Universitatea POLITEHNICA din București

Facultatea de Automatică și Calculatoare,
Catedra de Calculatoare

LUCRARE DE DIPLOMĂ

Sistemul AuthentiCop de Detecție a Plagiatului în Corpusuri de Specialitate: Algoritmi și Procesări de Date

Conducători Științifici:
Conf. dr. ing. Răzvan Rughiniș
As. drd. ing. Traian Rebedea

Autor:
Adrian Scoică

București, 2012

University POLITEHNICA of Bucharest

Automatic Control and Computers Faculty,
Computer Science and Engineering Department

BACHELOR THESIS

The AuthentiCop System for Plagiarism Detection in Specialized Corpora: Algorithms and Data Processing

Scientific Advisers:
Conf. PhD Eng. Răzvan Rughiniș
Eng. Traian Rebedea

Author:
Adrian Scoică

Bucharest, 2012

I would like to thank my supervisors,
Răzvan Rughiniș and Traian Rebedea,
for the guidance, support, invaluable advice
and understanding they have offered
me throughout the development of
this project in particular and
my formative university years in general.

Abstract

The Webster English Dictionary[1] defines plagiarism as the activity of "stealing and passing off the ideas or words of another as one's own". Plagiarism in academic writing is considered to be one of the worst breaches of professional conduct in western culture. Thus, despite the fact that the problem of automatically detecting plagiarism in written documents is an open, computationally hard problem in the field of Natural Language Processing, it is also one of particular practical interest.

AuthentiCop is a software system that we built for detecting instances of plagiarism in specialized corpora. This project aims at studying the characteristics that differentiate such corpora from general text samples and the way various successful solutions to the general problem of plagiarism detection were improved and implemented for use on specialized writing within AuthentiCop.

The study has been conducted on text from academic papers from the field of Computer Science. The algorithms studied and their benchmarks were taken from the Plagiarism Detection section of the PAN 2011 Workshop[2] and the English Wikipedia article base was used as an external corpus for drawing statistics.

[1] http://www.merriam-webster.com/dictionary/plagiarism
[2] http://pan.webis.de/

Contents

List of Figures

List of Tables

Notations and Abbreviations

CS – Computer Science
DOC – Microsoft Document Format
DTD – Document Type Definition
HTML – Hyper Text Markup Language
ODF – Open Document Format
PDF – Portable Document Format
tf – term frequency

Chapter 1

Introduction

1.1 Motivation

Plagiarism in academic writing is considered by current ethical standards to be an act of dishonesty and a breach of professional conduct that undermines the quality of education. There are severe punishments in place aimed at discouraging students from submitting work that is not their own original creation.

However, as the amount of written material in any given field of study increases at a rate which would make it impossible for a human to read over the course of an entire lifetime, enforcing anti-plagiarism rules in practice is a process which must be aided by automated tools.

We admit that the final decision on what constitutes a true case of plagiarism and what is a coincidental similarity in wording cannot be trusted to a software application due to the complex nature of language. However, an automatic detector can help guide the human reader's attention towards the most likely pieces of text which might arise suspicion, so that the task of checking against plagiarism becomes manageable from the reviewer's perspective.

Various solutions have currently been proposed to the problem of plagiarism detection, as of the moment of writing this paper. These solutions have proven to be successful to different degrees when used in practice, but most of them address the general problem and thus further improvements can be made if we consider the case of specialized corpora.

The AuthentiCop system was developed because of the need for a plagiarism detection solution that is optimised to handle academic papers from a specific field - such as Computer Science - and which use specialized language and terminology.

1.2 The Plagiarism Detection Problem

1.2.1 Formal Definition

The problem of plagiarism detection can be formally defined as follows: let

- *Source* be a set of text documents from which plagiarism may have occured;
- *Suspicious* be a set of text documents that we suspect may contain plagiarised passages.

The *Source* set could be either an explicitly enumerated corpus (such as the documents in a database) or an implicitly specified one (such as the entire World Wide Web).

Following the notations and definitions in the international PAN workshop on plagiarism detection[18], the problem of plagiarism detection can be stated as finding all the tuples

$$case = \langle s_{plg}, d_{plg}, s_{src}, d_{src} \rangle$$

where:

- $d_{plg} \in Suspicious$ is a document that contains plagiarism;
- $d_{src} \in Source$ is a document that served as a source for plagiarism;
- $s_{plg} \in d_{plg}$ is a passage that was plagiarised;
- $s_{src} \in d_{src}$ is the passage that s_{plg} has been plagiarised after.

Similarly, let

$$detection = \langle r_{plg}, d_{plg}, r_{src}, d'_{src} \rangle$$

be an instance of plagiarism detected by a plagiarism detection system where passage r_{plg} from document d_{plg} is claimed to be plagiarised after passage r_{src} from documet d'_{src}. We say that *detection detects case* iff:

- d_{src} is the same as d'_{src};
- passages r_{plg} and s_{plg} are not disjoint;
- passages r_{src} and s_{src} are not disjoint.

1.2.2 Performance Metrics

In order to objectively quantify the performance of various components of the AuthentiCop system, concrete performance metrics were defined and used for benchmarking.

For the purpose of assessing various pre-processing methods, the language models of the post-processed documents were compared. The properties of the specialized corpora AuthentiCop has to be optimized for were described in terms of the language model:

- Number of lexems present in the corpus and their respective term frequencies;
- Impact of normalization techniques on the language model;
- Impact of stopword removal on the language model;
- Impact of stemming on the language model.

For the purpose of evaluating the candidate selection stage of the AuthentiCop pipeline (responsible for search space reduction), the PAN 2011 corpus on plagiarism detection and its golden standard were used[18]. In the following definitions, R is taken to represent the set of detections and S is taken to represent the set of plagiarism cases:

- **Precision** is a measure which intuitively describes the proportion of true plagiarism detections out of the total number of detections output by a given algorithm (the better it is, the smaller the number of false positives). Precision can be defined in two ways:
  - **Micro-precision** takes into account the length of the plagiarised passages detected. Micro-precision is defined as:

$$prec_{micro}(S, R) = \frac{|\bigcup_{(s,r)\in(S\times R)}(s \cap r)|}{|\bigcup_{r\in R} r|}$$

  - **Macro-precision** only takes into account the cardinality of the plagiarism passages detected, regardless of their length. Macro-precision is defined as:

$$prec_{macro}(S,R) = \frac{1}{|R|}\Sigma_{r\in R}\frac{|\bigcup_{s\in S}(s\cap r)|}{|r|}$$

- **Recall** is a measure which intuitively describes the proportion of true plagiarism detections which have been detected by a given algorithm (the better it is, the smaller the number of plagiarisms which slip undetected). Similarly to precision, recall can be defined in two ways:
  - **Micro-recall** takes into account the length of the plagiarised passages detected. Micro-recall is defined as:

$$rec_{micro}(S,R) = \frac{|\bigcup_{(s,r)\in(S\times R)}(s\cap r)|}{|\bigcup_{s\in S} s|}$$

  - **Macro-recall** only takes into account the cardinality of the plagiarism passages detected, regardless of their length. Macro-recall is defined as:

$$rec_{macro}(S,R) = \frac{1}{|S|}\Sigma_{s\in S}\frac{|\bigcup_{r\in R}(s\cap r)|}{|s|}$$

- **Granularity** is a measure that quantitatively describes whether a large plagiarised passage is detected as a whole or as multiple smaller pieces. Granularity is defined as:

$$gran(S,R) = \frac{1}{|S_R|}\Sigma_{s\in S_R}|R_s|$$

  where S_R are instances detected by detections in R and R_s are all detections of passage s.

- **PlagDet** score is a measure that aggregates all previous measures in such a way that it allows an objective ranking of various methods. It is computed as follows:

$$plagdet(S,R) = \frac{F_\alpha}{log_2(1+gran(S,R))}$$

  where F_α is the harmonic mean of precision and recall.

1.3 Related Work

A great amount of research has been carried out on the topic of automatic plagiarism detection both in the academia and in the industry. There is a great deal of diversity in the solutions that currently make up the state of the art in the field, both in regard to algorithms used and the search strategies employed (eg. live web search as opposed to document retrieval from a previously compiled database).

In the following sections, I will go over four of the most successful solutions to the problem of plagiarism detection, highlighting key results and features, as well as shortfalls.

1.3.1 Turnitin

Turnitin[13] is an online service provided by iParadigms that enjoys great popularity within the North American educational systems, and for which a classroom integration scheme was implemented in these countries[6].

Turnitin is by far the most popular plagiarism detection service currently in operation on the web. However, as the software driving it is propietary, many details of its implementation remain undisclosed.

The service uses a database of papers previously submitted by the students. Each document uploaded to Turnitin is processed and a *fingerprint* is extracted from it[19]. Each newly submitted document is compared not only to the other documents in the database for similarity, but also to the entire World Wide Web during a crawling session. A report[20] ranked the web search component of Turnitin first among all other plagiarism detection tools studied.

