Masaryk University
Faculty of Informatics

Removing Boilerplate and Duplicate Content from Web Corpora

Ph.D. thesis

Jan Pomikálek

Brno, 2011

Acknowledgments

I want to thank my supervisor Karel Pala for all his support and encouragement in the past years.

I thank Luděk Matyska for a useful pre-review and also for being so kind and correcting typos in the text while reading it.

I thank the KrdWrd team for providing their Canola corpus for my research and namely to Egon Stemle for his help and technical support.

I thank Christian Kohlschütter for making the L3S-GN1 dataset publicly available and for an interesting e-mail discussion.

Special thanks to my NLPlab colleagues for creating a great research environment. In particular, I want to thank Pavel Rychlý for inspiring discussions and for an interesting joint research.

Special thanks to Adam Kilgarriff and Diana McCarthy for reading various parts of my thesis and for their valuable comments.

My very special thanks go to Lexical Computing Ltd and namely again to the director Adam Kilgarriff for fully supporting my research and making it applicable in practice.

Last but not least I thank my family for all their love and support throughout my studies.

Abstract

In the recent years, the Web has become a popular source of textual data for linguistic research. The Web provides an extremely large volume of texts in many languages. However, a number of problems have to be resolved in order to create collections (text corpora) which are appropriate for application in natural language processing. In this work, two related problems are addressed: cleaning a boilerplate and removing duplicate and near-duplicate content from Web data.

On most Web pages, the main content is accompanied by so-called boilerplate content, such as navigation links, advertisements, headers and footers. Including the boilerplate in text corpora results in an undesirable over-representation of some common boilerplate words and phrases, such as *home*, *search*, *print*, etc. This work provides an overview of commonly used boilerplate cleaning techniques and presents a novel heuristic based approach to this problem. The method is compared with other state-of-art algorithms on available data sets.

Many texts on the Web exist in multiple instances (e.g. mirrored websites, document revisions, quotations in discussion forums, etc). Duplicate data distorts corpus statistics and causes difficulties to users who search for linguistic data in Web corpora. Many researchers have addressed the problem of identifying duplicate and near-duplicate Web pages in the context of Web search engines. However, the problem is typically limited to identifying almost identical documents. Pairs of Web pages which contain significant amounts of both identical and different content are not considered near-duplicates by search engines. For text corpora, on the other hand, any duplicated data constitutes a problem. Since identifying an intermediate level of duplication in large text collections has not yet received much attention, this work aims to fill this gap.

Contents

Chapter 1

Introduction

1.1 Motivation

Text corpora—large collections of texts—have many applications in computational linguistics. Corpus data can be processed statistically by computer programs, e.g. to create language models in fields such as speech recognition or machine translation. Corpus data is also often used directly by humans with a specialised software called a corpus manager. Typical examples of the latter include finding corpus evidence for dictionary entries by lexicographers or examining the behaviour of words in contexts by language learners.

Distribution of words in a natural language follows the Zipf law [64]: In a list of words sorted by a frequency of occurrence in a descending order, the frequency of a word multiplied by its rank is roughly the same for all words on the list (the most frequent word is two times as frequent as the second most frequent word, three times as frequent as the third most frequent word, etc). Put differently, there are few words which are used frequently and many words which are used rarely. The same applies for other language phenomena, such as multi-word expressions, phrases or patterns.

The distribution of words and expressions in a natural language imposes one essential requirement on text corpora – a large size. Most language phenomena occur infrequently and a lot of textual data is needed for finding evidence about them. A huge amount of texts may be necessary for finding occurrences of some extremely rare expressions, such as *unbridled hedonism.* With respect to text corpora it is therefore often said that "more data is better data". Table 1.1 shows frequencies of selected words and multi-words expressions in corpora of different sizes.

In early days, text corpora have been created from printed resources, such as

corpus name	BNC	ukWaC	ClueWeb09 (7 %)
size [tokens]	112,181,015	1,565,274,190	8,390,930,801
nascent	95	1,344	12,283
effort	7,576	106,262	805,142
unbridled	75	814	7,228
hedonism	63	594	4,061
nascent effort	0	1	22
unbridled hedonism	0	2	14

Table 1.1: Frequencies of words and multi-word expressions in corpora of various sizes.

books and magazines. Digitalization of the texts was a tedious and time-consuming process which involved a lot of manual labor. The size of the corpora created in this way (e.g. the Brown Corpus [43]) was up to a few millions of words. Today, text corpora containing billions of words are fairly common. For building the corpora of this size, the texts on the Word Wide Web are practically the only possible resource.

The Web consists of billions of pages most of which contain texts in a natural language. Since these texts are typically available to anyone and in an electronic form, they are a perfect resource for building text corpora. Nevertheless, creating a clean Web corpus is still a challenging task. A number of steps have to be performed none of which is completely straightforward. First, the Web pages which are likely to contain the texts in the language of interest must be identified and downloaded. Next, the pages need to be represented in a unified character encoding which requires detecting the original encoding of each page. Any non-text content, such as HTML markup or JavaScript code has to be stripped as well as boilerplate content – navigation links, headers, footers, advertisements, etc. Language filtering must be performed to weed out texts not in the language of interest. Last but not least, duplicate and near-duplicate content should be eliminated.

This work opts to elaborate on two of these subproblems – removing boilerplate and eliminating duplicate and near-duplicate content. Both these issues represent challenges which have not yet received enough attention.

Fig. 1.1 illustrates the problem of boilerplate and duplicate content in Web corpora. More screenshots can be found in Appendix A.

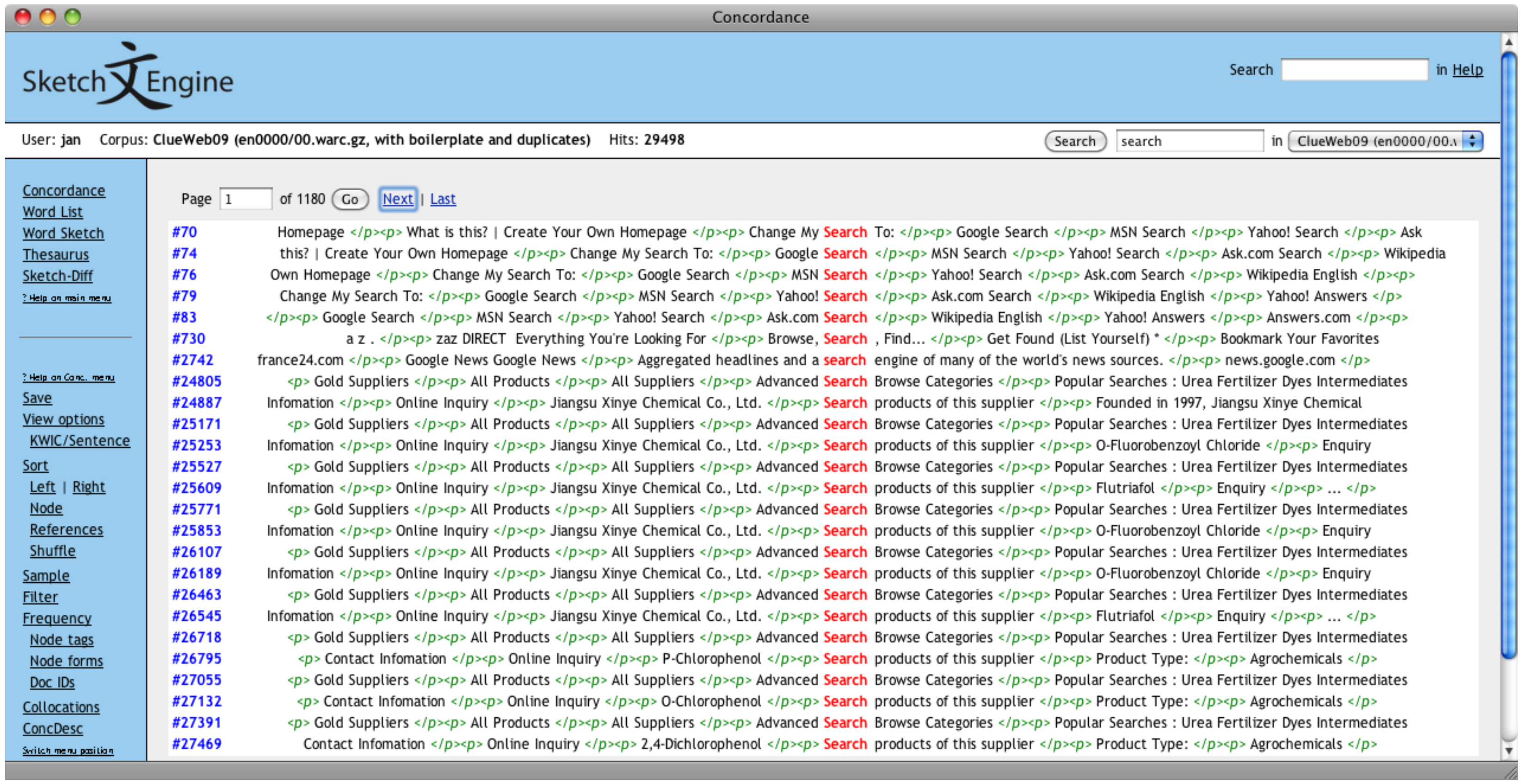
1.2 Text corpora

Different definitions of the term *text corpus* can be found in the literature. For instance, McEnery and Wilson [49] define the corpus as any collection of texts with the following four characteristics: sampling and representativeness, finite size, machine-readable, a standard reference. Since some of the characteristics are rather vague, it could be easily argued whether a given collection is a valid corpus according to this definition and/or if it is a valid corpus according to any other definition. Kilgarriff and Grefenstette [36] simplify matters greatly by defining a corpus simply as "any collection of texts". They point out that the important thing to find out is whether "corpus x is good for task y" rather than whether "x is a corpus at all". In this work, the term *corpus* (or *text corpus*) is used in accordance with the latter definition.

1.2.1 Annotation

Annotation is a desirable feature of text corpora, which makes them more useful for linguistic research. Structural tags are added to corpora to mark the boundaries of sentences, paragraphs, and documents. Metadata can be associated with structural tags to indicate the source of a text or its type (written, spoken), identify the speaker (in speech rewrites), etc. Apart from that, a part-of-speech tag and a lemma (base form) is a typically added to each word in the corpus. The process of associating words with part-of-speech tags and lemmas is called part-of-speech tagging and lemmatisation respectively, or simply tagging.

Corpus tagging can be done manually by humans, which, however, is only possible for smallish corpora. Manual tagging of corpora containing billions of words would be extremely expensive and time consuming and would require a vast number of man hours. Fortunately, algorithms exist for automatic part-of-speech tagging and lemmatisation such as the Brill tagger [11] or TreeTagger [57]. Though they are not hundred percent accurate they are adequate for most applications.



User: jan   Corpus: ClueWeb09 (en0000/00.warc.gz, with boilerplate and duplicates)   Hits: 29498

Search search in ClueWeb09 (en0000/00.\

Page 1 of 1180 Go Next | Last

```
#70     Homepage </p><p> What is this? | Create Your Own Homepage </p><p> Change My Search To: </p><p> Google Search </p><p> MSN Search </p><p> Yahoo! Search </p><p> Ask
#74     this? | Create Your Own Homepage </p><p> Change My Search To: </p><p> Google Search </p><p> MSN Search </p><p> Yahoo! Search </p><p> Ask.com Search </p><p> Wikipedia
#76     Own Homepage </p><p> Change My Search To: </p><p> Google Search </p><p> MSN Search </p><p> Yahoo! Search </p><p> Ask.com Search </p><p> Wikipedia English </p><p>
#79     Change My Search To: </p><p> Google Search </p><p> MSN Search </p><p> Yahoo! Search </p><p> Ask.com Search </p><p> Wikipedia English </p><p> Yahoo! Answers </p>
#83     </p><p> Google Search </p><p> MSN Search </p><p> Yahoo! Search </p><p> Ask.com Search </p><p> Wikipedia English </p><p> Yahoo! Answers </p><p> Answers.com </p><p>
#730    a z . </p><p> zaz DIRECT  Everything You're Looking For </p><p> Browse, Search , Find... </p><p> Get Found (List Yourself) * </p><p> Bookmark Your Favorites
#2742   france24.com </p><p> Google News Google News </p><p> Aggregated headlines and a search engine of many of the world's news sources. </p><p> news.google.com </p>
#24805  <p> Gold Suppliers </p><p> All Products </p><p> All Suppliers </p><p> Advanced Search Browse Categories </p><p> Popular Searches : Urea Fertilizer Dyes Intermediates
#24887  Infomation </p><p> Online Inquiry </p><p> Jiangsu Xinye Chemical Co., Ltd. </p><p> Search products of this supplier </p><p> Founded in 1997, Jiangsu Xinye Chemical
#25171  <p> Gold Suppliers </p><p> All Products </p><p> All Suppliers </p><p> Advanced Search Browse Categories </p><p> Popular Searches : Urea Fertilizer Dyes Intermediates
#25253  Infomation </p><p> Online Inquiry </p><p> Jiangsu Xinye Chemical Co., Ltd. </p><p> Search products of this supplier </p><p> O-Fluorobenzoyl Chloride </p><p> Enquiry
#25527  <p> Gold Suppliers </p><p> All Products </p><p> All Suppliers </p><p> Advanced Search Browse Categories </p><p> Popular Searches : Urea Fertilizer Dyes Intermediates
#25609  Infomation </p><p> Online Inquiry </p><p> Jiangsu Xinye Chemical Co., Ltd. </p><p> Search products of this supplier </p><p> Flutriafol </p><p> Enquiry </p><p> ... </p>
#25771  <p> Gold Suppliers </p><p> All Products </p><p> All Suppliers </p><p> Advanced Search Browse Categories </p><p> Popular Searches : Urea Fertilizer Dyes Intermediates
#25853  Infomation </p><p> Online Inquiry </p><p> Jiangsu Xinye Chemical Co., Ltd. </p><p> Search products of this supplier </p><p> O-Fluorobenzoyl Chloride </p><p> Enquiry
#26107  <p> Gold Suppliers </p><p> All Products </p><p> All Suppliers </p><p> Advanced Search Browse Categories </p><p> Popular Searches : Urea Fertilizer Dyes Intermediates
#26189  Infomation </p><p> Online Inquiry </p><p> Jiangsu Xinye Chemical Co., Ltd. </p><p> Search products of this supplier </p><p> O-Fluorobenzoyl Chloride </p><p> Enquiry
#26463  <p> Gold Suppliers </p><p> All Products </p><p> All Suppliers </p><p> Advanced Search Browse Categories </p><p> Popular Searches : Urea Fertilizer Dyes Intermediates
#26545  Infomation </p><p> Online Inquiry </p><p> Jiangsu Xinye Chemical Co., Ltd. </p><p> Search products of this supplier </p><p> Flutriafol </p><p> Enquiry </p><p> ... </p>
#26718  <p> Gold Suppliers </p><p> All Products </p><p> All Suppliers </p><p> Advanced Search Browse Categories </p><p> Popular Searches : Urea Fertilizer Dyes Intermediates
#26795  <p> Contact Infomation </p><p> Online Inquiry </p><p> P-Chlorophenol </p><p> Search products of this supplier </p><p> Product Type: </p><p> Agrochemicals </p>
#27055  <p> Gold Suppliers </p><p> All Products </p><p> All Suppliers </p><p> Advanced Search Browse Categories </p><p> Popular Searches : Urea Fertilizer Dyes Intermediates
#27132  <p> Contact Infomation </p><p> Online Inquiry </p><p> O-Chlorophenol </p><p> Search products of this supplier </p><p> Product Type: </p><p> Agrochemicals </p>
#27391  <p> Gold Suppliers </p><p> All Products </p><p> All Suppliers </p><p> Advanced Search Browse Categories </p><p> Popular Searches : Urea Fertilizer Dyes Intermediates
#27469  Contact Infomation </p><p> Online Inquiry </p><p> 2,4-Dichlorophenol </p><p> Search products of this supplier </p><p> Product Type: </p><p> Agrochemicals </p>
```

Figure 1.1: Search results in a corpus built from a collection of Web pages converted to plain text without any further processing.

1.2.2 Monolingual vs. multilingual corpora

Text corpora may contain data in a single language (monolingual corpora) or in multiple languages (multilingual corpora). Monolingual corpora are more common. A monolingual corpus can be understood as a representative sample of a language or its specific part. Users of monolingual corpora include researchers, lexicographers, language teachers and learners, etc. Monolingual corpora can be used for building language models, which are important resources for tasks, such as speech recognition, optical character recognition or machine translation.

Multilingual corpora can be subdivided into *parallel corpora* and *comparative corpora*. Parallel corpora contain equivalent texts in two or more languages. These texts are typically aligned at word, sentence or paragraph level. Parallel corpora are an indispensable resource for statistical machine translation. They can also support creating multilingual dictionaries.

Comparable corpora also contain texts in multiple languages. However, these texts are typically only similar (e.g. about the same topics), not equivalent, and thus they can not be aligned except for at a document level. While the comparable corpora are much easier to create than parallel corpora, the range of applications is more limited.

1.2.3 Some significant text corpora

- **Brown Corpus** was compiled by W.N. Francis and H. Kucera in 1961. It was the first computer readable, general corpus. It consists of 500 English texts in 15 different categories, with a total of 1,014,312 words. [43]

- **British National Corpus** (BNC)[1] is a 100 million word collection consisting of written (90 %) and spoken (10 %) texts from various sources. It is meant to represent the British English of the late twentieth century. The BNC was automatically part-of-speech tagged using the CLAWS tagger. It was completed in 1994.

- **Italian Web Corpus** (ItWaC) is a collection of texts mined from the Web, which contains almost 2 billion words. ItWaC was built by M. Baroni et al. in 2006 and tagged with the TreeTagger. [4]

- **Europarl** is a parallel corpus compiled of European Parliament proceedings. Its last release from 2010 consists of 11 European languages with up to 55 million words per language. The corpus is freely available for download in 10 aligned language pairs all of which include English. [39]

1.2.4 Corpus managers

As the sizes of text corpora now extend to billions of words, special software tools called corpus managers are required for handling these large amounts of data efficiently. There is a number of requirements on a good corpus manager, such as powerful query language, sorting concordances (the search results), computing basic statistics (word, bigrams and n-grams frequencies), finding collocations based on statistical measures such as T-score or MI-score, creating sub-corpora, etc. Some examples of existing corpus managers include Sketch Engine [37], IMS Corpus Workbench [58], SARA [1].

[1] `http://www.natcorp.ox.ac.uk/`

1.3 Web as corpus

The World Wide Web (WWW, or simply the Web) contains a huge amount of texts available in an electronic form to anyone. It is therefore an excellent resource for linguistic research. The methods for using the Web as a corpus range from simple search engine wrappers to large Web crawls and complex data processing. An overview of the typical approaches follows.

1.3.1 Search engine as a corpus manager

Web search engines, such as Google or AltaVista can be viewed as very simple corpus managers. Their main disadvantages (as corpus managers) are a weak query language and little context being provided for the search results. The query languages of search engines are designed for information retrieval and they typically serve their purpose. However, searching for syntactic patterns, such as “the verb *look* followed by a preposition and a noun”, is not possible.

There have been attempts to make the search engines more useful for linguistics by creating wrappers (e.g. KWiCFinder [23] or WebCorp [35]). These tools forward the input query to the search engine, collect the URLs from it, download Web pages from the URLs, process the Web pages (strip HTML tags, do part-of-speech tagging, lemmatisation) and present the results to user in a well-arranged form, such as keywords in context (KWIC).

Unfortunately, the weak query language of the search engines cannot be overcome by the wrappers. In addition, it may take quite a long time for the wrapper to process the output of a search engine, which makes the user wait after each query. Also, the responses of search engines to a particular query are not stable. Therefore, any research which makes use of these tools is hardly replicable.

1.3.2 BootCaT

Baroni and Bernardini [3] proposed a method for building smallish domain specific Web corpora. The implementation of the method is available[2] as a BootCaT toolkit[3]. A Web-based implementation also exists. [5]

The domain of a corpus is defined using a list of seed words which are characteristic for the domain. For instance, to create a corpus of texts related to playing a guitar one could use seed words such as *guitar*, *string*, *fret*, *tune*, *strum* and *chords*. The process of building a corpus is a step-by-step procedure:

1. Randomly combine input seed words into triples.
2. Use each triple as a query to a search engine.
3. Retrieve all URLs from the search engine; drop duplicates.
4. Download Web pages from the URLs.
5. Remove boilerplate; strip HTML.
6. Remove duplicate documents.
7. Do part-of-speech tagging and lemmatisation (optional).

[2]`http://bootcat.sslmit.unibo.it/`

[3]BootCaT is an acronym for “Simple Utilities to **Boot**strap **C**orpora **A**nd **T**erms from the Web”.

If there is not enough data in the corpus after the first iteration more data can be added by extracting keywords from the specialised corpus and using these as seed words for the next iteration. This procedure can be repeated until a corpus of satisfactory size is achieved. The keywords can be extracted with the help of a general language reference corpus by comparing word frequencies. The words with a higher relative frequency in the specialised corpus are likely to relate to its domain.

The output of the BootCaT is usually a smallish corpus of about one million words. This would be too small for a practical use as a general language corpus. However, the size is typically sufficient for a specialised corpus since most of the contained data is related to the area of interest. Such small domain specific corpus may often provide more evidence about relevant specialised terms than a much larger general language corpus.

Specialised corpora can be used in fields which sometimes focus on restricted domains, such as speech recognition (recognising parliament speeches or sport commentaries) and lexicography (compiling a dictionary of medical terminology).

1.3.3 Corpus Factory

The procedure utilised by BootCaT can also be used for building large general language corpora. Sharoff [59] used 500 frequent common lexical words in a language as seeds for compiling Web corpora for English, German, Russian and Chinese – each of a size similar to BNC. He explains that the seeds should be general medium-frequency words which do not indicate a specific topic.

The Corpus Factory [38] project aims to create corpora for many languages in a similar way. The key problem here is getting the list of seed words for a language where no existing corpus is available. We addressed this problem by using Wikipedia XML dumps which are available for many languages. We have built word frequency lists from the dumps and selected the mid-frequency words (ranks 1001-6000) as seed words. For most languages, only the words containing at least 5 characters have been used in order to reduce the words which are not unique for the language.

1.3.4 Web-crawled corpora

Building truly large Web corpora (say one billion words+) requires downloading millions of Web pages. This is typically achieved by *Web crawling*. A specialised software (a *Web crawler*, also just *crawler* or a *spider*) is used to "crawl" over Web pages by simulating clicking on the hypertext links. All the visited pages are downloaded.

While the Web crawling is straightforward in principle, a fast and robust Web crawler capable of handling terabyte downloads is not easy to implement. Though many open source Web crawlers exist, the production-ready ones are fairly rare. A popular choice in the Web-as-corpus community is the Heritrix crawler[4] developed by Internet Archive.

For performing a Web crawl, a list of starting URLs is required. These can be obtained from search engines using the techniques described above. One of the essential requirements on Web crawling for corpora is that mostly the Web pages containing the texts in the language of interest are downloaded. The most straightforward way to achieve this is by making the crawler stick to the first level domains of the countries where the language is prevalently used (e.g. `.de` and `.at` for German). In general, it is a good idea to employ further filtering based on content-type (e.g. only HTML pages), size (very large files usually do not contain useful data), etc.

[4] `http://crawler.archive.org/`

Once the data is downloaded, it has to be converted to plain text and cleaned of boilerplate content. A language filtering (e.g. based on frequencies of character triples[5]) should be applied to weed out pages not in the language of interest. Duplicate and near-duplicate data need to be removed.

The Web-crawled corpora have been pioneered by the WaCky community[6] who compiled large Web corpora for several languages (e.g. German, Italian, English, French), each approaching the size of 2 billion words. [4, 21, 7]

[5] `http://code.activestate.com/recipes/326576/`

[6] `http://wacky.sslmit.unibo.it/`

Chapter 2

Removing boilerplate

2.1 Introduction

The World Wide Web in its present form is an extremely rich source of linguistically interesting textual data. However, on most Web pages the main content is accompanied by non-informative parts, such as navigation menus, lists of links, headers and footers, copyright notices, advertisements, etc. These elements are usually referred to as boilerplate.

The boilerplate is known to cause problems if included in text corpora. The frequency count of some terms, such as *home*, *search*, *print*, is highly increased giving biased information about the language. Also, hits within boilerplate may be annoying when searching in corpora since they often provide no useful evidence about the phenomenon being investigated.[1] It is therefore highly desirable to remove the boilerplate when processing Web data for text corpora.

Boilerplate identification and removal has applications in other fields, too. Previous work has demonstrated that the performance of information retrieval tasks can be improved significantly if the hits within non-informative boilerplate sections are ignored or given lower weights [20, 42].

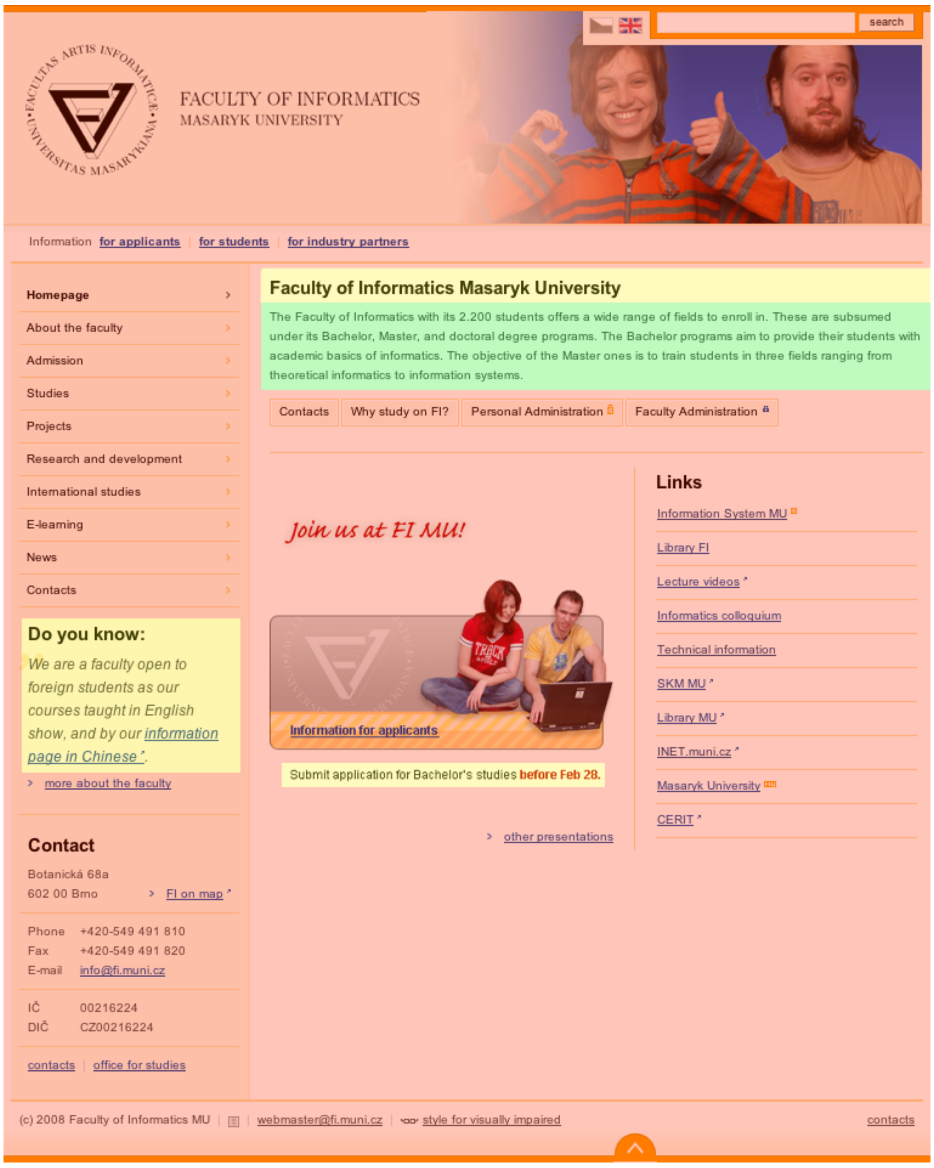
2.2 Definition of boilerplate

The boilerplate is usually defined rather vaguely as non-informative parts outside of the main content of a Web page, typically machine generated and repeated across the Web pages of the same website. While some elements such as navigation menus or advertisements are easily recognised as boilerplate, for some other elements it may be difficult to decide whether they are boilerplate or not in the sense of the previous definition. Imagine a Web page from a news site which, apart from a full text of one article, contains an abstract of another article with a link to the full text. Does this abstract qualify as boilerplate? Is it informative? Is it the main content?

It turns out that the notion of boilerplate is not easy to define with a single sentence. In the context of cleaning boilerplate from Web pages, the term typically denotes any elements which constitute noise for the application of the Web data. The exact definition is therefore application specific.

To the best of my knowledge, two fairly detailed specifications of boilerplate exist in a form of boilerplate annotation guidelines – the guidelines for the annotators of

[1]Imagine a lexicographer processing a dictionary entry for the word *home*.



search

FACULTY OF INFORMATICS
MASARYK UNIVERSITY

Information for applicants | for students | for industry partners

Homepage
About the faculty
Admission
Studies
Projects
Research and development
International studies
E-learning
News
Contacts

Do you know:

We are a faculty open to foreign students as our courses taught in English show, and by our information page in Chinese.

more about the faculty

Contact

Botanická 68a
602 00 Brno
FI on map

Phone +420-549 491 810
Fax +420-549 491 820
E-mail info@fi.muni.cz

IČ 00216224
DIČ CZ00216224

contacts | office for studies

Faculty of Informatics Masaryk University

The Faculty of Informatics with its 2.200 students offers a wide range of fields to enroll in. These are subsumed under its Bachelor, Master, and doctoral degree programs. The Bachelor programs aim to provide their students with academic basics of informatics. The objective of the Master ones is to train students in three fields ranging from theoretical informatics to information systems.

Contacts | Why study on FI? | Personal Administration | Faculty Administration

Join us at FI MU!

Information for applicants

Submit application for Bachelor's studies **before Feb 28.**

other presentations

Links

Information System MU
Library FI
Lecture videos
Informatics colloquium
Technical information
SKM MU
Library MU
INET.muni.cz
Masaryk University
CERIT

(c) 2008 Faculty of Informatics MU | webmaster@fi.muni.cz | style for visually impaired | contacts

Figure 2.1: Boilerplate content on a Web page. Green, red and yellow overlays indicate main content, boilerplate and disputable parts respectively.

the CleanEval competition gold standard[2] and the guidelines for using the KrdWrd annotation tool[3].

In short, the CleanEval guidelines instruct to remove boilerplate types such as:

- Navigation
- Lists of links
- Copyright notices
- Template materials, such as headers and footers
- Advertisements
- Web-spam, such as automated postings by spammers
- Forms
- Duplicate material, such as quotes of the previous posts in a discussion forum

The KrdWrd guidelines are more recent. They arose from the CleanEval guidelines, but they go one step further. Only the text made up of complete sentences is taken as the main content here. Elements such as lists and enumerations are classified as boilerplate unless they contain full sentences. Note that this is a significant difference from CleanEval where the inclusion of list items in the cleaned data is specifically required. Clearly, the KrdWrd aims to define the boilerplate in a way so that the remaining content constitutes good corpus data. The texts consisting of complete sentences are of the highest value for text corpora. Data such as lists or tables, on the other hand, are rather counterproductive for linguistic research.

The CleanEval's specification of boilerplate is more conservative and may, for instance, be appropriate for information retrieval, but less so for creating Web corpora.

As this work is done in the context of Web corpora, the KrdWrd's perspective on boilerplate is more relevant. Still, as the two views overlap to a high extent, I use the datasets originating from both KrdWrd and CleanEval for my experiments. The results on the KrdWrd's Canola dataset are considered more important though.

Note that CleanEval includes Web-spam and duplicate material in the definition of boilerplate. Little effort has been invested into detecting this kind of content by CleanEval contestants though. Only one system (Victor) tries to identify inter-page duplicate material. None of the participants has made an attempt to detect Web-spam. This is not surprising. While most boilerplate types can be identified based on structural features (surrounding HTML mark-up) and shallow text features (such as the number of tokens in a text block), a fundamentally different approach is required for detecting both Web-spam and duplicate content. Recognising Web-spam is a challenging task which is out of the scope of this work. Since a full separate chapter is dedicated to finding duplicated and near-duplicated texts, it is also not discussed here in the context of boilerplate cleaning.

2.3 Site-level methods

The approaches to automated boilerplate removal can be divided into two main groups – page-level and site-level. The page-level algorithms process each Web page individually. Within the site-level methods, multiple pages from the same website are processed at once.

[2] `http://cleaneval.sigwac.org.uk/annotation_guidelines.html`
[3] `https://krdwrd.org/manual/html/node6.html`

```
<html>
  <head>
    <title>HTML sample</title>
  </head>
  <body>
    <h1>Heading</h1>
    <p>Some <b>bold</b> and <i>italics</i> text.</p>
  </body>
</html>
```

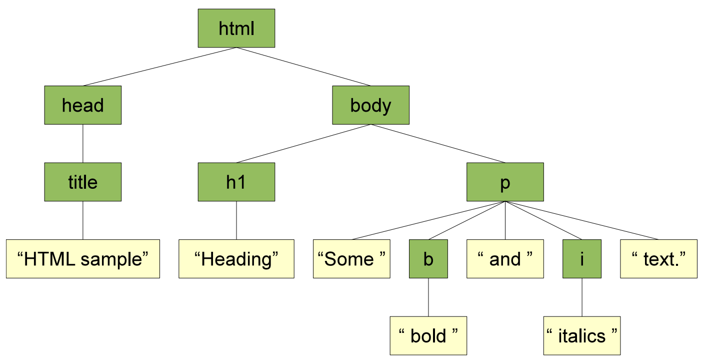


Figure 2.2: DOM tree for a sample HTML document. Text nodes containing only whitespace characters are omitted.

The site-level methods take advantage of the fact that the Web pages within the same website typically use the same or similar template. Consequently, the boilerplate sections show common characteristics across the website and can be identified by exploiting similarities among Web pages.

Bar-Yossef and Rajagopalan [2] segmented Web pages from the same website to semantically coherent blocks called pagelets by using a simple heuristic based on the number of contained links. For each pagelet, a fingerprint is computed using Broder's shingling method[4] [12]. The pagelets with equivalent fingerprints are grouped. This creates groups of identical or very similar pagelets. For each group with more than one pagelet, all Web pages are found which contain at least one pagelet from that group. For each collection of Web pages formed this way, hyperlinks are analysed. The collection is modelled as an undirected graph where nodes represent Web pages and an edge exists between any two pages which are connected with a link (i.e. one of the pages contains a link to the other). Such a graph is then split to components. For all components with more than one Web page, the contained pagelets (from the group used to form the collection) are marked as boilerplate.

Yi et al. [63] introduced a data structure called *style tree*. An HTML page can be represented with a DOM[5] tree – a tree structure which models the parent-child relations between HTML tags and where the text is contained in leave nodes called text nodes. Fig. 2.2 shows an example of a DOM tree. A *style tree* is an extension of a DOM tree and it represents one or more merged DOM trees. In a style tree, the nodes with the same parent (sibling nodes) are grouped into *style nodes*. The equivalent style nodes (equivalent sequences of HTML tags) at the same positions in the original trees are merged and their count is remembered in the merged style node. The subtrees of the style tree containing style nodes with a high count indicate frequently repeated HTML markup styles which are typical for boilerplate content.

[4] Broder's shingling is discussed in detail in Chapter 3.
[5] Document Object Model; `http://www.w3.org/DOM/`

Gibson et al. [25] also used a DOM tree based approach. For each DOM tree in a website, they extracted all subtrees and computed their hashes. Then, the number of occurrences of each hash was counted for the whole website. The HTML parts (DOM subtrees) correspondent to hashes with the count within some lower and upper thresholds are then marked as boilerplate. The upper threshold is set to prevent capturing very small HTML chunks (such as <`br`>) which occur frequently on most Web pages.

The same team also experimented with a text-based approach where HTML tags are ignored and only duplicated text chunks are exploited in order to find boilerplate parts. They used a sliding window of W characters; sampled each D-th fragment and computed its hash. Frequent hashes are then stored and used for scanning the pages of the website for boilerplate content. The sub-sampling is motivated by computational speed so that very large collections of Web pages can be processed in a reasonable time. The authors report a significant speedup over the DOM-based approach without loss in performance since both approaches produce very similar results.

Though some site-level boilerplate removal methods provide promising results, there are several drawbacks of the site-level approach. First, often there are variations in the template parts. Different subsections of the website may have different navigation links. The link to the current page may be omitted in the navigation menu. Sometimes these variations are substantial making it hard to impossible for the automated methods to recognise what the common template is. Second, apart from common templates, other boilerplate types are frequently present on Web pages, such as lists of links to relevant pages (outside the common template) or advertisements, as well as other parts which typically do not contain full sentences, such as contact information, tables, lists and enumerations, etc. Obviously, these elements are not repeated on the website and thus cannot be captured by the site-level algorithms.

