Machines: Learning with Many Relevant Features, Proceedings of ECML-98, 10th European Conference on Machine Learning, 1998, pp. 137-142

[10] M. Potthast, A. Barrón-Cedeño, B. Stein and P. Rosso, Cross-language plagiarism detection, Springer Science+Business Media B.V. 2010, Published online: 30 January 2010

[11] Turnitin: Leading Plagiarism Checker, http://www.turnitin.com, accessed on 10 May 2012

[12] SVOP Ltd, System for plagiarism detection, http://www.svop.sk/en/antiplag.aspx, accessed on 10 May 2012

[13] SafeAssign by BlackBoard, http://www.safeassign.com, accessed on 10 June 2012

[14] G. Salton and C. Buckley, Term weighting approaches in automatic text retrieval, Information Processing and Management, vol. 24, no. 5, 1988, pp. 513-523

[15] Snowball, http://snowball.tartarus.org, accessed on 10 June 2012

[16] S. Schleimer, D. S. Wilkerson, A. Aiken, Winnowing: Local Algorithms for Document Fingerprinting, SIGMOD '03 Proceedings of the 2003 ACM SIGMOD international conference on Management of Data, pp. 76-85

[17] S. Brin, J. Davis, H. Garcia-Molina, COPS: Copy Detection Mechanisms for Digital Documents, ACM International Conference on Management of Data (SIGMOD 1995), May 1995

[18] N. Shivakumar and H. Garcia-Molina, SCAM: Building a Scalable and Accurate Copy Detection Mechanism, Proceedings of 1st ACM International Conference on Digital Libraries (DL'96) , March 1996

[19] A. Si, H. Va Leong, R. W. H. Lau, CHECK: A Document Plagiarism Detection System, Proceedings of ACM Symposium for Applied Computing, February 1997, pp. 70-77

[20] A. R. Pereira Jr. and N. Ziviani, Syntactic Similarity of Web Documents, IEEE, Proc. of the 1st Latin American Web Congress, LA-WEB 2003

[21] C. Lyon, R. Barrett and J. Malcolm, A theoretical basis to the automated detection of copying between texts, and its practical implementation in the Ferret plagiarism and collusion detector, Plagiarism: Prevention, Practice and Policies, vol 2004

[22] Z. Ceska, The Future of Copy Detection Techniques, Proceedings of the 1st Young Researchers Conference on Applied Sciences (YRCAS 2007), November 2007, pp. 5–107

[23] S. Alzahrani and N. Salim, Fuzzy Semantic-Based String Similarity for Extrinsic Plagiarism Detection, Lab Report for PAN at CLEF 2010

[24] WordNet: A lexical database for English, http://wordnet.princeton.edu, accessed on 23 June 2012

[25] T. W. S. Chow and M. K. M. Rahman, Multilayer SOM With Tree-Structured Data for Efficient Document Retrieval and Plagiarism Detection, IEEE Transactions on Neural Networks, vol. 20, no. 9, September 2009, pp. 1385 - 1402

[26] M. Potthast, Technologies for Reusing Text from the Web, PhD Thesis, Bauhaus-Universität, Weimar, Germany, 2011

[27] R. W. Irving, Plagiarism and Collusion Detection using the Smith-Waterman Algorithm, Dept of Computing Science, University of Glasgow, 2004, pp. 1-24

[28] F. Bravo-Marquez, G. L'Huillier, S. A. Rios, J. D. Velasquez, and L. A. Guerrero, DOCODE-Lite: A Meta-Search Engine for Document Similarity Retrieval, R. Setchi et al. (Eds.): KES 2010, Part II, LNAI 6277, 2010, pp. 93–102

[29] Martin Potthast, Benno Stein, Matthias Hagen, Tim Gollub, and Maik Anderka, http://pan.webis.de/, accessed on 10 May 2012

[30] G. Oberreuter, G. L'Huillier, S. A. Ríos and J. D. Velásquez, FASTDOCODE: Finding Approximated Segments of N-Grams for Document Copy Detection, Lab Report for PAN at CLEF, 2010

[31] C. Grozea, M. Popescu, The Encoplot Similarity Measure for Automatic Detection of Plagiarism, Extended Technical Report, Fraunhofer Institute FIRST, Berlin, Germany, 2011

[32] C. Grozea, M. Popescu, The Encoplot Similarity Measure for Automatic Detection of Plagiarism, Notebook for PAN at CLEF 2011

[33] R. J. Bayardo, Y. Ma, R. Srikant, Scaling Up All Pairs Similarity Search, WWW '07 Proceedings of the 16th international conference on World Wide Web, pp. 131-140
[34] C. Xiao, W. Wang, X. Lin, J. Xu Yu, Efficient similarity joins for near duplicate detection, WWW '08 Proceedings of the 17th international conference on World Wide Web, pp. 131-140
[35] M. Potthast, A. Eiselt, A. Barrón-Cedeño, B. Stein, and P. Rosso, Overview of the 3rd International Competition on Plagiarism Detection, in CLEF Notebook Papers/Labs/Workshop, 2011
[36] Project Gutenberg - free ebooks, http://www.gutenberg.org/, accessed on 9 June 2012
[37] A. Barrón-Cedeño, M. Potthast, P. Rosso, B. Stein, A. Eiselt, Corpus and Evaluation Measures for Automatic Plagiarism Detection, in Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)
[38] Apache Tika, http://tika.apache.org/, accessed on 10 May 2012
[39] HipHop for PHP, https://github.com/facebook/hiphop-php/wiki/, accessed on 3 June 2012
[40] JsonCpp project page, http://jsoncpp.sourceforge.net/, accessed on 19 May 2012
[41] RapidXML, http://rapidxml.sourceforge.net/, accessed on 19 May 2012
[42] MySQL C API, Reference Manual, http://dev.mysql.com/doc/refman/5.0/en/c.html, accessed on 19 May 2012
[43] PAN @ CLEF 2011, http://www.uni-weimar.de/medien/webis/research/events/pan-11/pan11-web/plagiarism-detection.html, accessed on 15 April 2012
[44] B.W. Matthews, Comparison of the predicted and observed secondary structure of T4 phage lysozyme, Biochim, Biophys, Acta 1975, pp. 442–451
[45] Apache Lucene, http://lucene.apache.org/core/, accessed on 10 May 2012
[46] CLucene - lightning fast C++ search engine, http://clucene.sourceforge.net/, accessed on 26 May 2012
[47] Google Books Ngram Viewer, http://books.google.com/ngrams/datasets, accessed on 15 April 2012
[48] W. R. Pearson and D. J. Lipman, Improved tools for biological sequence comparison, *Proceedings of the* National Academy of Sciences of the United States of America, vol. 85, April 1988, pp. 2444–2448
[49] S. F. Altschul, W. Gish, W. Miller, E. W. Myers and D. J. Lipman, Basic Local Alignment Search Tool, Journal of Molecular Biology vol. 215, pp. 403–410
[50] Gensim – Topic Modelling for Humans, http://radimrehurek.com/gensim, accessed on 10 May 2012
[51] TextCat Language Guesser, http://odur.let.rug.nl/vannoord/TextCat/, accessed on 23 June 2012