Turnitin is built to detect some of the most frequently used plagiarism methods. In a study[14] the company released, the 10 most frequently employed plagiarism methods that the service detects were listed:

- cloning, taking another's work word for word;
- copying and pasting a large passage of text from a single source without any insertions and/or deletions;
- reusing a piece of text while replacing certain phrases and keywords, but without altering the structure of the text;
- paraphrasing other sources;
- self plagiarism from a previous work of the same author;
- combining cited sources with copied passages without citation;
- copying pieces of test from various other works without citation;
- citations to non-existent sources or inaccurate citations;
- a low proportion of original work in the final paper, in spite of properly cited sources;
- changes in the word ordering and deletions or insertions of text from other sources, with little or no added original work.

A few shortfalls of Turnitin which need addressing are:

- proper citations are sometimes not always recognized properly and are labeled as plagiarism;
- the fact that the service retains a copy of the students' work raises questions regarding privacy and copyrigt;
- a full report on the plagiarism status of a document needs around 24 hours to generate because of the time consuming web search component.

1.3.2 AntiPlag

AntiPlag[7] is a plagiarism detection tool developed in Slovakia by the Slovak Centre of Scientific and Technical Information and used by the Ministry of Education there.

The system had been under development since 2008 and its primary mode of operation is through information retrieval from a database of papers extended incrementally with each new submission.

Even though SVOP, the company managing the software, doesn't release implementation details, AntiPlan none the less remains an example for a successful anti plagiarism detection system.

1.3.3 Encoplot

Encoplot[10] is the name of an algorithm and a software system for automatic detection of plagiarism developed at the Weimar University by Cristian Grozea and Marius Popescu.

The software infrastructure used by Encoplot is a scalable one, as it is parallelised using the OpenMP library. The main steps of the processing pipeline are candidate pair ranking, followed by a detailed analysis stage which is built upon the Dotplot[11].

Having implemented no algorithm calibrations which would have been specific to the training corpus, the Encoplot method achieved precision and recall scores that ranked it second at the PAN 2011 competition[1]. Encoplot was especially effective at detecting manually plagiarised passages from the test corpus (having ranked first in this section), which makes this approach of particular practical interest to the problem that AuthentiCop is designed to solve.

One of the recognized shortfalls of Encoplot is its failure to recognize plagiarism cases across different languages due to the obfuscation introduced by translation engines. The method also currently only operates on enumerated corpora, having no web search component built into it.

1.3.4 Fastdocode

Fastdocode[9] stands for **Fast Document Copy Detection** and is a plagiarism detection method and algorithm developed at the University of Chile by Gabriel Oberreuter and his collaborators.

The Fastdocode method is composed of two main stages:

- an approximated segment finding algorithm, which aims at roughly identifying text areas which might be plagiarised from a given source;
- an exhaustive search, which takes the results of the previous step as inputs and computes actual text passages by extending and joining the previously determined areas.

The Fastdocode method relies on n-grams (specifically trigrams) for its computations. To reduce the size of the problem, the preprocessing stages of the algorithm filter out stopwords and infrequent ngrams, although the authors argue that the impact of this approach on the final results had not been adequatly studied.[9]

Fastdocode also ranked in the top three in the PAN 2011 competition, but just as the Encoplot method, it doesn't have any support for plagiarism detection from web sources. Unlike Encoplot, though, its parameters allow for calibration dependant on the language model of the corpus studied, and as such it might be possible to tune Fastdocode to have good results for a cetain type of documents, as AuthentiCop is aiming to achieve.

1.4 Project Proposal

Although there are a number of off the shelf solutions to the problem of plagiarism detection, most of them are either prohibitively expensive, closed-source, or simply not well suited to the case of specialized corpora.

The AuthentiCop project has two main objectives. One one hand, I aimed to discover how the traits and profile of corpora in the field of Computer Science are different from those of general texts. One the other hand, I explored how that information can be used to tune in algorithms for obtaining better performances at the task of plagiarism detection.

1.5 Summary

Chapter 1 presented with a brief overview of the problem of plagiarism detection and of the state of the art in the field and concluded with a statement of the project proposal.

Chapter 2 of this paper describes the main components of the AuthnetiCop system pipeline and the data flow through the system. A few notes are given on the general project layout and build system.

Chapter 3 speaks about the first steps in the pipeline and talks about solutions for document format conversion, stemming, normalization and the traits of the language model of a Computer Science corpus composed of academic theses.

In Chapter 4 we will discuss text and binary file formats for storing ngram models of documents and the process of building a Computer Science corpus from the Wikipedia databse to be used for keyword extraction and topic detection.

Chapter 5 speaks about the problem of search space reduction and an adaption of the Encoplot algorithm for the PAN 2011 corpus.

Finally, Chapter 6 sums up the efforts and identifies a few potential directions for further development.

Chapter 2

System Data Flow and Architecture

2.1 AuthentiCop Data Flow

2.1.1 Overview

The flow of data in the AuthentiCop plagiarism detection system can be accurately described by the chart in Figure 2.1 below. This section describes the main steps of information processing.

The system takes inputs in the form of suspicious documents. For the use case of academic papers in the field of Computer Science that AuthentiCop is tuned for, these documents will almost invariably be uploaded in PDF format, but other supported formats include HTML, XML, ODF, RTF and Microsoft Office proprietary formats.

2.1.2 Description of Stages

The **conversion (1)** stage of the processing pipeline takes care of retrieving the text content from the uploaded documents. Though not explicitly shown on the diagram, conversion also takes place for documents that are retrieved from the Web. These documents may come in a variety of formats, though most usually they are either HTML, XML or PDF. The end product of the conversion stage is a **plain text representation** of the document that retains all of the original content.

Since the plain text representation may contain a great amount of noise stemming from misinterpreted boilerplate sources, the raw text document must undergo preprocessing to filter out these tokens. Possible sources for noise include, without being limited to:

- page header and footer;
- page numbering;
- references on the bottom of the page;
- tables;
- table and figure captions.

The **preprocessing (2)** stage is tasked with lexing the source document and filtering out any non-word tokens and stopwords. In addition to filtering, the preprocessing also performs normalization and stemming on the tokens of the text. Without normalization and stemming, the plagiarism detector would only be able to detect instances of verbatim plagiarism. The

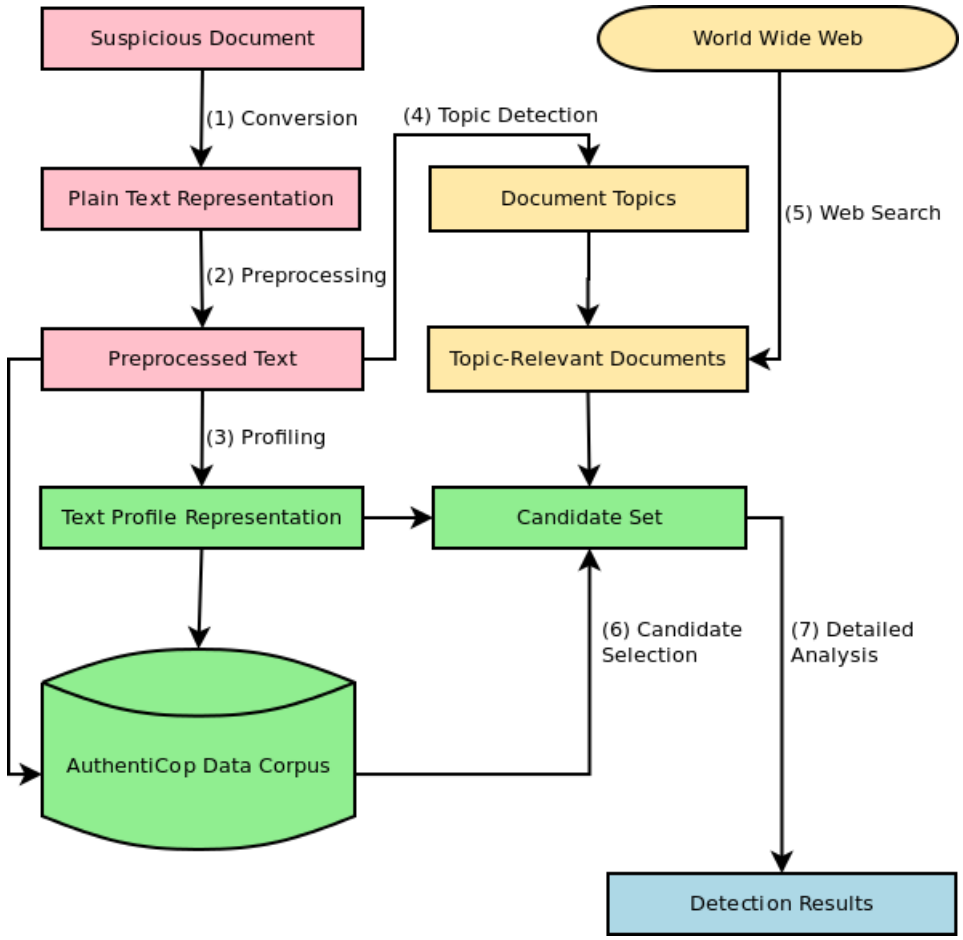

Figure 2.1: Dataflow in AuthentiCop

output of the preprocessing stage is a sanitized plain text document that is both saved into the AuthentiCop database for further reference and fed to the next stage in the processing pipeline: the profiling stage.