The site-level methods also introduce practical problems since there may not be enough pages from the same website available, either because they do not exist (the website is very small) or they have not been downloaded for a given dataset. Finally, it is clear that having to group the Web pages by website before processing is much more complicated than handling each page individually. It may be especially problematic should the processing be done while the data is still being collected.

2.4 Page-level methods

The page-level boilerplate removal methods take a single Web page as input. This makes them more flexible and easier to use than the site-level methods. Most algorithms operate in two steps – segmentation and classification.

2.4.1 Segmentation

In the segmentation step, the input HTML page is split into semantically coherent blocks. In the next steps, each block is classified as main content (clean text) or boilerplate. The blocks should be sufficiently homogeneous in terms of boilerplate content, i.e. each block should ideally contain either boilerplate or clean text, but not a mixture of both. Different segmentation algorithms are used across boilerplate removal methods as well as different levels of granularity. The segmentation granularity typically constitutes a trade-off between the homogeneity of blocks and the ease of classification. If the segmentation is too fine-grained, individual blocks may be very small and may not carry enough information for being classified reliably.

If on the other hand the segmentation is too coarse-grained, it is more likely that heterogeneous blocks will be formed.

Simple approaches to segmentation (preferred by most algorithms) include treating each DOM text node as a single block [48, 42] or splitting the page at specified HTML tags [25, 27, 30].

Cai et al. [14] introduced a vision based method for page segmentation – VIPS. They try to identify visually coherent rectangular blocks in the visual form of a web page (as rendered by Web browsers). This is done by splitting the original DOM tree in a top-down fashion until sufficiently coherent subtrees are formed (each representing a single block). The level of coherence is determined by using features such as contained tags (blocks containing certain tags, such as `<hr>`, are split), background colour (only a single background colour is allowed in a block) or size (too large blocks get split). Several other researchers have employed the VIPS algorithm in their boilerplate cleaning methods [20, 24, 30].

A number of unique segmentation heuristics can be found in the literature. Bauer et al. [9] accept a DOM node (and its subtree) as a segment if at least 10 % of its textual content is contained in its immediate children text nodes. Bar-Yossef and Rajagopalan [2] examine the child nodes of a given DOM node N. If some of the child nodes contains at least k links in its subtree (they used $k = 3$ in their experiments), each of the child nodes is processed recursively. Otherwise, the node N and its subtree form an atomic segment. Evert [19] converts a HTML page to text with lynx and identifies segments in the text dump (e.g. by splitting at empty lines).

Some algorithms do not fit well into the segment-classify concept. The BTE algorithm [22], for instance, finds a continuous part of a Web page which maximises its objective function. A similar approach is employed by Pasternack and Roth [50].

2.4.2 Classification

The classification step decides for each block whether it is main content (clean text) or boilerplate. Depending on the application of the cleaned data, more than two classes may be supported. Some CleanEval contestants, for instance, attempt to sub-classify the clean blocks as paragraphs, headers or list items.[6]

The commonly used classification methods include both heuristics and supervised machine learning techniques, such as CRF [26, 48], Logistic regression classifiers [15], SVM [17, 9], decision trees [30, 42], Naive Bayes [50].

The lists of features used for making the classification decisions (heuristic- or ML-based) by different boilerplate cleaning methods overlap to a high extent. The features can be divided into three main groups:

- **Structural features** are based on HTML markup. The most frequently used ones are tag density (the number of HTML tags divided by the number of words/tokens), number of links and link density (the proportion of tokens inside `<a>` tags), occurrence of certain tags and their count, parent tag type, etc.

- **Textual features** capture the properties of the text contained in the block. Typical textual features include number of words or characters, number of sentences, average sentence length, proportion of function words, frequency of certain keywords, proportion of capital letters or punctuation marks, etc.

- **Visual features** are derived from a visual form of a Web page as rendered by Web browsers. Most researchers make use of visual features such as size and

[6]This is due to the form of the cleaning task assignment. The CleanEval competition is discussed in detail later.

shape of a block, distance from the center of the page or from its borders, font size and weight, etc.

Note that extracting visual features may be both computationally expensive and problematic since (i) the Web page must be rendered, and (ii) external resources such as images or style sheets may be required. Most researchers avoid using visual features for these reasons. There is no evidence that systems employing visual features perform better than other systems.

To the best of my knowledge, only one thorough comparative study on features appropriate for boilerplate cleaning has been published. Kohlschütter et al. [42] compared a number of structural and textual features on a dataset of 621 manually annotated news articles from 408 different web sites. They used the information gain measure (Kullback-Leibler divergence) in order to identify the features with the highest ability to distinguish between main content and boilerplate. Simple features, such as number of words, link density, average word length, the proportion of capital letters and the proportion of full stops produced very high scores.

2.5 CleanEval

The goal of the CleanEval competition [6] held in 2007 was to support research in the area of boilerplate cleaning and encourage development of new boilerplate cleaning algorithms. The task assignment for the contestants was:

1. remove boilerplate (Text Only part)
2. remove boilerplate and add simple markup for headers, list items and paragraphs by inserting `<h>`, `<l>` and `<p>` tags respectively (Text and Markup part)

The competition included two categories – English and Chinese. The contestants could decide to participate in one of them or in both (only one system participated in Chinese).

In the development stage, a development dataset[7] was made available comprising 58 HTML pages in English and 51 in Chinese, all randomly selected from existing Web corpora. Additionally, text versions of the pages were provided as well as cleaned texts (gold standard) created by human annotators.

The final dataset includes 684 files for English and 653 for Chinese. Human annotations have been done by 23 students who had been provided with two pages of annotation guidelines. Each file was annotated by only a single annotator. The format of the gold standard files is a plain text with simple markup (headers, lists, paragraphs) where the boilerplate parts are removed. Since there is no direct linking to the original HTML files the evaluation against the gold standard is not straightforward.

The CleanEval scoring script computes the Text Only and Text and Markup score for each document and the results are averaged over the whole collection in order to get the final Text Only and the final Text and Markup score. The mean of these two values is the contestant's final score.

The Text Only score is computed by using a modified Levenshtein edit distance [45]. The scoring program counts the minimal number of operations (token insertions and deletions) to be performed on the cleaned file in order to transform it into the gold standard file. The operation count is then divided by the file length (in tokens).

[7] `http://cleaneval.sigwac.org.uk/devset.html`

The Text and Markup scoring is not discussed here since adding the structural markup is not essential for the boilerplate cleaning task. The detailed CleanEval report [6] can be consulted for information about the scoring.

The results of CleanEval provide a comparison of nine different systems. Surprisingly, seven of them achieve very close scores (ranging from 80.9 to 84.1 in the Text Only task) even though they employ very diverse boilerplate cleaning techniques. Unfortunately, the information value of the results is disputable. The final dataset contains documents of various sizes, even very short ones. The problem is that results on short documents are unstable. One of the extreme cases is the document number 278 which only contains the text "404 : Page Not Found". This is (apparently correctly) taken out as boilerplate in the gold standard leaving the document empty. Now, while failing to remove this short string from a long document should only lower performance by a few points, here it means a drop from 100 to 0. This result will influence the final score in exactly the same way as the result on any other document. Therefore, the final results may be biased. The problem could have been avoided by using the document lengths as weights.

Back in 2008, I tested my implementation[8] of the well-known BTE algorithm [22] on the CleanEval data collection. It came out as a surprise that the simple heuristic scored 85.41 Text Only[9] (reported in [21]) on the final dataset, outperforming all the CleanEval contestants, some of which employed very sophisticated methods. Later work revealed though that the performance of the BTE algorithm is problematic and the good results here are mostly due to the characteristics of the CleanEval collection and scoring method. I further discuss this issue in the evaluation section.

2.6 Algorithms

This section provides a brief description of the algorithms included in the evaluation. Those algorithms were selected, for which the implementation is available or they are easy to re-implement. While these constraints may seem limiting, still the selected algorithms represent a wide range of different approaches to boilerplate cleaning and most of them achieve top scores in the available comparative studies [6, 42].

2.6.1 BTE

BTE (Body Text Extraction) algorithm [22] removes boilerplate using a simple heuristic based on the density of HTML tags. The idea is that the main body contains only little formatting and is therefore sparse in terms of HTML tags. The navigation links, advertisements and alike, on the other hand, contain a lot of tags. The BTE algorithm views a HTML page as a sequence of bits B, where $B_n = 0$ if n-th token is a word and $B_n = 1$ if the token is a tag. Values i and j are searched, for which the objective function $T_{i,j}$ is maximal.

$$T_{i,j} = \sum_{n=0}^{i-1} B_n + \sum_{n=i}^{j} (1 - B_n) + \sum_{n=j+1}^{N-1} B_n$$

where N is the number of tokens on the Web page. The text between positions i and j is then extracted. This method has been applied for boilerplate stripping in a number of projects aimed at building corpora from the Web [3, 4, 5, 59, 21].

[8] http://nlp.fi.muni.cz/~xpomikal/cleaneval/bte-3.0/bte.py
[9] http://nlp.fi.muni.cz/~xpomikal/cleaneval/bte-3.0.log

2.6.2 Victor

Victor [48] utilises a supervised machine learning approach. It extracts the list of DOM text nodes from the input HTML file and classifies each either as *header*, *paragraph*, *list*, *continuation* or *other*. The first three classes represent content blocks (the subcategorisation is due to the CleanEval Text and Markup task assignment). The *continuation* class indicates the continuation of the previous content block. Boilerplate blocks are marked as *other*. The classification is done by using the Conditional Random Fields with a wide range of structural and textual features. Victor is implemented in Perl with the use of CRF++ toolkit[10] and the implementation is available for download[11]. The package contains a classifier trained on 51 HTML files, manually annotated in accordance with CleanEval annotation guidelines. It is worth noting that this system achieved the highest score in CleanEval and won the competition.

2.6.3 NCLEANER

NCLEANER [19] (a.k.a. StupidOS [18]) uses character-level n-gram language models based on conditional probabilities. The algorithm performs some regular expression pre-processing of the input HTML. The input is then converted to plain text with lynx[12]. After that, post-processing heuristics are applied which remove some easily recognisable boilerplate (such as fields separated with vertical bars) and segments are detected in the text dump (e.g. paragraphs are separated with empty lines by lynx). As the last step, both main content and boilerplate language models are applied to each segment and the model that calculates a higher probability assigns a class to the segment. The disadvantage of this approach is its language dependence. New models have to be trained in order to apply the algorithm to other languages. A language independent non-lexical version of NCLEANER has also been described and tested, but it is not considered in the evaluation since it does not perform as well as the lexical version. The Perl implementation of NCLEANER is available for download.[13]

2.6.4 boilerpipe

boilerpipe [42] is a family of boilerplate cleaning algorithms based on decision trees. The algorithms have been developed for extracting the full texts of articles on news sites. However, they are also applicable to cleaning boilerplate from general HTML pages.

The basic algorithm, **DefaultExtractor**, has been created by building a C4.8-based decision tree on the L3S-GN1 dataset (discussed in section 2.8.3) using 67 different features. Reduced-error pruning has then been applied to the tree to prevent overfitting. As a result, a small tree has been created which uses only two (apparently most discriminative) features – word count and link density. These features are extracted from the current block (the block being classified) as well as from the previous and the next block.

An HTML page is split to blocks at all tags except for links (<`a`> tags). The links are included in the blocks so that the link density can be computed (as the number of tokens inside the <`a`> tags divided by the total number of tokens in the block). The blocks are classified using the pruned decision tree which is described here as Algo-

[10] `http://crfpp.sourceforge.net/`
[11] `http://ufal.mff.cuni.cz/victor/`
[12] `http://lynx.browser.org/`
[13] `http://sourceforge.net/projects/webascorpus/files/NCleaner/`

```
curr_linkDensity <= 0.333333
| prev_linkDensity <= 0.555556
| | curr_numWords <= 16
| | | next_numWords <= 15
| | | | prev_numWords <= 4: BOILERPLATE
| | | | prev_numWords > 4: CONTENT
| | | next_numWords > 15: CONTENT
| | curr_numWords > 16: CONTENT
| prev_linkDensity > 0.555556
| | curr_numWords <= 40
| | | next_numWords <= 17: BOILERPLATE
| | | next_numWords > 17: CONTENT
| | curr_numWords > 40: CONTENT
curr_linkDensity > 0.333333: BOILERPLATE
```

Algorithm 2.1: Classification of blocks in the boilerpipe DefaultExtractor algorithm.

rithm 2.1. The DefaultExtractor algorithm is referred to as “NumWords/LinkDensity Classifier” in [42].

The authors present another similar algorithm which uses text density [40] rather than word count. It turns out though that the text density can be accurately approximated with the word count (with some upper boundary). It is not surprising that the difference between the performance of the two algorithms is negligible. I therefore selected only the word count based version for the evaluation since the word count is better defined (here as the number of white-space delimited character sequences) and easier to understand than the text density.

ArticleExtractor is an extension of the DefaultExtractor which adds some heuristic rules specific for processing HTML files containing news articles (e.g. identifying the beginning of the comments section by using indicator strings such as “User comments:”). In [42], this algorithm is referred to as “NumWords/LinkDensity + Main Content Filter”.

A version of the DefaultExtractor trained on the Canola dataset (see section 2.8.2) is available as **CanolaExtractor**. Java implementation of the boilerpipe algorithms can be downloaded from the project website[14].

2.6.5 Other algorithms

Other notable algorithms are mentioned here for reference. Pasternack and Roth [50] assign a value between -0.5 and 0.5 to each token in a document using a supervised classifier (Naive Bayes). Then, similarly as BTE, they identify the subsequence of tokens with the maximal total value.

The FIASCO system [9] segments a Web page to blocks using a simple heuristic. A number of structural, textual and visual features are then extracted from each block and an SVN classifier is applied.

Gao and Abou-Assaleh [24] employ the VIPS segmentation. They extract 9 features of various kinds from blocks and determine a block importance by computing the sum of weighted values of the features. Block similarity is then estimated for each two blocks by taking their mutual distance, horizontal offset, width and the values of extracted features into account and the block importances are adjusted to be similar for similar blocks. The blocks with the importance below certain threshold are removed.

Htmlcleaner [27] strips text-style tags (such as <`font`>, <`b`>, <`i`>, etc) from a page and splits it at the remaining tags. For each block, the number of anchored

[14] `http://code.google.com/p/boilerpipe/`

characters (nA) and the number of characters in function words (nFW) is counted. Then, a continuous sequence of blocks is found, for which the sum of ($nFW - nA$) values is maximal.

Chakrabarti et al. [15] build a DOM tree for the input page and assign a score between 0 and 1 to each subtree using logistic regression classifiers based on features such as closeness to the margins, link density and tag density. The scores are stored in the roots of the subtrees. Then a technique called isotonic smoothing is applied to adjust the scores in a way so that the score of each node is at least the score of its parent and at the same time the adjusted scores do not differ too much from the original values. Finally, the subtrees with the score below certain threshold are kept as main content.

2.7 jusText

The jusText algorithm was created as a simple exercise during my experiments with data de-duplication. At that time, the BTE algorithm was widely used for removing boilerplate within the Web as corpus community. It has been in use before CleanEval since few alternatives have been available and it remained in use even after CleanEval as it was discovered that it outperforms all CleanEval contestants [21].

In the de-duplication experiments on Web data (described in the next chapter), I typically used BTE during a preprocessing stage. A boilerplate was often found in the data identified as duplicated content. This is not surprising as a boilerplate is often repeated in Web pages within the same website. These experiments revealed a lot of boilerplate that the BTE failed to remove. In the search for a more effective algorithm, jusText has been created.

The algorithm uses a simple way of segmentation. The contents of some HTML tags are (by default) visually formatted as blocks by Web browsers. The idea is to form textual blocks by splitting the HTML page on these tags[15] with the hope that they will be sufficiently homogeneous in terms of good and boilerplate content. The evaluation section demonstrates that blocks of this kind can contain both, but it is too rare to constitute a problem.

Several observations can be made about the blocks created in this way:

1. Short blocks which contain a link are almost always boilerplate.
2. Any blocks which contain many links are almost always boilerplate.
3. Long blocks which contain grammatical text are almost always good whereas all other long block are almost always boilerplate.
4. Both good (main content) and boilerplate blocks tend to create clusters, i.e. a boilerplate block is usually surrounded by other boilerplate blocks and vice versa.

Deciding whether a text is grammatical or not may be tricky, but a simple heuristic can be used based on the volume of function words (stop words). While a grammatical text will typically contain a certain proportion of function words, few function words will be present in boilerplate content such as lists and enumerations.

[15] The full list of the used block-level tags includes: BLOCKQUOTE, CAPTION, CENTER, COL, COLGROUP, DD, DIV, DL, DT, FIELDSET, FORM, H1, H2, H3, H4, H5, H6, LEGEND, LI, OPTGROUP, OPTION, P, PRE, TABLE, TD, TEXTAREA, TFOOT, TH, THEAD, TR, UL. A sequence of two or more BR tags also separates blocks.

```
if link_density > MAX_LINK_DENSITY:
    return 'bad'

# short blocks
if token_count < LENGTH_LOW:
    if link_density > 0:
        return 'bad'
    else:
        return 'short'

# medium and long blocks
if stopwords_density > STOPWORDS_HIGH:
    if token_count > LENGTH_HIGH:
        return 'good'
    else:
        return 'near-good'
if stopwords_density > STOPWORDS_LOW:
    return 'near-good'
else:
    return 'bad'
```

Algorithm 2.2: Classification of blocks in jusText.

The key idea of the algorithm is that long blocks and some short blocks can be classified with very high confidence. All the other short blocks can then be classified by looking at the surrounding blocks.

2.7.1 Preprocessing

In the preprocessing stage, the contents of the <`header`> tag are removed as well as the contents of all <`style`> and <`script`> tags. This is for compatibility with the Canola dataset (discussed below). For compatibility with Victor, an option can be specified for removing the contents of the <`select`> tags, too. If the <`select`> tags are not removed completely, the preprocessing still marks their contents as bad (boilerplate). Also, blocks containing a copyright symbol (©) are marked as bad.

Note that one of the used evaluation method requires that the algorithm outputs both main content and boilerplate (with corresponding labels). Therefore, marking a block as boilerplate is not the same as removing it completely.

2.7.2 Context-free classification

After the segmentation and preprocessing, context-free classification is executed which assigns each block to one of four classes:

- *bad* – boilerplate blocks
- *good* – main content blocks
- *short* – too short to make a reliable decision about the class
- *near-good* – somewhere in-between short and good

The classification is done by the Algorithm 2.2. The token count is the number of space-separated items and the link density is defined as the proportion of tokens inside <`a`> tags (i.e. the same way as for the boilerpipe algorithms). The stop words density is simply the proportion of stop list words (the words are defined here as the longest non-overlapping continuous subsequences of alphabetical characters[16]).

The algorithm takes two integers (LENGTH_LOW, LENGTH_HIGH) and three floating point numbers (MAX_LINK_DENSITY, STOPWORDS_HIGH, STOPWORDS_LOW) as parameters. The former two set the thresholds for dividing the blocks by tokens count into short, medium-size and long. The latter two divide the blocks by the stop words density into low, medium and high. The default settings are:

- MAX_LINK_DENSITY = 0.2
- LENGTH_LOW = 10
- LENGTH_HIGH = 30
- STOPWORDS_LOW = 0.30
- STOPWORDS_HIGH = 0.32

The evaluation section explains why these values seem to be most appropriate for the algorithm.

For convenience, the assignment of classes for the medium and long blocks is also illustrated in Table 2.1 since the second part of the classification algorithm may be difficult to read.

block size (token count)	stopwords density	class
medium-size	low	bad
long	low	bad
medium-size	medium	near-good
long	medium	near-good
medium-size	high	near-good
long	high	good

Table 2.1: Assignment of classes for medium-size and long blocks.

2.7.3 Context-sensitive classification

The goal of the context-sensitive part of the algorithm is to re-classify the *short* and *near-good* blocks either as *good* or *bad* based on the classes of the surrounding blocks. The blocks already classified as *good* or *bad* serve as base stones in this stage. Their classification is considered reliable and is never changed.

The pre-classified blocks can be viewed as sequences of *short* and *near-good* blocks delimited with *good* and *bad* blocks. Each such sequence can be surrounded by two *good* blocks, two *bad* blocks or by a *good* block at one side and a *bad* block at the other. The former two cases are handled easily. All blocks in the sequence are classified as *good* or *bad* respectively. In the latter case, a *near-good* block closest to the *bad* block serves as a delimiter of the *good* and *bad* area. All blocks between the *bad* block and the *near-good* block are classified as *bad.* All the others are classified as *good.* If all the blocks in the sequence are *short* (there is no *near-good* block) they are all classified as *bad.*

The idea behind the context-sensitive classification is that boilerplate blocks are typically surrounded by other boilerplate blocks and vice versa. The *near-good* blocks

[16] An alphabetic sequence containing a hyphen (-), such as *de-duplication*, is considered a single word.

usually contain useful corpus data if they occur close to *good* blocks. The *short* blocks are typically only useful if they are surrounded by *good* blocks from both sides. They may, for instance, be a part of a dialogue where each utterance is formatted as a single block. While discarding them may not constitute a loss of significant amount of data, losing the context for the remaining nearby blocks could be a problem.

In the description of the context-sensitive classification, one special case has been intentionally omitted in order to keep it reasonably simple. A sequence of *short* and *near-good* blocks may as well occur at the beginning or at the end of the document. This case is handled as if the edges of the documents were *bad* blocks as the main content is typically located in the middle of the document and the boilerplate near the borders.

```
def get_context_sensitive_class(block):
    if block.pre_class == 'bad':
        return 'bad'
    if block.pre_class == 'good':
        return 'good'
    if block.pre_class == 'near-good':
        prev_class = get_class_of_prev_good_or_bad_block(block)
        next_class = get_class_of_next_good_or_bad_block(block)
        if prev_class == 'good' or next_class == 'good':
            return 'good'
        else:
            return 'bad'
    if block.pre_class == 'short':
        prev_class = get_class_of_prev_good_or_bad_block(block)
        next_class = get_class_of_next_good_or_bad_block(block)
        if prev_class == 'bad' and next_class == 'bad':
            return 'bad'
        if prev_class == 'good' and next_class == 'good':
            return 'good'
        if prev_class == 'bad' and next_class == 'good':
            if get_class_of_prev_non_short_block(block) == 'near-good':
                return 'good'
            else:
                return 'bad'
        if next_class == 'bad' and prev_class == 'good':
            if get_class_of_next_non_short_block(block) == 'near-good':
                return 'good'
            else:
                return 'bad'
```

Algorithm 2.3: Context-sensitive classification of pre-classified blocks.

Algorithm 2.3 describes the process of context-sensitive classification more formally. The semantics of the used methods is as follows. The `get_class_of_prev_good_or_bad_block` method returns the class of the nearest previous block which is *good* or *bad*, i.e. it skips all *short* and *near-good* blocks. The method `get_class_of_prev_non_short_block` returns the class of the nearest previous block which is not *short*. The `next` forms of the methods are defined analogically. All methods return the *bad* class if the edge of the document is reached while looking for a block in a given direction. An illustrative example of the application of the algorithm is provided in Fig. 2.3.

The time complexity of the algorithm is linear in the number of blocks since the values returned by the helper methods can be precomputed with two linear passes over the blocks. The classes of the previous *good_or_bad* and *non_short* blocks are easily determined by going from the first block to the last and remembering the last *good_or_bad* and the last *non_short* block encountered. Analogically for the classes of the next blocks by using the opposite direction.

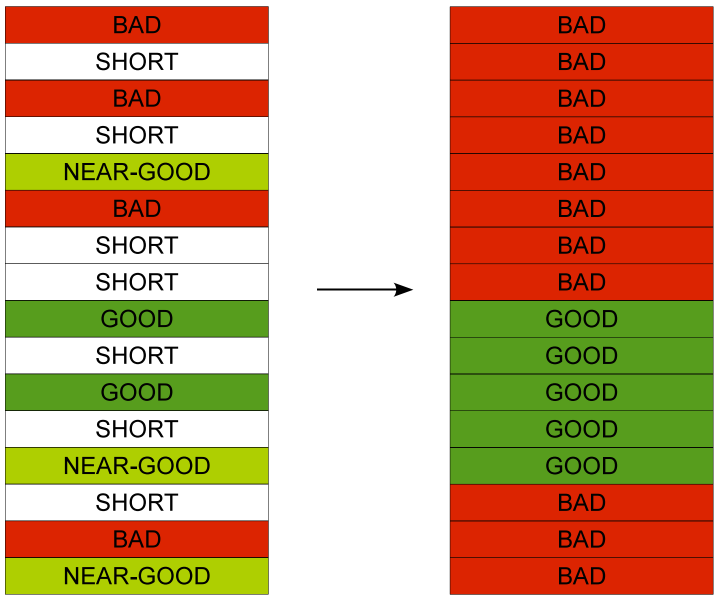


Figure 2.3: Example of context-sensitive classification.

2.8 Datasets

2.8.1 CleanEval

The dataset from the CleanEval competition is discussed in section 2.5. To the best of my knowledge few other publicly available datasets exist.

2.8.2 Canola

An interesting page annotation tool is developed within the KrdWrd project[17]. The tool is installed as a Firefox AddOn. It is designed in a clever way which makes the task of annotating Web pages easy and fast. It seems as a promising starting point to creation of new datasets for evaluation of boilerplate cleaning algorithms. The tool is already in use and first resources have already been created. The Canola corpus[18] consists of 219 Web pages each of which was annotated by at least five different human annotators.[19]

The annotations in the Canola corpus are done at the level of DOM tree text nodes. Since the boilerplate parts are almost always separated from the main content with some markup, it is practically guaranteed that each text node is homogeneous in terms of boilerplate and main content. Three annotation classes are supported – *bad* (1), *uncertain* (2) and *good* (3). Obviously, the *bad* and *good* refer to boilerplate and main content respectively. The *uncertain* class is meant for everything that does not belong in any of the other two categories. According to the annotation guidelines[20], all captions should specifically be included here.

[17] `https://krdwrd.org/`
[18] `https://krdwrd.org/trac/wiki/Corpora/Canola`
[19] The Canola corpus is not (yet) publicly available. I am indebted to the authors for providing the data for my experiments.
[20] `https://krdwrd.org/manual/html/node6.html`

For each Web page, the annotations are saved into two files: (i) The `.html` file contains the original document with <`kw`> tags surrounding each annotated part where the value of the `class` attribute indicates the class of the contained text nodes. In these files, only the merged results of all annotators are present and the final class is the result of majority voting. Fig. 2.4 shows a sample of the format. (ii) The `.stat` file contains the decisions of all annotators for each text node in the document. Each line in the file corresponds to a single text node and contains the following tab-separated values: the length of the node (in characters), the merged annotation result (same as in the `.html` file) and a list of the decisions of all annotators. A sample is shown in Fig. 2.5. Note that the samples have been created to be illustrative rather than taken from the corpus.

In the merged annotations, the level of agreement is expressed as an integer number smaller than or equal to 100. The value is henceforth referred to as *annotation reliability* (*AR*). The annotation reliability is a rounded percentage of annotators who assigned the most frequent class to given text node. The number is rounded to tens. For instance, if six annotators assigned the *good* class, two annotators the *bad* class and one annotator the *undecided* class, the majority is 66.67 % (6 out of 9) and the number is rounded to produce the final *AR* of 70. Note that the *AR* value of 100 indicates an unanimous agreement (maximal reliability).

The availability of multiple annotations for each document makes the collection extremely useful for evaluation. First, the merged annotations are much more reliable than annotations of a single annotator. Second, problematic parts can easily be recognised based on the level of annotators disagreement. High disagreement indicates that no clear decision can be made about the class of the text node in question. These text nodes can then be assigned lower weights in the evaluation or even completely ignored.

2.8.3 L3S-GN1

A dataset named L3S-GN1 has recently been made available for download[21]. The collection consists of 621 manually annotated news articles from 408 different websites. It is provided in the WARC format[22]. Both original HTML files and HTML files with annotations are available. Each annotated text block is surrounded with a <`span`> tag with a `class` attribute indicating the annotation class of the block. Six different classes are recognised:

- Not content (0)
- Headline (1)
- Full text (2)
- Supplemental (3)
- Related content (4)
- Comments (5)

The *non content* class represents a boilerplate. All other classes represent some part of the main content.

Some pages contain unannotated parts. These are to be interpreted as *non content.* The annotations are sometimes nested. According to the author [41], this is due to minor glitches in the annotation tool and the class of the innermost annotation tag is the intended one.

[21] `http://www.l3s.de/~kohlschuetter/boilerplate/`
[22] `http://bibnum.bnf.fr/WARC/`

```
<ul class="navigation">
  <li><a href="home.html"><kw class="krdwrd-tag-1-100">Home</kw></a></li>
  <li><a href="search.html"><kw class="krdwrd-tag-1-100">Search</kw></a></li>
</ul>
<h1><kw class="krdwrd-tag-2-80">Lorem ipsum</kw></h1>
<p>
  <kw class="krdwrd-tag-3-100">Lorem ipsum dolor sit amet, <strong>consectetur
  adipisici elit</strong>, sed eiusmod tempor incidunt ut labore et dolore magna
  aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris
  nisi ut aliquid ex ea commodi consequat. Quis aute iure reprehenderit in
  voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint
  obcaecat cupiditat non proident, sunt in culpa qui officia deserunt mollit
  anim id est laborum.</kw>
  <a href="lorem_ipsum_full.html"><kw
    class="krdwrd-tag-mrgtie-50">more...</kw></a>
</p>
<div class="footer">
  <kw class="krdwrd-tag-1-80">Copyright (c) 2010. All rights reserved.</kw>
</div>
```

Figure 2.4: Canola annotation format – `.html` part. Merged annotation classes are in blue (*mrgtie* indicates a tie in the majority voting). Annotators agreement is in fuchsia.

```
4    krdwrd-tag-1-100       krdwrd-tag-1 krdwrd-tag-1 krdwrd-tag-1 krdwrd-tag-1
6    krdwrd-tag-1-100       krdwrd-tag-1 krdwrd-tag-1 krdwrd-tag-1 krdwrd-tag-1
11   krdwrd-tag-2-80        krdwrd-tag-2 krdwrd-tag-1 krdwrd-tag-2 krdwrd-tag-2
27   krdwrd-tag-3-100       krdwrd-tag-3 krdwrd-tag-3 krdwrd-tag-3 krdwrd-tag-3
26   krdwrd-tag-3-100       krdwrd-tag-3 krdwrd-tag-3 krdwrd-tag-3 krdwrd-tag-3
375  krdwrd-tag-3-100       krdwrd-tag-3 krdwrd-tag-3 krdwrd-tag-3 krdwrd-tag-3
7    krdwrd-tag-mrgtie-50   krdwrd-tag-1 krdwrd-tag-3 krdwrd-tag-3 krdwrd-tag-1
40   krdwrd-tag-1-80        krdwrd-tag-1 krdwrd-tag-2 krdwrd-tag-1 krdwrd-tag-1
```

Figure 2.5: Canola annotation format – `.stat` part. Text node lengths, merged annotations results and individual decisions of four annotators.

2.9 Evaluation

Several technical problems had to be resolved in order to use the Canola corpus for evaluation. It turned out that Canola was created with an early version of the KrdWrd annotation framework. Due to minor glitches in this version some noise has been introduced into the collected data. The most notable is the problem with JavaScript. The downloaded HTML pages have been parsed with Gecko[23] and JavaScript has been executed. For some pages, the JavaScript had modified the DOM tree of the page (mostly by adding new elements) before the Web page has been saved to a database. In order to present a page to a user for annotation, it has been retrieved from the database (possibly already modified: original + JS execution) and parsed again, which in some cases caused further modifications (original + two JS executions). Consequently, some annotated pages differ from the original versions. In order to overcome this problem, the annotation tags (<`kw`>) have been stripped from the annotated pages and the boilerplate cleaning algorithms executed on these stripped pages rather than on the originals.

Another problem is with the identification of the text nodes. In order to take the decisions of all annotators into account for evaluation, the contents of the `.stat` files must be used. The only linking of these files to the `.html` files is in terms of the DOM tree text nodes – the line number in the `.stat` file corresponds to the text node position in the HTML file. It this therefore essential for the cleaning algorithm to be able to extract the list of text nodes from the HTML file and output the classification decision for each node. Only then can its output be aligned with the gold standard and evaluated.

Unfortunately, it is a widely known fact that well formed HTML pages are rare on the Web since most Web browsers use robust HTML parsers which make it possible for virtually any page to be displayed (somehow), no matter how broken HTML code it contains. However, it is obvious that these robust parsers have to use some heuristics in order to create parse trees of broken pages. Since different parsers may use different heuristics here, different parse trees may be created, each containing a (slightly) different list of text nodes.

The jusText algorithm is implemented in Python and uses the lxml[24] library for HTML parsing. It turned out that the level of misalignment of its output with the gold standard (`.stat` files) is significant. Manual inspection of the files revealed that this is partially due to the difference in the HTML parsing and partially due to some further glitches in the Canola dataset.

Out of the 219 documents in Canola only 97 documents processed with jusText contained the same number of text nodes as the gold standard. Victor, which uses yet another HTML parser (Perl module HTML::Parser[25]), aligned with the gold standard only on 90 files. The intersection with the files where jusText aligned produced 75 files. Since all other algorithms included in the evaluation on Canola have been implemented using the lxml library (same as jusText) their output should also align with these 75 files. This suggested that the 75 documents could be used as a test set and the remaining files as a development set. However, it turned out that splitting the collection in this way would introduce a strong bias in terms of boilerplate distribution. While the content annotated as boilerplate would only amount to 19 % in the test set, the development set would contain 32 % of boilerplate (measured in characters for merged annotations). A possible explanation is that broken pages which make consistent HTML parsing difficult tend to contain higher volume of boilerplate.

[23]`https://developer.mozilla.org/en/Gecko`
[24] `http://codespeak.net/lxml/`
[25] `http://search.cpan.org/~gaas/HTML-Parser-3.66/Parser.pm`

This bias would clearly be undesirable for the evaluation.

As the problems with using the `.stat` files for evaluation proved extremely difficult to overcome, these were ignored and the merged annotations in the `.html` were used files instead. Since the merged annotations indicate the level of agreement of the annotators they are almost equally useful as having the annotations of every single annotator available.

All of the evaluated algorithms, as well as most other known algorithms, treat DOM text nodes as atomic units, i.e. no algorithm classifies one part of a text node as main content and other part as boilerplate. It is therefore perfectly reasonable to do the evaluation at the level of text nodes as long as the same list of text nodes is produced by all the algorithms to be compared. This approach did not work well with the `.stat` files as (i) three HTML parsers have been involved (Gecko, lxml, HTML::Parser) and (ii) some noise seems to be included in the data, probably due to minor bugs in the program which produced them. The hope was that if the number of parsers is reduced to two and the noise is eliminated, the approach may actually be plausible.

The gold standard has been created by parsing the annotated `.html` files with the lxml parser, extracting all text nodes, finding the annotation for each text node in the enclosing `<kw>` tag and storing the results in a format similar to the `.stat` files. 22 out of the 219 files contained unannotated text nodes. These files have been excluded from the collection narrowing it down to 197 files. Since Victor removes the contents of the `<head>`, `<style>`, `<script>` and `<select>` tags in the preprocessing stage, which is a reasonable thing to do, the same preprocessing has been implemented in the other algorithms included in the evaluation as well as in the script which created the gold standard.

After creating the gold standard the `<kw>` tags have then been stripped from the `.html` files and the algorithms have been run on the stripped files. Since all implementations, except for Victor, used the lxml parser it was not surprising that they produced a perfect alignment with the gold standard. For Victor, the number of text nodes in the output has been compared with the gold standard. In the files where the number of nodes matched, the lengths of each pair of text nodes have been compared. The number of pairs where the difference in lengths was more than 5 % has been counted and if the total count was more than 5 % of all pairs, the file was said to be misaligned with the gold standard. Only 31 documents have been found where the number of nodes was different from the gold standard or the text nodes did not align well. This is a significant improvement over the level of misalignment with the `.stat` files. Manual inspection of the misaligned files revealed that the most frequent cause of misalignment is that Victor merges text nodes separated with empty tags into one. For example, the following piece of (rather weird) HTML code taken from one of the misaligned files is treated as a single text node by Victor:

```
We investigated the usefulness of echocardiographic
contrast<sup></sup>perfusion imaging in differentiating cardiac
masses.
```

Rather than investing more efforts into achieving a better alignment of the Victor's output with the gold standard the 31 documents have been examined for the amount of boilerplate content. This revealed that the problem with the bias observed before does not apply here. Therefore the 31 files have simply been inserted into the development set. The test set has been created by randomly sampling 100 documents from the remaining 166 files (197 minus 31). The other 66 files have been added to the development set increasing its size to 97 documents.

2.9.1 Evaluation method for Canola

Each page in the Canola corpus has been annotated by 5 to 12 annotators with the average of 7.28 annotations per page. The availability of multiple annotations is of a great value. The reliability of the annotation of each text node is indicated by the level of agreement of the annotators. Higher agreement expresses higher reliability.