The **profiling (3)** stage is responsible with building an internal representation of the document. This involves replacing the actual words in the document with an ID that uniquely represents the lexem in the AutentiCop database and then building and sorting the vector of document ngrams. This stage is necessary because both algorithms studied rely on ngram representations of the text and recomputing the ngram vectors every time the file gets loaded into memory wastes processors resources.

At the same time, the sanitized document is fed into a **topic detection (4)** module, which extracts keywords relevant to the content. The keywords will then be used to **query search engines (5)** on the web and retrieve extra source document candidates to be used during the detailed analysis stage.

These candidate documents are added to a set of candidate documents retrieved from the AuthentiCop database during the **candidate selection (6)** stage.

The candidate selection stage is the first out of two stages that comprise the actual plagiarism detection pipeline, the second one being the detailed analysis stage. Candidate selection is important because running a detailed analysis against all the documents in the AuthentiCop databse is computationally unfeasable and unnecessary. The role of candidate selection is to

perform **search space reduction** so that only the source documents which have the best chance of having been plagiarised from are used in the detailed analysis stage.

Ultimately, the candidate set and the suspicious document undergo the **detailed analysis (7)** stage, during which the suspicious document is compared in detail to every candidate and the plagiarised sections are marked accordingly. The output of the detailed analysis stage is an XML file which maps plagiarised sections to their souces in the source documents according to the formal definition in Section 1.2.1.

2.2 AuthentiCop Architecture

2.2.1 Overview

Managing a system as complex as AuthentiCop presents with additional challenges. The source code is written in multiple languages, with the most common being C++ modules for the binaries, Python scrips and Bash shell scripts for the backend section, and PHP and JavaScript for the front end. Moreover, the C++ codebase is especially large and in an effort to avoid code duplication, it comprises of a large number of modules that implement data structures, algorithms and auxiliary systems such as the logging system.

A brief description of the main components of the application is given in Figure 2.2.

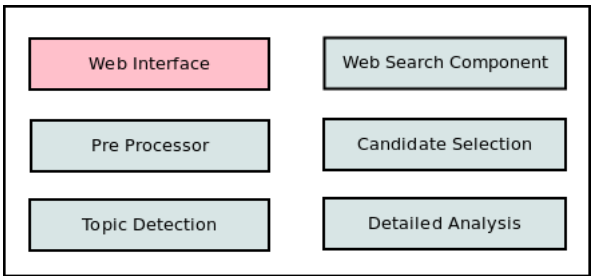

Figure 2.2: Architecture of the AuthentiCop components

The system's architecture contains components which roughly correspond to the main stages of data processing. However, the system also features a graphical interface which enables the user to upload documents and to view the results.

2.2.2 Build System for C++ Source Code

In order to prevent the build and include dependencies from getting out of control, AuthentiCop has its own build system. Each unit of code is organized into its own directory, also called a **module**. Each module directory subtree has the following elements:

- a **module_init** file, decribing the type of module and its dependencies;
- a **header file** file, describing the API that the module exposes to other modules.
- a **src/** directory which contains the C++ implementation files that code the module's functionality.

An actual example of module file organization can be seen in Figure 2.3 for the **ngram-indexes** module.

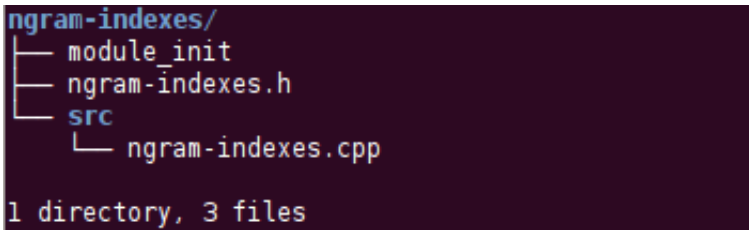

Figure 2.3: Example of module organization

The **module_init** file specifies dependencies and whether the module should be compiled into an object file (***.o**) or an executable. Based on the dependencies specified in the module file, a special **deps.h** file will be generated which in turns includes the headers of all required modules. The clear advantage to this approach is that the implementation files needn't specify full paths for module interfaces, but rather include the **deps.h** file alone.

To finish up the example, the module_init file for the **ngram-indexes** module is given in the following listing:

```
1 module_type('OBJ')
2
3 require_module('core/logging')
4 require_module('core/utils')
5 require_module('core/dictionary')
6 require_module('core/ngram')
7 require_module('core/hash')
8 require_module('core/lexer')
```

Chapter 3

Conversion, Preprocessing and Analysis

3.1 Format Conversion

As shown in Figure 2.2, the designed method of input for documents to be analysed for plagiarism is the web interface. Because AuthentiCop is designed and optimised for plagiarism detection in academic papers in the field of CS, and because papers are usually submitted in PDF format, with DOC and ODF as less frequent but viable alternatives, the first stage in the document processing pipeline is the format conversion stage. It is during this stage that the text content of the documents is extracted and then fed into the preprocessor for the next stage.

A number of third party conversion tools have been analysed for the purpose of text retrieval. This section describes the top two choices: **PoDoFo**[2] and **Apache Tika**[3] and speaks about their evaluation and performances.

3.1.1 The PoDoFo C++ Conversion Libraries

PoDoFo stands for **Portable Document Format** and is an Open Source library for creating and parsing PDF documents. It has been in active development since May 2nd, 2006 and it is currenty at version 0.9.1 with the last stable release of the source code published on April 26th, 2011. The source code and documentation are hosted on SourceForce[1] and are publicly accessible.

Being written entirely in C++ and being completely platform-independent, PoDoFo immediately stood out as a viable solution that could be linked directly into the project's C++ modules without complicating the existing architecture. Its API is very clean and easy to use, with basic text retrieval possible with as little as under 10 lines of code, as can be observed in the Listing 3.1 below.

However, despite its apparent ease of use, the library does not work out of the box, has a flaky support for UTF-8 letter encodings and generally does not preserve the original whitespace layout in the document. Even after significant changes brought to the source code of the parser in the area of whitespace management, the retrieved text incorrectly identifies paragraph limits.

[1] http://podofo.sourceforge.net/download.html

Table 3.1: Counts of Documents parsed and classified according to the tf of the word *the*

Parser	Document Classified as English	Documents Classified as Romanian
PoDoFo	0	856
Tike	289	567

```
#include "TextExtractor.h"

/* ... */
TextExtractor extractor;
try {
  extractor.Init(pdfFileName);
} catch(PdfError& e) {
  e.PrintErrorMsg();
  return e.GetError();
}
/* ... */
```

Listing 3.1: Example of parsing a PDF file using the PoDoFo library

3.1.2 Apache Tika for Document Conversion

Apache Tika is a product of the Apache Software Foundation used for metadata retrieval and text content retrieval from a variety of document formats. The first release of Apache Tika was in January 2008[3], and it has been in active development alongside Apache Lucene, with the last stable release occuring on March 23rd, 2012.

Both the source code and a runnable JAR archive can be downloaded freely from the official Apache Tika website. The JAR file can be run out of the box and will perform text retrieval from most widely used formats (among others: HTML, XML, DOC, ODF, PDF, EPF, RTF) into an either plain text, XML or HTML output.

In general, Tika preserves the whitespace layout of the original PDF documents intact, with the exception of breaking paragraphs into individual lines. Full UTF-8 support is also not perfect, but its behavior is more predictable than in the case of PoDoFo. With Tika, there is also less chance that the conversion should fail altogether because of unrecognized fonts or formats in the source PDF file.

3.1.3 Performance Analysis: Procedures, Metrics and Results

In order to objectively quantify the performance of PoDoFo versus Apache Tika, I designed a side-by-side test which I ran on a corpus of 857 papers on topics from Computer Science. A fraction of the papers were written in English, with the majority written in Romanian.

During the first stage of the test, I ran both parsers on the document corpus and then classified the output as follows:

- **English**, if the of the word *the* in the output is larger than 1.00%[1];
- **Romanian**, otherwise.

A brief count of the classification results from the experiment in given in Table 3.1.

The results of the experiment indicate that PoDoFo is of no practical use because of the number of documents it failed to parse (among which all of the English language documents, defaulting to Romanian in their classification).

After the initial results with PoDoFo, I decided to evaluate in more detail the accuracy of the parsing done by Tika. First I calculated the Language Models for the Romanian and for the English set of documents. Let

$$LM_R = \{< w, freq_w > | w \in Words, freq_w \in (0, 1)\}$$
$$LM_E = \{< w, freq_w > | w \in Words, freq_w \in (0, 1)\}$$

be the two language models. Two histogram charts of the 30 most frequent words in the two languages, including stop words and with no normalization and no stemming are given in Figure 3.2 and Figure 3.3, respectively.

The two histograms confirm that parse errors during conversion did not have a significant impact on the lower ranks of the language models. With this confirmation, I turned my attention to the less frequently occuring lexems, as they are much more prone to being affected by parse errors. Since the language models for both Romanian and English were rather sizeable and not completely disjoint (with the Romanian-derived language model containing a significant amount of English loanwords), I decided to verify their soundness (and thus, the accuracy of Tika's retrieval) by using Zipf's law.