In the evaluation, a threshold of the AR[26] is set and only those nodes which meet or exceed this value are taken into account. All other nodes are ignored. Throughout the experiments, the results based on four thresholds have been monitored – 100, 80, 50 and 0. Note that the value of AR is always positive. Therefore, with the threshold of 0 all text nodes are used. Different thresholds produce different absolute values of the results (both precision and recall typically decrease with a decreasing threshold). However, the relative differences are not significant and graphs show similar trends. For this reason and also for the sake of easier readability, only the results at the extreme ends are reported, i.e. for the thresholds of 100 and 0. In the following text, the relevant annotations will also be referred to as *reliable annotations* and *all annotations* respectively.

As the Canola corpus allows for *undecided* annotations, the text nodes where the merged annotation is *undecided* are ignored in the evaluation. The same applies for the nodes where the majority voting is tied.

The results are presented using common information retrieval metrics – precision, recall and F_1. The results are weighted by the lengths of the text nodes in characters. The rationale is that failing to remove a long boilerplate block is a more serious error than failing to remove a short one. The same logic applies for incorrectly removed main content blocks. Thus the precision is defined as the number of characters correctly classified as main content divided by the total number of characters classified as main content. The recall is the number of characters correctly classified as main content divided by all main content characters. F_1 is the harmonic mean of the precision and recall:

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

When processing Web data for corpora we typically have enough data available. Therefore we may prefer to trade precision for recall since losing some good data is not as much of a problem as including rubbish data. For this reason, the value of the $F_{0.5}$ measure is also reported, which weights precision twice as much as recall.

$$F_{0.5} = (1 + (0.5)^2) \cdot \frac{precision \cdot recall}{(0.5)^2 \cdot precision + recall}$$

All reported results are micro-averaged, i.e. the scores are computed for the whole dataset as if it were a single big document, rather than computing a score for each document separately and averaging the values (macro-averaging).

2.9.2 Canola statistics

Table 2.2 shows the volume of text nodes and characters for each of the three annotation classes and for both reliable annotations ($AR = 100$) and all annotations ($AR > 0$). The statistics are reported separately for the test set (100 files) and the development set (97 files). The columns labeled as *undecided* contain the values of the text nodes classified as *undecided* by the majority of annotators. The tied annotations are not included here. They comprise 314 text nodes and 12,044 characters for the test set and 424 nodes and 14,043 characters for the development set.

[26] Annotation reliability (AR) is defined in section 2.8.2.

		text nodes			characters		
		good	bad	undec.	good	bad	undec.
test	$AR = 100$	3,182	6,379	144	702,723	102,278	3,811
	$AR > 0$	5,016	11,812	729	879,338	260,406	17,319
dev	$AR = 100$	3,152	7,058	94	572,697	119,066	2,075
	$AR > 0$	5,179	13,051	811	755,022	279,350	23,819

Table 2.2: Annotation statistics for the Canola corpus.

2.9.3 Classification features

This section analyses the classification features used by the jusText algorithm, namely the link density, stop words density and number of tokens. The goal of the analysis is to confirm the intuition that all these features can be effectively used for distinguishing between the boilerplate and the main content.

A simple algorithm has been implemented which splits the document to blocks on the block-level tags (the same way jusText does) and classifies each block either as boilerplate or main content based on given feature and given threshold. For link density, the blocks with the value above the threshold are classified as boilerplate, the others as main content. For stop words density and number of tokens, the values below the threshold indicate boilerplate.

The evaluation has been performed on the Canola development set. The results for both reliable and all annotations are reported in Fig. 2.6. Apparently, both versions of the evaluation reveal the same trends, though absolute values are higher for reliable annotations. The baseline for the dataset is determined by classifying all data as main content. This produces the F_1 and $F_{0.5}$ of 90.81 % and 86.06 % respectively for reliable annotations and 84.43 % and 77.21 % for all annotations. It is easy to see that all three classifiers exceed the baseline with a wide range of thresholds in terms of both F_1 and $F_{0.5}$. This confirms the hypothesis that all three features are effective for boilerplate filtering.

An interesting development can be observed in the link density filtering graphs. The precision is almost constant throughout the graph. It decreases only slightly towards the high thresholds. The recall, on the other hand, grows rapidly for thresholds lower than 0.15. Then it gets close to perfection for $AR = 100$ and to very high values (over 97 %) for $AR > 0$ and the growth almost stops here. This indicates that blocks with the link density above a reasonably set threshold (somewhere between 0.2 and 0.5) could be safely discarded as boilerplate with almost no loss in recall. Most of the lost good blocks would be those which raise some doubts about the correct classification (there is no unanimous agreement of the annotators) and some of them may actually not be annotated correctly. Note that this kind of filtering is the first step which the context-free classification in jusText performs.

In both stop words density filtering and token count filtering the recall is near 100 % for low thresholds. This indicates that, similarly as with link density filtering, the blocks with low stop words density as well as short blocks could be discarded with little loss in recall. On the other hand, as already mentioned above, for short blocks, it is hard to distinguish between main content and boilerplate without using the context. While the token count filtering graph indicates that most short blocks will be boilerplate, we may prefer not to drop them all immediately since the good ones may contain an important context for the surrounding blocks. Discarding all short blocks may have a negative impact on the document fragmentation as discussed later in the evaluation section.

In the stop words density filtering, the recall keeps near maximum up to the

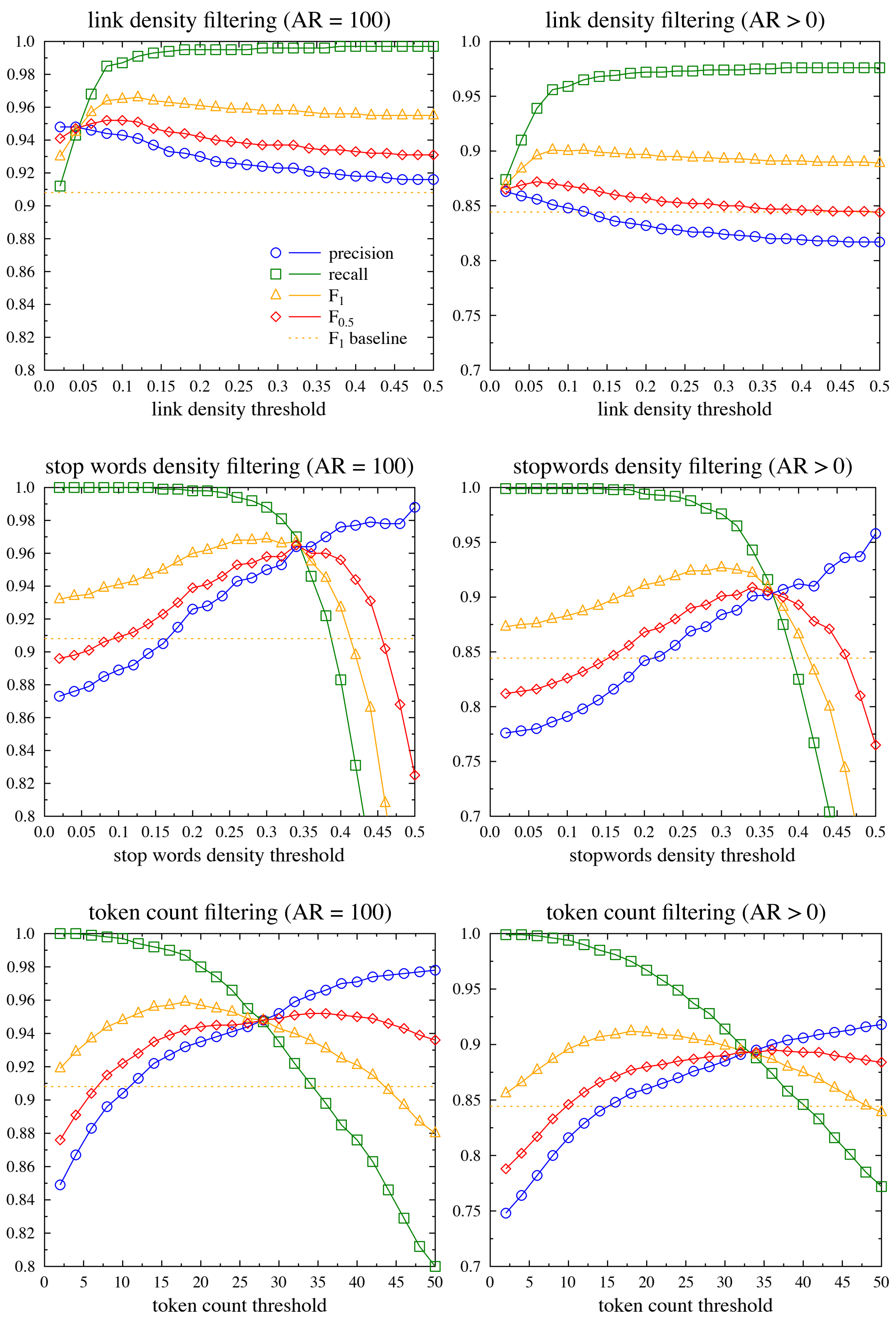


Figure 2.6: Boilerplate filtering using simple features.

	$AR = 100$				$AR > 0$			
classifier	thresh.	F_1	thresh.	$F_{0.5}$	thresh.	F_1	thresh.	$F_{0.5}$
baseline	-	90.81	-	86.06	-	84.43	-	77.21
link density	0.12	95.57	0.08	95.21	0.12	90.11	0.06	87.15
word count	18	95.85	36	95.20	18	91.17	36	89.46
sw density	0.30	96.89	0.34	96.51	0.30	92.75	0.34	90.94
combined	15; 0.25	98.15	20; 0.30	98.18	15; 0.30	94.23	20; 0.30	93.10

Table 2.3: Maximal values of F_1 and $F_{0.5}$ reached by the simple classifiers.

threshold of around 0.3. Then it starts dropping rather rapidly. In the token count filtering, on the other hand, the recall decreases almost linearly with increasing threshold.

Fig. 2.7 shows the results of a combined classifier which utilises both token count and stop words density for boilerplate filtering. Only the results for reliable annotations are reported as the graphs for all annotations show similar trends. It is easy to see that the performance of the combined classifier is far superior to using any of the two features separately. The best results can be achieved by setting the word count threshold between 15 and 20 and the stop words density threshold between 0.25 and 0.30. It is not surprising that the F_1 prefers lower thresholds whereas $F_{0.5}$ prefers higher.

In correspondence with the previous results, setting the stop words ratio threshold to 0.40 or higher causes a high loss in recall, which has a negative impact on both F_1 and $F_{0.5}$. On the other hand, it can be seen that setting the threshold to 0.40 results in an almost perfect precision for the token count of 40 (or higher). Though the recall is too low here to be practical for boilerplate filtering, it is still high enough (79 %) for identifying most of the good blocks without making almost any mistakes. This is very useful for the jusText algorithm which needs to classify good blocks with high confidence in the context-free classification phase. The values of 0.40 and 40 therefore seem good candidates for setting the values of the STOPWORDS_HIGH and LENGTH_HIGH parameters respectively.

Table 2.3 shows the maximal values of F_1 and $F_{0.5}$ reached by each of the simple classifiers and the thresholds which produced these maximal values.

2.9.4 Block coherence

The previous section presents results on text blocks created by segmenting the input documents at block level HTML tags. The same segmentation method is used by the jusText algorithm. One disadvantage of this approach is that some blocks may contain a mixture of main content and boilerplate. The classification of such blocks can never be perfectly correct. Intuition suggests that this is not a serious problem since these blocks should be rather rare. A simple test has been performed to verify this expectation.

18,126 text blocks have been extracted from 197 files in the Canola development and test sets. Then the blocks which contained at least one text node annotated as good and at least one text node annotated as bad have been identified and counted. This has been done for reliable annotations as well as for all annotations. For reliable annotations, only 6 mixed blocks have been found. All these blocks contained a “Back to Top” link at the end, correctly annotated as boilerplate, while the rest of the blocks contained good text. Several similar cases have been found among the 218 blocks identified as mixed by using all annotations. The trailing boilerplate links could be

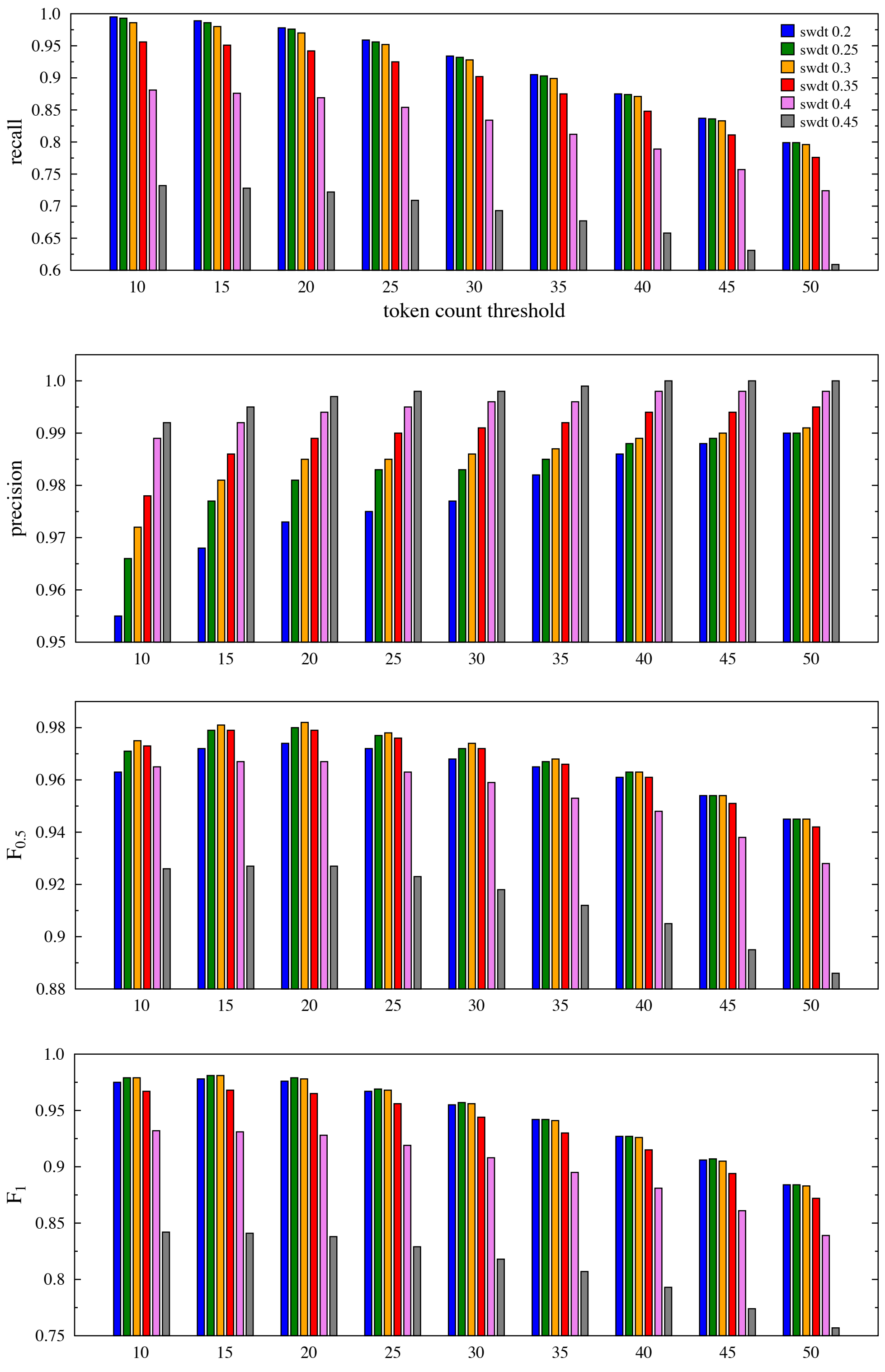


Figure 2.7: Boilerplate filtering based on token count and stop words density.

easily removed by using a simple heuristic, but since the 218 blocks constitute only 1.2 % of all blocks, the problem can probably be safely ignored. Moreover, most of the 218 blocks seem to be identified as mixed due to incorrect annotations. For example, the document with ID 790 contains the following post to a discussion forum about computer games:

> I really have no idea how to play this one, but I do know I'm planting trees! Weeeeeeeeeeeeee! Oh wait, we did the wee thing yesterday, sorry. (weeeeeee?)

Though this is a single post, the last token "(weeeeeee?)" is formatted with a smaller font and therefore forms a separate text node. The majority of annotators classified this text node as bad while the rest of the post was classified as good. Though the text is rather awkward, it clearly contains full sentences and forms a coherent unit. Therefore, all of it should be kept as main content. Many other "mixed" blocks can be found which are difficult to annotate rather than incoherent.

2.9.5 jusText parameter tuning

justText algorithm is driven by five input parameters. In order to determine which values of the parameters are most appropriate and how changing each parameter value influences the results, a set of tests have been performed on the Canola development set. First, the parameters have been sorted by their expected impact on the results – the parameters with the lowest expected impact first and the ones with the highest impact last:

1. MAX_LINK_DENSITY
2. LENGTH_LOW
3. STOPWORDS_LOW
4. LENGTH_HIGH
5. STOPWORDS_HIGH

The initial values have been set partially based on the results presented in the previous section and partially based on author's intuition. Then, the value of the first parameter has been determined which maximises the $F_{0.5}$ score while keeping the values of the other parameters fixed. The parameter has then been set to the best value and the best value for the second parameter has been determined in the same way. This process has been repeated two times for all parameters. Since the second iteration produced only minor changes in the parameter values, no further iterations have been performed. When finding the best value for a given parameter, the $F_{0.5}$ scores for both reliable annotations and all annotations have been taken into account. In most cases, both evaluation methods preferred the same value of the parameter. Where two different values have been preferred, a value somewhere in-between has been selected. This typically resulted in a negligible loss in any of the two maximal $F_{0.5}$ scores.

Table 2.4 shows the development of the parameter values and the $F_{0.5}$ scores produced by each combination of parameters. In the second step of the second iteration the process branched. The graph of the $F_{0.5}$ score based on the value of the LENGTH_LOW parameter showed three local maxima – for the value of 10, 0 and 18. This can be seen in Fig. 2.8. As the local maximum for LENGTH_LOW=18 was slightly lower than the other two local maxima, it was not investigated any

	1.1	1.2	1.3	1.4	1.5	2.1	2.2a	2.3a	2.4a	2.5a
MAX_LINK_DENSITY	**0.20**	0.20	0.20	0.20	0.20	**0.20**	0.20	0.20	0.20	0.20
LENGTH_LOW	10	**10**	10	10	10	10	**10**	10	10	10
STOPWORDS_LOW	0.20	0.20	**0.18**	0.18	0.18	0.18	0.18	**0.30**	0.30	0.30
LENGTH_HIGH	40	40	40	**30**	30	30	30	30	**30**	30
STOPWORDS_HIGH	0.40	0.40	0.40	0.40	**0.32**	0.32	0.32	0.32	0.32	**0.32**
$F_{0.5}$ / $AR = 100$ [%]	98.83	98.83	98.84	99.19	99.26	99.26	99.26	99.19	99.19	99.19
$F_{0.5}$ / $AR > 0$ [%]	93.19	93.19	93.21	93.62	93.87	93.87	93.87	94.10	94.10	94.10

							2.2b	2.3b	2.4b	2.5b
MAX_LINK_DENSITY							0.20	0.20	0.20	0.20
LENGTH_LOW							**0**	0	0	0
STOPWORDS_LOW							0.18	**0.30**	0.30	0.30
LENGTH_HIGH							30	30	**26**	26
STOPWORDS_HIGH							0.32	0.32	0.32	**0.32**
$F_{0.5}$ / $AR = 100$ [%]							99.31	99.33	99.41	99.41
$F_{0.5}$ / $AR > 0$ [%]							94.33	94.42	94.44	94.44

Table 2.4: Optimising jusText parameters.

further. The best scores were produced by LENGTH_LOW=0. Additionally, the next steps in the iteration of this branch improved the score even further. The effect of setting LENGTH_LOW=0 is that no *short* blocks are created by the context-free classification. Though this seems to have a positive impact on precision, there is also a negative impact on document fragmentation. Since the improvement over the $F_{0.5}$ score achieved by the other branch (for LENGTH_LOW=10) is not significant, both final sets of parameters are evaluated on the test set and compared with the other algorithms.

An interesting observation is that the value of STOPWORDS_LOW is nearly equivalent to the value of STOPWORDS_HIGH in both final sets of parameters. This suggests that a single threshold for the stop words density could be sufficient for the algorithm. This would reduce the number of parameters by one and simplify the context-free classification. All blocks, except the *short* ones, would be classified as *bad* if they did not reach the stop words density threshold or as *near-good* or *good* otherwise (depending on their length).

The influence of each parameter on precision and recall is shown in Fig. 2.8. The figure contains five pairs of graphs, one for each parameter. The parameter settings correspond to the values in Table 2.4 in the columns labeled as 2.1, 2.2a, 2.3a, 2.4a and 2.5a respectively (light-grey background in the table). The bold value indicates the variable in each of the graphs.

2.9.6 Canola dataset

For the evaluation on the Canola corpus, the BTE and boilerpipe DefaultExtractor algorithms have been implemented in Python with the lxml library in a way so that they included the predicted class of each text node in the output. For DefaultExtractor this was easy to achieve since it segments the input to blocks which consist of one or more text nodes. In order to differentiate from the original Java implementation, the lxml based implementation is referred to as DefaultExtractor$_{lxml}$ in the following (or DefExt$_{lxml}$ in figures and tables where shorter name is needed).

The BTE does not operate at the level of text nodes. It identifies the main content as a continuous block of tokens which maximises its objective function. Still it is easy to see that the best continuous block will always consist of whole text nodes. If one of the text nodes at the boundaries was trimmed the objective function would not be maximal.

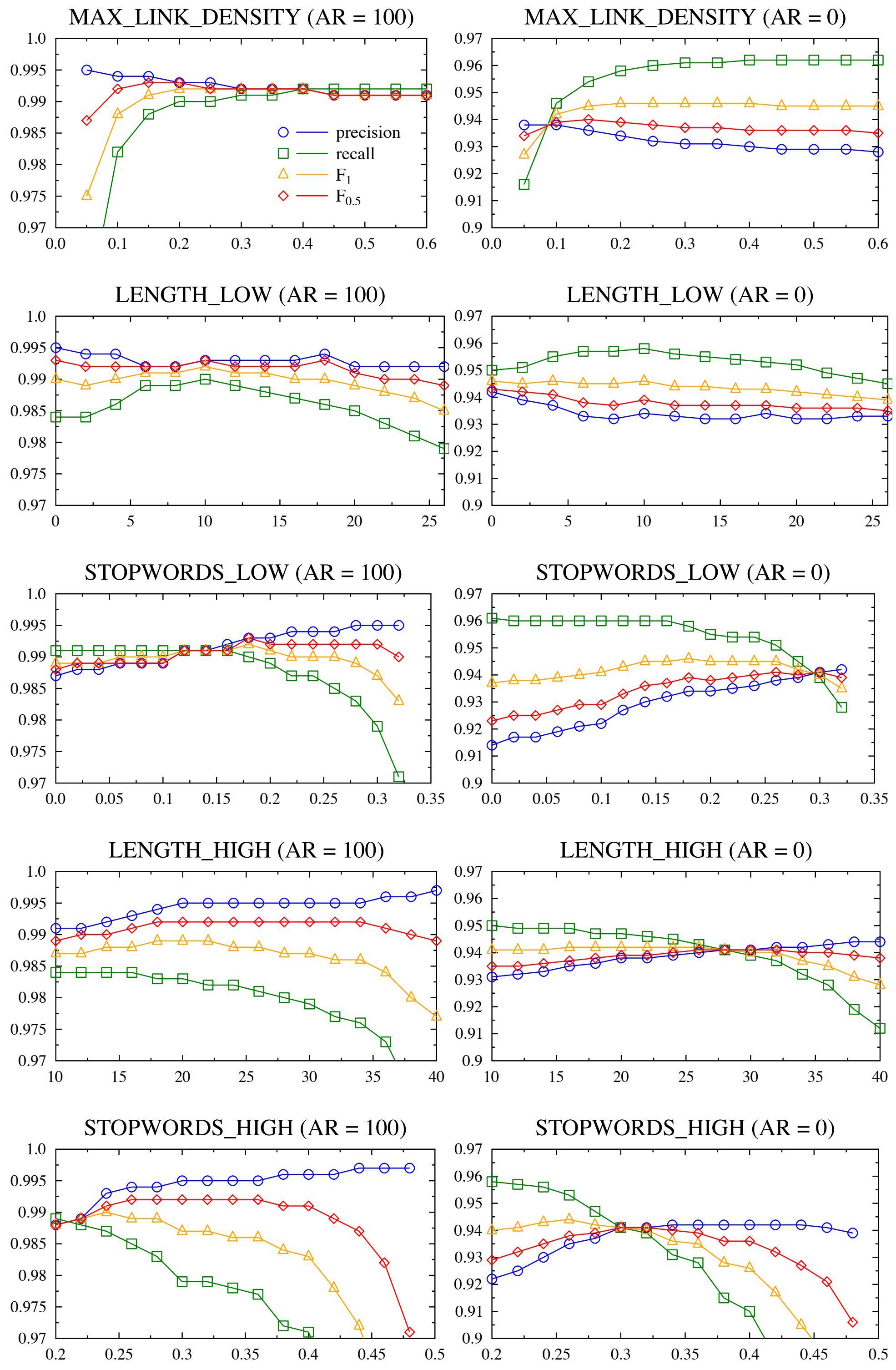


Figure 2.8: Influence of jusText parameters on precision and recall.

	$AR = 100$				$AR > 0$			
	P	R	F_1	$F_{0.5}$	P	R	F_1	$F_{0.5}$
baseline	86.67	100.00	92.86	89.04	76.48	100.00	86.67	80.26
tc_{20}-$\text{sw}_{0.30}$-filter	98.97	96.40	97.66	98.44	**95.78**	93.58	94.67	**95.34**
BTE	94.93	**99.37**	97.10	95.79	84.04	**98.29**	90.61	86.55
DefExt_{lxml}	97.35	94.75	96.03	96.82	90.86	93.05	91.94	91.29
Victor_{Canola}	97.20	97.29	97.24	97.21	94.48	94.76	94.62	94.53
$\text{Victor}_{CleanEval}$	93.13	98.74	95.85	94.20	87.04	97.75	92.08	88.99
$\text{Victor}_{CleanEval2}$	93.10	98.72	95.83	94.18	86.45	98.14	91.93	88.56
jusText_A	**99.41**	96.99	98.18	98.91	95.13	94.36	94.74	94.97
jusText_B	99.32	97.39	**98.34**	**98.92**	95.44	94.64	**95.04**	95.28

Table 2.5: DOM text node level evaluation on the Canola corpus.

Victor classifies each text node individually. However, it only outputs cleaned plain texts. Fortunately, as the source code is well structured, it was not difficult to analyse it and implement a custom output function which produces the predicted class for each text node.

The format of the training data used by Victor is HTML with annotations. Each text node is enclosed in a `<span>` tag which indicates the annotation class as the value of the `class` attribute. As this is very similar to the annotation format used in Canola, it was easy to convert the annotated pages from the development set so that they could be used for training Victor. This new model (Victor_{Canola}) was trained to use only two classes (main content and boilerplate). The text nodes annotated as *undecided* and those where the majority voting is tied were not used for training. The relevant `<kw>` tags were not converted to `<span>` tags. Since Victor ignores the contents of unknown HTML tags during training, leaving the `<kw>` tags prevented Victor from complaining about unannotated text nodes.

The original model ($\text{Victor}_{CleanEval}$) trained for CleanEval was also tested. As this model predicts one of five classes (header, paragraph, list, continuation and other), the output function was adjusted to print any of the former four classes as the main content class and the other class as boilerplate. Since the Victor package also contains the training data used for CleanEval, the main content classes have been merged into one in the annotations and the modified data (containing only two different classes) was used for training another model ($\text{Victor}_{CleanEval2}$).

jusText was tested with two sets of parameters which came up as optimal from the parameter tuning on the development set. jusText_A corresponds to the configuration specified in the upper part of the rightmost column of Table 2.4. The lower part of the same column represents the parameter values for jusText_B.

The combined filter based on token count and stop words density is also included in the evaluation as tc_{20}-$\text{sw}_{0.30}$-filter. The thresholds which produced the best $F_{0.5}$ score on the development set are used.

The baseline is represented by an algorithm which classifies all data as main content. The results of the evaluation are presented in Table 2.5 and in Fig. 2.9.

It can be seen that both versions of jusText outperform all other algorithms in terms of both F_1 and $F_{0.5}$. They also achieve very high values of precision. Especially for the reliable annotations the precision is nearly perfect. Surprisingly, equally good results are achieved by the simple tc_{20}-$\text{sw}_{0.30}$-filter.

The Victor's models trained on the original dataset performed rather poorly on this collection. It is also quite interesting that re-training the model on two merged classes brought almost no difference in performance. The model trained on the Canola

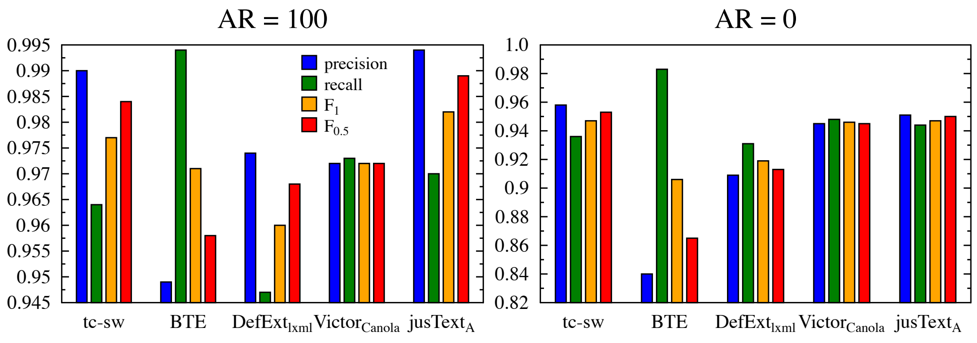


Figure 2.9: Text node level scores for selected algorithms.

development set, on the other hand, achieved very good scores. This may be due to two reasons. First, more training data was provided for Victor_{Canola} (97 documents) than for $\text{Victor}_{CleanEval}$ (51 documents). Second, and probably more importantly, there are minor differences between Canola and CleanEval datasets in the definition of boilerplate (see section 2.2). As CleanEval's view of boilerplate is more conservative, it is not surprising that the CleanEval models achieve lower precision but they on the other hand achieve slightly higher recall.

It is interesting that Victor_{Canola} stays behind jusText on reliable annotations, but it catches up on all annotations. This indicates that it performs very well in average, but it misclassifies quite a lot of text nodes which are easy to classify. A possible reason is that the model has been trained on all annotations rather than only on the reliable ones.

The BTE algorithm achieves the best recall out of all algorithms. The precision is rather poor though. This makes the algorithm a good choice for application in information retrieval where recall will typically be preferred over precision. However, it may be less appropriate for cleaning data for Web corpora where high precision is the key factor. This is an important finding since BTE has been widely used for exactly this application [3, 4, 5, 59, 21].

The $\text{DefaultExtractor}_{lxml}$ achieves average results in terms of both precision and recall. Still the performance is quite impressive considering that the algorithm is extremely simple and requires no language resources (such as a stop list).

NCLEANER could not be evaluated at the level of text nodes since it would be hard to reengineer it so that it produced a classification decision for each text node. In order to compare it with the other algorithms, the cleaneval.py[27] script created by Evert [19] has been used. The script evaluates a cleaned plain text against the plain text of the cleaned gold standard in a similar way as the CleanEval evaluation script. However, rather than using the minimum edit distance, it computes the word level precision and recall from a word level alignment of the cleaned text with the gold standard. This makes the results easier to understand. The scores are weighted by lengths in words, which should produce similar results as weighting by lengths in characters and the scores should therefore be comparable with the results at the text nodes level. Another advantage of the cleaneval.py is that it runs much faster than the original CleanEval evaluation script and requires less computational resources.

In order to create a gold standard in a plain text format it was necessary to decide for each text node whether it should be kept or discarded. This is a problem for nodes annotated as *undecided* and for tied annotations. Therefore, two gold

[27] `http://sourceforge.net/projects/webascorpus/files/CleanEval/cleaneval.py-1.0/`

	with undecided				without undecided			
	P	R	F_1	$F_{0.5}$	P	R	F_1	$F_{0.5}$
baseline	77.41	100.00	87.27	81.07	76.00	100.00	86.36	79.83
tc$_{20}$-sw$_{0.30}$-filter	**95.32**	92.48	93.88	**94.74**	**94.96**	93.85	94.40	**94.74**
BTE	85.40	**97.98**	91.26	87.65	84.10	**98.27**	90.63	86.60
DefExt$_{lxml}$	90.66	92.45	91.55	91.01	89.57	93.04	91.27	90.24
DefExt	89.96	92.78	91.35	90.51	88.88	93.38	91.08	89.74
CanolaExt	89.96	96.71	93.22	91.23	89.06	97.52	93.1	90.63
ArticleExt	95.26	77.55	85.50	91.10	94.09	78.02	85.31	90.37
Victor$_{Canola}$	93.92	93.67	93.79	93.87	93.34	94.81	94.07	93.63
Victor$_{CleanEval}$	86.93	97.55	91.94	88.86	85.60	97.84	91.31	87.80
Victor$_{CleanEval2}$	86.41	97.86	91.78	88.48	85.11	98.18	91.18	87.44
NCLEANER	85.90	94.79	90.13	87.54	85.07	95.62	90.04	86.99
jusText$_A$	94.04	95.17	**94.60**	94.26	93.28	96.15	**94.70**	93.84
jusText$_B$	93.88	94.90	94.39	94.08	93.03	95.79	94.39	93.57

Table 2.6: Word level evaluation on the Canola corpus.

standards have been created, one with the undecided nodes included and one without the undecided nodes. Algorithms are evaluated against both gold standards. As there are not so many undecided nodes in the dataset, the differences between scores are not significant.

The cleaneval.py script was run with the `-a` flag, which removes non-ASCII characters from both the evaluated output and the gold standard before comparing them. This is to prevent misalignments which are due to problems with character encoding. NCLEANER adds a simple markup to its output along the lines of the CleanEval Text and Markup cleaning task. Since there was no markup in the gold standard, NCLEANER received lower scores from the evaluation script. Therefore, the markup has been stripped from NCLEANER's output before the final evaluation to make it fair. Table 2.6 shows the results. Java implementations of the boilerpipe algorithms—DefaultExtractor (DefExt), ArticleExtractor (ArticleExt) and CanolaExtractor (CanolaExt)—have also been included in this evaluation.

It can be seen that the scores are very close to the ones for $AR > 0$ in Table 2.5. This indicates that the evaluation method used by cleaneval.py is as reliable as creating an exact alignment with the text nodes of the gold standard. Therefore, as long as a gold standard is available in the form of optimally cleaned data (in a plain text format), cleaneval.py should be the preferred evaluation method since aligning the text nodes introduces technical problems which are difficult to overcome. Still, where multiple annotations are available, it is very useful to apply an evaluation method which takes the agreement of the annotators into account since it can then be seen whether mismatches of the output of a cleaning algorithm with the gold standard are due to poor performance or due to unreliable annotations. This is especially important for development as the weak spots of an algorithm can be easily recognised by evaluating it against reliable annotations.

NCLEANER does not perform very well at the Canola dataset. It achieves low precision scores and at the same time the recall is not high above average. The F_1 and $F_{0.5}$ values are the lowest of all tested algorithms.

The lxml based Python implementation of the boilerpipe DefaultExtractor performs slightly better than the original Java implementation, probably thanks to the robustness of the lxml parser. The CanolaExtractor improves over DefaultExtractor significantly in terms of recall without any loss in precision. Note, however, that

	jusText$_A$	jusText$_B$	jusText$_C$	jusText$_D$
MAX_LINK_DENSITY	0.20	0.20	0.20	0.20
LENGTH_LOW	10	0	10	10
STOPWORDS_LOW	0.30	0.30	0.25	0.20
LENGTH_HIGH	30	26	25	20
STOPWORDS_HIGH	0.32	0.32	0.25	0.25

Table 2.7: jusText configurations.

this version of the algorithm has been trained on the full Canola dataset (including the test set). Though this probably does not constitute a major problem since the final decision tree rules are fairly general, a minor bias in the results may apply. The ArticleExtractor achieves very good precision, but the recall is poor.

2.9.7 CleanEval dataset

The evaluation on the CleanEval dataset was also done with the cleaneval.py script in exactly the same way as on the Canola corpus. The gold standard is available here in a plain text format with simple markup. I stripped the markup before the evaluation. In order to get a baseline, the original HTML files have been converted to plain text with lynx[28].

Two files numbered 103 and 250 have been removed from the dataset as the manual cleanup contained incorrect data. The files have been identified by checking the recall scores of the lynx, which were very low here. Note that the micro-averaged recall of the baseline is still not 100 %. This has already been observed by Evert [19] who reported recall of only 95.15 %. The problem has been attributed to using a different tool to lynx for generating the text dumps which served as a basis for creating the gold standard. This seems a logical explanation for receiving a recall below 100 %. Still, the recall is significantly higher in my experiments than previously reported. This might be due to removing the two erroneous file or due to using more appropriate lynx parameters.

It turned out that the algorithms which performed well on Canola achieve very low recall on CleanEval, which results in poor F scores. Therefore, two more configurations of jusText (jusText$_C$ and jusText$_D$) have been tested with the parameters set for a higher expected recall. The parameter values have been derived from the tests on the Canola dataset without any attempts for fine-tuning. Table 2.7 gives an overview of the values of parameters for all used jusText configurations.

In order to test whether more training data helps Victor to improve scores, the full Canola dataset has been used to train another model (Victor$_{CanolaFull}$). Table 2.8 shows the results of the evaluation.

It is apparent that the results are completely different from the results on Canola. First thing to notice is the much higher baseline. While on Canola the baseline algorithm achieves the precision of around 76 %, it is more than 88 % on CleanEval. Clearly, the CleanEval dataset contains much less data annotated as boilerplate. This is mainly due to the CleanEval's more conservative understanding of boilerplate. Additionally, some gold standard files (e.g. 114, 180, 250) contain long lists of links, which have not been cleaned as boilerplate. The problem might be that the annotation guidelines do not make it completely clear whether these lists should be cleaned or not. On one hand, they certainly qualify as "internal and external link lists". On the other hand, where the lists of links are included in the main part of

[28] `http://lynx.browser.org/`; `lynx -dump -pseudo_inlines -nolist -nonumbers`

	P	R	F_1	$F_{0.5}$
baseline (lynx)	88.55	99.33	93.63	90.51
tc_{20}-$\text{sw}_{0.30}$-filter	96.12	86.20	90.89	93.96
BTE	93.61	**96.03**	**94.81**	94.08
DefExt_{lxml}	95.49	91.52	93.46	94.67
DefExt	93.32	90.90	95.01	87.13
CanolaExt	93.48	92.37	94.24	90.57
ArticleExt	88.77	79.76	96.01	68.21
Victor_{Canola}	96.02	88.01	91.84	94.30
$\text{Victor}_{CanolaFull}$	95.92	87.82	91.69	94.18
$\text{Victor}_{CleanEval}$	94.88	93.95	94.41	94.69
$\text{Victor}_{CleanEval2}$	94.70	94.13	94.41	94.59
NCLEANER	95.43	92.53	93.96	94.84
jusText_A	**96.70**	88.03	92.16	94.83
jusText_B	96.66	88.81	92.57	94.98
jusText_C	96.39	91.11	93.68	95.29
jusText_D	96.07	92.41	94.21	**95.31**

Table 2.8: Boilerplate cleaning results on CleanEval.

the page rather than in the template part, they are not exactly “extraneous to the proper, coherent contents of the page”. At any rate, most of the evaluated algorithms clean such lists as boilerplate and lose extra points in recall.

It is easy to see that the CleanEval dataset prefers algorithms which are balanced towards good recall performance. Ultimately, keeping all data is a fairly good strategy. This explains why BTE achieves top F_1 score here though it performs rather poorly on Canola. Victor trained for CleanEval is the second best algorithm in terms of F_1, but it outperforms BTE in terms of $F_{0.5}$ as it achieves higher precision while keeping the recall reasonable. Interestingly, NCLEANER gets higher precision than $\text{Victor}_{CleanEval}$ here, though the opposite is true for Canola. Another remarkable observation is that $\text{Victor}_{CanolaFull}$ is slightly behind Victor_{Canola} rather than ahead of it. This indicates that 97 annotated documents provide sufficient amount of training data for Victor.

The algorithms which achieve top scores on Canola (jusText_A, jusText_B, Victor_{Canola}, tc_{20}-$\text{sw}_{0.30}$-filter) get very low recall values on CleanEval. They seem to “overclean” the dataset. On the other hand, all four algorithms achieve precision scores over 96 % (a value that is not reached by any of the remaining algorithms) and thus they catch up very well in terms of $F_{0.5}$ scores. The results of jusText_C and jusText_D show that by lowering the thresholds, a more favourable trade-off between precision and recall can be achieved which improves the F_1 close to the top scores and creates top $F_{0.5}$ scores. This indicates that the algorithm is fairly flexible.

2.9.8 L3S-GN1 dataset

The evaluation on the L3S-GN1 dataset also used the cleaneval.py script and the procedure was identical to evaluating on Canola and CleanEval. The gold standard has been created by extracting the text nodes annotated as one of the five main content classes and saving them to a plain text file. $\text{Victor}_{CanolaFull}$ and $\text{Victor}_{CleanEval2}$ have not been included in this evaluation since the previous results suggest that their performance is almost identical to Victor_{Canola} and $\text{Victor}_{CleanEval}$ respectively. The results are presented in Table 2.9.

	P	R	F_1	$F_{0.5}$
baseline	62.69	100.00	77.07	67.75
tc$_{20}$-sw$_{0.30}$-filter	92.01	79.99	85.58	89.33
BTE	89.41	**94.68**	91.97	90.42
DefExt$_{lxml}$	94.25	91.25	**92.73**	93.63
DefExt	93.91	91.04	92.45	93.32
CanolaExt	87.86	91.14	89.47	88.50
ArticleExt	**97.77**	80.37	88.22	**93.71**
Victor$_{Canola}$	95.04	81.84	87.95	92.07
Victor$_{CleanEval}$	89.37	92.72	91.01	90.02
NCLEANER	80.24	91.90	85.67	82.33
jusText$_A$	95.88	84.03	89.56	93.25
jusText$_B$	95.45	85.31	90.10	93.23
jusText$_C$	94.81	89.18	91.91	93.63
jusText$_D$	93.66	90.74	92.17	93.06

Table 2.9: Boilerplate cleaning results on L3S-GN1.

Note that the baseline is lower for L3S-GN1 compared to the previous two data sets. This indicates that the data contains more boilerplate content.

The ArticleExtractor, which is trained specifically for processing news articles, achieves the highest precision here. On the other hand, similarly as on the other datasets, recall is rather low. Unlike Canola, the tc$_{20}$-sw$_{0.30}$-filter algorithm performs worse here than all the variants of jusText. This suggests that the simple heuristic is not sufficient for general use and the additional rules employed by jusText are well justified.

Again, it can be seen that lowering jusText thresholds trades precision for recall nicely without a significant loss in the overall performance. The $F_{0.5}$ stays almost constant for all sets of input parameters and on par with the top scoring algorithms.

2.9.9 Language independence

The evaluation has been done on English datasets only. Unfortunately, no annotated data is available for other languages, except for the Chinese part of CleanEval. Since none of the evaluated algorithms would work correctly on Chinese, this dataset has not been used. Indeed, though some of the tested algorithms, such as BTE, boilerpipe or Victor, are nearly language independent, they still need to be able to split the text to words in order to compute the length of a block, the link density or the tag density. While for many languages proper tokenisation can be quite accurately approximated by splitting the text at white spaces, this does not work for languages such as Chinese or Japanese which do not insert spaces between words. Additionally, the number of words needed for expressing the same thing may vary among languages. Therefore, the thresholds derived from word count which work well for one language may not work so well for other language. Still, at least within the European languages the differences should not be so significant as to constitute a problem. Thus, the above-mentioned algorithms can be expected to work well on these languages.

The most problematic algorithm, with respect to language independence, is NCLEANER. Since it uses n-gram language models to distinguish between main content and boilerplate, it requires separate training data for each language. Though the author claims that NCLEANER works reasonably well even with minimal amount of manually cleaned training data, it can be argued that preparing the data is not

	$AR = 100$				$AR > 0$			
	P	R	F_1	$F_{0.5}$	P	R	F_1	$F_{0.5}$
jusText$_{default}$	99.41	96.99	98.18	98.91	95.13	94.36	94.74	94.97
jusText$_{BNC100}$	99.49	96.26	97.85	98.82	96.21	93.32	94.75	95.62
jusText$_{BNC500}$	98.92	97.76	98.34	98.69	94.28	95.62	94.95	94.54

Table 2.10: Using jusText with different stop lists.

worth the effort since the algorithm does not perform as well as some of the other algorithms which can be applied to most languages immediately.

jusText makes use of a list of functional words (stop list) for a given language. This, of course, also makes it language dependent. However, a proper stop list can be approximated by taking a list of the most frequent words in the language. Such list can be easily derived from any reasonably large language sample (i.e. any longish document or a small corpus). Text samples for many languages are easy to find on the Web, e.g. on Wikipedia. In the Corpus Factory project [38] we have successfully used Wikipedia for creating lists of medium frequency words for various languages.

In order to evaluate how sensitive jusText is to the contents of the stop list, a simple test has been performed. In all the previous experiments, the list of functional words downloaded from `armandbrahaj.blog.al`[29] was used. The list contains 319 words. For the experiment, the 100 and 500 most frequent words have been taken from the British National Corpus. jusText has then been evaluated on the Canola test set using the parameter values of jusText$_A$ with each stop list. Results are presented in Table 2.10.

It can be seen that using the most frequent words instead of the functional words works very well. Of course, there is a significant overlap in the stop lists as the functional words are very frequent. Taking the top 100 BNC words slightly improves precision at the cost of recall. Taking the top 500 words, on the other hand, has exactly the opposite effect. It can be assumed that if around 300 most frequent words have been used (roughly the size of the default stop list), similar results as with the default stop list would have been achieved.

The base versions of the stop lists only contained lowercase letters. Before using them with jusText, each word has been duplicated with the first letter capitalised (e.g. both *you* and *You* were contained in the final stop list). Matching against the stop list is then case sensitive. That way, the stop words at the beginning of a sentence are matched, but words in all-caps texts are not.

2.9.10 Document fragmentation

Removing boilerplate may leave gaps in the cleaned texts. Consequently, it may be difficult to understand some parts of a cleaned text as the context information is missing. This may be especially problematic if the cleaning algorithm generates false negatives, i.e. it cleans good text as boilerplate.

For example, the document 694 in Canola contains the following three consequent text nodes:

1. A few days ago, I mentioned that I'd begun playing on one of the older text-based MMOGs, Gemstone IV. Many of you

2. commented on the Loading forums

[29] `http://armandbrahaj.blog.al/2009/04/14/list-of-english-stop-words/`

	avg fragment length	median fragment length	avg fragments per document
perfect cleaning	1315.1	279	6.98
tc_{20}-$sw_{0.30}$-filter	1059.8	385	8.27
BTE	11095.6	7611	1.00
$DefExt_{lxml}$	671.2	68	13.81
$Victor_{Canola}$	637.9	126	14.13
$jusText_A$	2304.3	794	3.88
$jusText_B$	1609.7	630	5.54

Table 2.11: Document fragmentation (Canola test set).

3. about your own old experiences with text and how they compared with my own. ...

The second text node is a link and thus the sentence gets split into three nodes here. It turns out that $Victor_{Canola}$ removes the second text node as boilerplate. The fact that the node is rather short and forms a link probably takes its "boilerplateness" score too high. This is clearly a mistake which results in a minor loss in recall. However, the mistake does not only cause a minor data loss. The surrounding nodes of the incorrectly cleaned node are also affected as an important part of a sentence is removed and the remaining part alone is not syntactically correct and does not make much sense. Therefore, not only good data has been lost, but also noise has been introduced into the corpus.

This problem can be overcome to a high extent by using a more coarse-grained segmentation, e.g. by splitting the input only at block level HTML tags rather than taking each text node as a separate unit. Still, even at the level of such text blocks a loss of context may occur, such as removing short utterances from a dialogue.

In order to estimate possible loss of context, fragmentation of the cleaned texts is measured. A cleaned document is viewed as a set of continuous sequences of good text separated by gaps after removed text (presumably boilerplate). Each such sequence forms a continuous fragment of text. To get an idea about the level of fragmentation, three values are followed:

- Average length of a fragment (in characters, for the whole dataset)
- Median length of a fragment
- Average number of fragments per document

The results for selected algorithms on the Canola test set are shown in Table 2.11. The fragmentation of a perfect cleaning is reported for reference. This is the fragmentation achieved by an imaginary algorithm which removes all blocks annotated as *bad* in the gold standard. (Undecided nodes are included in the output.)

It is not surprising that BTE achieves the best scores. Since it always extracts a single continuous cleaned block from a Web page, no fragmentation occurs. On the other hand, as demonstrated in the previous experiments, this comes at the cost of a rather low precision.

The algorithms which operate at the text nodes level (Victor) or close to it (boilerpipe) suffer from a high fragmentation. This suggests that though both algorithms show a very decent performance in terms of precision and recall, they may tend to output scattered pieces of texts taken out of the context. Such pieces may constitute equally undesirable corpus content as boilerplate. Still it is hard to say how significant the negative impact of this problem is. A detailed analysis of the negative impact of fragmentation would require manual annotation of semantically coherent blocks in the gold standard. At any rate, manual inspection of the output of the two above-mentioned algorithms reveals that the problems with lost context are not rare.

The results of the tc_{20}-$sw_{0.30}$-filter indicate that using a more coarse-grained segmentation helps to decrease the fragmentation significantly. Further improvement can be achieved by smoothing the classification of blocks. Such smoothing happens as a positive side-effect of jusText's context-sensitive classification. It can also be seen that classifying the *short* blocks solely based on the context (jusText$_A$) reduces the fragmentation even more, though of course it comes at the cost of a minor loss in precision.

2.10 Summary

This chapter presented a novel approach to boilerplate cleaning – the jusText algorithm. Comparison with other state-of-art methods on available datasets revealed that the algorithm achieves competitive results. Most notably, it enjoys very high precision scores and at the same time keeps the level of data fragmentation low. This makes the method appropriate for cleaning Web data for text corpora. Is has also been demonstrated that a trade-off between precision and recall can easily be adjusted by modifying the values of jusText input parameters without suffering a major loss in overall performance. This indicates that the algorithm is flexible and can certainly be applied even when high recall is required.

Though the jusText algorithm is not completely language independent, it has been shown that its requirements for language resources are easily satisfied. It only needs a list of approximately 100 most frequent words for a given language.

The evaluation revealed that using simple features, namely token count and stop words density, is very effective for boilerplate cleaning. An algorithm which simply discards text blocks with token count or stop words density below certain thresholds achieves very good results.

An important finding is that the BTE algorithm—widely used for cleaning Web corpus data—achieves much lower precision scores than other algorithms, which makes it a rather inappropriate choice for this application.

Finally, evaluation at the level of DOM text nodes proved technically very difficult due to minor differences between HTML parsers with respect to invalid HTML pages. For future experiments, a more robust evaluation method should be developed. The cleaneval.py script seems a good starting point. It would be useful to extend it in a way which would enable taking multiple annotations of the same data into account.

For convenience, the evaluation results are graphed in Appendix B.

Chapter 3

Duplicate and near-duplicate content

3.1 Introduction

Many texts are present on the Web in multiple instances. Common types of duplicates and near-duplicates include mirrored websites, multiple presentation styles of the same Web page (e.g. one for viewing in a Web browser and one for printing), document revisions, similar or identical news articles at different news sites, quotations of previous posts in online discussion forums, etc. Duplicate content is problematic for many applications of Web data. For instance, imagine a Web search engine which indexes all Web pages without worrying about duplicates. It could then easily happen that the first page of Web search results for a given query would only contain hits in identical documents. Such a search engine would not be very useful. The situation is similar with Web corpora. Here, users might get many duplicate concordance lines when searching in a corpus containing duplicate texts. Moreover, duplicate content may bias results derived from statistical processing of corpus data by artificially inflating frequencies of some words and expressions. Identifying and removing duplicate and near-duplicate texts is therefore essential for using the Web data in text corpora.

When talking about duplicates, it is important to distinguish between *naturally* and *artificially* repeated texts. It is perfectly normal that some language phenomena, such as words, expressions and even full sentences are used repeatedly and independently. Such recurrences are natural and are not considered duplicates (at least in the scope of this work). On the other hand, there are texts which are taken copy-and-paste from other sources (and possibly slightly modified) rather than created independently. Such a repetition is no longer a normal language use. It is extremely unlikely that two different language users would independently produce two completely identical paragraphs or even full documents. When referring to duplicates and near-duplicates, these copy-and-paste recurrences are meant. Some researchers also use the term *co-derivative* texts.

Identifying duplicate and near-duplicate texts is fairly easy for small data collections. Here, we can simply compare each pair of documents (or other units, such as paragraphs). Deciding whether two documents are identical is trivial. It is also not difficult to decide whether two documents are similar (near-duplicate). Many metrics are available for measuring the similarity of documents, such as Levenshtein edit distance [45], Broder's resemblance [12] or cosine similarity of document vector space models[1]. For small data sets, the similarity of each pair of documents can be computed and the pairs with the similarity value above certain threshold reported as near-duplicates. However, the problem gets difficult for large data collections where comparing each pair of documents would be prohibitively expensive. Today, Web corpora comprising billions of words are fairly common. Such corpora are created from millions of Web pages and would require trillions (10^{12}) pairwise document comparisons to be performed.

For finding exact duplicates, document checksums (fingerprints) can be used. The checksum is an output of a hash function, such as MD5 [55]. Two identical documents will always have the same checksum whereas there is typically a high probability for the checksums of two different documents to be different. The advantage of using checksums is that they are shorter than the original documents and it is cheaper to compare them. Moreover, checksums can be easily stored in memory. Duplicated documents can then be identified and removed with a single pass over the data set by probing the checksum of each document against the checksums of previously seen documents.

The situation is more complicated with near-duplicates. With standard hash functions, a change in a single byte of the input changes the output completely. It is therefore not possible to assume that the checksums of similar documents will be similar or even identical. The problem of finding near-duplicated documents has been addressed by many researchers—mostly in the context of Web search engines—by using special fingerprinting techniques. In the following section, an overview of the most popular methods is provided.

3.2 Related work

3.2.1 Broder's shingling algorithm

Broder's algorithm [12] uses a special fingerprinting scheme designed specifically for detecting near-duplicate documents. Each document is represented as a set of shingles (or n-grams), which are sequences of any n consequent words. For example, for $n = 6$, the text "what shall we do with a drunken sailor" contains three shingles:

$$(what, shall, we, do, with, a)$$
$$(shall, we, do, with, a, drunken)$$
$$(we, do, with, a, drunken, sailor)$$

For a document D, S_D is a set of all shingles which occur in D. A document similarity measure called resemblance is then computed as follows:

$$r(A, B) = \frac{|S_A \cap S_B|}{|S_A \cup S_B|}$$

The resemblance takes values in the interval $[0, 1]$. A pair of near-duplicate documents will typically contain many common shingles and only a few different ones. Thus their resemblance should be close to 1. Unrelated documents, on the other hand, will contain no common shingles, except for by chance, and their resemblance should be 0 or near it. It is intuitively clear that high resemblance values are fairly good indicators of duplicate and near-duplicate documents, especially if reasonably long shingles are used and thus the probability of independent recurrence is low. The problem of finding near-duplicate documents can therefore be reduced to finding document pairs with a high resemblance.

[1]`http://en.wikipedia.org/wiki/Vector_space_model`

The core idea of the Broder's algorithm is that it is not necessary to compute the resemblance value exactly in order to decide whether two documents are near-duplicate. It suffices to find out whether their resemblance is above a predefined threshold. For almost identical documents, the resemblance can be accurately approximated by using only small samples of the shingles. This reduces the computational costs significantly. The idea is that if we select a shingle with a specific feature (such as the lowest hash value) from each of two near-duplicate documents, there is a good chance that the same shingle will be selected from both. It can actually be shown that the probability of selecting the same shingle is equivalent to the resemblance of the two documents. Formally, let us compute a hash of each shingle and represent a document as a set of hashes rather than a set of shingles. Thus, $S_D \subseteq \{0, \dots, m-1\}$ where $m = 2^b$ and b is the bit-length of the used hashes[2]. Then

$$\Pr(\min\{S_A\} = \min\{S_B\}) = \frac{|S_A \cap S_B|}{|S_A \cup S_B|} = r(A, B)$$

Additionally, if $\pi \in S_m$ is a permutation which satisfies the *min-wise independence condition* [13] then also

$$\Pr(\min\{\pi(S_A)\} = \min\{\pi(S_B)\}) = r(A, B)$$

See [12] for a proof. By taking a set of t independent permutations $\pi_1, \dots, \pi_t$, a sketch of a document A can be created as

$$\bar{S}_A = (\min\{\pi_1(S_A)\}, \min\{\pi_2(S_A)\}, \dots, \min\{\pi_t(S_A)\})$$

The resemblance of documents A and B can then be estimated by computing how many elements $\bar{S}_A$ and $\bar{S}_B$ have in common:

$$r(A, B) \approx \frac{|\bar{S}_A \cap \bar{S}_B|}{t}$$

The problem is that a fairly large t needs to be used in order to make the estimation accurate.

For example, let us assume we want to find all document pairs with a resemblance higher than 0.5 and mark them as near-duplicated. For a given t, we should expect those pairs of documents whose sketches contain more than $t/2$ equal elements to be near-duplicated. However, if the number of equal elements is close to $t/2$ the classification may be unreliable due to inaccuracy of the estimation of the resemblance, i.e. the document pairs which contain more than $t/2$ equal elements may be accepted as near-duplicates even though their actual resemblance is below 0.5 and vice versa.

The probability of error increases with a decreasing value of t. This is demonstrated in Fig. 3.1. It shows the probability that a pair of documents with a given resemblance will be accepted as a near-duplicate pair (i.e. have more than $t/2$ common sketch elements) for different values of t. This probability is a special case of equation 3.1 (see below) for $s = 1$. It can, for instance, be seen that for $t = 100$, there is a very little chance of misclassifying document pairs with resemblances below 0.4 and above 0.6. For $t = 10$, on the other hand, there is a fairly high risk of misclassifying all document pairs except for those with very low or very high resemblance values.

For fast in-memory processing of large document collections, a compact representation of the documents is needed. Document sketches of 100 or more (64-bit)

[2] According to Broder [12], 64 bits should be sufficient in practice. The probability of hash collision is then negligible and can be ignored.

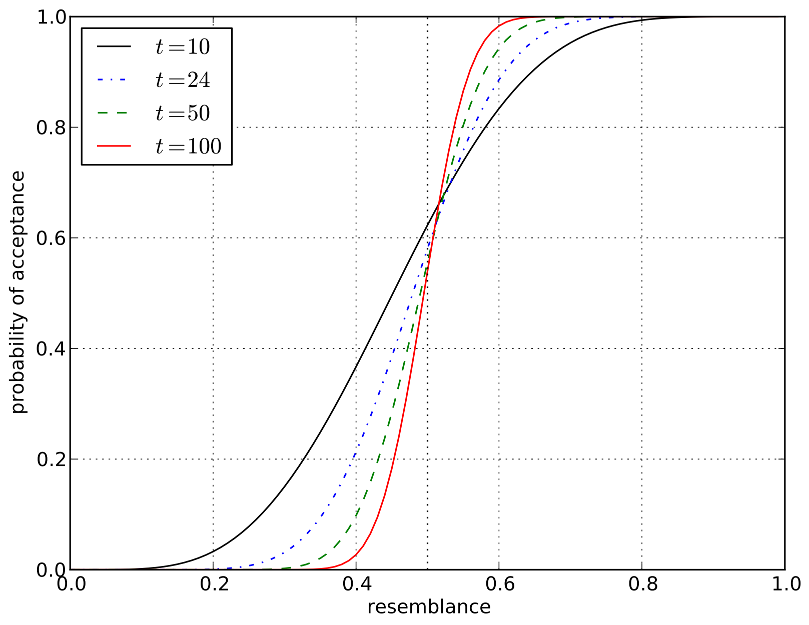


Figure 3.1: Probability of acceptance ($P_{t,1,t/2}$) for different values of t.

elements may not be compact enough. In order to further reduce the size of document representation, Broder's algorithm divides every document sketch into k groups of s elements each. A hash of each group—a *super-shingle*—is then computed and the list of k super-shingles is stored instead of the full sketch. It can be shown that for two documents A and B with a resemblance ρ the probability that they have r or more equal super-shingles is

$$P_{k,s,r} = \sum_{i=r}^{k} \binom{k}{i} \rho^{s \cdot i} (1 - \rho^{s})^{k-i} \tag{3.1}$$

The equation is graphed in Fig. 3.2 with several different sets of parameters. It can be seen that for reasonably selected parameters, a very accurate filtering can be achieved.

Unfortunately, this scheme has been designed only for matching almost identical documents and it does not work well for assessing document pairs with medium resemblance values. Since we are matching whole groups of shingles, it is obvious that for a reasonable chance of getting at least one match, either the documents have to be very similar or we have to sample a lot of groups and thus lose the advantage of compact document representation. As demonstrated in Fig. 3.2, shifting the probability function towards lower resemblance thresholds without losing the sharpness of the filter can only be achieved by increasing the value of the parameter k.

While the Broder's algorithm has desired properties for application in Web search engines where only almost identical pages are of interest[3], it is problematic to use it if a wider range of duplicate content needs to be considered. This issue is further discussed in section 3.3.

[3]The algorithm has been designed for the AltaVista Web search engine.

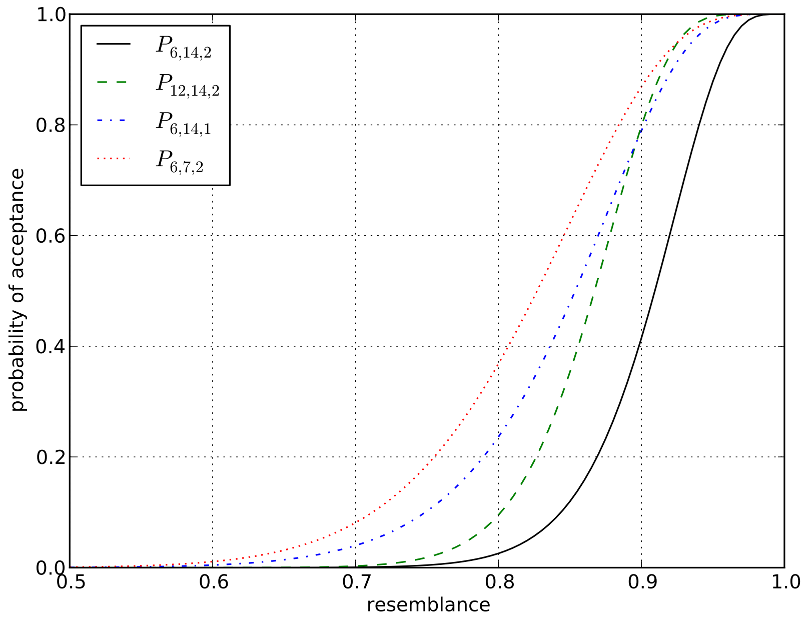


Figure 3.2: Probability of acceptance ($P_{k,s,r}$) for different parameter settings.

Finding near-duplicate document pairs

Once the sketches of documents are created, finding near-duplicate pairs is fairly straightforward. The procedure is the same no matter whether basic sketches (of permuted hashes of shingles) or sketches of super-shingles are used. In both cases, the near-duplicates are identified by testing whether the number of equivalent elements—*features*—in two sketches is above a predefined threshold. First, (feature, doc-ID) pairs are created for all documents and sorted. Then it is easy to find, for each feature, a list of IDs of documents which contain the feature. Each such list is then expanded into pairs of documents known to share given feature: (doc-ID$_1$, doc-ID$_2$). All these pairs are then sorted; the equivalent pairs are merged and their count is remembered. If the count is above a predefined threshold, a near-duplicate document pair is found.

For large collections, this process is only feasible if the number of (doc-ID$_1$, doc-ID$_2$) pairs is reasonable. Thus, it is necessary to select the parameters of the fingerprinting scheme in a way so that it does not generate too many document pairs sharing a feature. Again, this is problematic for low and medium resemblance thresholds (say around 0.5). The lower the resemblance threshold, the more near-duplicate document pairs exist and consequently a higher number of documents sharing a feature has to be generated in order to find the near-duplicate pairs.

3.2.2 Charikar's algorithm

Charikar [16] proposed a hashing scheme which produces similar hash values for similar objects. Though this is an unusual feature for a hash function, it is very useful for identifying near-duplicates. The input to the hash function is a vector (typically in a high dimensional space) and the output is a fixed size bit-string. The hashing is done so that the Hamming distance is small for bit-strings of similar vectors and large for bit-strings of different vectors. The problem of exploiting similarities in a high dimensional vector space can then be reduced to computationally much less expensive problem of finding pairs with a small Hamming distance in a set of (short) bit-strings. Since text documents can be easily represented as vectors (typically as frequency vectors of the contained words), applying the method to identification of near-duplicate documents is straightforward.

The hash of a vector is computed as follows. For each dimension i (typically a word when working with text documents), a random vector $\vec{u}_i = (u_i^1, \ldots, u_i^b)$ is created, where $u_i^j \in \{-1, 1\}$ for any $j \in \{1, \ldots, b\}$. This vector represents a projection of the dimension and is used consistently for any input. An input n-dimensional vector $\vec{v} = (v_1, \ldots, v_n)$ is projected into a b-dimensional space by projecting each of its dimensions as follows:

$$\vec{w} = p_u(\vec{v}) = \sum_{i=1}^{n} v_i \cdot \vec{u}_i$$

The b-dimensional vector $\vec{w} = (w_1, \ldots, w_b)$ is then converted into a b-bit string $s = s_1 \ldots s_b$ by simply converting the vector dimensions with non-negative values to 1 bits and those with negative values to 0 bits, i.e.:

$$s_i = \begin{cases} 1 & w_i \geq 0 \\ 0 & w_i < 0 \end{cases}$$

Hamming distance problem

Though determining the Hamming distance of two bit-strings is a computationally cheap operation, for large data collection, it may still be prohibitively expensive to compare all pairs. For finding the pairs with a small Hamming distance, several efficient methods exist.

Henzinger [29] divided each bit-string into k non-overlapping pieces of equal length and computed the Hamming distance for all pairs of bit-strings which had at least one piece in common. Such pairs can be found easily by generating (bit-string-piece, doc-ID) pairs and sorting them (similarly as described in the previous section). To avoid false positives, this should be done separately for each of the k positions in the original bit-strings (it does not make sense to compare a piece at the beginning of one bit-string with a piece at the end of another). This approach is guaranteed to find all pairs with a Hamming distance of $(k - 1)$ or smaller. Such pairs cannot have k different pieces as each of them would have to differ in at least a single bit and thus the Hamming distance would be at least k.

The value of k has to be reasonably small so that the number of bit-string pairs to be assessed is reduced to a feasible amount.

Manku et al. [47] described a more general approach which indexes bit-strings in a way so that, for a query bit-string q, all bit-strings within some (small) Hamming distance from q can be found efficiently. The method makes use of the fact that if we take a sorted table of 2^d random f-bit strings and look only at the d most significant bits, then most of possible combinations exist and few combinations are duplicated – these d most significant bits work as "almost a counter". Since the table is sorted, it is easy to find all bit-strings which match with a query in d' most significant bits for some integer d'. If the difference between d and d' is small, a small number of matches can be expected. The matched bit-strings can then be compared with the query and those which differ in at most k positions can be identified. This way, all the bit-strings within the Hamming distance of k from the query will be found which differ only in the $(f - d')$ least significant bits. Of course, the bit-strings with differences in the d' most significant bits are missed. This can be fixed by repeating the process with permuted bit-strings.

A list of t permutations $\pi_1, \ldots, \pi_t$ is created. For each permutation π_i, a table T_i is formed by permuting all bit-strings in the original table and sorting them. An integer p_i is associated with each table T_i. The scheme is constructed so that for a query bit-string q, any bit-strings within a Hamming distance k from q can be found in one of the tables by probing the p_i most significant bits of $\pi_i(q)$ in T_i.

Many different setups are possible for given input parameters. There is always a trade-off between the number of used tables and the number of bit-strings generated by probing a table. One sample setup presented in [47] is as follows. Let us have 2^{34} bit-strings of length 64 and $k = 3$. Each table is created by splitting the bit-strings to 4 blocks (16 bits each) and selecting one of them. The remaining 48 bits are again divided into 4 blocks (12 bits each) and again one is selected. This can be done in $4 \cdot 4 = 16$ different ways and leads to creating 16 tables. For each table, the permutation is defined so that the 16 bits of the first selected blocks and the 12 bits of the second selected blocks are moved to the beginning of the bit-string (the 28 most significant bits). These 28 bits are used for probing equally permuted queries. On average, a probe generates $2^{34-28} = 64$ (permuted) bit-strings. Since two bit-strings which differ in at most $k = 3$ positions must have at least one of four blocks identical and the 16 tables together cover any selection of blocks, it is guaranteed that by probing all tables, all bit-strings within the Hamming distance k will be found.

3.2.3 Pugh and Henzinger's algorithm

Pugh and Henzinger [54] patented a fingerprinting method which splits an input document into a fixed number (n) of lists of words. For each word, a hash is computed which selects one of the n lists and the word is inserted into that list. For example, let us have the sentence "eat to live but do not live to eat" and $n = 3$. The lists could be created as follows:

$$(eat, but, eat)$$
$$(to, live, live, to)$$
$$(do, not)$$

Note that the same words always end up in the same list. Also, the word order is preserved within each list.

For each document, n lists of words are created; a hash of each list is computed and the n hash values are stored as a fingerprint of the document. The hash function should be so that the probability of collision is very low and at the same time the hash values are reasonably small in order to reduce the storage requirements as well as computational costs. The fingerprints of two near-duplicate documents can then be expected to contain one or more equal hashes. Note that in a general case, the fingerprints can be built upon other units than words, such as characters, shingles, sentences, etc. For simplicity, words are used throughout.

An alternative list population method uses a separate hash function H_i for each list L_i. A word w is inserted into L_i if $H_i(w) = true$. Thus, a word can be inserted into zero, one, or more lists rather than into exactly one list. Let the number of lists be n and let each hash function H_i return $true$ with a probability p. It can then be shown that the probability $P_{n,p,k}$ that two documents which differ in k words will have at least one common list is

$$P_{n,p,k} = 1 - (1 - (1 - p)^k)^n$$

The values of n and p can be adjusted in order to achieve desired properties of the fingerprinting scheme. The probability function $P_{n,p,k}$ is graphed in Fig. 3.3

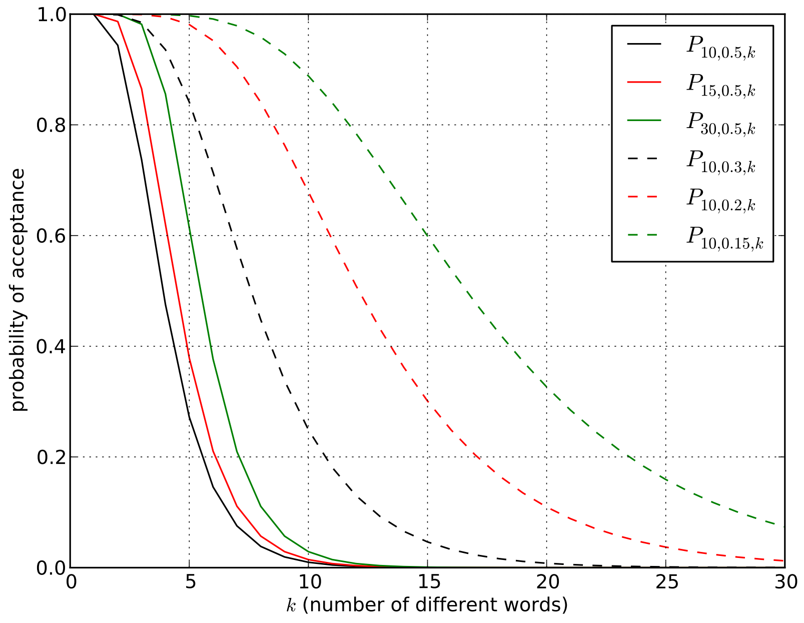


Figure 3.3: Probability of acceptance $P_{n,p,k}$ as a function of the number of different words k.

for various values of n and p, with the k as a variable. It can be seen that for detecting small differences (up to a few words), a small number of lists (n) is sufficient. However, similarly as with Broder's shingling algorithm, detecting more significant differences is problematic. Either the number of lists has to be increased (increasing the size of the fingerprints), or the p has to be decreased, but then the sharpness of the filter deteriorates quickly.

The fingerprinting scheme does not take the document length into account. Two long documents which differ in k words are considered equally similar as two short documents which differ in the same number of words. This contradicts the intuition since the long documents have more content in common and thus should be considered more similar. In order to compensate for the document length, the authors suggest using adaptive hash functions for fingerprinting which return *true* with a lower probability p for longer documents. This increases the probability for pairs of long documents to be identified as near-duplicates.

3.2.4 Other algorithms

A number of other algorithms have been described which are based on similar principles as the Broder's algorithm. Some pieces of each document (such as sequences of characters, words or lines) are selected according to some criteria and their hashes are stored as a fingerprint. Equivalent hashes in two fingerprints then indicate similarities in the related documents.

Shivakumar and Garcia-Molina [60] created a simple fingerprinting scheme for Web documents by storing 32-bit hashes of (i) each two lines and (ii) each four lines of each file.

Manber [46] developed a system called *sif* which finds similar files in a file system. The system extracts all 50-byte substrings s_i from a file and computes their 30-bit hashes $h(s_i)$. The hashes which have the 8 least significant bits set to 0 ($h(s_i)$ mod $256 = 0$) are selected and stored as a fingerprint.

Heintze [28] observed that the problem of the previous approach is that long sequences may occur which do not contain any substrings selected by the algorithm. Eventually, no substring may be selected from a whole file. He proposed two other methods for creating document fingerprints. The first simply selects from each document the 100 hashes with the lowest value. The second method tries to select hashes of substrings which are infrequent in the whole collection. The rationale is that using infrequent substrings reduces the number of false positives when matching the fingerprints. In order to identify the infrequent substrings the frequencies of five letter sequences are computed and the 100 substrings starting with the least frequent five letter prefixes are selected for the fingerprint.