Le Quan Ha and F. J. Smith proved in [15] that Zipf's law holds even when applied to syllable ngrams in English or to single characters in Chinese, provided that the syllables and characters are taken from real language samples. We expect that a text sample obtained from the preprocessing stage might contain errors that would distort the Language Model in the high ranks by amplifying a phenomena called *hapax legomena* (introducing erroneous words which only occur once or twice in the text). As indicated by Figure 3.1, there are two abnormalities in the language models:

- for the higher ranks, we see some evidence of *hapax legomena*, which was confirmed by looking at the log data and finding lexems such as *realizaconexiunea* (missing space between words) or ——————*Anid* (misinterpreted text).
- for the lower ranks, there is a plateau region on the curves. This can be explained by the lack of normalization and stemming, as evident from Figure 3.2, which contains both *The* and *the*, as well as *is* and *be*.

3.2 Preprocessing

3.2.1 Normalization

The process of normalization in natural language processing refers to changing the case of letters in a given word so as to bring the word to a canonical form. Solving the problem of normalization in its own right requires knowledge of the semnatics of words, namely whether they represent Named Entities (i.e. they are proper nouns) or not. However, the plagiarism detection algorithms used in AuthentiCop would have no benefit from this information and in fact would be led astray by the uppercase letters at the beginning of sentences in cases where obfuscation changes the first word in the sentence.

[1]The actual frequency of the word *the* in English is placed at around 6.17% according to [12]

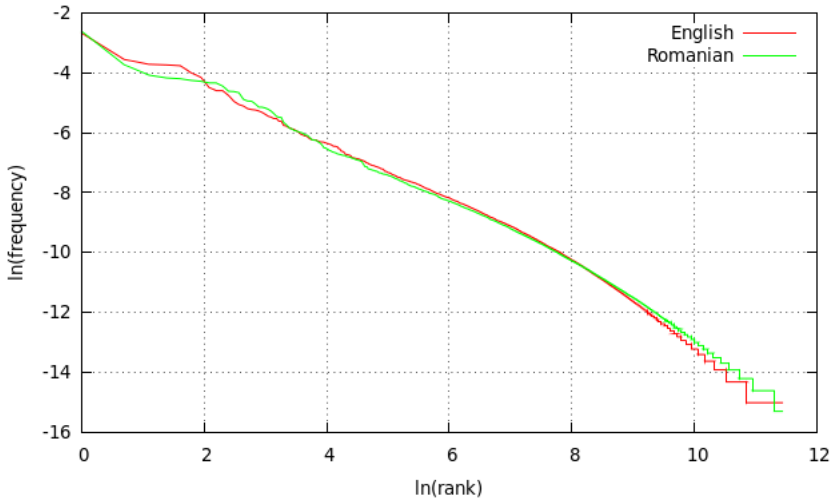

Figure 3.1: Verification of Zipf's law on pre-processed corpora

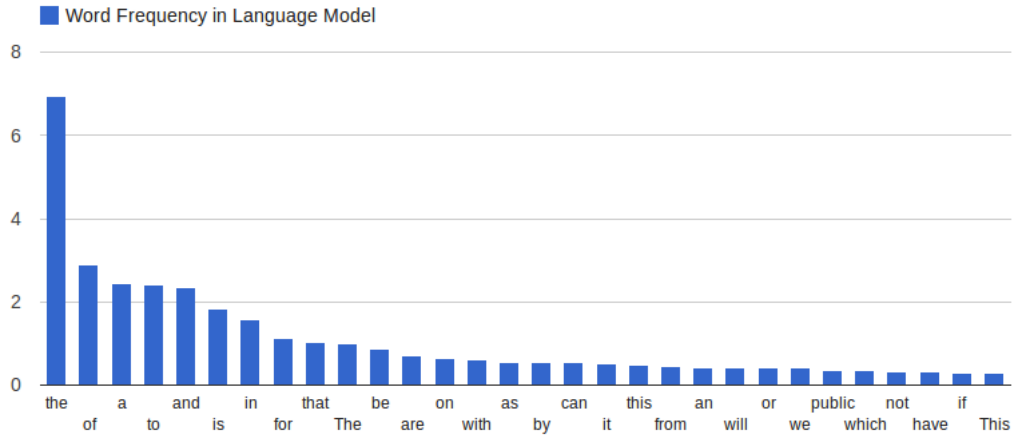

Figure 3.2: Language Model (top 30 lexems) - English, no normalization and stemming

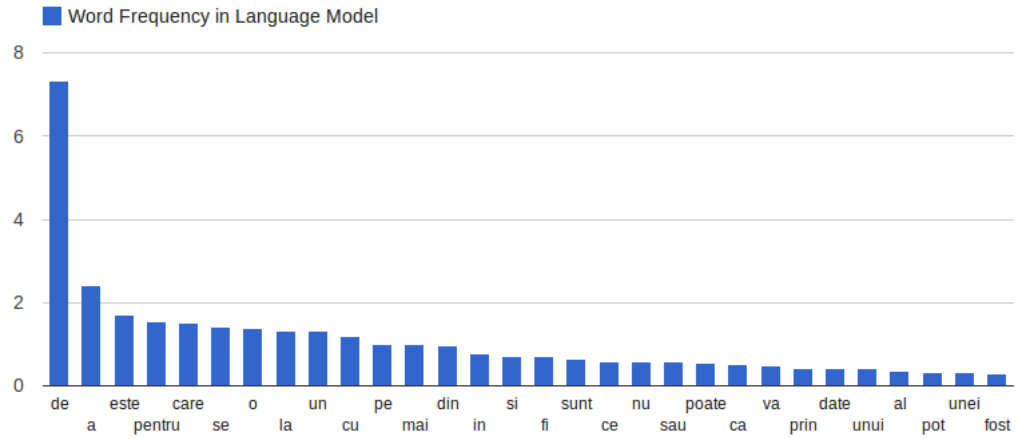

Figure 3.3: Language Model (top 30 lexems) - Romanian, no normalization, and stemming

Table 3.2: Impact of normalization on the first 10 ranks in each of the derived language models.

Rank	Lexem	Frequency	Rank change	Lexem	Frequency	Rank change
1	the	7.98137%	–	de	7.55475%	–
2	of	2.91625%	–	a	2.52178%	–
3	a	2.63528%	–	pentru	1.81789%	↑ 1
4	to	2.46376%	–	este	1.77239%	↓ 1
5	and	2.37233%	–	se	1.54906%	↑ 1
6	in	1.85986%	↑ 1	care	1.51822%	↓ 1
7	is	1.84017%	↓ 1	o	1.51514%	–
8	for	1.25415%	–	un	1.43823%	↑ 1
9	that	1.02780%	–	la	1.38915%	↓ 1
10	be	0.86723%	↑ 1	cu	1.22596%	–

My decision was to render all letters to their lowercase versions. The correction brought to the language models by normalization is significant. The English language model shrank from 92267 lexems down to 75168 lexems, a change of 18.53%, while the Romanian language model shrank from 149935 individual lexems down to 124151 lexems, a change of 17.196%.

A summary of the effects of normalization on the first 10 ranks of the two language models is given in Table 3.2.

3.2.2 Stemming

In addition to normalization, the most important step in the data pre-processing pipeline is **stemming**. Stemming is the process of reducing a given lexeme to a base, canonical form (for example, *features* can be reduced to *featur*). By subjecting the text samples to stemming before running the plagiarism detection algorithms, we take the first step at making sure that the results produced are insensitive to rephrasing. The language models derived after stemming all individually retrieved lexems from the PDF files are also smaller and more relevant, following Zipf's law more closely.

Stemming should not be confused with **lemmatization**, which is the process of correctly reducing a word to its uninflected dictionary representative (also called a **lemma**). The problem of lemmatization is a very difficult one to solve correctly, requiring both knowledge of the context in which the lexem is encountered (for example, *saw* could be reduced to either the verb *see* or the noun *saw*) and knowledge of all the irregularities in inflection rules in a given language.

By comparison, stemming attempts a crude transformation by subsequently applying heuristics which truncate what are believed to be suffixes to a given stem. Stemming, unlike lemmatization, does not produce valid word forms and it is not expected to be error-free, but rather a fast and approximate process for reducing different word forms to the same class of equivalence.

A stemmer is described by a set of rules and their relative order of application on a given lexeme. For English, a simple example of rules which would reduce a given noun to the singular form is described below. The application order specifies that out of a set of matching rules, the one with the longest matching suffix is triggered first (so that, for example, te word *duchess* wouldn't get erroneously reduced to the stem *duches* as indicated by the final rule in the example below). This ultimately gives the freedom to hardcode special cases individually, although it is rarely done in practice.

$$SSES \rightarrow SS$$

$$IES \rightarrow I$$

$$SS \rightarrow SS$$

$$S \rightarrow \oslash$$

Among the stemming algorithms available for English, the best known one is *Porter's Algorithm*[17] which has been empirically proven to give good results, although there are also newer stemmers, such as the *Paice Stemmer*[16].