Schleimer et al. [56] proposed an algorithm called *Winnowing* which also addresses the problem of not selecting any substring from long sequences of characters (when using the *0 mod p* criteria as in aforementioned *sif*). At the same time, the number of substrings selected by *Winnowing* is proportional to the length of a document (rather than fixed). This is important for exploiting similarities between documents of different lengths (e.g. if one document is contained in the other). By using fingerprints of a fixed number of substrings (or their hashes), it can easily happen that none (or only few) of the substrings selected from a short document will be selected from a long document even if the short document is fully contained in the long one.

The *Winnowing* algorithm uses a sliding window of t characters. For each sequence of t characters, all substrings of length k are identified and the one with the lowest hash value is selected. If the same substring (at the same position) is selected from more than one window, it is only used once. This happens very often since the consequent windows overlap to a high extent and thus the number of selected substrings is reduced. It can be shown that, on average, the proportion of selected substrings (density) is

$$d = \frac{2}{t - k + 1}$$

At the same time, it is obvious that the algorithm guarantees that at least one substring is selected for each sequence of t characters.

3.2.5 Comparative studies

Henzinger [29] conducted a comparison of Broder's and Charikar's algorithm with respect to identifying near-duplicate Web pages in a very large dataset. The task was to identify the pairs of Web pages which differ only by a session id, a timestamp, an execution time, a message id, a visitor count, a server name, a URL, or any combination of the previous; i.e. the focus was only on almost identical Web pages. The parameters of both algorithms were set so that they produced equally sized fingerprints of 48 bytes per page and so that they identified roughly the same number of pairs of Web pages as near-duplicates (i.e. the algorithms were set up for the same expected recall). Both algorithms were tested on a collection of 1.8 billion Web pages. The pairs identified as near-duplicated were subsampled and manually evaluated as correct (true near-duplicates), incorrect (false positives) or undecided. The algorithms achieved precision of 0.38 (Broder) and 0.5 (Charikar). It turned out that most of the incorrectly identified near-duplicate pairs originate from the same website whereas the near-duplicate pairs at different sites are usually identified correctly. Henzinger also proposed a combined algorithm which achieves precision of 0.79.

Potthast and Stein [52] compared a number of fingerprinting methods on Wikipedia documents. They created a data set of 80 million revisions of 6 million distinct Wikipedia articles. The revisions of the same article were considered near-duplicates. The results suggest that chunking-based methods (Broder's algorithm and alike) perform better than similarity hashing (e.g. Charikar's algorithm) in terms of precision and recall. Note that this is the opposite of Henzinger's results, but of course different kinds of document similarities are exploited here. Unfortunately, the report is rather brief and it does not specify some important details, such as the parameters of the evaluated methods.

3.2.6 Plagiarism detection

Plagiarism is typically defined as presenting other author's ideas or work as one's own original work. A common form of plagiarism is copying parts of other author's texts into one's own document without citing the source. The texts can be copied unchanged or with various modifications which try to mask the plagiarism. Such modifications may include changing the word order, replacing some words with synonyms, rephrasing some parts of the texts, or translating texts from other language. The task of a plagiarism detection software is to find the plagiarised parts in a written text.

The problem of plagiarism detection certainly overlaps with the problem of finding near-duplicate texts to some extent. In both cases, the copy-and-paste texts or text chunks with or without small modifications have to be dealt with. However, there are also significant differences. First, when finding near-duplicates (at least in the context of text corpora) only the texts which are reasonably similar, such as document revisions, are of interest. Some forms of plagiarism, such as translations from other language or re-phrasing other author's ideas in one's own words, can hardly be understood as duplicate corpus data since from the linguistic point of view such texts are unique. In general, plagiarism detection algorithms have to deal with a much wider range of "duplicates" than algorithms for removing near-duplicate data from text corpora.

Also, an anti-plagiarism program has to be able to tell, for each part of a text identified as plagiarised, where exactly does it come from so that the plagiarism can be proved. When removing duplicates from a text corpus, on the other hand, it suffices to know if the currently processed text is contained somewhere in the already processed corpus data. It is not necessary to identify all the pairs of duplicate and near-duplicate documents or paragraphs.

Another difference is in the requirement on incremental processing. A plagiarism detection system will typically need to store a collection of documents in external memory and whenever a new document is added, it should be examined for plagiarism against all already contained documents. For text corpora, on the other hand, it typically suffices to de-duplicate all the contained documents in a single run, since it is not usual that data is added to text corpora, and when it is, it is not on document by document basis. Rather than that, batches of documents would typically be added time to time in which case the full collection can be re-processed each time provided the processing is reasonably fast.

Last but not least, when processing corpus data, it is necessary not only to identify the duplicate text parts but also remove them. The following sections explain that this is not as easy as it may seem at first sight. No similar problem needs to be addressed in the context of plagiarism detection.

As can be seen, though the two problems share some common features, they are different in many aspects. Therefore, this work mentions plagiarism detection only briefly and does not elaborate on it, nor does it try to compare the algorithm for removing near-duplicate data (introduced in the next) with existing plagiarism detection systems.

As a notable example of a plagiarism detection system, the one developed at Masaryk University should be mentioned [32, 33, 34]. This software has been supporting plagiarism detection in a document storage of the Masaryk University Information System (IS MU) since August 2006. Since 2008 it has also been a part of the Czech nationwide registry of theses Theses.CZ. In 2010, the system won the PAN-10 International Competition on Plagiarism Detection [53] and as such should be considered the state of art.

The core of the algorithm exploits document similarities by finding out how many 5-grams of tokens (chunks) the documents have in common. First, the pairs of documents which share at least 20 chunks are preselected. Within these, the concept of *valid intervals* is used to identify the plagiarised parts. A valid interval is a part of document containing a sufficiently dense sequence of chunks which also occur in another document. The gap between each two chunks in the sequence must be at most 50 tokens (the density criterion). A pair of matching valid intervals from two documents is considered a plagiarised part.

The algorithm is implemented using an inverted index on hashes of 5-grams. For each hash, a list of documents in which the 5-grams mapped to that hash occur is stored. Apart from that, the information about the length of each 5-gram and its position in the document is also remembered in the inverted index (associated with each document entry). This data structure allows for efficient pre-selection of candidate documents (sharing at least 20 chunks) as well as for efficient creation of valid intervals and identifying the plagiarised parts.

The algorithm can be parallelised as follows. Each node in a cluster is responsible for a specified range of hashes and maintains the index for the matching 5-grams. This applies to both creating the index and holding it in memory afterwards. The documents to be checked for plagiarism can then be distributed among the same nodes for processing. A node can communicate with other nodes to get access to the parts of the inverted index which it does not store locally.

The system allows for batch processing of a given collection of documents as well as for incremental processing. Details can be found in [33].

In 2009, the common document base of IS MU and Theses.CZ contained 1,300,000 documents. Checking the full collection for plagiarism took about three hours on a cluster of 45 PCs. [34] As of May 2011, there are about 3 million documents in the database which amounts to ca 10 billion tokens. [31] It is not known how long does it currently take to process the full database from scratch since the anti-plagiarism software has only been doing incremental updates since 2009. [31]

The developers have also experimented with cross-lingual plagiarism detection by using Google Translate and with intrinsic plagiarism detection using a style change function based on profiles of character trigrams. [32]

3.3 Duplicate content in text corpora

Most of the research on identifying duplicate and near-duplicate content in large data collections has been done in the context of Web search engines. Here, the goal is to identify identical and almost identical pairs of Web pages, i.e. those which only differ in small details, such as a timestamp or a visitor count. Some of the methods described in the previous are effective for this task and they scale up very well since minor differences between documents can be detected by using very small document fingerprints.

Section 3.6.2 shows that the almost identical pages represent only a small percentage of Web documents which have a significant amount of content in common. Pairs of Web pages with an intermediate level of similarity (say 50-80 % of shared content) are fairly frequent. While these pairs are probably of little interest for Web search engines, they definitely cannot be ignored when creating Web corpora as they contain a lot of undesirable duplicate texts.

The previous section demonstrated on some of the described algorithms that it is problematic to apply them to detecting intermediate level of similarities. The attempts to do so inevitably lead either to a significant loss in accuracy or to increasing the size of the fingerprints and reducing the scalability of the method. It is easy to see why this happens. The algorithms select some small samples of the original documents and estimate the similarities of documents by matching the samples. For almost identical documents, a few samples are sufficient since it is likely that the identical parts of the documents will be sampled and matched. However, as the documents become less similar, the risk of sampling different parts increases. This reduces the reliability of the method unless more samples are used. The document sampling principle is common for most near-duplicate detection fingerprinting schemes and the described problem can be expected to apply in all of these schemes.

3.4 Making use of duplicate n-grams

Bernstein and Zobel [10] suggested that most of the near-duplicate detection algorithms select inappropriate document samples for fingerprints. They pointed out that those samples which occur only once in the whole data set are not of any use for identifying near-duplicate documents. Including these unique samples in document fingerprints only increases their size and/or harms the accuracy.

The researchers proposed an iterative algorithm (SPEX) for finding all shingles (n-grams) which occur at least twice in the data set. For reasonably long shingles, the duplicated ones represent only a small percentage in most real world document collections. Bernstein and Zobel demonstrated this on a 476 MB collection of newswire articles from the Los Angeles Times. Out of 65,900,076 distinct shingles of length eight (8-grams) only 907,981 (less than 1.4 %) occurred two or more times.

Since the duplicated shingles are fairly rare, it is possible to use them for creating reasonably sized document fingerprints. Such fingerprints can be used for computing the values of document similarity functions (such as resemblance) exactly rather than estimating them. Smaller fingerprints can be created by subsampling. Though then the similarities can no longer be computed exactly, it should still be possible to achieve more accurate estimates than from equally sized fingerprints based on virtually random sampling methods.

3.4.1 SPEX

The SPEX algorithm computes the list of duplicated shingles in an iterative way. The core idea is as follows. A shingle S of length n can only be duplicated if both of its sub-shingles S_1 and S_2 of length $n-1$ are duplicated. If S_1 or S_2 is unique then it is clear that S must be unique, too. For example, the bigram *nascent effort* can only be duplicated if each of the words *nascent* and *effort* occurs at least twice in the collection.

SPEX starts from a list of duplicated unigrams (single words). Since the number of distinct words is usually small even for large data collections, it is possible to

n	N_n^{unq}	N_n^{prn}	T_n^{dup}	N_n^{dup}	M_n
1	191,299	0	278,079	93,751,313	469,378
2	10,130,676	373,298	4,603,645	83,807,882	14,639,102
3	40,876,464	17,291,952	8,061,474	53,058,040	**36,249,631**
4	70,777,744	55,301,874	5,881,395	23,152,706	29,418,739
5	85,278,904	80,918,395	2,945,432	8,647,492	13,187,336
6	90,085,317	89,244,690	1,555,553	3,837,025	5,341,612
7	91,442,538	91,309,145	1,080,612	2,475,750	2,769,558
8	91,864,665	91,842,834	917,628	2,049,569	2,020,071
9	92,047,840	92,041,981	844,990	1,862,340	1,768,477
10	92,158,604	92,155,864	800,768	1,747,522	1,648,498

Table 3.1: SPEX iterations on BNC

hold a counter for each word in a hash table. Thus, finding the duplicated words is straightforward. However, with the increasing length of shingles, the number of distinct ones increases quickly. The number of distinct long shingles (such as 10-grams) is usually close to the size (number of words) of the whole data set. For large collections, this may easily exceed a normal capacity of RAM. Therefore, SPEX prunes the shingles known to be unique based on the results of the previous iteration. For each n-gram, both $(n-1)$-grams are extracted and if one of them is not found among the duplicated $(n-1)$-grams (identified in the previous iteration), the n-gram is ignored. This reduces memory requirements to some extent.

In [51] we have shown that the memory requirements of SPEX are still quite high, especially throughout the third iteration. The problem is that the number of unique trigrams is typically high whereas much less unique bigrams exist since many get repeated as a result of a normal language use. Thus, when the list of duplicated trigrams is built, a counter for many unique trigrams is needed which are not pruned since they contain naturally duplicated bigrams. This is illustrated in Table 3.1 which shows the details of SPEX iterations on the British National Corpus (BNC). The following values are monitored for each iteration:

- N_n^{unq} – Number of unique n-grams (this is equal to the number of unique n-gram types).
- N_n^{dup} – Number of duplicate n-grams (each n-gram being computed as many times as it occurs in the collection).
- T_n^{dup} – Number of duplicate n-gram types.
- N_n^{prn} – Number of unique n-grams which could be pruned based on the results of the previous iteration.
- M_n – Number of shingles which had to be stored in memory during the n-th iteration. This includes the duplicate n-grams, the unique n-grams which could not be pruned, and the duplicate $(n-1)$-grams from the previous iteration. Thus, $M_n = T_n^{dup} + (N_n^{unq} - N_n^{prn}) + T_{n-1}^{dup}$.

It can be seen that during the third iteration, more than 36 million shingles have to be stored in memory (39 % of all distinct 10-grams). The experiment has been repeated for 1 % and 10 % samples of BNC. The number of shingles stored in memory during the third iteration was 0.39 million and 4.3 million respectively. This suggests that the memory requirements increase roughly linearly with the corpus size. For very large corpora (10 billion words+), the memory requirements may be prohibitively high.

3.4.2 Finding duplicate n-grams with a suffix array

While it is clear that duplicate n-grams are very useful for finding near-duplicate data, the high memory requirements of the SPEX algorithm are problematic. We have therefore developed an alternative algorithm for finding duplicate n-grams with a constant amount of memory. [51]

An obvious way to get a list of duplicate n-grams is to generate all n-grams and sort them. External sorting can be used for large collections. The input is split into sufficiently small chunks which can be sorted in memory one by one. Each sorted chunk is dumped to a hard disk. As the last step, the sorted chunks are merged to form the final sorted list. The problem of this approach is that a large amount of data has to be processed. Note that the size of the list of all n-grams is n times the size of the whole corpus.

We have proposed two optimisations to the process. First, rather than sorting n-grams, we build a suffix array[4] using a suffix sorting algorithm. This can be done with much less memory than sorting n-grams. Thus, larger chunks can be processed and the number of chunks is reduced. Generating the list of sorted n-grams from a suffix array is straightforward. Second, we compress the temporary files (sorted chunks) to reduce the required disk space.

We start by mapping words to numeric IDs in order to work with integers rather than with character strings. The IDs are assigned in an increasing order starting from 0. Each newly encountered word is mapped to the next unassigned ID. (The first word in the corpus is mapped to 0; the second word is mapped to 1 unless it is the same as the first word, etc.) Note that frequent words are encountered early and thus they get small IDs.

Once the words are mapped to IDs we can think of the whole corpus as a long sequence of integers. We split it to smaller subsequences which can be processed in memory. For each subsequence, a suffix array is built using an efficient suffix sorting algorithm [44]; a list of sorted n-grams (represented as sequences of n integers) is generated from the suffix array and stored in a temporary file. When saving the sorted n-grams, Elias codes (delta and gamma) [62] are used to represent the word IDs. This compresses the data since frequent words are mapped to small integers and these are represented by short bit-strings in Elias codes. Additional compression is achieved by omitting the longest common prefix with the previous n-gram.

In order to build a suffix array, 12 bytes per word are required (4 bytes to represent the word ID and 8 bytes in auxiliary data structures). Thus, by using 1 GB of memory, we can process chunks of roughly 89.5 million words. When sorting n-grams directly, we would need approximately $6n$ bytes per n-gram (assuming 5 bytes per word plus word separators) or $4n$ bytes per n-gram by using word enumeration (assuming 32-bit integers). For 10-grams, the latter (more favourable) approach still requires 40/12 or 3.33 times more memory than suffix sorting, i.e. 3.33 GB of RAM would be needed for processing 89.5 million words.

According to our experimental results, the size of the compressed temporary files is about 3 times the size of the corpus. Direct external sort of n-grams would require n times the size of the corpus.

We have successfully applied the suffix sorting based algorithm to extract dupli-

[4]`http://en.wikipedia.org/wiki/Suffix_array`

Input text size in millions of words	Running time	# of duplicate 10-gram types
1	1.4s	9,585
10	13.6s	126,372
100	2m48s	2,581,827
1,000	38m21s	31,266,338
9,292	18h19m	403,920,712

Table 3.2: Results of duplicate n-grams detection using suffix sorting.

cate 10-grams[5] from a corpus of more than 9 billion words. It took 18.5 hours on a single 2.4 GHz machine and required no more than 1.5 GB RAM. Running times for smaller data samples are summarised in Table 3.2.

Since the time complexity of the proposed algorithm is close to linear and the memory requirements are constant, the algorithm should be practical even for processing much larger corpora.

3.5 Removing duplicate data

An obvious approach to removing duplicate and near-duplicate documents is as follows:

1. Identify near-duplicate document pairs.
2. Create clusters of near-duplicate documents by using transitivity.
3. Keep one document for each cluster and discard all other documents.

One problem which needs to be addressed, especially for low similarity thresholds, is that different documents may end up in the same cluster (imagine a similarity threshold of 0.5 and documents A, B, C where A and B have one half of the content in common, B and C have one half in common, but A and C are completely different). Consequently, it may be difficult to decide which document should be kept for the cluster. It may even be possible to keep more than one document without exceeding the similarity threshold (such as A and C in the previous example).

A simpler alternative approach to removing near-duplicate documents is possible provided we only care about removing duplicate and near-duplicate data from the corpus and we do not need to know which pairs of documents are near-duplicated. Consider the following algorithm: Let G represent a set of n-grams. For each document, extract all n-grams and compute the proportion of the n-grams which exist in G. If it is above a predefined threshold, discard the document. Otherwise, keep the document and insert all its n-grams to G.

In other words, for each document, we check the amount of its content which is duplicate to some of the already processed data. If the amount is too high, the document is discarded. It is not important whether the matched n-grams come from a single document or from multiple documents. The main thing is that by including a document which contains n-grams already present in the corpus, some of the corpus data gets duplicated (if long n-grams are used, the recurrence is unlikely to be natural and thus it is undesirable). By excluding the document, the duplicate

[5]We have ignored common function (stoplist) words and punctuation when building the 10-grams. Section 3.6.4 discusses the consequences of doing so.

data is eliminated. A filtering threshold represents a trade-off between the amount of allowed duplicate data and the amount of discarded non-duplicate data.

The described algorithm represents the core idea but it is not practical. The reason is that the set G would need to store almost all n-grams and for large corpora it may easily grow out a normal RAM capacity. In order to make the algorithm practical, we can make use of the fact that we can find duplicate n-grams efficiently. Recall that duplicate n-grams are typically rare and thus it is possible to fit them all into a memory efficient data structure even when processing very large data. The previous algorithm can then be refined as follows: Let G_{2+} represent a set of n-grams initialised to contain all duplicate n-grams. For each document, extract all n-grams and compute the proportion of the n-grams which exist in G_{2+}. If it is above a predefined threshold, discard the document. Otherwise, keep the document.

The algorithm makes use of the fact that duplicate n-grams are indicative of the duplicate content. The duplicate parts of a document can be identified by simply comparing against the set of duplicate n-grams. The problem is that the algorithm discards all instances of the duplicate data rather than keeping one. In order to prevent this, one more refinement is necessary which results in the final algorithm described formally as Algorithm 3.1.

An auxiliary set of n-grams (G_{out}) is used. Whenever a document is processed and kept (sent to output), the duplicate n-grams contained in the document are moved from the set G_{2+} to G_{out}. When deciding whether a document should be kept or removed as duplicate, the duplicate n-grams in G_{out} are used. Thus, a document is only removed if it contains too many duplicate n-grams which already exist in the output. Since the duplicate n-grams are only moved from one data structure to another, the memory requirements are not increased (or at least not significantly).

The order in which the documents are processed is important. The documents containing duplicate content are more likely to be kept if they are processed early (before the duplicate content is brought to output by other documents). Before starting the de-duplication algorithm, the proportion of duplicate n-grams is determined for each document and the documents are reordered so that the ones with the lowest proportion of duplicate n-grams come first. The rationale is twofold. First, the documents with less duplicate content are preferred over the documents with more duplicate content. Second, especially at news sites, index pages which contain excerpts of other pages (articles) are common. During the de-duplication, if the index page comes first, it is sent to output and the referenced pages may be discarded since each of them shares some content with the index page. Reordering the documents is likely to ensure that the referenced pages are processed before the index page.

3.5.1 Storing duplicate n-grams in a Judy array

The implementation of the Algorithm 3.1 uses two techniques to reduce the amount of memory needed for storing the sets of n-grams. First, hashes of n-grams are stored rather than raw n-grams. Second, the hashes of n-grams are stored in a memory efficient data structure called Judy array [8, 61].

Judy is a complex associative array data structure. It is designed to enable an implementation which uses cheap CPU operations. In particular, it tries to maximise the use of CPU cache and thus reduces the number of CPU instructions needed for accessing the memory. Judy arrays are therefore generally very fast.

The main advantage of a Judy array with respect to the aforementioned algorithm is its memory efficiency. It is claimed to be more memory efficient than most other data structures of the kind, such as B-trees or hash tables.

Judy1 is a form of a Judy array which maps integers to Boolean values and

```
Input: $\mathcal{D}, G_{2+}, t$
Output: $\mathcal{D}' \subseteq \mathcal{D}$ such that none of the documents $D \in \mathcal{D}'$ has more than $t$ percent
  of content in common with any subset of $\mathcal{D}'$ not containing $D$.

  {$\mathcal{D}$ represents the corpus to be processed as a set of documents.}
  {$G_{2+}$ is a set of all duplicate n-grams in $\mathcal{D}$.}
  {$t$ is a threshold defining the amount of acceptable duplicate content.}
  {$|D|$ represents the number of words in a document $D$.}
  {$n$ is the length of $n$-grams.}
  {$ngrams(D) = (g_1, g_2, \ldots, g_{|D|-n+1})$ is the list of all n-grams in $D$.}

  $G_{out} \leftarrow \emptyset$
  for $D \in \mathcal{D}$ do
    {determine the amount of duplicate content}
    $I \leftarrow \emptyset$
    for $g_i \in ngrams(D)$ do
      if $g_i \in G_{out}$ then
        {update the positions of words within duplicate n-grams}
        $I \leftarrow I \cup \{i, \ldots, i + n - 1\}$
      end if
    end for
    if $|I|/|D| < t$ then
      {the amount of duplicate content is acceptable}
      output $D$
      {move the duplicate n-grams contained in the document
      from $G_{2+}$ to $G_{out}$}
      for $g_i \in ngrams(D)$ do
        if $g_i \in G_{2+}$ then
          $G_{2+} \leftarrow G_{2+} \setminus \{g_i\}$
          $G_{out} \leftarrow G_{out} \cup \{g_i\}$
        end if
      end for
    end if
  end for
```

Algorithm 3.1: Removing near-duplicate documents using a list of duplicate n-grams.

thus it is appropriate for representing sets of integers. The implementation of the described de-duplication algorithm uses two instances of Judy1 for storing the hashes of duplicate n-grams.

A free Judy array implementation in C is available[6] as well as a Python interface – PyJudy[7]. On a 64-bit CPU architecture, 64-bit integer keys can be used and thus n-gram hashes of lengths up to 64 bits can be stored in Judy1.

I have set up an experiment to analyze the memory requirements of the Judy array. I generated a large number of integers to be stored in Judy1 and monitored how memory allocation increases as more integers are added. Two different techniques were used for estimating the memory consumption. First, a Linux kernel system call `getrusage` was used and the value of `ru_maxrss` recorded (the maximum resident set size utilized). Second, the `MemUsed()` method of the aforementioned Judy implementation was called, which should return the number of bytes allocated by the data structure. It turned out that the values are never exactly the same, but they are highly correlated.

To speed up the computation, random integers have been generated rather than hashes of n-grams. This should not have any impact on the results of the experiment since hashes of up to 64 bits are in fact fact random 64-bit integers. It turned out that the size (bit length) of the stored integers affects the memory efficiency of the Judy array significantly. Therefore, the experiment was repeated with three different bit lengths of the stored integers – 63-bit, 60-bit and 56-bit. In each run, 10 billion integers were generated and stored in Judy1. A machine with 100 GB RAM was used for the computation. Fig. 3.4 shows the development of memory consumption.

Note that all used lengths of hashes provide a very large hash range which makes a probability of hash collision negligibly low even if amounts of hashes as high as 10 billion are considered. In the experiment, it happened only 686 times that the same 56-bit integer (out of the 10 billion) was generated more than once. For 60-bit integers there have been only 39 “collisions” and there has been no “collision” for 63-bit integers.

The results suggest that there are three stages of the memory usage development. In the first stage the memory requirements get quickly very low to roughly 6 bytes per hash (B/hash). At some point, second stage starts where memory requirements quickly increase to ca 20 B/hash. After achieving this local maximum, memory requirements start to decrease again and seem to converge to ca 6-8 B/hash. This is the third stage. The lengths of the stages as well as the speed of convergence during the third stage seems to be highly dependent on the bit lengths of the stored hashes (integers).

For 56-bit hashes, the first stage is the longest and for a major part of it only 6 to 7 bytes per hash are required. This lasts until about 900 million hashes get stored when memory usage raises quickly and decreases only very slowly after achieving the maximum of 20 B/hash.

For 60-bit hashes, on the other hand, the second stage comes very early and the local maximum of relative memory usage is achieved around 35 million of stored hashes. Then, however, Judy becomes very memory efficient again soon. After storing more than 700 million hashes the memory requirements get below 10 B/hash and from there keep gradually decreasing to as low as 6 B/hash.

Surprisingly, the results for 63-bit hashes are somewhere in-between the former two. Due to an extreme complexity of the Judy array it would be very difficult to analyse why this happens and it is out of scope of this work. At any rate, a conclusion which can be drawn from the results is that for storing less than 900

[6]`http://judy.sourceforge.net/`
[7]`http://www.dalkescientific.com/Python/PyJudy.html`

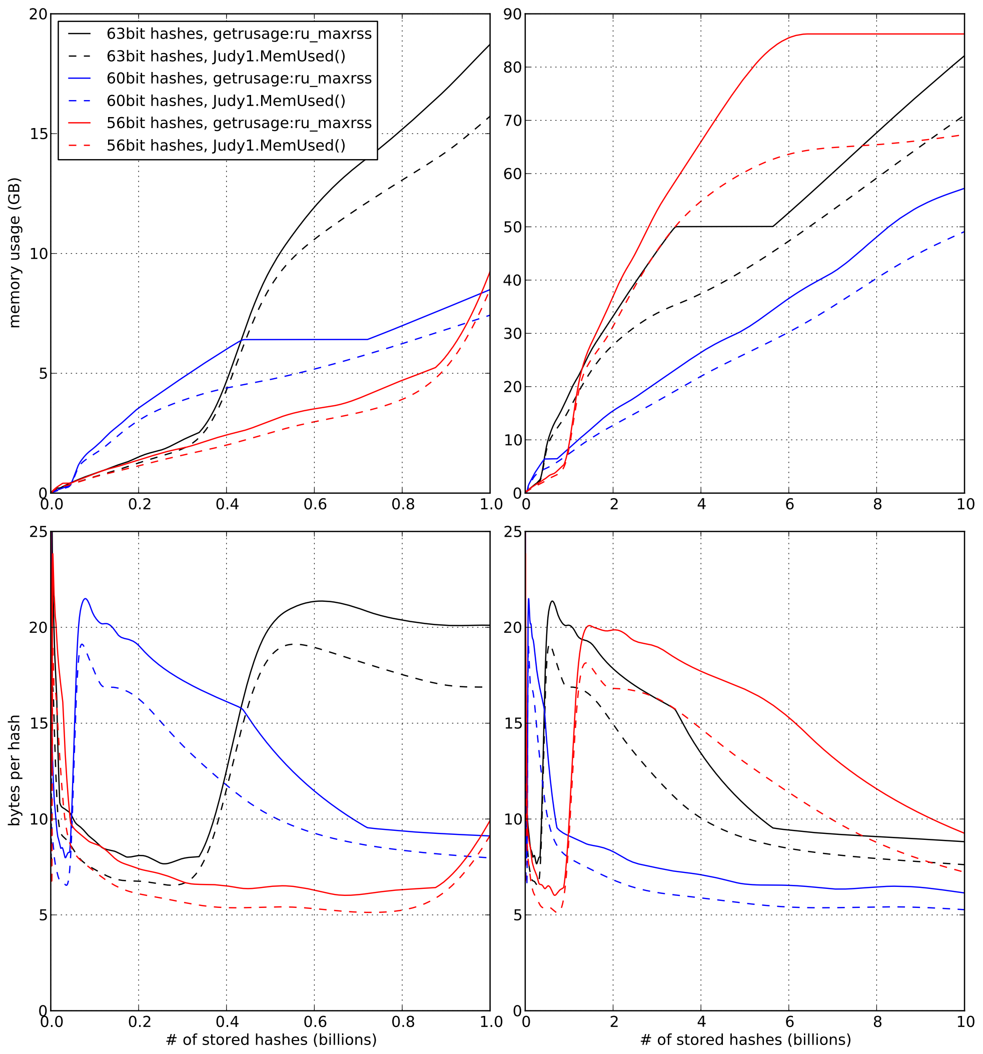


Figure 3.4: Judy array memory usage for various lengths of stored hashes.

million of records, 56-bit hashes are appropriate. For larger volume of data, 60-bit hashes should be preferred.

Note that with an appropriate set up Judy array not only does not add any memory overhead, but in fact achieves a slight compression of the stored data while remaining very fast at all times.

Now if we get back to the de-duplication algorithm, an important fact is that having 100 GB of RAM available, one can store much more than 10 billion (60-bit) hashes of duplicate n-grams in Judy1. Provided the proportion of duplicate n-grams is typically less than 10 % (see sections 3.6.4 and 3.7), a collection of more than 100 billion words can be processed with the de-duplication algorithm at such memory constraints.

It might be that other bit lengths of the hashes (such as 58-bit or 59-bit) would result in even more plausible memory usage. This has not been tested, but it may be a subject to future work.

3.5.2 Removing duplicate text blocks

Removing near-duplicate data on a document level is often problematic, especially with respect to documents with an intermediate level of duplicate content. Consider a largish document with 50 % of duplicate content and 50 % of unique content. By keeping the document we allow duplicate texts in the corpus. By discarding the document we lose good texts. In an ideal case we would want to discard only the duplicate part and keep the unique part. An obvious way to do so is to work with smaller units than documents, such as paragraphs or even sentences. This is fairly straightforward. However, it introduces a problem of fragmentation.

A full document is typically a sufficiently independent unit in the sense that it can be fully understood by a reader without requiring any external information (except for eventual background knowledge in the relevant area). The meaning of a sentence or a short text block taken out of context, on the other hand, can often be unclear. It may be difficult to recognise the senses of the words in such a text. Being unable to do so constitutes a problem for some areas of linguistics, such as lexicography[8].

When applying a de-duplication algorithm on units smaller than full documents, it can happen that some documents are broken into small fragments (stubs) for which the context information is not available since the surrounding text blocks have been removed as duplicates. Depending on the intended use of the corpus it may be desirable to remove the stubs.

The problem of document fragmentation has been discussed in section 2.9.10. It has also been demonstrated that the context-sensitive classification of the jusText algorithm (section 2.7.3) reduces the fragmentation to a high extent. This part of the algorithm can be easily reused for the same purpose in the de-duplication process.

The idea is to apply the de-duplication algorithm to *near-good* and *good* blocks identified by jusText. The blocks containing too much duplicate content are re-classified as *duplicate*. Then the context-sensitive classification is applied where the *duplicate* blocks are treated exactly the same way as *bad* blocks. As a result, the *short* and some *near-good* blocks near the *duplicate* blocks are removed. This prevents creating stubs. Figure 3.5 illustrates the situation.

[8]For a lexicographer who uses corpus evidence for processing a dictionary entry, it is essential to be able to recognise the senses of the occurrences of given word in the corpus.

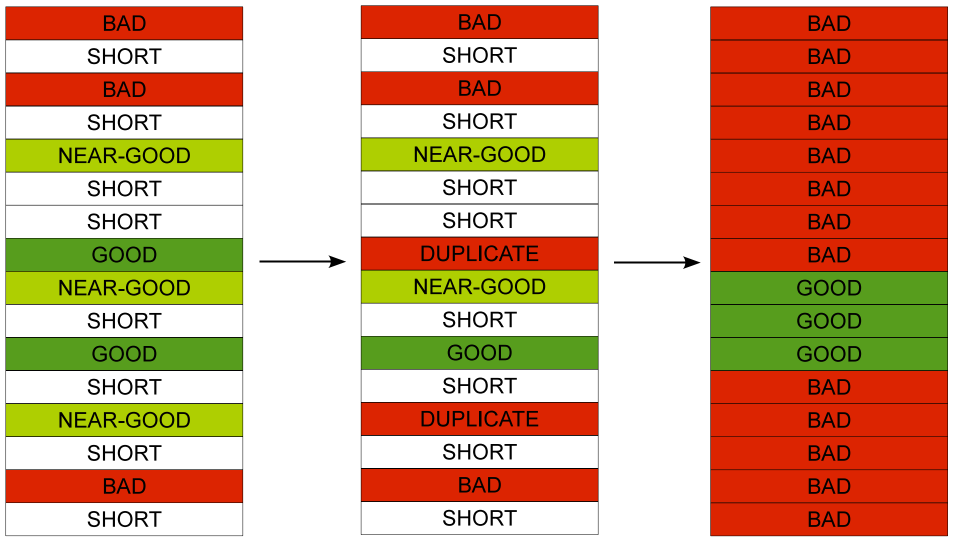


Figure 3.5: Example of context-sensitive classification with duplicate blocks.

3.6 Case study: ClueWeb09

ClueWeb09[9] is a collection of one billion Web pages in ten languages, collected by massive Web crawling in January and February 2009. The data is available for research purposes. Since the size of the data set is enormous (5 TB gzip compressed, 25 TB uncompressed) it is distributed on four 1.5 TB hard drives for a fee proportional to the value of the hard drives.

ClueWeb09 is an excellent resource for experiments on Web data. Not only does it save researchers from having to do their own Web crawls, but also provides unchanging data and makes it possible for any experiments on the data to be reproducible.

The Web pages are stored in WARC archives, each of a size roughly 1 GB uncompressed. For most of the experiments reported in this section, only a single WARC file from the English part of ClueWeb09 has been used, namely the file `en0000/00.warc.gz`. Since for most experiments it was necessary to process the input data many times with different parameter settings, using larger data would not be practical.

In ClueWeb09, the Web pages have been ordered by URL before saving. As a result, the pages from the same domain are grouped together. This is an important fact for de-duplication experiments since most of the duplicate and near-duplicate pages occur within the same domain. A smallish collection of Web pages randomly sampled from the whole ClueWeb09 may easily contain no duplicates at all.

The data sample was first processed with jusText and the documents containing only boilerplate were removed. Next, language detection was applied to weed out non-English texts. I used the Python module Trigram[10] which computes a similarity score (0 to 1) of an input text to a language model based on frequencies of triples of characters. The documents with a score below 0.6 were removed.[11] The remaining data comprises 21,776 Web pages from 2,722 different domain. The distribution of domains follows a power law. The most frequent domain is `blog.pennlive.com` with

[9]`http://boston.lti.cs.cmu.edu/Data/clueweb09/`

[10]`http://code.activestate.com/recipes/326576-language-detection-using-character-trigrams/`

[11]The threshold of 0.6 was chosen based on a manual inspection of the data sorted by the language model scores. It turned out that as a positive side effect of the language filtering occasional JavaScript and alike content was also removed.

1,473 occurrences. There are 1416, 319 and 163 domains with one, two and three occurrences respectively.

The texts were tokenised using a simple regular expression based tokeniser. The tokeniser splits input to tokens such as words, punctuation marks, numbers and other entities often found on the Web, such as e-mail addresses or URLs. After removing boilerplate the corpus contains 13,745,880 tokens.

3.6.1 Exact duplicates

In order to remove exact duplicates, a MD5 checksum was computed for each document[12]. For each group of documents with the same checksum only one document was kept. This procedure identified 877 types of duplicate documents with a total count of 2,950. Thus, 2,073 documents were removed reducing the corpus from 21,776 to 19,703 documents. In term of tokens, the corpus size was reduced to 13,105,727.

3.6.2 Resemblance

As explained in section 3.2.1, one way to find near-duplicate documents is to estimate resemblances of pairs of documents. For large data collections, only those pairs with a resemblance close to 1 can be identified efficiently. However, for the small ClueWeb09 sample it was possible to compute resemblances for all pairs of documents. The motivation to do so was to show that document pairs with an intermediate level of similarity do exist on the Web and are much more frequent than pairs with a high level of similarity.

The resemblances were computed on 10-grams of tokens (ignoring the punctuation). For details, see section 3.2.1. This was done after removing the exact duplicates. The results are plotted in Fig. 3.6. The graph shows for each value of resemblance the number of pairs of documents with a resemblance greater than that value. Note that the y-scale is logarithmic. It can be seen that the number of pairs with a resemblance greater than 0.5 is roughly an order of magnitude higher than the number of pairs with a resemblance greater than 0.9. It can be concluded that algorithms focused on finding nearly identical documents (with resemblances close to 1) fail to detect a lot of duplicate data here. Some of the results reported later in this section provide further evidence.