In order to integrate stemming in AuthentiCop's data processing pipeline, the **Snowball** stemmers available for download on the project's official website[1] for English and Romanian were used. The English stemmer uses an improved implementation of Porter's initial 1980 algorithm, while support for stemming in Romanian was only introduced in March 2007.

Snowball has support for both UTF-8 as well as other less common encodings. The stemming rule files are compiled into C source code which can then be statically compiled to produce a library that is linked into the client code. The following code section in Listing 3.2 demonstrates the main steps of creating and using a stemmer compiled from Snowball.

```
1  /* The Snowball library interface. */
2  #include "libstemmer.h"
3
4  /* Example of creating a stemmer for English (UTF-8 encoding). */
5  struct sb_stemmer* stemmer = sb_stemmer_new("english", NULL);
6
7  /* Example of stemming a lexem of type std::string. */
8  const sb_symbol* stemmed = sb_stemmer_stem(stemmer,
9      (const sb_symbol*) lexem.c_str(), lexem.length());
10 if (stemmed != NULL)
11   lexem = std::string((const char*)stemmed);
12
13 /* Unused stemmers waste a large amount of memory. */
14 sb_stemmer_delete(stemmer);
```

Listing 3.2: Example of lexem stemming in C++ using C Snowball stemmers

3.3 Corpus Statistics and Analysis

In order to be able to take advantage of the traits of specialized corpora similar to that which AuthentiCop is designed for (academic papers in the field of Computer Science), it is important to give an overview of the parameters of such a corpus with all preprocessing steps enabled.

For the corpus of 857 papers described in Section 3.1.3, the results were summed up in Table 3.3

The numbers are in stark contract to their counterparts for generalized corpora. After running part 1 of the source documents for the externat plagiarism detection task in the training corpus of the PAN 2011 competition through the preprocessing stage and aggregating the data, I discovered that the language model contains roughly 132538 lexemes, almost 3.15 times more than in the case of the specialized CS corpus.

When analyzed in closer detail, the two language models proved to be surprisingly different. Of the 42127 lexemes in the CS paper language model, a staggering 28890 (almost 68.57%) were

[1] http://snowball.tartarus.org/

Table 3.3: Traits of a Corpus Composed of 857 CS Academic Papers in Romaian and English.

	English	Romanian
Number of Documents	289	567
Lexemes (total)	2830507	3670260
Unique Lexemes (no pre-processing)	92267	149935
Unique Lexemes (normalization and stemming)	42197	62927
Contraction	54.266%	58.030%
Hapax Legomena (no pre-processing)	40898	68965
Hapax Legomena (normalization and stemming)	18281	30339
Contraction	55.300%	56.008%

absent from the general English PAN-derived language model. The numbers may first lead one to believe that there might be a mistake in the measurements, but the explanation lies in the fact that the frequencies of these terms only add up to 6.26284% of the terms in the CS corpus. Similarly, the added frequencies for all of the 119231 (almost 89.95%) lexemes which were present in the PAN corpus and absent from the CS corpus only make up for 7.15403% of the total words in that corpus.

After a closer inspection of the data, it was revealed that a large portion of the lexemes found only in the CS papers were attributable to one of the following categories:

- technical vocabulary which is normaly not encountered in daily speech (eg. *ethernet*, *broadband*, *algorithm*, etc.);
- lexemes derived by the PDF parser from code or pseudo-code samples (eg. variable names, function names, various notations and abbreviations, etc.);
- typographical errors.

Similarly, the lexemes found in the literature corpus alone are either rare words or words that are unlikely to be found in academic language in the field of Computer Science (eg. *coronation*, *corpulence*, *ex-mayor*, etc.) or typographical errors.

To compute the overall distance between the two language models, I used the *cosine metric*. Given two normalized language models:

$$LM_{CS} = \{< w, freq_w > | w \in Words(CS), freq_w \in [0,1]\}$$
$$LM_{Gen} = \{< w, freq_w > | w \in Words(General), freq_w \in [0,1]\}$$

we define the *distance* between the two language models to be the angle between vectors LM_{CS} and LM_{Gen} in the space $(Words(CS) \bigcup Words(General)) \times [0,1]$. Specifically,

$$distance = \arccos(\frac{LM_{CS} \cdot LM_{Gen}}{|LM_{CS}||LM_{Gen}|})$$

Since LM_{CS} and LM_{Gen} are normalized, they both have the modulus equal to 1. Furthermore, by expanding the dot product, we get:

$$distance = \arccos(\Sigma_{w \in Words(CS) \bigcup Words(General)}(freq_{w,Words(CS)} * freq_{w,Words(General)}))$$

where $freq_{w,LM}$ is defined to be 0.00 iff $w \notin Words(LM)$.

For the two corpora studied, I got:

$$distance = \arccos(0.949566) = 0.3189474157rad = 18.274340\circ$$

Which proves that the two language models are in fact, different in their makeup. This conclusion has broad implications for the plagiarism detection algorithms, indicating that the candidate selection stage of the pipeline has good potential for identifying possible plagiarism sources from a large database written in general language (as is, for instance, Wikipedia or the broader World Wide Web).

Chapter 4

Profiling and Topic Detection

4.1 Profiling

4.1.1 Overview

As previously stated in Section 2.1.2, the AuthentiCop system maintains a databse of text documents from which candidates are drawn for comparison to the suspicious documents. In order to speed up computation and avoid redundancy, each document is saved in the database in two forms:

- a copy of the original file, for reference;
- a document profile containing information derived from the original file.

Saving the document profile alongside the original document serves two different optimization functions.

On one hand, conversion and preprocessing, which bear a significant I/O load, are only done on the documents once, upon inserting them into the database. In this regard, profiling acts as a cache system for the document database.

However, most importantly, profiling the document can bring a much more significant optimization by eliminating the need to deal with character strings in the algorithms. Keeping document profiles that would contain character strings is costly because the lexems would have to be parsed every time the profile were brought up for inspection. Comparisons between character strings are far more expensive than comparisons between 32-bit integers, and they also take up more space in the working memory. Since profiles are meant to store ngrams from the document, the memory waste would be further amplified by a factor equal to the size of the ngrams.

To demonstrate the impact and importance of the profiling stage in the pipeline, the following sections discuss two file formats for document profiles which I have created and their use.

4.1.2 The text ngrams file format

The algorithms based on Dotplot and Encoplot perform computations based on the ngram representation of the documents. For each input document, the sorted list of ngrams is required. For each of the ngrams, some positional information is also necessary: the number of the first

lexem in the document where the ngram starts, the byte offset in the preprocessed document for it, and also the length of the ngram in Bytes:

$$record =< ngram, lexem\_offset, byte\_offset, byte\_size >$$

where $lexem\_offset, byte\_offset, byte\_size \in [0, 2^{32} - 1]$ and $ngram \in LM^n$, with n being the size of the ngrams considered (commonly, 3, 4 or 5).

The former three variables in the tuple are naturally expressed as integers. However, it would also be optimal to store the acutal ngrams in a more compact form than simply listing their lexems in plain text. We use a global dictionary computed across the entire database to assign a unique number to each different lexem, thus a profile becomes a string of records, with each $record \in [0, 2^{32} - 1]^{n+3}$.

The text profile format also specifies a header with meta-information useful for debugging purposes (such as the number of tuples, the time and date of creation, the original file name and a short description of the tuples listed). A sample containing the first three records of a text **ngram** profile is given for reference in Listing 4.1. Note that lines starting with a hash symbol are considered comments and ignored.

```
1 # 3322
2 # 3
3 # Thu May 24 00:53:12 2012
4 # Original file: part1-source/source-document00330.txt
5 # Lexem[0] Lexem[1] Lexem[2] LexemOffset ByteOffset ByteSize
6 1 94 8580 1472 8434 16
7 1 437 1264 1864 10636 17
8 1 812 94 1363 7808 15
```

Listing 4.1: Sample text ngram file

An implementation detail worth mentioning concerns the relationship between the lexem IDs and their relative collation order in the dictionary. Even though sorting through the list of ngrams would be faster if the lexem IDs were assigned in ascending collation order, we cannot benefit from that approach because the size of the dictionary is liable to growth as more documents with potentially never before seen lexems are uploaded into the database. Therefore, the lexem ID to dictionary mappings still needs to be loaded in memory for sorting. The dictionary is kept in a **dict** file, the syntax of which is demonstrated through the sample lines in Listing 4.2.

None the less, two different ngrams can still be compared for equality based solely on the IDs of their lexems.

```
1  # 339330
2  # Mon May  7 19:49:09 2012
3  # Lexem LexemID
4  # (collation order is indicated by the line number)
5  ...
6  zy 64672
7  zyg 225502
8  zygomatic 320239
9  zygon 225378
10 ...
```

Listing 4.2: Sample text ngram file

4.1.3 The binary ngrams file format

When dealing with a corpus of several Gigabytes in size, storing profiles in text files becomes cumbersome because of the need to parse these files each time they are loaded into memory. The time spent parsing is significant especially in the candidate selection stage because the algorithm needs to look at $|Suspicious| + |Source|$ profiles. For the PAN 2011 corpus, this means 22286 documents. Since the profiles in themselves, despite being smaller than the original documents, are still comparable in size, it would be impossible to keep all of them into memory. Each document will have to be loaded into memory multiple times (as argued in the following chapter, this number would optimally be $K = \sqrt{\max(|Suspicious|, |Source|)}$), causing unnecessary and wasteful parsing.

The simpler solution is to store the profiles in binary format. Since we have successfully eliminated character strings from the profile representation, we can also safely predict the size and alignment of the tuples in the file. The syntax no longer allows for comments, as the files are never meant to be opened for inspection in a text editor. Instead, the file syntax follows the description below.

```
1 unsigned 32bit integer: Count = number of tuples in file
2 unsigned 32bit integer: Size = size of the ngrams
3 (unsigned 32bit integer) * Size: Tuples = the array of tuples
```

Listing 4.3: Syntax of a binary ngram file

This way, loading the files into memory becomes as easy as resizing an array and copying the file contents to that array. A seemingly faster solution would be to map the contents of the file to the virtual memory of the process and let the operating system handle the caching, but as we are certain that we will be iterating through the entire profile a number of times on the order of K, it turns out that runtime is not visibly improved.

There is one more main concern regarding the storage of profiles in this case, which is endianness. As we are storing tuples of unsigned 32 bit integers, extra care must be taken not to migrate databases containing such profiles to machines which use a different endianness. I chose not to use a specific endianness, as is commonly done in networking applications, because the need to convert would have defeated the purpose of fast and seamless loading into memory from disk and because of the assumption that the profiles will be generated on the same machine which uses them.

Besides faster load times for the files, using binary profiles also improves disk space usage. A better overview of this information is given in Table 4.1.

Table 4.1: Comparison of disk space for text and binary profile formats.

	Text Profile Format	Binary Profile Format	Binary vs. Text Improvement
Maximum Size	13,060,948 B	10,430,116 B	20.142%
Average Size	896,434 B	753,705 B	11.872%
Minimum Size	14,658 B	13,920 B	5.034%
Total Size	**448,217,470 B**	**376,852,544 B**	**15.921%**

The data in the table is derived from the profiles of a sample of 500 documents from the PAN corpus. Aside from the time complexity gains, there is an almost 16% improvement in total storage size when using binary profiles. The table also proves that using binary profile formats brings better compression for longer documents than for shorter ones.

4.2 Topic Detection Using Wikipedia

4.2.1 Overview

Wikipedia is a freely accessible web-based encyclopedia and knowledge base which is written by a community of anonymous volunteers from all over the world. New articles are continuously added and existing articles are improved or expanded every day, and since it was first launched in 2001 it has grown to become one the largest reference websites on the Internet. As of the time of writing this thesis, there were more than 85,000 active contributors and the article database spanned 280 languages amassing a total of over 21,000,000 articles[1].

As a resource for natural language processing research, Wikipedia proves to be invaluable as it provides with a large, good quality corpus for drawing statistics. In the summer of 2012, the database contained approximately 4,000,000 articles written in English alone, totalling an XML dump with only the most recent revision for each article of close to 34 Gigabytes. This volume of data allows us to build an accurate language model, to draw information on semantic analysis and, for the case of AuthentiCop, also to search for further possible plagiarism sources in the candidate selection stage.

4.2.2 Wikipedia Export

The contents of Wikipedia are published under a Creative Commons License, which grants any user the liberty to share, adapt, and possibly republish them under a compatible license. Wikipedia has a dedicated page for giving support on handling the export and import of the contents of its database.

It is currently possible to download database dumps for the articles in each of languages that the encyclopedia is featured in. These dumps are accessible from Wikimedia and are generally updated on a weekly basis. For the lastest version of a dump in a given language, one can simply make an HTTP request for a file under the location:

http://dumps.wikimedia.org/*XX*wiki/latest/

where ***XX*** should be replaced with a two-letter language code (i.e. ***en***, ***ro***, etc.).

The dumps available for download come in a range of formats, with the most trafficked ones being XML and SQL. Because the files are plain text and tend to be fairly large, they are distributed in compressed form and there are is a choice of archive formats which includes *bzip2* (the free implementation of the Burrows-Wheeler compression algorithm) and *gz* (GNU zip).

The export tools offer a wide array of options for selecting which information should be included. The complete Wikipedia export will include at least the following information on each of the articles:

- The article proper, including:
  - page title;
  - metainformation about timestamps and contributors;
  - information about the latest revisions of the article;
  - the contents of the article.
- Comments on the article;
- The history of resolving edit conflicts.

[1] According to http://en.wikipedia.org/wiki/Wikipedia:About

This information can inflate the size of the dump despite the fact that it isn't valuable from the point of view of corpus analysis. For the purpose of retrieving a corpus for the AuthentiCop system, I had to generate a special export in XML format which only contained information about the page titles and the most recent revision of the article content.

Furthermore, we can reduce the size of the download if we use the specialized export functionality service. The service allows the user to fill in the titles of articles he or she may want included in the dump. Since within the AuthentiCop system we are focusing on specialized corpora, it is optimal to download only the fraction of the Wikipedia database concerning articles from the specialized field in question, rather than downloading the entire dump and then filtering it. The problem boils down to being able to enumerate all the titles of articles from the field of Computer Science. In order to solve this problem, a closer inspection of Wikipedia Categories must be taken.

4.2.3 Wikipedia Taxonomy

Wikipedia implements a *category system* that is generated automatically from category tags situated at the bottom of wikipedia pages. The purpose of the category system is to provide a hierarchical classification of all wikipedia pages based on their topic. The root category for the entire content hosted on Wikipedia is the *Contents* category. There are three main category systems being used in parallel:

- The *Fundamental Category* system, which acts as an ontology and has four main subcategories that organize all knowledge in the encyclopedia:
  - Concepts;
  - Life;
  - Matter;
  - Society.
- The *Wikipedia Categories* system, which acts as a meta category system for wikipedia pages, with subcategories such as *Set categories* or *Tracking categories*.
- The *Categories Index* system, which is a manually edited category system with a far greater branch factor and smaller depth in the tree structure and which provides an index of all categories classified by their topic.

Wikipedia categories are specified in the export tool by the string **Category:** followed by the category name. For extracting pages relating to topics in Computer Science, I used the Categories Index classification and considered all the subtrees of the **Category:Computer_science** category.

Wikipedia does not currently expose a tool or an API to recursively list all articles under a certain category. To get a listing of all the pages under the Computer Science category, I used a crawler script available on **toolserver.org**[1].

The results of running the script with increasing values for the depth parameter between 2 and 7 are summarized in Table 4.2. The number of article titles crawled increases exponentially with a branch factor averaging 2.177, which decreases as the depth progresses. However, it is important to note that the crawling does not add relevant articles for depths greater than 4 because of the presence of many categories which list names of organizations, journals, conferences or professionals and which make for poor candidate documents in a plagiarism detection system and bear little weight when computing language model parameters. As depth progresses, the articles naturally become more specific, smaller in size and less informative in content.

[1] http://toolserver.org/ dschwen/intersection/index.php

Table 4.2: Number of articles crawled under the Computer Science category by depth.

Depth	Number of Articles	Query Time (in seconds)
1	1,064	0.00
2	6,280	0.02
3	17,831	1.00
4	**39,745**	**3.00**
5	86,215	60.00
6	154,863	73.00
7	287,125	197.00

For building a reference database of Wikipedia articles, I used a depth equal to 4 and I purged the final list of articles to remove subcategories of list categories comprising of named entities, leaving a dump of 8,729 articles.

4.2.4 Wikipedia Dump Processing

The most common plain text formats for exporting articles from Wikipedia are SQL and XML. SQL is used mainly for migrating Wikipedia and is not of particular interest for the purpose of building a corpus. The XML dump, on the other side, can be easily parsed to extract the contents of the articles. As the XML dump can be fairly large, a SAX parser is needed. Such a parser is implemented in a straightforward way based on the DTD of the file, which is given for reference in Appendix A.1.

After XML parsing, the output text obtained is WikiMedia source code, which is plain text with metadata and adnotations. To give an actual example, the first paragraph in the article for Computer Science in its original form is rendered as in Example 4.4, whereas what we really need is a cleaned-up version with all the markup stripped away from the plain text contents of the files and the HTML entities replaced accordingly.

```
1  <text xml:space="preserve" bytes="9591">The following outline is
       provided as an
2  overview of and topical guide to computer science:
3
4  '''[[Computer science]]''' (also called '''computing science''') &
       amp;ndash;
5  study of the theoretical foundations of [[information]] and
6  [[computation]] and their implementation and application in
7  [[computer system]]s. One well known
8  subject classification system for [[computer science]] is the [[ACM
       Computing
9  Classification System]] devised by the [[Association for Computing
       Machinery]].
10 The [[ACM computer science body of knowledge]] is a recommended
       curriculum for a
11 university-level computer science course. [...] </text>
```

Listing 4.4: Dump with WikiMedia Markup

I achieved this final parsing step with the help of two tools:

- **wp2txt** - an online, open source project written in Ruby by Yoichiro Hasebe from the Doshisha University of Japan[4], which interprets the XML and strips most WikiMedia markups from the plain text sections;

- **split2articles** - a simple parser written by me which parses the output of **wp2txt** and interprets the remaining WikiMedia markups, moving the contents of each article to a separate file with a name the same as the name of the article.

wp2txt is currently pusblished on GitHub[5] under an MIT license and is written using *rubygems*. However, I decided to fork an older version of the wp2txt project from 2009 because I found it easier to tweak the source code for working with locally archived XML dumps. Since the output plain text articles would already be stored separately on disc, I wanted to reduce the amount of wasted storage while building the Wikipedia-based corpus.