3.6.3 Lengths of n-grams

Many de-duplication algorithms use n-grams of words as basic units for exploiting duplicates and near-duplicates. However, there is no consistency in the length of the used n-ngrams (the value of n). Some researchers propose using 5-grams, others 8-grams or 10-grams. To the best of my knowledge, no studies have been published on the impact of the length of n-grams on the results of de-duplication. This and the following section try to shed some light on the issue.

De-duplication algorithms employ n-grams since they are indicative of duplicate content. In general, the longer repeated n-gram is found the more likely it is that it indicates copy-and-paste duplication rather than natural independent use of a phrase. On the other hand, if the n-grams are too long they may fail to detect equivalent texts with sparse minor changes, such as document revisions. For instance, imagine that 50-grams are used for de-duplication. Then if one makes a copy of a document

[12]Note that this is different from checksum-based de-duplication at the level of full Web pages (including HTML mark-up) which is typically performed during Web crawling. Here, only the plain text content excluding the boilerplate is used for computing the checksums.

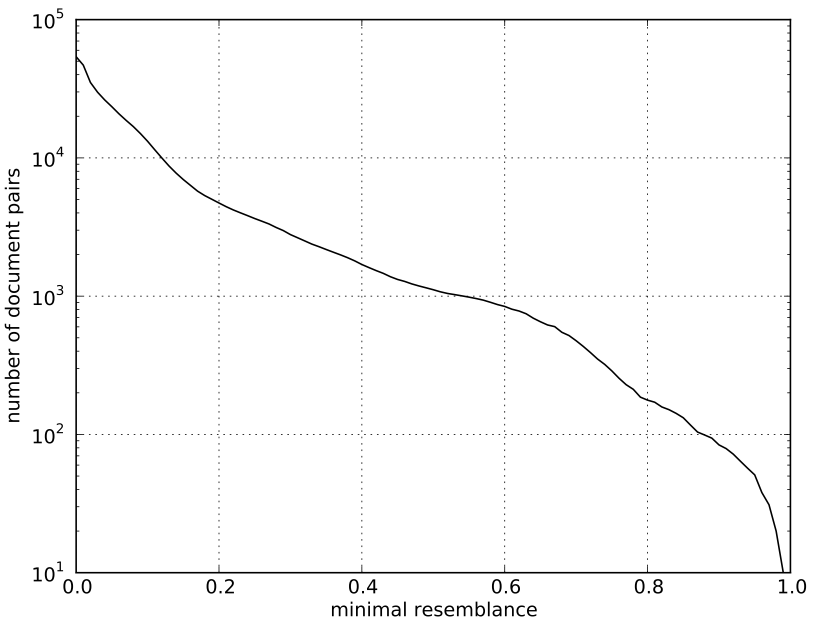


Figure 3.6: Numbers of pairs of documents with a resemblance above given threshold in a sample of ClueWeb09.

and changes at least every 50th word, the two documents will appear as completely different to the algorithm.

I have used the ClueWeb09 sample to analyse the natural and artificial repetition of n-grams of various lengths. The experiment is based on the assumption that if an n-gram occurs repeatedly only within Web pages from a single domain it is more indicative of a copy-and-paste duplication than an n-gram repeated on Web pages from multiple domains. For each n-gram with at least two occurrences in the corpus, the URLs of the documents where the n-gram occurred were exploited and the n-gram was labeled as *single-domain* if all the URLs belonged to a single domain, or *cross-domain* otherwise.

When building the n-grams, three different strategies were used. First, the n-grams were built using all tokens in the corpus. Second, the tokens starting with punctuation characters (most often just single punctuation marks) were ignored and only the remaining tokens were used for building n-grams. Third, both punctuation marks and stop words[13] were ignored. The latter two strategies attempt to reduce the number of naturally repeated n-grams by excluding frequent tokens. The corpus contains 1,500,347 tokens (12.65 %) starting with a punctuation character and 5,054,664 stop words (42.62 %). Thus, by excluding both punctuation and stop words the corpus size is reduced by 55.28 %.

Table 3.3 contains the counts of distinct *cross-domain* and *single-domain* n-grams for various values of n and for each of the three above-mentioned strategies for building n-grams. It can be seen that the counts of *single-domain* n-grams are almost equivalent for all values of n. The counts of the *cross-domain* n-grams on the other hand decrease rapidly with an increasing length. This is not surprising since as the

[13]Same as in the previous experiments, the list of 319 stop words downloaded from `http://armandbrahaj.blog.al/2009/04/14/list-of-english-stop-words/` was used.

	all tokens		no punctuation		no punct.+stopw.	
n	cross-d.	single-d.	cross-d.	single-d.	cross-d.	single-d.
3	774,760	792,523	595,824	738,310	41,561	489,901
4	394,821	1,010,704	252,422	884,571	9,854	456,955
5	137,481	1,042,718	76,793	895,167	5,482	435,271
6	46,231	1,022,239	25,581	876,567	4,350	420,821
7	21,188	997,904	13,614	857,464	3,790	409,791
8	14,378	978,150	10,531	841,848	3,410	400,694
9	12,166	962,192	9,321	828,879	3,125	392,656
10	11,064	948,876	8,610	817,673	2,876	385,385
11	10,324	937,220	8,087	807,587	2,656	378,666
12	9,748	926,836	7,651	798,373	2,460	372,319

Table 3.3: Counts of cross-domain and single-domain n-grams of various lengths.

n-grams get longer the probability of getting repeated "by chance" decreases and thus the number of naturally repeated n-grams is reduced whereas the duplicate texts generate roughly equivalent number of repeated n-grams for all reasonably small values of n (especially if the duplicate data is contained in a small number of longish text blocks rather than in many short blocks). The results indicate that the assumption of *single-domain* n-grams being more indicative of copy-and-paste duplicates is valid.

The results also demonstrate that by excluding very frequent tokens from n-grams the number of duplicate n-grams decreases. There are two apparent reasons for this. For simplicity, let us assume that we exclude every second token from a corpus (note that this is almost the case when excluding both punctuation and stop words). Then first, the size of the corpus is reduced to one half and consequently also the number of n-grams is reduced by one half. Second, building n-grams in the reduced corpus is very similar to building 2n-grams in the original corpus. It can be seen from the results that the counts of 10-grams at "all tokens" reduced by one half roughly match the counts of 5-grams at "no punctuation and stop words".

The results indicate at which length of n-grams the natural recurrence (or recurrence by chance) can be expected to be negligibly low. It is where the rapid decrease of *cross-domain* frequency stops. This length is around 8 for "all tokens", around 7 for "no punctuation" and around 5 for "no punctuation and stop words". This of course does not yet make it clear which lengths are appropriate for de-duplication. This is explored in the next section.

3.6.4 De-duplication

I have repeatedly applied the algorithm for removing duplicate text blocks described in section 3.5.2 on the ClueWeb09 sample. I used a threshold of 0.5 for filtering duplicated blocks, i.e. all text blocks were marked as duplicate which contained more than 50 % of duplicate texts. A jusText content-sensitive classification was then applied to remove the duplicate blocks and the surrounding stubs.

In each run, a different list of duplicate n-grams was used with the de-duplication algorithm. The n-grams were extracted with various lengths (3, 5, 7 and 10) and with various token exclusion strategies (all tokens, excluding punctuation, excluding stop words and excluding both). Three indicators have been used for evaluating the results of de-duplication: (i) the memory requirements of the de-duplication algorithm, (ii) the reduction of the size of the corpus, and (iii) the amount of duplicated data left in the corpus after de-duplication.

excl stop	excl punct	n	duplicate n-grams used for de-dupl		corpus size after de-dupl (in tokens)		duplicate 10-grams after de-dupl	
yes	yes	10	346,471	2.92%	10,393,096	87.64%	110,519	4.91%
		7	366,091	3.09%	10,351,934	87.30%	92,033	4.09%
		5	387,490	3.27%	10,303,425	86.89%	78,717	3.50%
		3	465,761	3.93%	10,163,985	85.71%	58,013	2.58%
	no	10	458,541	3.87%	10,362,671	87.39%	95,393	4.24%
		7	484,300	4.08%	10,306,078	86.91%	77,809	3.46%
		5	520,960	4.39%	10,226,234	86.24%	65,007	2.89%
		3	714,597	6.03%	9,742,150	82.15%	32,937	1.46%
no	yes	10	733,287	6.18%	10,330,505	87.11%	80,758	3.59%
		7	769,956	6.49%	10,278,241	86.67%	68,799	3.06%
		5	858,641	7.24%	10,180,833	85.85%	53,830	2.39%
		3	1,207,802	10.19%	6,124,874	51.65%	11,942	0.53%
	no	10	848,819	7.16%	10,299,759	86.86%	71,317	3.17%
		7	897,395	7.57%	10,231,065	86.28%	59,714	2.66%
		5	1,042,084	8.79%	10,052,234	84.77%	42,840	1.90%
		3	1,422,984	12.00%	3,425,905	28.89%	2,826	0.13%
rsmbl. de-dupl					11,782,150	99.36%	2,144,192	95.34%
no de-dupl					11,858,518	100.00%	2,249,057	100.00%

Table 3.4: The influence of the length and type of n-grams on the results of de-duplication with a threshold of 0.5.

The results are reported in Table 3.4. The table contains four multi-columns separated with double-lines. The first multi-column specifies the type of duplicate n-grams. The second column labeled “duplicate n-grams used for de-dupl” shows the number of distinct duplicate n-grams, i.e. the number of n-grams which the de-duplication algorithm needs to store in memory. The percentage values indicate the number of n-grams relative to the corpus size in tokens. The column labeled “corpus size after de-dupl (in tokens)” shows the size of the corpus after de-duplication, i.e. it indicates the data loss. The last column labeled “duplicate 10-grams after de-dupl” shows the number of duplicate 10-grams (built on all tokens) in the corpus after de-duplication. Total count is reported rather than number of distinct 10-grams, i.e. each 10-gram with at least two occurrences is counted as many times as it occurs in the corpus. Since 10-grams have a very low natural recurrence (as demonstrated in the previous section) they are good indicators of the amount of duplicated data. For reference, the number of duplicate 10-grams before de-duplication is shown in the last row of the table.

It can be seen that the results of most configurations are plausible since they reduce the duplicate content significantly without losing too much corpus data. The only problematic configurations are those based on 3-grams, especially when stop words are not ignored. Clearly, the natural recurrence of 3-grams is too high and thus generates too many false positives for de-duplication to be practical.

One can observe that shorter duplicate n-grams are more frequent and thus impose higher memory requirements on the de-duplication algorithm, but they also remove more duplicate content. The effect of ignoring stop words when building n-grams seems to be plausible. It reduces the memory requirements significantly while keeping the amount of removed duplicate data high. For instance, compare the row “yes-yes-5” with “no-yes-10”. The former configuration has lower memory requirements (387,490 duplicate 5-grams vs. 733,297 duplicate 10-grams) and still leaves slightly

excl stop	excl punct	n	duplicate n-grams used for de-dupl		corpus size after de-dupl (in tokens)		duplicate 10-grams after de-dupl	
yes	yes	10	346,471	2.92%	10,320,981	87.03%	74,270	3.30%
		7	366,091	3.09%	10,243,969	86.38%	50,145	2.23%
		5	387,490	3.27%	10,144,283	85.54%	36,172	1.61%
		3	465,761	3.93%	9,707,401	81.86%	19,140	0.85%
	no	10	458,541	3.87%	10,259,809	86.52%	55,518	2.47%
		7	484,300	4.08%	10,150,155	85.59%	35,191	1.56%
		5	520,960	4.39%	9,966,524	84.05%	22,920	1.02%
		3	714,597	6.03%	7,374,281	62.19%	4,300	0.19%
no	yes	10	733,287	6.18%	10,204,473	86.05%	38,835	1.73%
		7	769,956	6.49%	10,097,921	85.15%	29,633	1.32%
		5	858,641	7.24%	9,654,407	81.41%	18,496	0.82%
		3	1,207,802	10.19%	2,041,319	17.21%	1,211	0.05%
	no	10	848,819	7.16%	10,144,899	85.55%	29,922	1.33%
		7	897,395	7.57%	9,986,641	84.21%	21,678	0.96%
		5	1,042,084	8.79%	8,890,957	74.98%	8,302	0.37%
		3	1,422,984	12.00%	951,365	8.02%	45	0.00%
no de-dupl					11,858,518	100.00%	2,249,057	100.00%

Table 3.5: The influence of the length and type of n-grams on the results of de-duplication with a threshold of 0.25.

less duplicate content (78,717 vs. 80,758 duplicate 10-grams after de-duplication). On the other hand, the configurations which do not leave out tokens when building n-grams are slightly more accurate, i.e. they remove more duplicate content while keeping the total corpus size higher. Compare the rows "no-no-10" and "yes-yes-5". The former leaves less duplicate content (3.17 % vs. 3.50 %) while the size of the corpus size after de-duplication is almost the same for both configurations (86.86 % vs. 86.89 % of the original size). Similarly for the lines "no-yes-5" and "yes-yes-3".

It can be concluded that there is no clear winner among the configurations. All have their pros and cons. A best choice may depend on a particular application. In fact, there is fairly little difference in the results and any configuration should work quite well in practice as long as the pathological ones (3-grams) are avoided. The fact that the results of de-duplication are reasonable for many different types of n-grams suggests that the de-duplication algorithm should work well also for data in languages other than English where the structure of n-grams is different.

For comparison with traditional de-duplication algorithms (focusing only on almost identical pairs of documents), I have identified all pairs of documents with a resemblance higher than 0.9. I then created clusters of near-duplicate documents using transitivity and I discarded all but one document for each cluster. The results are shown in the row labeled "rsmbl. de-dupl" in the Table 3.4. The number of duplicate 10-grams was reduced only to 95.34 %. Compare with 5 % or less achieved by any configuration of the block level de-duplication. Apparently, the difference is huge.

I have repeated the experiment and changed the threshold used by the de-duplication algorithm from 0.5 to 0.25 and 0.1. The results are presented in Table 3.5 and Table 3.6 respectively. Lowering the threshold of course leads to a higher reduction of the duplicate content, but also to a higher data loss. Longer n-grams have to be used in order to keep the amount of lost data reasonable. The threshold of 0.25 seems to offer the best trade-off between the reduction of the duplicate content and

excl stop	excl punct	n	duplicate n-grams used for de-dupl		corpus size after de-dupl (in tokens)		duplicate 10-grams after de-dupl	
yes	yes	10	346,471	2.92%	10,253,794	86.47%	58,589	2.61%
		7	366,091	3.09%	10,092,657	85.11%	26,856	1.19%
		5	387,490	3.27%	9,843,495	83.01%	13,628	0.61%
		3	465,761	3.93%	8,259,996	69.65%	2,959	0.13%
	no	10	458,541	3.87%	10,103,124	85.20%	30,435	1.35%
		7	484,300	4.08%	9,863,436	83.18%	10,163	0.45%
		5	520,960	4.39%	9,353,513	78.88%	3,969	0.18%
		3	714,597	6.03%	3,758,889	31.70%	152	0.01%
no	yes	10	733,287	6.18%	10,032,801	84.60%	16,848	0.75%
		7	769,956	6.49%	9,712,796	81.91%	9,245	0.41%
		5	858,641	7.24%	7,726,170	65.15%	2,268	0.10%
		3	1,207,802	10.19%	695,868	5.87%	510	0.02%
	no	10	848,819	7.16%	9,889,527	83.40%	5,763	0.26%
		7	897,395	7.57%	9,354,001	78.88%	3,023	0.13%
		5	1,042,084	8.79%	5,931,515	50.02%	569	0.03%
		3	1,422,984	12.00%	324,076	2.73%	0	0.00%
no de-dupl					11,858,518	100.00%	2,249,057	100.00%

Table 3.6: The influence of the length and type of n-grams on the results of de-duplication with a threshold of 0.1.

the data loss.

As an alternative evaluation of the amount of duplicate content left in the corpus after de-duplication, I have computed the resemblances of the documents in the de-duplicated corpus. This has been done in exactly the same way as reported in the previous section. I have only done this evaluation for a single configuration – a threshold of 0.5, 10-grams with punctuation and stop words removed (the first data row of the Table 3.4), i.e. the most conservative settings. The highest value of resemblance encountered was 0.09. There were only 15 pairs of documents with a resemblance higher than 0.05.

3.6.5 Increasing data size

In order to examine the scalability of the described de-duplication algorithm it has been applied to a much larger portion of the ClueWeb09 data set. A major part of the English ClueWeb09 is divided into subdirectories named `en0000` to `en0124`. The first 10 subdirectories (`en0000` to `en0009`) comprising 36,559,057 Web pages and 977 GB of uncompressed data were used for the experiment. This is roughly 7 % of the English pages available in the full ClueWeb09.

Distribution of domains follows a power law. Out of 2,369,698 distinct domains, there are 1,023,914 with a single occurrence, 280,658 with two occurrences and 170,154 with three occurrences. The three most frequent domains are `en.wikipedia.org` (38,887 pages), `dictionary.reference.com` (34,686 pages) and `dir.yahoo.com` (30,428 pages).

The processing was exactly the same as described in the previous two sections. Block level de-duplication used a threshold of 0.5 and duplicate 10-grams built with ignoring the punctuation and stop words. All the computations were done on a single machine with two quad-core Intel Xeon 2.13 GHz CPUs and 100 GB RAM.

The preprocessing (boilerplate removal, language filtering, tokenisation) is easy

phase	processing time	used CPU cores
removing boilerplate (jusText)	26h 37min	8
language filtering (Trigram)	5h 22min	8
tokenisation	15h 54min	8
removing exact duplicates	12h 44min	1
duplicate n-grams extraction	9h 49min	1
block level de-duplication	69h 8min	1

Table 3.7: Processing times for ClueWeb09_English_1.

stage	word count		token count
original corpus	24,633,016,767		n/a
after removing boilerplate	11,363,624,711	46.1 %	n/a
after language filtering	9,586,878,336	84.4 %	n/a
after removing exact duplicates	8,983,831,725	93.7 %	10,359,277,678
after block level de-duplication	7,279,959,176	81.0 %	8,390,930,801

Table 3.8: Corpus size throughout processing. The percentage values are relative to the previous row.

to parallelise since the input can be split into multiple parts which can be processed independently. The preprocessing used eight processes running in parallel each of which fully loaded one CPU core. It took roughly 2 days to preprocess the data. The details about the processing times can be found in Table 3.7.

The original corpus contained 24.63 billion words[14] out of which 11.36 billion (46.1 %) were identified as main content and 13.27 billion (53.9 %) as boilerplate. Language filtering reduced the corpus size further to 9.57 billion words.

Next, the exact duplicates were removed using MD5 checksums, duplicate 10-grams were extracted and as the last step the block level de-duplication was performed. All this processing used only a single process since it cannot be easily parallelised and took nearly 4 days in total.

After preprocessing, the corpus contained 11,837,454 documents. Among these, 658,512 types of fully duplicated documents were identified with a total of 1,682,969 instances. By keeping only one instance of each document type the number of documents was reduced to 10,812,997. This amounts to 8.98 billion words.

By applying the suffix array sorting algorithm, 374 million types of duplicate 10-grams were identified. The block level de-duplication algorithm needed 6.3 GB RAM for storing the 10-grams in memory. Corpus size was decreased to 7.28 billion words and duplicate content was reduced from 913,415,626 duplicate 10-grams to 97,232,210 (10.6 %). Note that same as in the previous section, number of all instances of the duplicate 10-grams is reported. Moreover, these 10-grams are built using all tokens (unlike the 10-grams used for de-duplication where punctuation and stop words are ignored). The block level de-duplication also slightly increased the fragmentation (see section 2.9.10) from 3.56 to 3.85 text blocks per document.

The results of corpus processing are summarised in Table 3.7 and Table 3.8. The sizes of the corpus are reported both as word counts and token counts. Recall that word count (in a slight abuse of terminology) represents the number of whitespace separated sequences of characters whereas token count is the number of entities output by a tokeniser (words, punctuation, URLs, email addresses, etc). Token count

[14]The "words" were counted the same way as the UNIX `wc -w` command does, i.e. as the number of whitespace separated character strings.

is typically slightly higher than word count. Since tokenisation has been performed after language filtering, token counts are not available for the previous stages.

The processing required roughly 6 days of computing on a single machine (2 days preprocessing, 4 days de-duplication) with a maximal usage of 6.3 GB RAM (during block level de-duplication). The processing time could be shortened significantly in two ways. First, the preprocessing phase allows for a massive paralellisation and could be shortened by simply utilising more CPU power. Second, most of the algorithms have been implemented in Python and little effort has been invested to optimising the code.[15] It is likely that by carefully profiling the code and rewriting the critical parts to a lower level language, such as C or C++, a fraction of the current running time can be achieved. This is a subject to future work.[16]

Memory requirements were fairly high since 60-bit hashes of n-grams have been stored. As explained in section 3.5.1, 56-bit hashes would have been more appropriate for storing 374 million records and would have reduced the RAM usage from 6.3 GB to less than 3 GB. However, this has not been a known fact at the time of ClueWeb09 procecessing since it preceded the detailed analysis of Judy array memory usage.

3.7 Processing other large Web corpora

Throughout the year 2010, three other large Web corpora have been compiled for English, German and Italian, named enTenTen, deTenTen and itTenTen respectively[17]. The corpus creation started from large Web crawls using the open source crawler Heritrix[18]. The crawls were restricted to the `.de` and `.at` domains for German, and `.it` for Italian. For English, the `.au`, `.ca`, `.uk` and `.us` domains have been used as well as major generic domains (`.com`, `.edu`, `.gov`, `.net` and `.org`) which also contain mostly texts in English. The downloaded pages were then processed in the same way as the ClueWeb09 data. Table 3.9 summarises the results and also provides side-by-side comparison with ClueWeb09.

It can be seen that the amount of data removed by various stages of the processing differs significantly among the four data sets. This can be partially attributed to a different performance of the processing tools on various languages, especially with respect to removing boilerplate. Here, for both English collections (enTenTen and ClueWeb09) the proportion of content classified as good is around 46 % while for the other two collections it is much lower (around 34 %). It is hard to say whether this is because jusText tends to filter out more data for Italian and German or because the two collections simply contain more boilerplate. At any rate, it would not be difficult to achieve balance by adjusting jusText thresholds.

As for duplicate content, the results of removing exact duplicates suggest that the processing tools are not to be blamed for the differences. It is clear that the Italian and German corpora contain the highest amount of duplicate content, or at least the highest number of fully duplicated documents. ClueWeb09, on the other hand, contains very little duplicates. The final size of the corpus is roughly one third of the original. This is a much better result than for all the other corpora. The data of German deTenTen was reduced to almost 10 % by the processing.

[15]The only exception is the suffix array sorting algorithm for extracting duplicate n-grams which is implemented in C++ in a very efficient way which leaves little room for improvement.

[16]As of May 2011, shortly before submitting this thesis, the code for block level de-duplication has already been rewritten into C. The new implementation reduced the time required for processing a 10 billion tokens collection from 69 to 6 hours.

[17]The family name *TenTen* has been proposed by Adam Kilgarriff and it represents the target size of 10^{10} (10 billion) words we aimed to meet when creating the corpora.

[18]`http://crawler.archive.org/`

It is apparent that the source data of the TenTen corpora contain a lot of noise. As mentioned above, this data has been collected by Web crawling of relevant first level domains. Following suggestions of other researchers [4, 21, 7], too small (less than 5 kB) and too large (more than 1 MB) Web pages have not been processed since the small pages typically do not contain enough texts to be worth processing and the large pages usually contain automatically generated data. No further restrictions have been applied on the Web crawling. Thus, a lot of data which was later filtered out (or reduced to a fraction) has been downloaded in vain. By analysing the downloaded pages while crawling, the branches which receive good data could be identified and preferred. It is likely that this would lead to a much better ratio between the amount of downloaded data and the final size of the corpus. Such approach to Web crawling is a subject to future research.

3.8 Summary

It has been argued that currently available de-duplication algorithms are only capable of finding pairs of almost identical documents. They fail to detect documents with an intermediate level of similarity. It has been shown that documents which contain a significant amount of both identical and different texts are frequent on the Web. Since any duplicate texts are undesirable for text corpora, these documents should not be ignored.

I have shown how to use a list of duplicate n-grams for removing near-duplicate content at any level of similarity. An efficient algorithm for computing the list of duplicate n-grams using suffix array sorting has been presented. It has been demonstrated that RAM requirements for representing the duplicate n-grams in memory can be kept very low by storing their hashes in a Judy array. An impact of the bit-length of the hashes on the memory efficiency of the storage has been discussed.

The proposed de-duplication algorithm can be applied both on a document or a paragraph (text block) level. In the latter case, processed documents may suffer from fragmentation. It has been explained how to overcome this problem by reusing the context-sensitive classification of the jusText algorithm presented in the previous chapter.

The performance and scalability of the proposed boilerplate cleaning and de-duplication algorithms has been demonstrated on four large data collections in three different languages.

corpus name	enTenTen		itTenTen		deTenTen		ClueWeb09 (7 %)	
language	English		Italian		German		English	
data size	645 GB		1360 GB		1455 GB		977 GB	
Web pages	13,638,928		33,459,999		43,160,992		36,559,057	
original corpus size [words]	14,648,716,440		22,555,059,478		22,927,528,414		24,633,016,767	
after removing boilerplate	6,805,296,135	46.5 %	7,740,199,568	34.3 %	7,769,459,528	33.9 %	11,363,624,711	46.1 %
after language filtering	5,975,184,997	87.8 %	7,135,930,675	92.2 %	7,242,302,420	93.2 %	9,586,878,336	84.4 %
after removing exact duplicates	4,091,383,071	68.5 %	4,216,156,603	59.1 %	4,129,201,031	57.0 %	8,983,831,725	93.7 %
after block level de-duplication	3,145,958,284	76.9 %	2,591,205,687	61.5 %	2,441,347,841	59.1 %	7,279,959,176	81.0 %
tokens before block dedupl.	4,765,119,530		5,017,409,779		4,880,335,291		11,363,624,711	
10-grams used for block dedupl.	212,864,389	4.5 %	295,369,351	5.9 %	331,258,395	6.8 %	374,264,251	3.3 %
dupl. 10-grams bef. block dedupl.	527,908,460		662,183,382		625,069,294		913,415,626	
dupl. 10-grams aft. block dedupl.	41,211,739	7.7 %	66,420,547	10.0 %	36,401,954	5.8 %	97,232,210	10.6 %

Table 3.9: Details of processing Web corpora. The percentage values are relative to the previous row.

Chapter 4

Conclusion

In this work, two important problems have been addressed relevant to creation of large Web corpora – cleaning boilerplate and removing duplicate and near-duplicate content.

The part of the work devoted to boilerplate cleaning proposes a novel heuristic based algorithm called jusText. The algorithm processes the input in two stages first of which classifies each text block independently based on its length, number of links and proportion of function words. The second stage then uses the context (the classes of surrounding blocks) to assign classes to those blocks which could not be reliably classified in the first stage.

The evaluation uses three different data collections and compares the jusText algorithm with several other state-of-aft methods. It was demonstrated that jusText achieves at least equally good results as the best of its competitors on all of the used data sets. Apart from that, it has a positive impact on fragmentation of the cleaned documents and prevents the texts taken out of context to be sent to output. Last but not least, the algorithm proved to be flexible in terms of the trade-off between precision and recall which can be easily adjusted by simply setting the algorithm's input parameters.

The second major part of the work focuses on detecting and removing duplicate and near-duplicate data. It explains that the current methods for identifying and removing duplicate and near-duplicate documents (widely used by search engines) are not appropriate for a corpus creation work-flow since they focus only on almost identical documents. It has been argued and demonstrated on a sample data set that Web collections contain a vast amount of document pairs with an intermediate level of similarity which bring a lot of duplicate data into text corpora if ignored by the de-duplication algorithm.

It has been shown how to tackle the problem of intermediate similarities effectively. The proposed algorithm builds on the ideas of Bernstein and Zobel [10] who have argued that duplicate n-grams are very useful for detecting near-duplicate documents. We have extended their work by designing a more efficient algorithm for finding the list of duplicate n-grams [51]. Also, a novel algorithm for removing near-duplicate content has been created, which only needs to store the duplicate n-grams in memory (rather than a inverted index on the duplicate n-grams). This reduces memory requirements and increases scalability of the method. To save even more memory, a memory efficient data structure Judy array is employed.

The algorithms developed in this work have been included in processing several large Web corpora for various languages (English, German, Italian, Slovak). All these corpora have practical applications. Some are being used in a EU funded machine translation project PRESEMT[1]. Other are included in the Sketch Engine[2] – a Web based corpus query system.

The described boilerplate cleaning and de-duplication tools have been made open source and are available for download at `http://code.google.com/p/justext/` and `http://code.google.com/p/onion/`. I believe that the presented research and the developed tools will contribute to creating cleaner, larger and generally more useful Web corpora.

[1]`http://presemt.eu/`
[2]`http://www.sketchengine.co.uk/`

Bibliography

[1] G. Aston and L. Burnard. *The BNC handbook: Exploring the British National Corpus with SARA*. Edinburgh University Press, 1998.

[2] Z. Bar-Yossef and S. Rajagopalan. Template detection via data mining and its applications. In *Proceedings of the 11th international conference on World Wide Web*, page 591. ACM, 2002.

[3] M. Baroni and S. Bernardini. BootCaT: Bootstrapping corpora and terms from the web. *Proceedings of LREC 2004*, pages 1313–1316, 2004.

[4] M. Baroni and A. Kilgarriff. Large linguistically-processed web corpora for multiple languages. *Proceedings of European ACL*, 2006.

[5] M. Baroni, A. Kilgarriff, J. Pomikálek, and P. Rychlý. WebBootCaT: instant domain-specific corpora to support human translators. *Proceedings of EAMT 2006*, pages 247–252, 2006.

[6] M. Baroni, F. Chantree, A. Kilgarriff, and S. Sharoff. Cleaneval: a competition for cleaning web pages. In *Proceedings of the Conference on Language Resources and Evaluation (LREC), Marrakech*, 2008.

[7] M. Baroni, S. Bernardini, A. Ferraresi, and E. Zanchetta. The WaCky Wide Web: A collection of very large linguistically processed Web-crawled corpora. *Language Resources and Evaluation*, 43(3):209–226, 2009. ISSN 1574-020X.

[8] D. Baskins. A 10-minute description of how Judy arrays work and why they are so fast, 2002. URL `http://judy.sourceforge.net/doc/10minutes.htm`.

[9] D. Bauer, J. Degen, X. Deng, P. Herger, J. Gasthaus, E. Giesbrecht, L. Jansen, C. Kahna, T. Kriiger, R. Martin, et al. FIASCO: Filtering the Internet by Automatic Subtree Classification, Osnabruck. In *Building and Exploring Web Corpora: Proceedings of the 3rd Web as Corpus Workshop, incorporating Cleaneval*, volume 4, pages 111–121. Presses univ. de Louvain, 2007.

[10] Y. Bernstein and J. Zobel. A scalable system for identifying co-derivative documents. *Proc. String Processing and Information Retrieval Symp.*, pages 55–67, 2004.

[11] E. Brill. A simple rule-based part of speech tagger. *Proceedings of the Third Conference on Applied Natural Language Processing*, pages 152–155, 1992.

[12] A.Z. Broder. Identifying and filtering near-duplicate documents. *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, pages 1–10, 2000.

[13] A.Z. Broder, M. Charikar, A.M. Frieze, and M. Mitzenmacher. Min-wise independent permutations (extended abstract). In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 327–336. ACM, 1998.

[14] D. Cai, S. Yu, J.R. Wen, and W.Y. Ma. Extracting content structure for web pages based on visual representation. In *Proceedings of the 5th Asia-Pacific web conference on Web technologies and applications*, pages 406–417. Springer-Verlag, 2003.

[15] D. Chakrabarti, R. Kumar, and K. Punera. Page-level template detection via isotonic smoothing. In *Proceedings of the 16th international conference on World Wide Web*, page 70. ACM, 2007.

[16] M.S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, page 388. ACM, 2002.

[17] S. Debnath, P. Mitra, N. Pal, and C.L. Giles. Automatic identification of informative sections of web pages. *IEEE transactions on knowledge and data engineering*, pages 1233–1246, 2005.

[18] S. Evert. StupidOS: A high-precision approach to boilerplate removal. In *Building and Exploring Web Corpora: Proceedings of the 3rd Web as Corpus Workshop, incorporating Cleaneval*, volume 123, page 123. Presses univ. de Louvain, 2007.

[19] S. Evert. A lightweight and efficient tool for cleaning web pages. *Proceedings of the Sixth International Language Resources and Evaluation (LREC08), Marrakech, Morocco*, 2008.

[20] D. Fernandes, E.S. de Moura, B. Ribeiro-Neto, A.S. da Silva, and M.A. Gonçalves. Computing block importance for searching on web sites. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pages 165–174. ACM, 2007.

[21] A. Ferraresi, E. Zanchetta, M. Baroni, and S. Bernardini. Introducing and evaluating ukwac, a very large web-derived corpus of english. In *Workshop Programme*, page 47, 2008.

[22] A. Finn, N. Kushmerick, and B. Smyth. Fact or fiction: Content classification for digital libraries. In *DELOS Workshop: Personalisation and Recommender Systems in Digital Libraries*, 2001.

[23] W. Fletcher. Concordancing the Web with KWiCFinder. *Proc. 3rd North American Symposium on Corpus Linguistics and Language Teaching*, 2001.

[24] W. Gao and T. Abou-Assaleh. GenieKnows Web Page Cleaning System. In *Building and Exploring Web Corpora: Proceedings of the 3rd Web as Corpus Workshop, incorporating Cleaneval*, page 135. Presses univ. de Louvain, 2007.

[25] D. Gibson, K. Punera, and A. Tomkins. The volume and evolution of web page templates. In *Special interest tracks and posters of the 14th international conference on World Wide Web*, pages 830–839. ACM, 2005.

[26] J. Gibson, B. Wellner, and S. Lubar. Adaptive web-page content identification. In *Proceedings of the 9th annual ACM international workshop on Web information and data management*, pages 105–112. ACM, 2007.

[27] C. Girardi. Htmlcleaner: Extracting the Relevant Text from the Web Pages. In *Building and Exploring Web Corpora: Proceedings of the 3rd Web as Corpus Workshop, incorporating Cleaneval*, volume 4, pages 141–143. Presses univ. de Louvain, 2007.

[28] N. Heintze. Scalable Document Fingerprinting (Extended Abstract). In *Proc. USENIX Workshop on Electronic Commerce*. Citeseer, 1996.

[29] M. Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 284–291. ACM, 2006.

[30] K. Hofmann, W. Weerkamp, C. Fairon, H. Naets, A. Kilgarriff, and GM de Schryver. Web corpus cleaning using content and structure. In *Building and Exploring Web Corpora: Proceedings of the 3rd Web as Corpus Workshop, incorporating Cleaneval*, page 145. Presses univ. de Louvain, 2007.

[31] J. Kasprzak. (personal communication), 2010.

[32] J. Kasprzak and M. Brandejs. Improving the Reliability of the Plagiarism Detection System. *Lab Report for PAN at CLEF*, pages 359–366, 2010.

[33] J. Kasprzak, M. Brandejs, and J. Brandejsová. Distributed aspects of the system for discovering similar documents. In *ITA 09: Proceedings of the Third International Conference on Internet Technology and Applications*, 2009.

[34] J. Kasprzak, M. Brandejs, and M. Křipač. Finding plagiarism by evaluating document similarities. In *Proceedings of the SEPLN*, volume 9, pages 24–28, 2009.

[35] A. Kehoe and A. Renouf. WebCorp: Applying the Web to Linguistics and Linguistics to the Web. *Proceedings of WWW2002*, pages 7–11, 2002.

[36] A. Kilgarriff and G. Grefenstette. Introduction to the Special Issue on Web as Corpus. *Computational Linguistics*, 29(3):1–15, 2003.

[37] A. Kilgarriff, P. Rychlý, P. Smrž, and D. Tugwell. The Sketch Engine. *Proceedings of Euralex*, pages 105–116, 2004.

[38] A. Kilgarriff, S. Reddy, J. Pomikálek, and A. PVS. A corpus factory for many languages. *Proc. LREC, Malta*, 2010.

[39] P. Koehn. Europarl: A parallel corpus for statistical machine translation. In *MT summit*, volume 5. Citeseer, 2005.

[40] C. Kohlschütter. A densitometric analysis of web template content. In *Proceedings of the 18th international conference on World wide web*, pages 1165–1166. ACM, 2009.

[41] C. Kohlschütter. (personal communication), 2010.

[42] C. Kohlschütter, P. Fankhauser, and W. Nejdl. Boilerplate detection using shallow text features. In *Proceedings of the third ACM international conference on Web search and data mining*, pages 441–450. ACM, 2010.

[43] H. Kucera and W.N. Francis. *Computational Analysis of Present-Day American English.* Brown University Press Providence, RI, 1967.

[44] N.J. Larsson and K. Sadakane. Faster suffix sorting. *Theoretical Computer Science*, 387(3):258–272, 2007.