4.2.5 Topic Detection

The final task of topic detection is compiling a list of keywords which could further be used to formulate queries to a search engine in hope of expanding the list of candidate documents. For speeding up the task, I decided to use a keyword extraction algorithm that does not make round trips to the database for extra information on the language model of the corpus. Research on Keyword Extraction from a single document has previously been carried out by Y. Matsuo and M. Ishizuka[21]. Their methods rely on statistical information about word co-occurence within the document.

The input to the algorithm is a stemmed plain text document.

- First, the list of the most frequent 30% unigrams and bigrams is computed;
- Secondly, the list of frequent terms is then clustered based on the **mutual information** formula:

$$M(w_1, w_2) = \log \frac{P(w_1, w_2)}{P(w_1)P(w_2)}$$

  Two terms are clustered together if their mutual information is above log(2.0).

- For each cluster, we define the **expectance probability** as the probability of a selected term to belong to that particular given cluster:

$$p_c = n_{terms\_in\_cluster}/N_{total\_terms}$$

- The final step is computing a ranking measure called the $\chi^{'2}$ value, according to the formulae proposed by Matsuo, and retaining only the top 10 entries.
- The list of candidates is also enriched with the name of the article and the top-ranking 3 named entities in the first 10% of the text.

The list of keywords is then stored and during the candidate selection stage of the algorithm, it is compared to the list of keywords extracted from the suspicious document. For each pair $(kw_{suspicious}, kw_{wikipedia\_article})$, we compute the mututal information metric in the suspicious document and for each of the wikipedia keywords $kw_{wikipedia\_article}$, we compute $best\_match = \max(M(k', kw_{wikipedia\_article}))$ and retain retain only the highest-ranking 10% keywords.

The final list of keywords encodes the topic of the document and can be used to build queries for retrieving other possible source candidates for plagiarism.

Chapter 5

Candidate Selection

5.1 Problem Formulation and Test Data

As described in Section 2.1.2, the **candidate selection** stage has the purpose of reducing the search space for the more computationally intensive **detailed analysis** processing stage in the pipeline. Given a suspicious document $d_{suspicious}$ and a set of possible source documents, D_{source}, the candidate selection stage aims at detecting a subset $Cand_{source} \subseteq D_{source}$ such that:

- all documents that $d_{suspicious}$ is plagiarised after are included in $Cand_{source}$;
- as few documents as possible that $d_{suspicious}$ is not plagiarised after are included in $Cand_{source}$.

Let PS_{source} be the set of documents that $d_{suspicious}$ was plagiarised after. We can formulate the precision and recall of a candidate selection algorithm based on the following adaptions of the formulae:

- precision aims to quantify the number of false positives:

$$prec = \frac{|PS_{source} \cap Cand_{source}|}{|PS_{source}|}$$

- recall aims to quantify the number of plagiarism sources which were actually picked up by the selection:

$$recall = \frac{|PS_{source} \cap Cand_{source}|}{|Cand_{source}|}$$

I used the PAN 2011 corpus and the associated Golden Standard for evaluating the performances of the candidate selection algorithm implementations. One important note here is that in the case of candidate selection, it is considered an acceptable tradeoff to increase recall at the cost of precision. This is the equaivalent of making sure to include all plausible sources for plagiarism such that the detailed analysis stage will pick them up at the cost of putting together a larger workload on the system.

5.2 The Encoplot Algorithm

5.2.1 Description

The Encoplot[10] algorithm was designed by Cristian Grozea and Marius Popescu and was successfully used in the PAN 2009 competition where it had the best performance of all other algorithms in the plagiarism detection category [8] and in the PAN 2011 competition, where it earned the second place.

The Encoplot algorithm is inspired by the better known Dotplot method. Given two strings of symbols, A and B of length l_A and l_B, respectively, a Dotplot among twe two documents is a matrix $M_{l_A x l_B}$ of black and white pixels. We set by convention to assign the pixel at (i, j) the color *black* whenever $A[i]$ matches $B[j]$ and the color *white* otherwise.

To use a real example, consider the two short paragraphs below:

```
1 Intellectual honesty is the admission that humanity is linked
2 together in a kind of collective learning process. Very little
3 is discovered "de␣novo," that is, without a solid foundation in
4 other researchers' previous exploration and understanding.
5 Citation is an act of humility and an act of appreciation for
6 what other scholars have pieced together about the nature of a
7 particular problem or an aspect of some phenomenon.
```

Listing 5.1: Sample source text

```
1 Intellectual honesty means that humanity is linked together in
2 a kind of joint learning process. Not very much is discovered
3 new without really understanding other scholars' previous research
4 and knowledge. Citing shows you are grateful and appreciate what
5 other researchers have figured out about a particular issue.
```

Listing 5.2: Slightly obuscated example of the same text

The second example is a slightly ofuscated version of the text in the first paragraph. If we run all the preprocessing steps on the two paragraphs, stripping punctuation and stemming the words, we end up with two strings of lexems. The dotplot of the two strings is given in Figure 5.1.

Figure 5.1: Dotplot graph of the original paragraph (vertical axis) versus the obfuscated paragraph (horizontal axis)

The pairs of matching lexems are then clusterized and the resulting clusters are verified for density and linearity. The decision of candidate selection is made based on the number of resulting clusters and their size.

However, the Dotplot method has certain disadvantages which make it undesirable in a large-scale system. Given the nature of the plotting, the algorithm has quadratic runtime and memory usage. Considering that an average document has on the order of $5,000$ words, the whole process easily takes up to $1.00sec$ runtime per document pair, which would mean a runtime on the order of days for the whole PAN 2011 corpus alone.

After a closer inspection of Figure 5.1, two features stand out:

- most matches in plagiarised passages occur in an orderly fashion along a line;
- most of the matches outside the so-called lines are irrelevant.

The idea behind the Encoplot algorithm follows naturally from these two observations. Instead of using all of the lexem matchings in the document, the strings of lexems in each document are sorted using a stable sort algorithm and then a merge is performed to only retain lexem matches in increasing order of their position in the original text. Thus, the plot generated by the Encoplot method will never match a lexem twice and will never contain matches outside of those present in a Dotplot matrix. This reduces the space complexity from quadratic to linear while preserving the important features indicative of plagiarism: linear patterns of dots on the plot.

It should be noted that in the Encoplot method, the dots are not lexems, but rather ngrams. I have experimented with various ngram sizes and stopword filtering parameters. Filtering out stopwords has the beneficial effect of lowering the noise on the plot and allowing the ngrams to be shorter. However, sometimes filtering out the stopwords interferes with the clustering process because the apparent density within a cluster is decreased, and as such may hinder detection. In my implementation, I relied on keeping stopwords and using a 4-gram model instead.

5.2.2 Pseudocode and Implementation Details

The main steps of the algorithm are:

Building and sorting the ngram arrays. This is done in the profiling stage and the profiles are simply loaded from the disc and interpreted according to the dictionary file they were compiled with.

Merging the ngram arrays and keeping only matching elements. This step happens for each pair of documents in the corpus, yielding a quadratic factor which multiplies the next step. Since the number of documents to be merged can be fairly large, they cannot be all kept in memory. Furthermore, disc I/O operations are very time consuming, so we need to devise a way by which to minimise the number of times a certain profile gets loaded into memory from disc.

We achieve this with a caching strategy in which we load blocks of size $\sqrt{N}$ into memory and process all possible pairs before loading the next block. This reduces the number of times any file is loaded into memory from $O(N)$ to $O(\sqrt{N})$, bringing a significant speedup in overall execution.

Besides caching, I also implemented a heuristic that speeds up the merge routine. This follows the natural observation that given any two documents forming a candidate pair, each of the documents in the pair will have a large number of ngrams that are not shared by the other document. However, in the traditional merge algorithm, all of these ngrams would be inspected and then discarded, one by one, in linear time. The heuristic I implemented speeds up the

merge by trying to discard a number of ngrams that grows in powers of 2 whenever a mismatch is detected between the two ngram arrays.

The pseudocode to the sublinear merge algorithms is given in Listing ??.

```
1  while (length(array1) > 0 and length(array2) > 0) {
2    h1 <- head(array1)
3    h2 <- head(array2)
4    if (h1 == h2) {
5      result <- result + (h1, h2);
6      drop(array1, 1);
7      drop(array2, 1);
8    } else if (h1 < h2) {
9      k <- max{array1[2^k] < h2}
10     drop(array1, 2^k);
11   } else {
12     k <- max{array2[2^k] < h1}
13     drop(array2, 2^k);
14   }
15 }
```

Listing 5.3: Pseudocode of the sublinear merge algorithm

Clustering the dots left over by the merge process. All of the pairs output by the merge process are projected on the axis of the source document. The reason for projecting on the component of the source document is that plagiarised passages are more likely to get matching ngrams which are consecutive in the source document due to the obfuscation through paraphrasing in the suspicious document.

A cluster of dots is defined to be a subsequence in the merged ngram string that has a density (ratio between total length of matching ngrams and total text span) above 40% and a total text span of at least 512 characters for the source document. The corresponding ngrams in the suspicious document are first trimmed of outlier points and then evaluated to have a density of above 30%. The difference in densities is explained by the amount of obfuscation which is expected in a plagiarised text.

The authors recommend doing this in a Monte Carlo optimization loop by selecting a seed dot pair and trying to grow a segment around it before the density falls below a threshold. If the size of the resulting cluster does not exceed the acceptance threshold, then the cluster is discarded and another attempt is made. A number of 30 iterations would be allowed before declaring that no cluster large enough exists in order to declare the document pair a candidate pair for detailed analysis.

However, I discovered that for the PAN corpus, there is a gain in accuracy and a time penalty of only around 22% when trying to select the first available dot match pair for seed and then discarding all pairs covered by growing the seed so as to avoid redundancy.

Despite the optimizations, the runtime for the candidate selection stage is still significant, at around 40 minutes for a batch of 500 source documents and 500 suspicious documents on a dual core AMD Turion 64bit processor with 1 GB of RAM memory running Linux.

Chapter 6

Conclusion and Future Development

6.1 Conclusion

The project described the necessary architecture and dataflow of an automatic plagiarism detection system for academic writing in the field of Computer Science, investigated the traits and language models of spcialized corpora in the field, and identified means of building and using such corpora for the purpose of building the AuthentiCop system.

Solutions have been described, built, run and evaluated for all leading steps in the processing pipeline (format conversion, preprocessing, profiling and keywords extraction for topic detection), and the Encoplot algorithm for candidate selection has been investigated and tuned.

The goals of the project has been successfully achieved. The differences between the general language model and the corpus-specific language model have been analyzed, the main steps of the processing pipeline have been described and their implementation has been documented.

In the end, I found the analysis of the AuthentiCop system to be especially challenging and rewarding due to the algorithmic nature of the problems, the sheer size of the data and the opportunity to study new concepts.

6.2 Future Development

The most important direction for future development is parallelizing the implementations so as to make the solution more scalable. One of the most limiting factors during development and testing was the runtime on the order of hours or days between successive development iterations due to limited hardware resources and the large size of the data corpus.

Next, integration should be carried out with a search engine API to facilitate online document retrieval as part of the candidate selection stage. At the same time, the system could be made language-independent between Romanian and English by using an automatic translation engine as part of the preprocessing pipeline.

Thirdly, alternative algorithms that also account for semantic information rather than just the lexical makeup of the documents should be taken into consideration for both the candidate selection stage and the detailed analysis one.

Appendix A

Appendix

A.1 Wikipedia Dump - Document Type Defition

```
<!ELEMENT mediawiki (siteinfo?,page*)>
<!-- version contains the version number of the format (currently
    0.3) -->
<!ATTLIST mediawiki
  version  CDATA  #REQUIRED
  xmlns CDATA #FIXED "http://www.mediawiki.org/xml/export-0.3/"
  xmlns:xsi CDATA #FIXED "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation CDATA #FIXED
    "http://www.mediawiki.org/xml/export-0.3/ http://www.mediawiki.
       org/xml/export-0.3.xsd"
>
<!ELEMENT siteinfo (sitename,base,generator,case,namespaces)>
<!ELEMENT sitename (#PCDATA)>      <!-- name of the wiki -->
<!ELEMENT base (#PCDATA)>          <!-- url of the main page -->
<!ELEMENT generator (#PCDATA)>     <!-- MediaWiki version string -->
<!ELEMENT case (#PCDATA)>          <!-- how cases in page names are
    handled -->
   <!-- possible values: 'first-letter' | 'case-sensitive'
                         'case-insensitive' option is reserved for
                            future -->
<!ELEMENT namespaces (namespace+)> <!-- list of namespaces and
    prefixes -->
  <!ELEMENT namespace (#PCDATA)>     <!-- contains namespace prefix
      -->
  <!ATTLIST namespace key CDATA #REQUIRED> <!-- internal namespace
      number -->
<!ELEMENT page (title,id?,restrictions?,(revision|upload)*)>
  <!ELEMENT title (#PCDATA)>         <!-- Title with namespace
      prefix -->
  <!ELEMENT id (#PCDATA)>
  <!ELEMENT restrictions (#PCDATA)>  <!-- optional page restrictions
      -->
<!ELEMENT revision (id?,timestamp,contributor,minor?,comment,text)>
  <!ELEMENT timestamp (#PCDATA)>     <!-- according to ISO8601 -->
  <!ELEMENT minor EMPTY>             <!-- minor flag -->
```

```
27    <!ELEMENT comment (#PCDATA)>
28    <!ELEMENT text (#PCDATA)>          <!-- Wikisyntax -->
29    <!ATTLIST text xml:space CDATA  #FIXED "preserve">
30  <!ELEMENT contributor ((username,id) | ip)>
31    <!ELEMENT username (#PCDATA)>
32    <!ELEMENT ip (#PCDATA)>
33  <!ELEMENT upload (timestamp,contributor,comment?,filename,src,size)>
34    <!ELEMENT filename (#PCDATA)>
35    <!ELEMENT src (#PCDATA)>
36    <!ELEMENT size (#PCDATA)>
```

Listing A.1: The Document Type Definition of Wikipedia XML Exports

Bibliography

[1] http://www.webis.de/research/events/pan-11, Last accessed on July 1st, 2012.

[2] http://podofo.sourceforge.net/about.html, Last accessed on July 1st, 2012.

[3] http://tika.apache.org/, Last accessed on July 1st, 2012.

[4] http://wp2txt.rubyforge.org/, Last accessed on July 1st, 2012.

[5] https://github.com/yohasebe/wp2txt/, Last accessed on July 1st, 2012.

[6] Department of political science policy on academic misconduct, ubc. http://www.politics.ubc.ca/undergraduate/program-information/plagiarism-and-turnitin.html, Last accessed on July 1st, 2012.

[7] SVOP Ltd. Bratislava. http://old.svop.sk/antiplag_eng.html, Last accessed on July 1st, 2012.

[8] Marius Popescu Cristian Grozea, Christian Gehl. Encoplot: Pairwise sequence matching in linear time applied to plagiarism detection. 3rd PAN Workshop. Uncovering Plagiarism, Authorship and Social Software Misuse. p. 10, 2009.

[9] Sebastian A. Rios Gabriel Oberreuter, Gaston L'Huillier. Fast docode: Finding approximated segments of n-grams for document copy detection. Lab Report for PAN at CLEF 2010, 2010.

[10] Cristian Grozea and Marius Popescu. The encoplot similarity measure for automatic detection of plagiarism. Notebook for PAN at CLEF 2011, 2011.

[11] Jonathan Hefman. Dotplot patterns: A literal look at pattern languages. Theory and Practice of Object Systems (TAPOS'95), pp. 3141, 1995.

[12] Susan Hunston. Word frequencies in written and spoken english: Based on the british national corpus. Language Awareness, 11, 2, 152-157, 2002.

[13] LLC iParadigms. Turnitin partners with education council to develop common core grading rubrics. http://pages.turnitin.com/rs/iparadigms/images/Turnitin_RELEASE_062512_EPLC.pdf, Last accessed on July 1st, 2012.

[14] LLC iParadigms. White paper: The plagiarism spectrum. http://pages.turnitin.com/rs/iparadigms/images/Turnitin_WhitePaper_PlagiarismSpectrum.pdf, Last accessed on July 1st, 2012.

[15] F. J. Smith Le Quan Ha E. I. Sicilia-Garcia Ji Ming. Extension of zipf's law to word and character n-grams for english and chinese. Computational Linguistics and Chinese Language Processing, 2003.

[16] C. D. Paice. Another stemmer. SIGIR Forum; 24, 56-61, 1990.

[17] Martin F. Porter. An algorithm for suffix stripping. Program, 14(3) pp 130−137, 1980.

[18] Martin Potthast Benno Stein Alberto Baron-Cedeno Paolo Rosso. An evaluation framework for plagiarism detection. Proceedings of the 23rd International Conference on Computational Linguistics, COLING 2010, 2010.

[19] Butler University Sally Neal. Turnitin.com: An exploration of a plagiarism detection tool. Presentation at the IOLUG 2004 Spring Program, 2004.

[20] Gerein Marla Satterwhite Robin. Downloading detectives: Searching for on-line plagiarism. http://www2.coloradocollege.edu/library/Course/downloading_detectives_paper.htm, Last accessed on July 1st, 2012.

[21] M. Ishizuka Y. Matsuo. Keyword extraction from a single document using word co-occurrence statistical information. International Journal on Artificial Intelligence Tools, 2003.