[45] V. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics-Doklady*, volume 10, 1966.

[46] U. Manber. Finding similar files in a large file system. In *Proceedings of the USENIX winter 1994 technical conference*, pages 1–10. San Fransisco, CA, USA, 1994.

[47] G.S. Manku, A. Jain, and A. Das Sarma. Detecting near-duplicates for web crawling. In *Proceedings of the 16th international conference on World Wide Web*, page 150. ACM, 2007.

[48] M. Marek, P. Pecina, and M. Spousta. Web page cleaning with conditional random fields. In *Building and Exploring Web Corpora: Proceedings of the 3rd Web as Corpus Workshop, incorporating Cleaneval*, page 155. Presses univ. de Louvain, 2007.

[49] T. McEnery and A. Wilson. *Corpus Linguistics.* Edinburgh University Press, Edinburgh, 1996.

[50] J. Pasternack and D. Roth. Extracting article text from the web with maximum subsequence segmentation. In *Proceedings of the 18th international conference on World wide web*, pages 971–980. ACM, 2009.

[51] J. Pomikálek and P. Rychlý. Detecting co-derivative documents in large text collections. In *Proceedings of LREC.* Citeseer, 2008.

[52] M. Potthast and B. Stein. New Issues in Near-duplicate Detection. *Data Analysis, Machine Learning and Applications*, pages 601–609, 2008.

[53] Martin Potthast, Alberto Barrón-Cedeño, Andreas Eiselt, Benno Stein, and Paolo Rosso. Overview of the 2nd International Competition on Plagiarism Detection. In Martin Braschler and Donna Harman, editors, *Notebook Papers of CLEF 2010 LABs and Workshops, 22-23 September, Padua, Italy*, sep 2010. ISBN 978-88-904810-0-0.

[54] W. Pugh and M.H. Henzinger. Detecting duplicate and near-duplicate files, April 29 2008. US Patent 7,366,718.

[55] R. Rivest. RFC1321: The MD5 message-digest algorithm. *RFC Editor United States*, 1992.

[56] S. Schleimer, D.S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85. ACM, 2003.

[57] H. Schmid. Probabilistic part-of-speech tagging using decision trees. *Proceedings of International Conference on New Methods in Language Processing*, 12, 1994.

[58] B. Schulze and O. Christ. The IMS Corpus Workbench. *Institut fur maschinelle Sprachverarbeitung, Universitat Stuttgart*, 1994.

[59] S. Sharoff. Creating general-purpose corpora using automated search engine queries. In *Wacky! Working papers on the Web as Corpus*, pages 63–98, Bologna, Italy, 2006. GEDIT. ISBN 88-6027-004-9.

[60] N. Shivakumar and H. Garcia-Molina. Finding near-replicas of documents on the web. *The World Wide Web and Databases*, pages 204–212, 1999.

[61] A. Silverstein. Judy IV Shop Manual, 2002. URL `http://judy.sourceforge.net/doc/shop_interm.pdf`.

[62] I.H. Witten, A. Moffat, and T.C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.

[63] L. Yi, B. Liu, and X. Li. Eliminating noisy information in web pages for data mining. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 296–305. ACM, 2003.

[64] G.K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley, 1949.

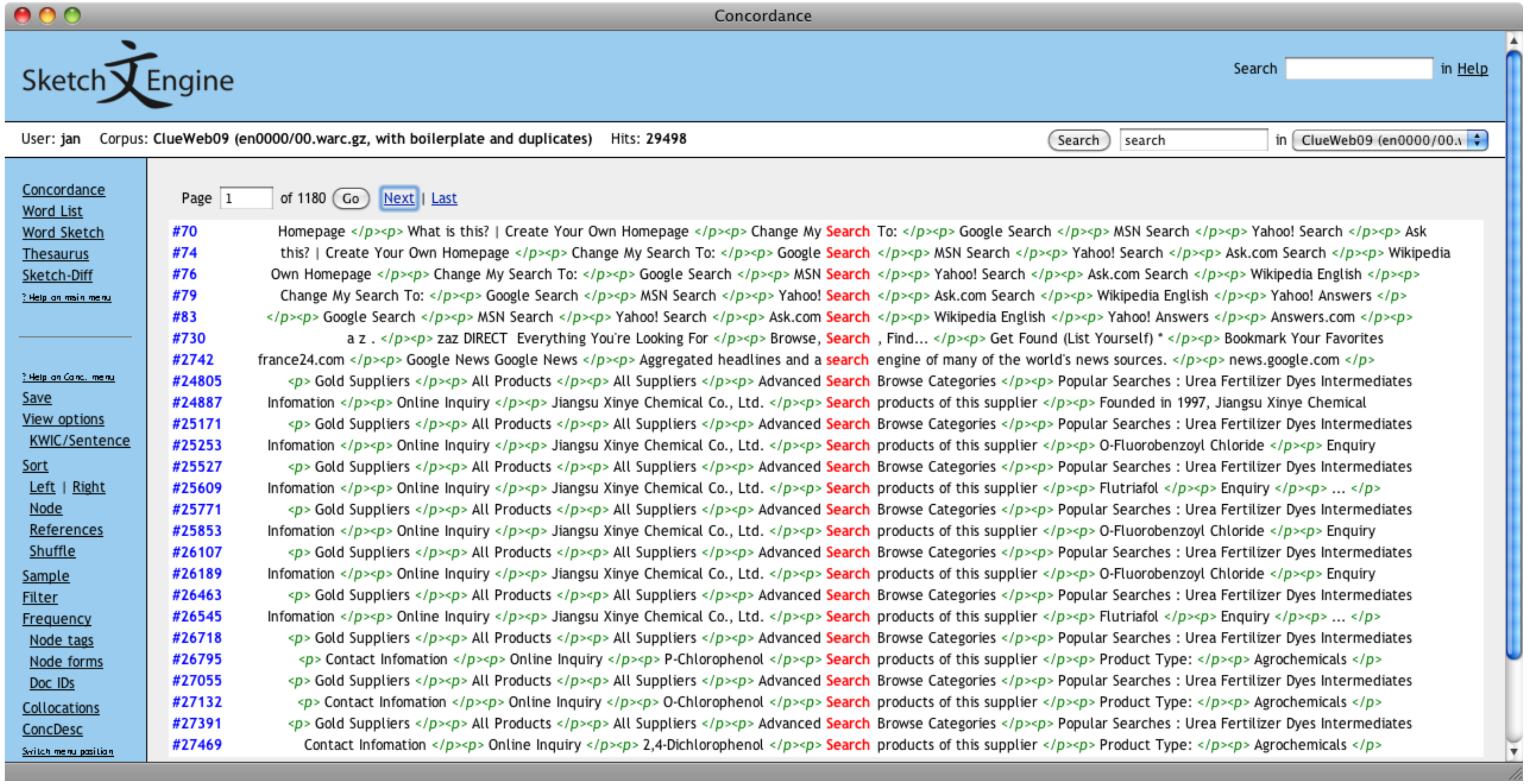
Appendix A

Screenshots

The following screenshots illustrate the problem of boilerplate and duplicate content in text corpora. Figures A.1, A.3 and A.5 show results of corpus searches for the word form *search* in a small sample of ClueWeb09 data (`en0000/00.warc.gz`, see section 3.6) before removing boilerplate and duplicates. The original Web pages have only been converted to plain text and block level HTML tags used as paragraph boundaries (`</p><p>`). No further processing has been done.

Fig. A.1 shows the first page of concordance lines, i.e. the first 25 occurences of *search* in the corpus. In Fig. A.3, the search results are shuffled (to reduce the amount of displayed duplicate content). Fig. A.5 shows the middle part of all concordance lines sorted by the right context of the keyword *search*. Sorted concordances typically reveal duplicate content in a corpus.

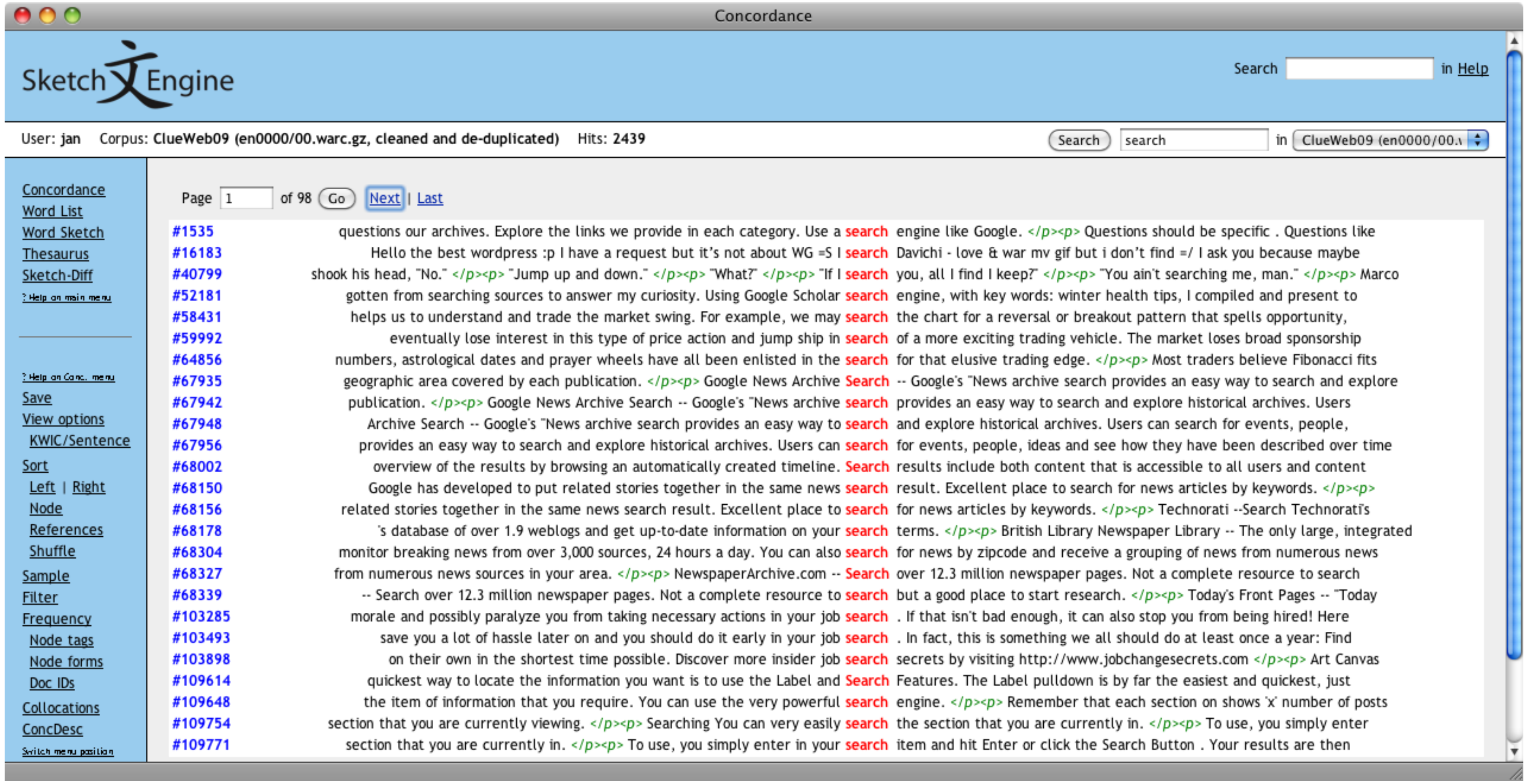
Figures A.2, A.4 and A.6 then show the same kind results for the same corpus after removing boilerplate with jusText and applying the text blocks de-duplication algorithm.



User: jan Corpus: **ClueWeb09 (en0000/00.warc.gz, with boilerplate and duplicates)** Hits: **29498**

```
#70     Homepage </p><p> What is this? | Create Your Own Homepage </p><p> Change My Search To: </p><p> Google Search </p><p> MSN Search </p><p> Yahoo! Search </p><p> Ask
#74     this? | Create Your Own Homepage </p><p> Change My Search To: </p><p> Google Search </p><p> MSN Search </p><p> Yahoo! Search </p><p> Ask.com Search </p><p> Wikipedia
#76     Own Homepage </p><p> Change My Search To: </p><p> Google Search </p><p> MSN Search </p><p> Yahoo! Search </p><p> Ask.com Search </p><p> Wikipedia English </p><p>
#79     Change My Search To: </p><p> Google Search </p><p> MSN Search </p><p> Yahoo! Search </p><p> Ask.com Search </p><p> Wikipedia English </p><p> Yahoo! Answers </p>
#83     </p><p> Google Search </p><p> MSN Search </p><p> Yahoo! Search </p><p> Ask.com Search </p><p> Wikipedia English </p><p> Yahoo! Answers </p><p> Answers.com </p><p>
#730    a z . </p><p> zaz DIRECT  Everything You're Looking For </p><p> Browse, Search , Find... </p><p> Get Found (List Yourself) * </p><p> Bookmark Your Favorites
#2742   france24.com </p><p> Google News Google News </p><p> Aggregated headlines and a search engine of many of the world's news sources. </p><p> news.google.com </p>
#24805  <p> Gold Suppliers </p><p> All Products </p><p> All Suppliers </p><p> Advanced Search Browse Categories </p><p> Popular Searches : Urea Fertilizer Dyes Intermediates
#24887  Infomation </p><p> Online Inquiry </p><p> Jiangsu Xinye Chemical Co., Ltd. </p><p> Search products of this supplier </p><p> Founded in 1997, Jiangsu Xinye Chemical
#25171  <p> Gold Suppliers </p><p> All Products </p><p> All Suppliers </p><p> Advanced Search Browse Categories </p><p> Popular Searches : Urea Fertilizer Dyes Intermediates
#25253  Infomation </p><p> Online Inquiry </p><p> Jiangsu Xinye Chemical Co., Ltd. </p><p> Search products of this supplier </p><p> O-Fluorobenzoyl Chloride </p><p> Enquiry
#25527  <p> Gold Suppliers </p><p> All Products </p><p> All Suppliers </p><p> Advanced Search Browse Categories </p><p> Popular Searches : Urea Fertilizer Dyes Intermediates
#25609  Infomation </p><p> Online Inquiry </p><p> Jiangsu Xinye Chemical Co., Ltd. </p><p> Search products of this supplier </p><p> Flutriafol </p><p> Enquiry </p><p> ... </p>
#25771  <p> Gold Suppliers </p><p> All Products </p><p> All Suppliers </p><p> Advanced Search Browse Categories </p><p> Popular Searches : Urea Fertilizer Dyes Intermediates
#25853  Infomation </p><p> Online Inquiry </p><p> Jiangsu Xinye Chemical Co., Ltd. </p><p> Search products of this supplier </p><p> O-Fluorobenzoyl Chloride </p><p> Enquiry
#26107  <p> Gold Suppliers </p><p> All Products </p><p> All Suppliers </p><p> Advanced Search Browse Categories </p><p> Popular Searches : Urea Fertilizer Dyes Intermediates
#26189  Infomation </p><p> Online Inquiry </p><p> Jiangsu Xinye Chemical Co., Ltd. </p><p> Search products of this supplier </p><p> O-Fluorobenzoyl Chloride </p><p> Enquiry
#26463  <p> Gold Suppliers </p><p> All Products </p><p> All Suppliers </p><p> Advanced Search Browse Categories </p><p> Popular Searches : Urea Fertilizer Dyes Intermediates
#26545  Infomation </p><p> Online Inquiry </p><p> Jiangsu Xinye Chemical Co., Ltd. </p><p> Search products of this supplier </p><p> Flutriafol </p><p> Enquiry </p><p> ... </p>
#26718  <p> Gold Suppliers </p><p> All Products </p><p> All Suppliers </p><p> Advanced Search Browse Categories </p><p> Popular Searches : Urea Fertilizer Dyes Intermediates
#26795  <p> Contact Infomation </p><p> Online Inquiry </p><p> P-Chlorophenol </p><p> Search products of this supplier </p><p> Product Type: </p><p> Agrochemicals </p>
#27055  <p> Gold Suppliers </p><p> All Products </p><p> All Suppliers </p><p> Advanced Search Browse Categories </p><p> Popular Searches : Urea Fertilizer Dyes Intermediates
#27132  <p> Contact Infomation </p><p> Online Inquiry </p><p> O-Chlorophenol </p><p> Search products of this supplier </p><p> Product Type: </p><p> Agrochemicals </p>
#27391  <p> Gold Suppliers </p><p> All Products </p><p> All Suppliers </p><p> Advanced Search Browse Categories </p><p> Popular Searches : Urea Fertilizer Dyes Intermediates
#27469  Contact Infomation </p><p> Online Inquiry </p><p> 2,4-Dichlorophenol </p><p> Search products of this supplier </p><p> Product Type: </p><p> Agrochemicals </p>
```

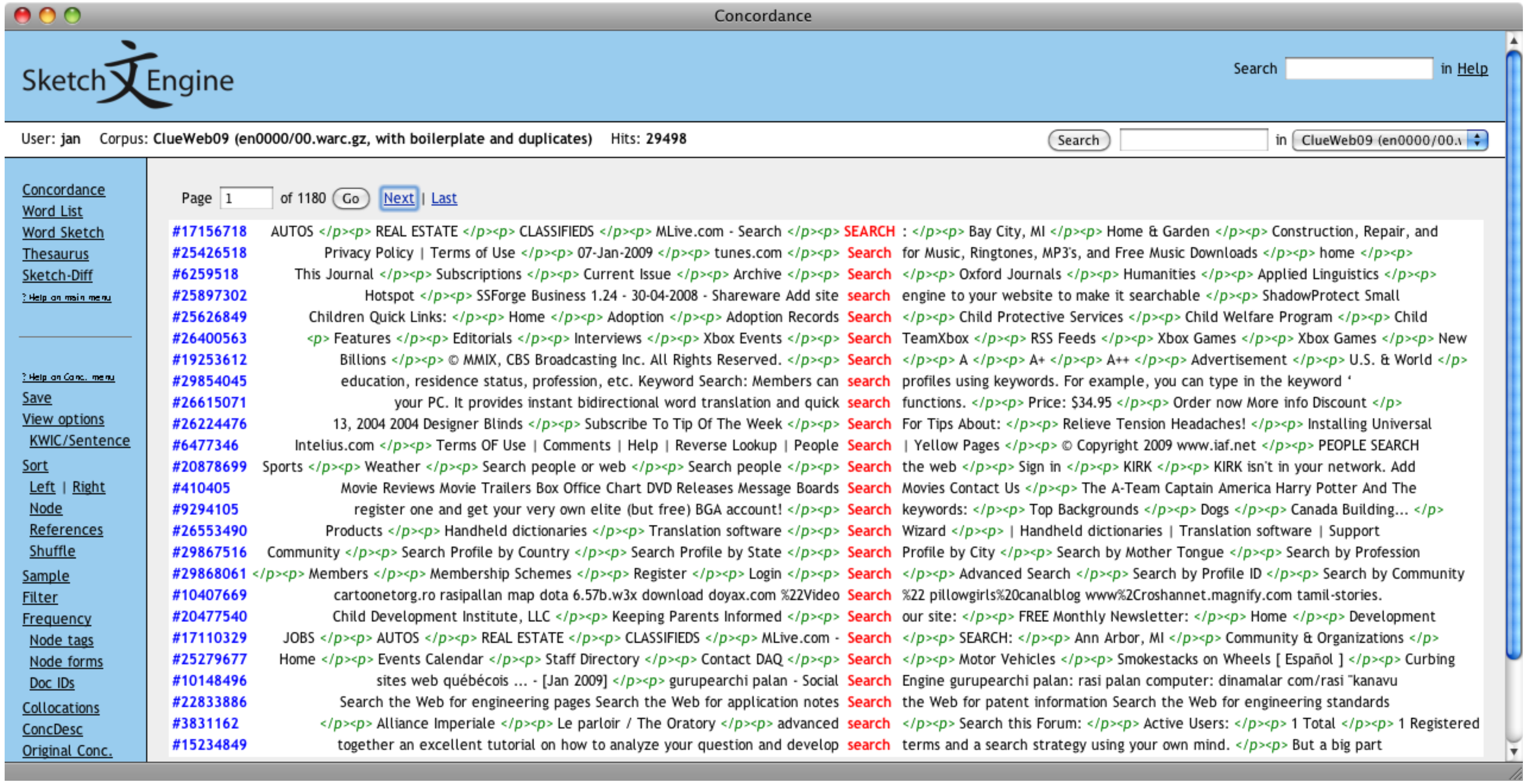
Figure A.1: Before removing boilerplate and duplicates. The first page of concordance lines.



User: jan   Corpus: **ClueWeb09 (en0000/00.warc.gz, cleaned and de-duplicated)**   Hits: **2439**

```
#1535    questions our archives. Explore the links we provide in each category. Use a search engine like Google. </p><p> Questions should be specific . Questions like
#16183   Hello the best wordpress :p I have a request but it's not about WG =S I search Davichi - love & war mv gif but i don't find =/ I ask you because maybe
#40799   shook his head, "No." </p><p> "Jump up and down." </p><p> "What?" </p><p> "If I search you, all I find I keep?" </p><p> "You ain't searching me, man." </p><p> Marco
#52181   gotten from searching sources to answer my curiosity. Using Google Scholar search engine, with key words: winter health tips, I compiled and present to
#58431   helps us to understand and trade the market swing. For example, we may search the chart for a reversal or breakout pattern that spells opportunity,
#59992   eventually lose interest in this type of price action and jump ship in search of a more exciting trading vehicle. The market loses broad sponsorship
#64856   numbers, astrological dates and prayer wheels have all been enlisted in the search for that elusive trading edge. </p><p> Most traders believe Fibonacci fits
#67935   geographic area covered by each publication. </p><p> Google News Archive Search -- Google's "News archive search provides an easy way to search and explore
#67942   publication. </p><p> Google News Archive Search -- Google's "News archive search provides an easy way to search and explore historical archives. Users
#67948   Archive Search -- Google's "News archive search provides an easy way to search and explore historical archives. Users can search for events, people,
#67956   provides an easy way to search and explore historical archives. Users can search for events, people, ideas and see how they have been described over time
#68002   overview of the results by browsing an automatically created timeline. Search results include both content that is accessible to all users and content
#68150   Google has developed to put related stories together in the same news search result. Excellent place to search for news articles by keywords. </p><p>
#68156   related stories together in the same news search result. Excellent place to search for news articles by keywords. </p><p> Technorati --Search Technorati's
#68178   's database of over 1.9 weblogs and get up-to-date information on your search terms. </p><p> British Library Newspaper Library -- The only large, integrated
#68304   monitor breaking news from over 3,000 sources, 24 hours a day. You can also search for news by zipcode and receive a grouping of news from numerous news
#68327   from numerous news sources in your area. </p><p> NewspaperArchive.com -- Search over 12.3 million newspaper pages. Not a complete resource to search
#68339   -- Search over 12.3 million newspaper pages. Not a complete resource to search but a good place to start research. </p><p> Today's Front Pages -- "Today
#103285  morale and possibly paralyze you from taking necessary actions in your job search . If that isn't bad enough, it can also stop you from being hired! Here
#103493  save you a lot of hassle later on and you should do it early in your job search . In fact, this is something we all should do at least once a year: Find
#103898  on their own in the shortest time possible. Discover more insider job search secrets by visiting http://www.jobchangesecrets.com </p><p> Art Canvas
#109614  quickest way to locate the information you want is to use the Label and Search Features. The Label pulldown is by far the easiest and quickest, just
#109648  the item of information that you require. You can use the very powerful search engine. </p><p> Remember that each section on shows 'x' number of posts
#109754  section that you are currently viewing. </p><p> Searching You can very easily search the section that you are currently in. </p><p> To use, you simply enter
#109771  section that you are currently in. </p><p> To use, you simply enter in your search item and hit Enter or click the Search Button . Your results are then
```

Figure A.2: After removing boilerplate and duplicates. The first page of concordance lines.



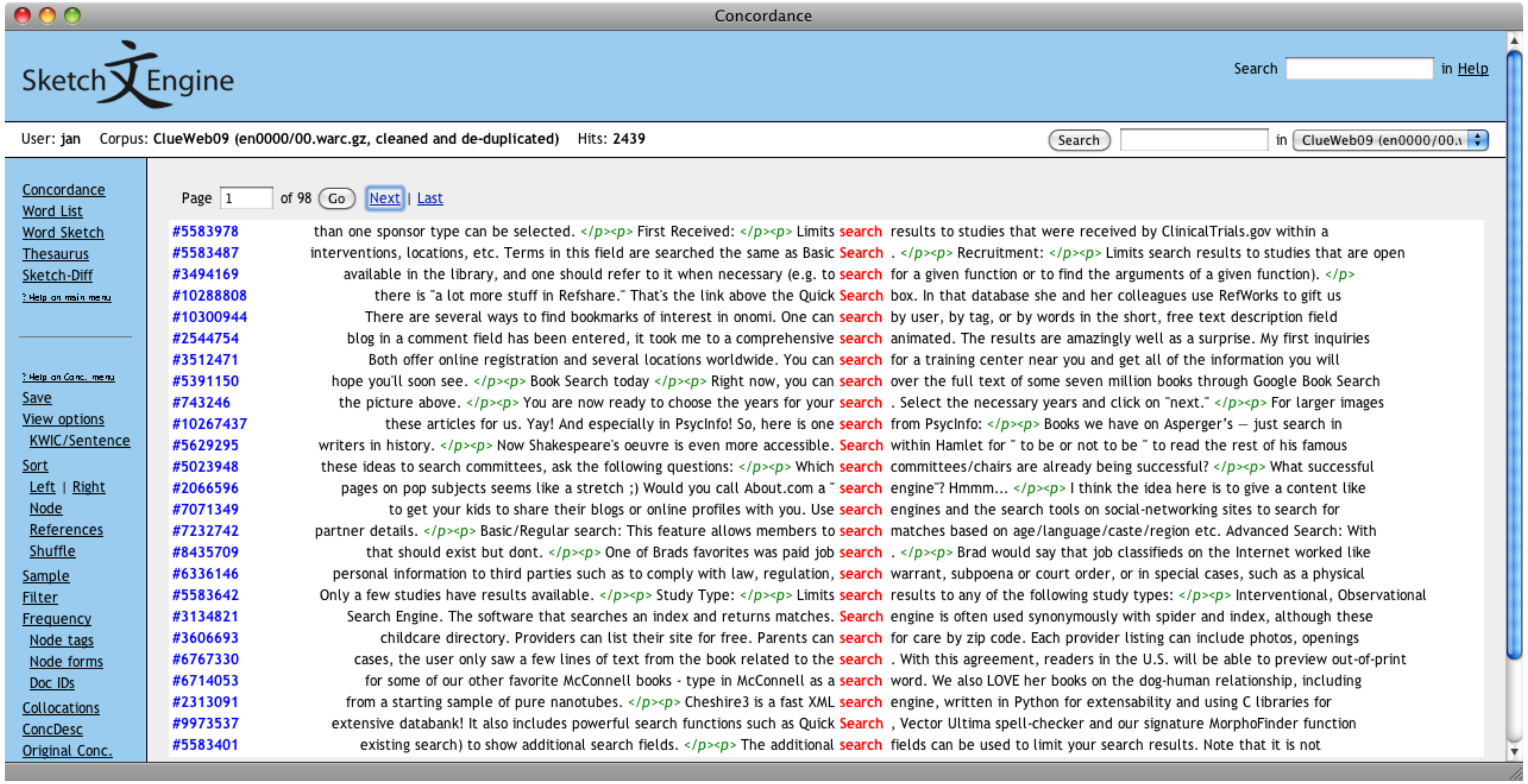
User: jan Corpus: ClueWeb09 (en0000/00.warc.gz, with boilerplate and duplicates) Hits: 29498

ClueWeb09 (en0000/00.\

Page 1 of 1180

```
#17156718  AUTOS </p><p> REAL ESTATE </p><p> CLASSIFIEDS </p><p> MLive.com - Search </p><p> SEARCH : </p><p> Bay City, MI </p><p> Home & Garden </p><p> Construction, Repair, and
#25426518  Privacy Policy | Terms of Use </p><p> 07-Jan-2009 </p><p> tunes.com </p><p> Search for Music, Ringtones, MP3's, and Free Music Downloads </p><p> home </p><p>
#6259518   This Journal </p><p> Subscriptions </p><p> Current Issue </p><p> Archive </p><p> Search </p><p> Oxford Journals </p><p> Humanities </p><p> Applied Linguistics </p><p>
#25897302  Hotspot </p><p> SSForge Business 1.24 - 30-04-2008 - Shareware Add site search engine to your website to make it searchable </p><p> ShadowProtect Small
#25626849  Children Quick Links: </p><p> Home </p><p> Adoption </p><p> Adoption Records Search </p><p> Child Protective Services </p><p> Child Welfare Program </p><p> Child
#26400563  <p> Features </p><p> Editorials </p><p> Interviews </p><p> Xbox Events </p><p> Search TeamXbox </p><p> RSS Feeds </p><p> Xbox Games </p><p> Xbox Games </p><p> New
#19253612  Billions </p><p> © MMIX, CBS Broadcasting Inc. All Rights Reserved. </p><p> Search </p><p> A </p><p> A+ </p><p> A++ </p><p> Advertisement </p><p> U.S. & World </p>
#29854045  education, residence status, profession, etc. Keyword Search: Members can search profiles using keywords. For example, you can type in the keyword '
#26615071  your PC. It provides instant bidirectional word translation and quick search functions. </p><p> Price: $34.95 </p><p> Order now More info Discount </p>
#26224476  13, 2004 2004 Designer Blinds </p><p> Subscribe To Tip Of The Week </p><p> Search For Tips About: </p><p> Relieve Tension Headaches! </p><p> Installing Universal
#6477346   Intelius.com </p><p> Terms OF Use | Comments | Help | Reverse Lookup | People Search | Yellow Pages </p><p> © Copyright 2009 www.iaf.net </p><p> PEOPLE SEARCH
#20878699  Sports </p><p> Weather </p><p> Search people or web </p><p> Search people </p><p> Search the web </p><p> Sign in </p><p> KIRK </p><p> KIRK isn't in your network. Add
#410405    Movie Reviews Movie Trailers Box Office Chart DVD Releases Message Boards Search Movies Contact Us </p><p> The A-Team Captain America Harry Potter And The
#9294105   register one and get your very own elite (but free) BGA account! </p><p> Search keywords: </p><p> Top Backgrounds </p><p> Dogs </p><p> Canada Building... </p>
#26553490  Products </p><p> Handheld dictionaries </p><p> Translation software </p><p> Search Wizard </p><p> | Handheld dictionaries | Translation software | Support
#29867516  Community </p><p> Search Profile by Country </p><p> Search Profile by State </p><p> Search Profile by City </p><p> Search by Mother Tongue </p><p> Search by Profession
#29868061  </p><p> Members </p><p> Membership Schemes </p><p> Register </p><p> Login </p><p> Search </p><p> Advanced Search </p><p> Search by Profile ID </p><p> Search by Community
#10407669  cartoonetorg.ro rasipallan map dota 6.57b.w3x download doyax.com %22Video Search %22 pillowgirls%20canalblog www%2Croshannet.magnify.com tamil-stories.
#20477540  Child Development Institute, LLC </p><p> Keeping Parents Informed </p><p> Search our site: </p><p> FREE Monthly Newsletter: </p><p> Home </p><p> Development
#17110329  JOBS </p><p> AUTOS </p><p> REAL ESTATE </p><p> CLASSIFIEDS </p><p> MLive.com - Search </p><p> SEARCH: </p><p> Ann Arbor, MI </p><p> Community & Organizations </p>
#25279677  Home </p><p> Events Calendar </p><p> Staff Directory </p><p> Contact DAQ </p><p> Search </p><p> Motor Vehicles </p><p> Smokestacks on Wheels [ Español ] </p><p> Curbing
#10148496  sites web québécois ... - [Jan 2009] </p><p> gurupearchi palan - Social Search Engine gurupearchi palan: rasi palan computer: dinamalar com/rasi "kanavu
#22833886  Search the Web for engineering pages Search the Web for application notes Search the Web for patent information Search the Web for engineering standards
#3831162   </p><p> Alliance Imperiale </p><p> Le parloir / The Oratory </p><p> advanced search </p><p> Search this Forum: </p><p> Active Users: </p><p> 1 Total </p><p> 1 Registered
#15234849  together an excellent tutorial on how to analyze your question and develop search terms and a search strategy using your own mind. </p><p> But a big part
```

Figure A.3: Before removing boilerplate and duplicates. Shuffled concordances.

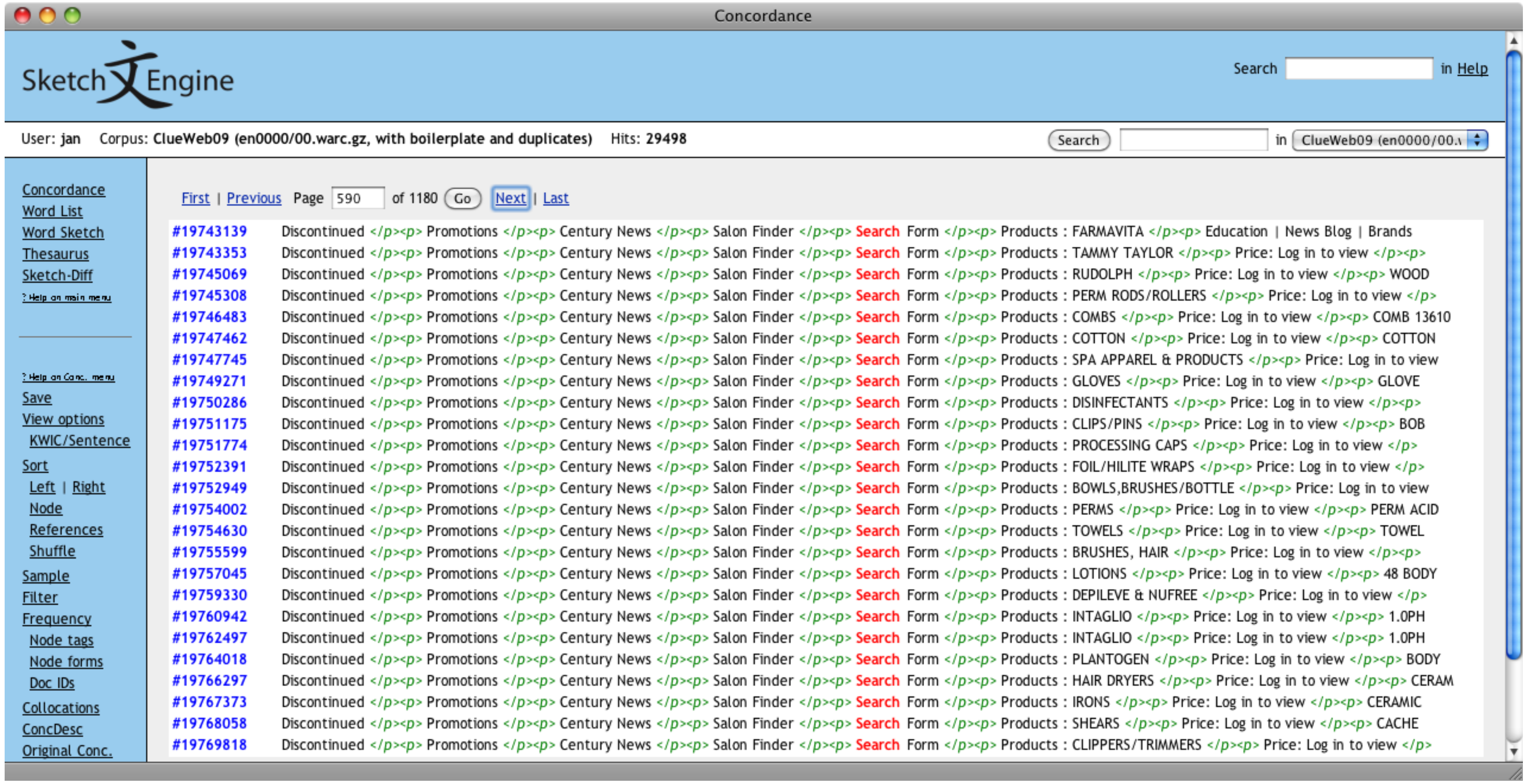


User: jan Corpus: ClueWeb09 (en0000/00.warc.gz, cleaned and de-duplicated) Hits: 2439

Page 1 of 98

#5583978	than one sponsor type can be selected. </p><p> First Received: </p><p> Limits	search	results to studies that were received by ClinicalTrials.gov within a
#5583487	interventions, locations, etc. Terms in this field are searched the same as Basic	Search	. </p><p> Recruitment: </p><p> Limits search results to studies that are open
#3494169	available in the library, and one should refer to it when necessary (e.g. to	search	for a given function or to find the arguments of a given function). </p>
#10288808	there is "a lot more stuff in Refshare." That's the link above the Quick	Search	box. In that database she and her colleagues use RefWorks to gift us
#10300944	There are several ways to find bookmarks of interest in onomi. One can	search	by user, by tag, or by words in the short, free text description field
#2544754	blog in a comment field has been entered, it took me to a comprehensive	search	animated. The results are amazingly well as a surprise. My first inquiries
#3512471	Both offer online registration and several locations worldwide. You can	search	for a training center near you and get all of the information you will
#5391150	hope you'll soon see. </p><p> Book Search today </p><p> Right now, you can	search	over the full text of some seven million books through Google Book Search
#743246	the picture above. </p><p> You are now ready to choose the years for your	search	. Select the necessary years and click on "next." </p><p> For larger images
#10267437	these articles for us. Yay! And especially in PsycInfo! So, here is one	search	from PsycInfo: </p><p> Books we have on Asperger's – just search in
#5629295	writers in history. </p><p> Now Shakespeare's oeuvre is even more accessible.	Search	within Hamlet for " to be or not to be " to read the rest of his famous
#5023948	these ideas to search committees, ask the following questions: </p><p> Which	search	committees/chairs are already being successful? </p><p> What successful
#2066596	pages on pop subjects seems like a stretch ;) Would you call About.com a "	search	engine"? Hmmm... </p><p> I think the idea here is to give a content like
#7071349	to get your kids to share their blogs or online profiles with you. Use	search	engines and the search tools on social-networking sites to search for
#7232742	partner details. </p><p> Basic/Regular search: This feature allows members to	search	matches based on age/language/caste/region etc. Advanced Search: With
#8435709	that should exist but dont. </p><p> One of Brads favorites was paid job	search	. </p><p> Brad would say that job classifieds on the Internet worked like
#6336146	personal information to third parties such as to comply with law, regulation,	search	warrant, subpoena or court order, or in special cases, such as a physical
#5583642	Only a few studies have results available. </p><p> Study Type: </p><p> Limits	search	results to any of the following study types: </p><p> Interventional, Observational
#3134821	Search Engine. The software that searches an index and returns matches.	Search	engine is often used synonymously with spider and index, although these
#3606693	childcare directory. Providers can list their site for free. Parents can	search	for care by zip code. Each provider listing can include photos, openings
#6767330	cases, the user only saw a few lines of text from the book related to the	search	. With this agreement, readers in the U.S. will be able to preview out-of-print
#6714053	for some of our other favorite McConnell books - type in McConnell as a	search	word. We also LOVE her books on the dog-human relationship, including
#2313091	from a starting sample of pure nanotubes. </p><p> Cheshire3 is a fast XML	search	engine, written in Python for extensability and using C libraries for
#9973537	extensive databank! It also includes powerful search functions such as Quick	Search	, Vector Ultima spell-checker and our signature MorphoFinder function
#5583401	existing search) to show additional search fields. </p><p> The additional	search	fields can be used to limit your search results. Note that it is not

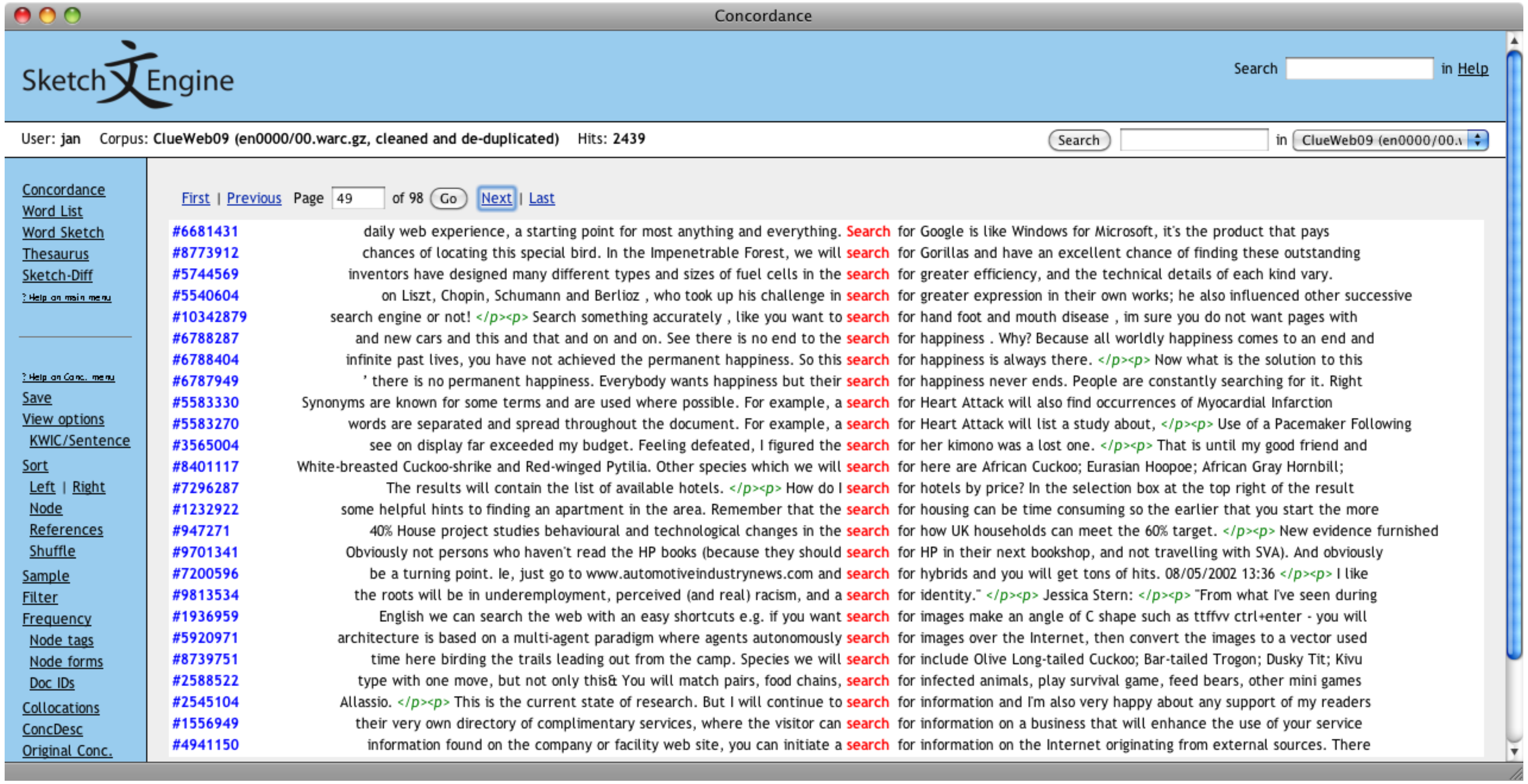
Figure A.4: After removing boilerplate and duplicates. Shuffled concordances.



User: jan Corpus: ClueWeb09 (en0000/00.warc.gz, with boilerplate and duplicates) Hits: 29498

```
#19743139   Discontinued </p><p> Promotions </p><p> Century News </p><p> Salon Finder </p><p> Search Form </p><p> Products : FARMAVITA </p><p> Education | News Blog | Brands
#19743353   Discontinued </p><p> Promotions </p><p> Century News </p><p> Salon Finder </p><p> Search Form </p><p> Products : TAMMY TAYLOR </p><p> Price: Log in to view </p><p>
#19745069   Discontinued </p><p> Promotions </p><p> Century News </p><p> Salon Finder </p><p> Search Form </p><p> Products : RUDOLPH </p><p> Price: Log in to view </p><p> WOOD
#19745308   Discontinued </p><p> Promotions </p><p> Century News </p><p> Salon Finder </p><p> Search Form </p><p> Products : PERM RODS/ROLLERS </p><p> Price: Log in to view </p>
#19746483   Discontinued </p><p> Promotions </p><p> Century News </p><p> Salon Finder </p><p> Search Form </p><p> Products : COMBS </p><p> Price: Log in to view </p><p> COMB 13610
#19747462   Discontinued </p><p> Promotions </p><p> Century News </p><p> Salon Finder </p><p> Search Form </p><p> Products : COTTON </p><p> Price: Log in to view </p><p> COTTON
#19747745   Discontinued </p><p> Promotions </p><p> Century News </p><p> Salon Finder </p><p> Search Form </p><p> Products : SPA APPAREL & PRODUCTS </p><p> Price: Log in to view
#19749271   Discontinued </p><p> Promotions </p><p> Century News </p><p> Salon Finder </p><p> Search Form </p><p> Products : GLOVES </p><p> Price: Log in to view </p><p> GLOVE
#19750286   Discontinued </p><p> Promotions </p><p> Century News </p><p> Salon Finder </p><p> Search Form </p><p> Products : DISINFECTANTS </p><p> Price: Log in to view </p><p>
#19751175   Discontinued </p><p> Promotions </p><p> Century News </p><p> Salon Finder </p><p> Search Form </p><p> Products : CLIPS/PINS </p><p> Price: Log in to view </p><p> BOB
#19751774   Discontinued </p><p> Promotions </p><p> Century News </p><p> Salon Finder </p><p> Search Form </p><p> Products : PROCESSING CAPS </p><p> Price: Log in to view </p>
#19752391   Discontinued </p><p> Promotions </p><p> Century News </p><p> Salon Finder </p><p> Search Form </p><p> Products : FOIL/HILITE WRAPS </p><p> Price: Log in to view </p>
#19752949   Discontinued </p><p> Promotions </p><p> Century News </p><p> Salon Finder </p><p> Search Form </p><p> Products : BOWLS,BRUSHES/BOTTLE </p><p> Price: Log in to view
#19754002   Discontinued </p><p> Promotions </p><p> Century News </p><p> Salon Finder </p><p> Search Form </p><p> Products : PERMS </p><p> Price: Log in to view </p><p> PERM ACID
#19754630   Discontinued </p><p> Promotions </p><p> Century News </p><p> Salon Finder </p><p> Search Form </p><p> Products : TOWELS </p><p> Price: Log in to view </p><p> TOWEL
#19755599   Discontinued </p><p> Promotions </p><p> Century News </p><p> Salon Finder </p><p> Search Form </p><p> Products : BRUSHES, HAIR </p><p> Price: Log in to view </p><p>
#19757045   Discontinued </p><p> Promotions </p><p> Century News </p><p> Salon Finder </p><p> Search Form </p><p> Products : LOTIONS </p><p> Price: Log in to view </p><p> 48 BODY
#19759330   Discontinued </p><p> Promotions </p><p> Century News </p><p> Salon Finder </p><p> Search Form </p><p> Products : DEPILEVE & NUFREE </p><p> Price: Log in to view </p>
#19760942   Discontinued </p><p> Promotions </p><p> Century News </p><p> Salon Finder </p><p> Search Form </p><p> Products : INTAGLIO </p><p> Price: Log in to view </p><p> 1.0PH
#19762497   Discontinued </p><p> Promotions </p><p> Century News </p><p> Salon Finder </p><p> Search Form </p><p> Products : INTAGLIO </p><p> Price: Log in to view </p><p> 1.0PH
#19764018   Discontinued </p><p> Promotions </p><p> Century News </p><p> Salon Finder </p><p> Search Form </p><p> Products : PLANTOGEN </p><p> Price: Log in to view </p><p> BODY
#19766297   Discontinued </p><p> Promotions </p><p> Century News </p><p> Salon Finder </p><p> Search Form </p><p> Products : HAIR DRYERS </p><p> Price: Log in to view </p><p> CERAM
#19767373   Discontinued </p><p> Promotions </p><p> Century News </p><p> Salon Finder </p><p> Search Form </p><p> Products : IRONS </p><p> Price: Log in to view </p><p> CERAMIC
#19768058   Discontinued </p><p> Promotions </p><p> Century News </p><p> Salon Finder </p><p> Search Form </p><p> Products : SHEARS </p><p> Price: Log in to view </p><p> CACHE
#19769818   Discontinued </p><p> Promotions </p><p> Century News </p><p> Salon Finder </p><p> Search Form </p><p> Products : CLIPPERS/TRIMMERS </p><p> Price: Log in to view </p>
```

Figure A.5: Before removing boilerplate and duplicates. Concordances sorted by right context of the keyword.



User: jan Corpus: ClueWeb09 (en0000/00.warc.gz, cleaned and de-duplicated) Hits: 2439

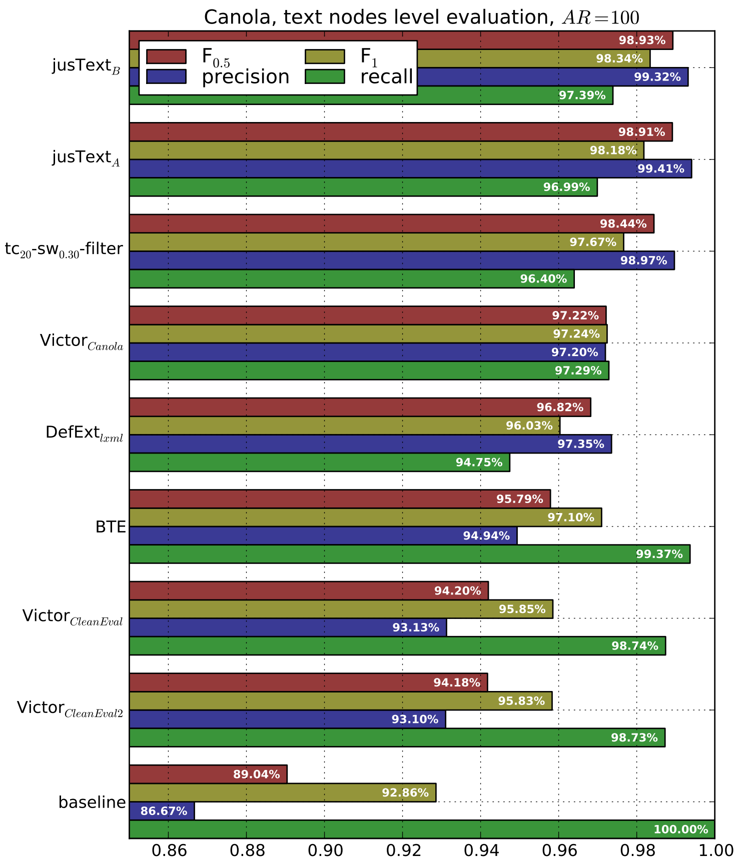
```
#6681431     daily web experience, a starting point for most anything and everything. Search for Google is like Windows for Microsoft, it's the product that pays
#8773912     chances of locating this special bird. In the Impenetrable Forest, we will search for Gorillas and have an excellent chance of finding these outstanding
#5744569     inventors have designed many different types and sizes of fuel cells in the search for greater efficiency, and the technical details of each kind vary.
#5540604     on Liszt, Chopin, Schumann and Berlioz , who took up his challenge in search for greater expression in their own works; he also influenced other successive
#10342879    search engine or not! </p><p> Search something accurately , like you want to search for hand foot and mouth disease , im sure you do not want pages with
#6788287     and new cars and this and that and on and on. See there is no end to the search for happiness . Why? Because all worldly happiness comes to an end and
#6788404     infinite past lives, you have not achieved the permanent happiness. So this search for happiness is always there. </p><p> Now what is the solution to this
#6787949     ' there is no permanent happiness. Everybody wants happiness but their search for happiness never ends. People are constantly searching for it. Right
#5583330     Synonyms are known for some terms and are used where possible. For example, a search for Heart Attack will also find occurrences of Myocardial Infarction
#5583270     words are separated and spread throughout the document. For example, a search for Heart Attack will list a study about, </p><p> Use of a Pacemaker Following
#3565004     see on display far exceeded my budget. Feeling defeated, I figured the search for her kimono was a lost one. </p><p> That is until my good friend and
#8401117     White-breasted Cuckoo-shrike and Red-winged Pytilia. Other species which we will search for here are African Cuckoo; Eurasian Hoopoe; African Gray Hornbill;
#7296287     The results will contain the list of available hotels. </p><p> How do I search for hotels by price? In the selection box at the top right of the result
#1232922     some helpful hints to finding an apartment in the area. Remember that the search for housing can be time consuming so the earlier that you start the more
#947271      40% House project studies behavioural and technological changes in the search for how UK households can meet the 60% target. </p><p> New evidence furnished
#9701341     Obviously not persons who haven't read the HP books (because they should search for HP in their next bookshop, and not travelling with SVA). And obviously
#7200596     be a turning point. Ie, just go to www.automotiveindustrynews.com and search for hybrids and you will get tons of hits. 08/05/2002 13:36 </p><p> I like
#9813534     the roots will be in underemployment, perceived (and real) racism, and a search for identity." </p><p> Jessica Stern: </p><p> "From what I've seen during
#1936959     English we can search the web with an easy shortcuts e.g. if you want search for images make an angle of C shape such as ttffvv ctrl+enter - you will
#5920971     architecture is based on a multi-agent paradigm where agents autonomously search for images over the Internet, then convert the images to a vector used
#8739751     time here birding the trails leading out from the camp. Species we will search for include Olive Long-tailed Cuckoo; Bar-tailed Trogon; Dusky Tit; Kivu
#2588522     type with one move, but not only this& You will match pairs, food chains, search for infected animals, play survival game, feed bears, other mini games
#2545104     Allassio. </p><p> This is the current state of research. But I will continue to search for information and I'm also very happy about any support of my readers
#1556949     their very own directory of complimentary services, where the visitor can search for information on a business that will enhance the use of your service
#4941150     information found on the company or facility web site, you can initiate a search for information on the Internet originating from external sources. There
```

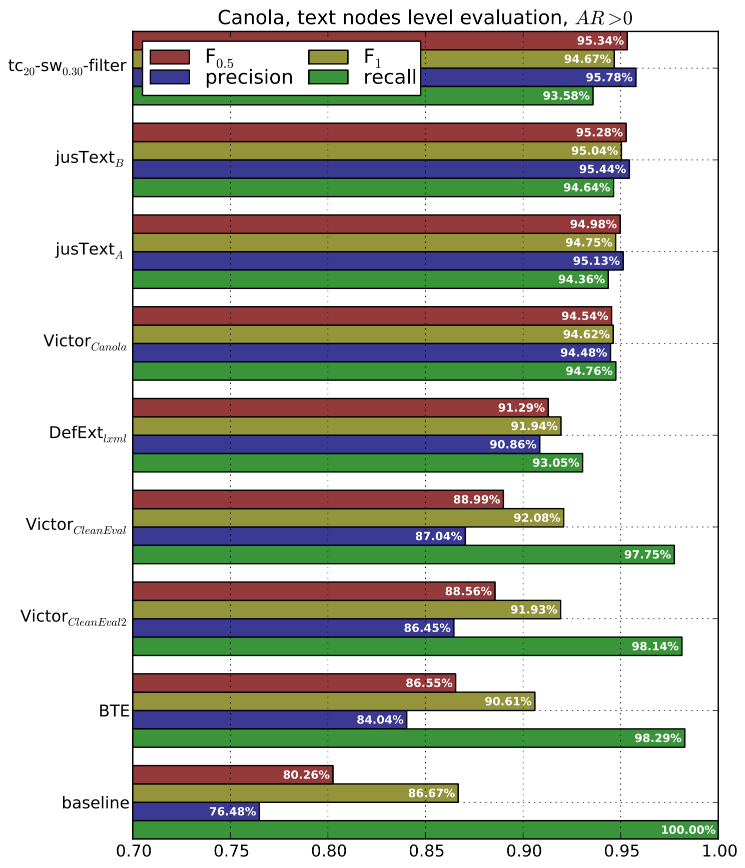
Figure A.6: After removing boilerplate and duplicates. Concordances sorted by right context of the keyword.

Appendix B

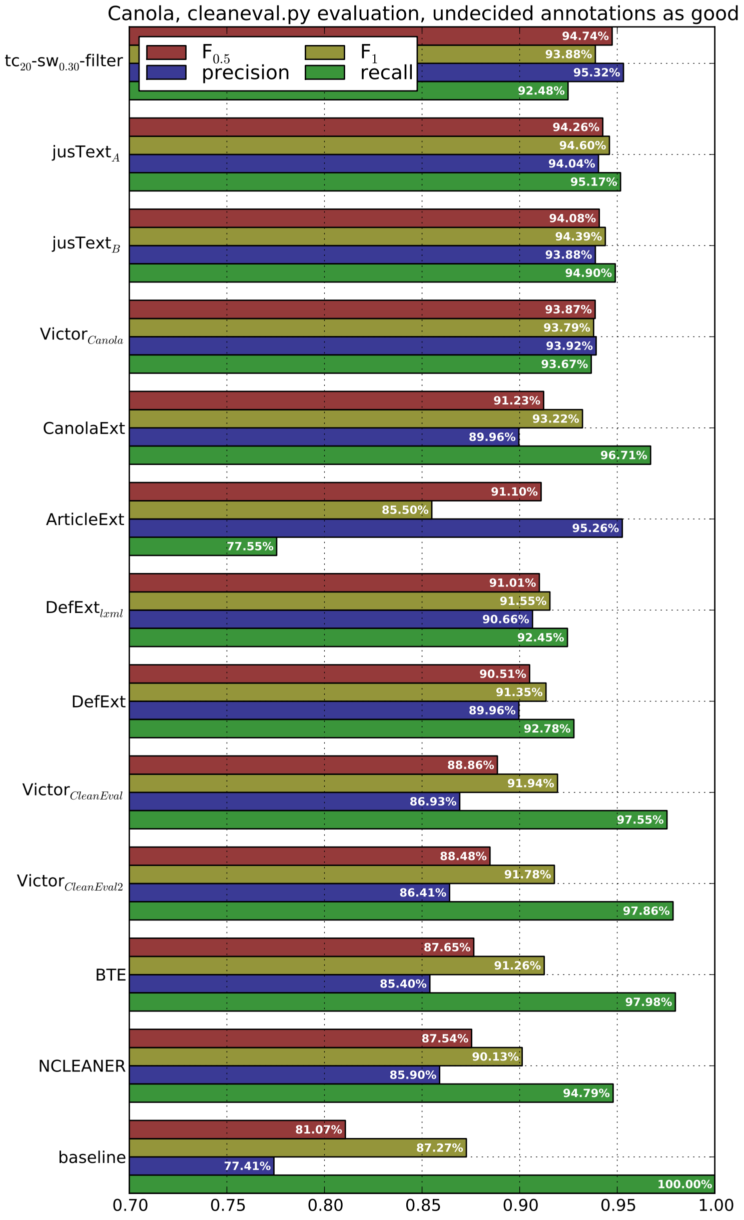
Removing boilerplate in figures

In the figures below, algorithms are ordered by $F_{0.5}$ scores. The details about the evaluation can be found in sections 2.9.6, 2.9.7, and 2.9.8.

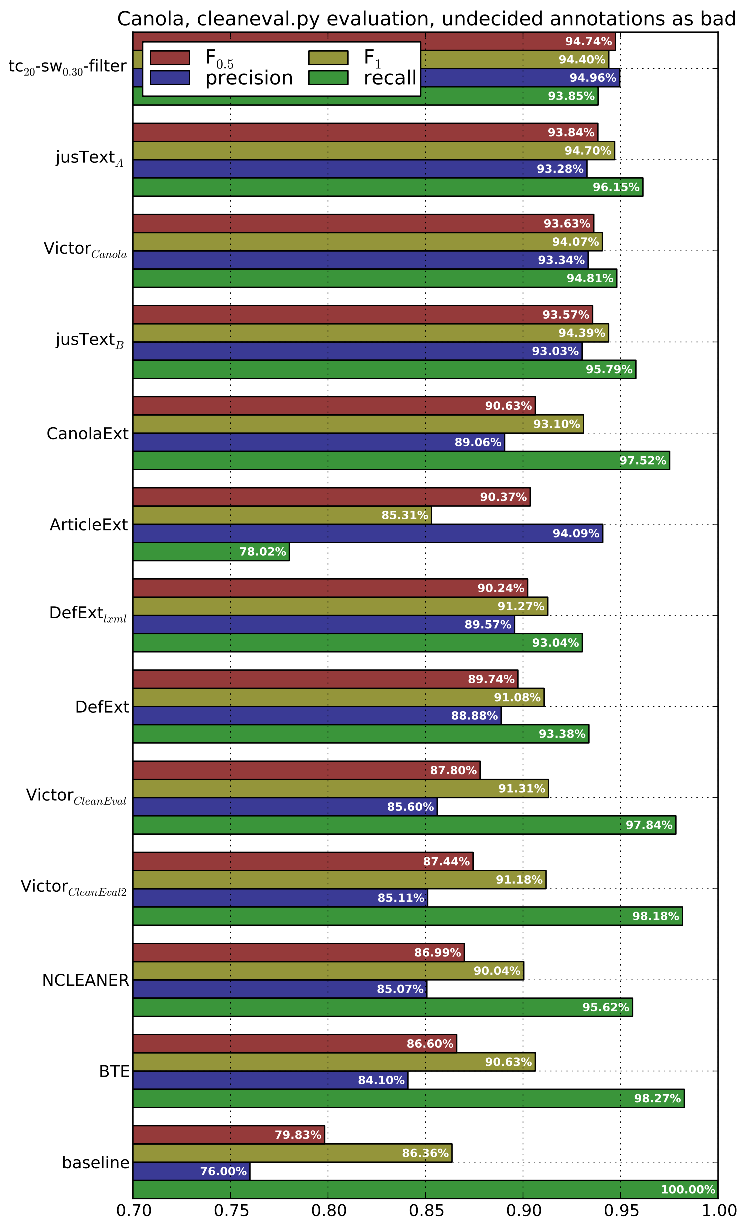


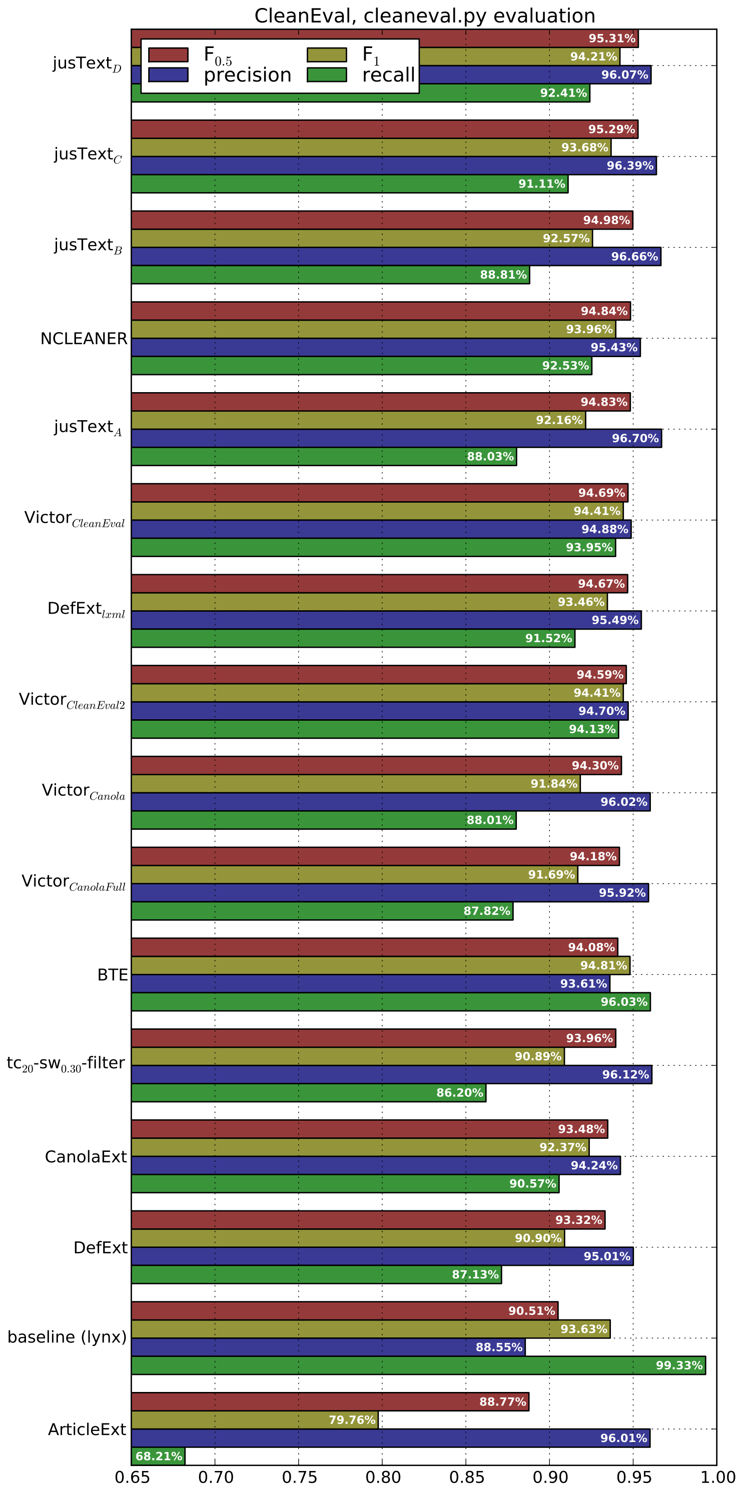

Canola, text nodes level evaluation, $AR>0$
$F_{0.5}$
F_1
precision
recall
tc$_{20}$-sw$_{0.30}$-filter
95.34%
94.67%
95.78%
93.58%
jusText$_B$
95.28%
95.04%
95.44%
94.64%
jusText$_A$
94.98%
94.75%
95.13%
94.36%
Victor$_{Canola}$
94.54%
94.62%
94.48%
94.76%
DefExt$_{lxml}$
91.29%
91.94%
90.86%
93.05%
Victor$_{CleanEval}$
88.99%
92.08%
87.04%
97.75%
Victor$_{CleanEval2}$
88.56%
91.93%
86.45%
98.14%
BTE
86.55%
90.61%
84.04%
98.29%
baseline
80.26%
86.67%
76.48%
100.00%
0.70
0.75
0.80
0.85
0.90
0.95
1.00

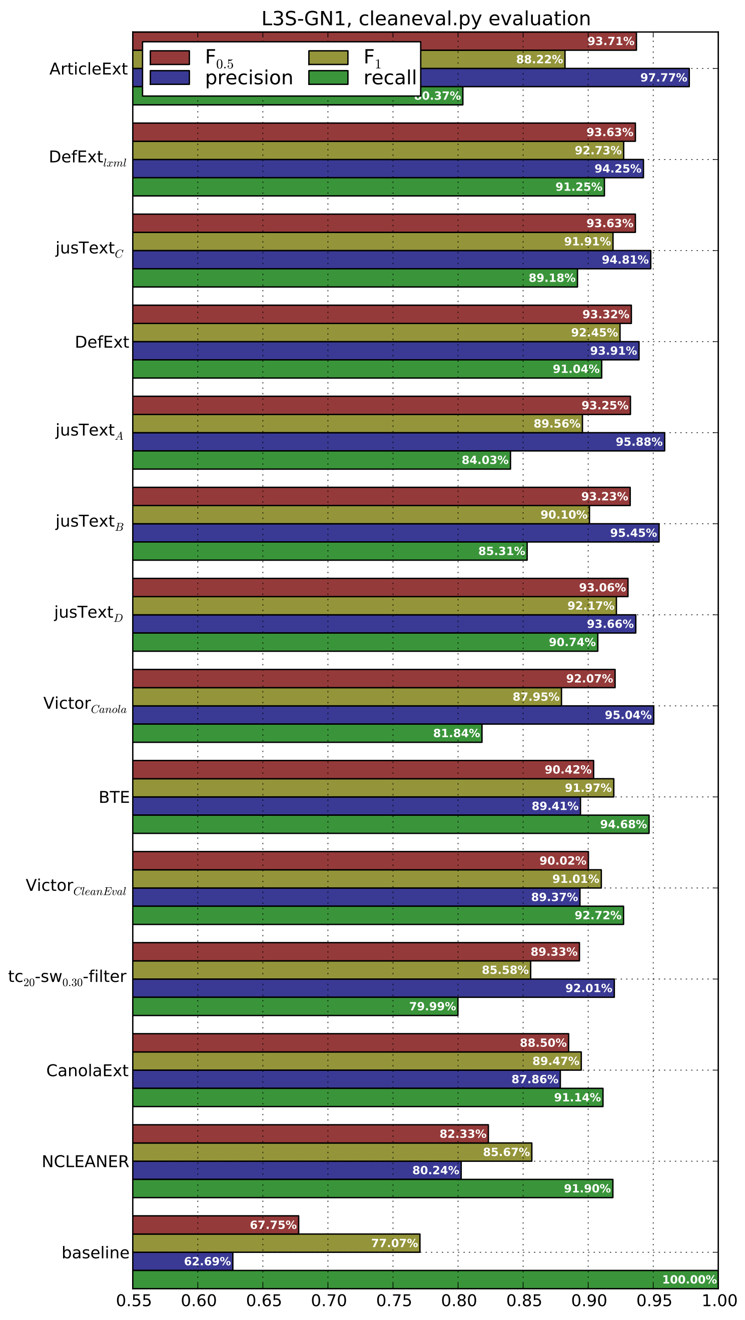
Canola, cleaneval.py evaluation, undecided annotations as good



	$F_{0.5}$	F_1	precision	recall
tc$_{20}$-sw$_{0.30}$-filter	94.74%	93.88%	95.32%	92.48%
jusText$_A$	94.26%	94.60%	94.04%	95.17%
jusText$_B$	94.08%	94.39%	93.88%	94.90%
Victor$_{Canola}$	93.87%	93.79%	93.92%	93.67%
CanolaExt	91.23%	93.22%	89.96%	96.71%
ArticleExt	91.10%	85.50%	95.26%	77.55%
DefExt$_{lxml}$	91.01%	91.55%	90.66%	92.45%
DefExt	90.51%	91.35%	89.96%	92.78%
Victor$_{CleanEval}$	88.86%	91.94%	86.93%	97.55%
Victor$_{CleanEval2}$	88.48%	91.78%	86.41%	97.86%
BTE	87.65%	91.26%	85.40%	97.98%
NCLEANER	87.54%	90.13%	85.90%	94.79%
baseline	81.07%	87.27%	77.41%	100.00%


Canola, cleaneval.py evaluation, undecided annotations as bad
$F_{0.5}$
F_1
precision
recall
tc$_{20}$-sw$_{0.30}$-filter
94.74%
94.40%
94.96%
93.85%
jusText$_A$
93.84%
94.70%
93.28%
96.15%
Victor$_{Canola}$
93.63%
94.07%
93.34%
94.81%
jusText$_B$
93.57%
94.39%
93.03%
95.79%
CanolaExt
90.63%
93.10%
89.06%
97.52%
ArticleExt
90.37%
85.31%
94.09%
78.02%
DefExt$_{lxml}$
90.24%
91.27%
89.57%
93.04%
DefExt
89.74%
91.08%
88.88%
93.38%
Victor$_{CleanEval}$
87.80%
91.31%
85.60%
97.84%
Victor$_{CleanEval2}$
87.44%
91.18%
85.11%
98.18%
NCLEANER
86.99%
90.04%
85.07%
95.62%
BTE
86.60%
90.63%
84.10%
98.27%
baseline
79.83%
86.36%
76.00%
100.00%
0.70
0.75
0.80
0.85
0.90
0.95
1.00


CleanEval, cleaneval.py evaluation
$F_{0.5}$
F_1
precision
recall
jusText_D
95.31%
94.21%
96.07%
92.41%
jusText_C
95.29%
93.68%
96.39%
91.11%
jusText_B
94.98%
92.57%
96.66%
88.81%
NCLEANER
94.84%
93.96%
95.43%
92.53%
jusText_A
94.83%
92.16%
96.70%
88.03%
$\text{Victor}_{CleanEval}$
94.69%
94.41%
94.88%
93.95%
DefExt_{lxml}
94.67%
93.46%
95.49%
91.52%
$\text{Victor}_{CleanEval2}$
94.59%
94.41%
94.70%
94.13%
Victor_{Canola}
94.30%
91.84%
96.02%
88.01%
$\text{Victor}_{CanolaFull}$
94.18%
91.69%
95.92%
87.82%
BTE
94.08%
94.81%
93.61%
96.03%
$\text{tc}_{20}\text{-sw}_{0.30}\text{-filter}$
93.96%
90.89%
96.12%
86.20%
CanolaExt
93.48%
92.37%
94.24%
90.57%
DefExt
93.32%
90.90%
95.01%
87.13%
baseline (lynx)
90.51%
93.63%
88.55%
99.33%
ArticleExt
88.77%
79.76%
96.01%
68.21%
0.65
0.70
0.75
0.80
0.85
0.90
0.95
1.00



L3S-GN1, cleaneval.py evaluation
$F_{0.5}$
F_1
precision
recall
ArticleExt
93.71%
88.22%
97.77%
80.37%
$DefExt_{lxml}$
93.63%
92.73%
94.25%
91.25%
$jusText_C$
93.63%
91.91%
94.81%
89.18%
DefExt
93.32%
92.45%
93.91%
91.04%
$jusText_A$
93.25%
89.56%
95.88%
84.03%
$jusText_B$
93.23%
90.10%
95.45%
85.31%
$jusText_D$
93.06%
92.17%
93.66%
90.74%
$Victor_{Canola}$
92.07%
87.95%
95.04%
81.84%
BTE
90.42%
91.97%
89.41%
94.68%
$Victor_{CleanEval}$
90.02%
91.01%
89.37%
92.72%
tc_{20}-$sw_{0.30}$-filter
89.33%
85.58%
92.01%
79.99%
CanolaExt
88.50%
89.47%
87.86%
91.14%
NCLEANER
82.33%
85.67%
80.24%
91.90%
baseline
67.75%
77.07%
62.69%
100.00%
0.55
0.60
0.65
0.70
0.75
0.80
0.85
0.90
0.95
1.00