# Towards Automated Creation and Management of a South African English Web Corpus

Submitted in partial fulfilment
of the requirements of the degree of

Bachelor of Arts (Honours)

of Rhodes University

Gareth Terence Bryant Dwyer

*Grahamstown, South Africa*
October 2014

**Abstract**

A corpus is a large collection of classified text, from which knowledge about how natural language is used can be extracted. Corpora are used by linguists and lexicographers to analyse language and to compile dictionaries. Although corpora can be built from a variety of data sources, the World Wide Web is particularly suitable as data exist in large quantities and are already in digital form.

For this research, a system was implemented which gathers and stores data from South African Web sites, and furthermore monitors a set of online feeds for new data, and provides a variety of analysis functionality through a web interface. Using the system, an evolving web corpus of South African English was created.

This research details a breakdown of steps and components which were required to build and maintain an evolving language-specific web corpus, including the gathering and storing of data, the cleaning or 'boilerplate removal' of Web pages, near and exact deduplication of content, and the various analysis tools implemented which are commonly used by language researchers.

### ACM Computing Classification System Classification

Thesis classification under the ACM Computing Classification System (2012 version, valid through 2014):
[500] Information systems Deduplication
[500] Information systems Data cleaning
[300] Information systems Near-duplicate and plagiarism detection
[300] Information systems Similarity measures

**General-Terms:** Corpus, Corpora, Natural Language, Concordancer, Boilerplate, Deduplication, Web Crawling

**Acknowledgements**

Many thanks go to everyone involved in the financial and moral support that allowed me to spend this highly enjoyable year at Rhodes University, especially to close friends and family (thanks Mom and Dad).

Thank you to my supervisor James Connan for all the guidance and advice, and to all the wonderful staff of the Hamilton Building, which has been something of a home to me this year. Mention also goes to those people who knew me only as a CV and who decided to generously award me various bursaries.

And last but not least, a special mention to Red Cafe: Lou & Co., no matter where I end up, I am sure that some of my fondest memories will always be of the copius amounts of coffee and time consumed on your wooden deck.

May good karma and appreciation ever rain down upon you all.

* * *

This work was undertaken in the Distributed Multimedia CoE at Rhodes University, with financial support from Telkom SA, Tellabs, Genband, Easttel, Bright Ideas 39, THRIP and NRF SA (TP13070820716). The authors acknowledge that opinions, findings and conclusions or recommendations expressed here are those of the author(s) and that none of the above mentioned sponsors accept liability whatsoever in this regard.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Corpus Definitions and Use

A corpus is a large body of classified text, from which knowledge about how language is being used can be extracted. Corpora have become essential for lexicographers, who use them to write accurate entries for dictionaries. For example, Oxford University Press (OUP, 2014) state, "The Oxford English Corpus is at the heart of dictionary-making in Oxford". A Web corpus is a corpus which sources data from the World Wide Web, usually gathering data through the use of a Web crawler. A language-specific corpus is a corpus which contains texts from only a single language, or language variant (in our case, South African English). While a corpus may be built from data gathered at a single point in time, or within some limited time frame, an evolving corpus is one to which new texts are constantly added. Evolving corpora allow for more interesting language analysis, including analysis into how language changes and evolves over time, and the influences of these changes.

## 1.2 The Need for a Corpus of South African English

A corpus of South African English (SAE) is needed by the Dictionary Unit for South African English (DSAE), who are the publishers of the *Oxford South African Concise Dictionary* and acknowledged authority on South African English, and also by the English Language and Linguistics Department of Rhodes University (RU, 2014a,b). The DSAE

are currently collecting all data used in compiling dictionaries for South African English manually and storing this data in spreadsheets, a solution which does not scale well, and severely limits the amount of data that can be gathered, as well as the depth and usefulness of the analysis which can be run.

## 1.3 Potential Challenges and Research Focus

Creating a corpus with a majority of its content sourced from the World Wide Web is not a straightforward task. This is noted by Liu & Curran (2006, p. 234), who state:

> There are many challenges in creating a Web corpus, as the World Wide Web is unstructured and without a definitive directory. No simple method exists to collect a large representative sample of the Web.

Creators of language-specific corpora face the additional challenge of collecting not just a representative sample of the Web, but instead a representative sample of a subsection of the Web, that is, Web pages which are written in a specific language or language variant. Another challenge is found in so-called 'boilerplate', which is the extraneous data found in a majority of Web pages. With Web pages becoming increasingly rich and varied in content, it is now a non-trivial task to automatically identify the main textual content of a Web page and separate this from extraneous data such as headers, footers, adverts, sidebars, viewer comments, and other data commonly included within the Hyper Text Markup Language (HTML) of a Web page. With sites becoming richer and more complex, dynamic data is becoming increasingly common. This content is injected into the HTML of a Web page dynamically using technology such as Asynchronous JavaScript and XML (AJAX), usually when the viewer of the page presses a button on the page, or even when the viewer activates certain JavaScript 'events', such as an 'on-scroll event'. This can be problematic in the automatic collection of Web page data as Web crawlers or any software which accesses a Web page without using a Web browser does not by default trigger these dynamic controls, and the automated system will therefore often retrieve only an incomplete version of the page. The final challenge that this research will aim to overcome is that of duplication. With press agencies selling articles to multiple clients, and with plagiarism becoming ever easier, there is a large amount of duplicate or very similar content to be found on the Web. Duplicate or near-duplicate content in a corpus can badly skew linguistic analysis, and it therefore needs to be identified and removed.

While identical content is trivial to find and remove by using hashing and checking for collisions, identifying near-duplicate content is far more difficult, as calculating similarity measures for every article in a corpus against every other article becomes infeasible as the corpus size grows.

## 1.4 Organization of Research

This research addresses each of the challenges mentioned above individually, before showing how the proposed solutions can be combined into a single system. The sections are addressed in the following order:

- Chapter 2: Gathering and Storing Data
  *A qualitative evaluation of methods to gather and store data.*
- Chapter 3: Cleaning Data
  *A comparison of boilerplate removal and data cleaning algorithms.*
- Chapter 4: Deduplication
  *A comparison of deduplication algorithms.*
- Chapter 5: Analysis Tools
  *A description of theory and implementation of linguistic analysis tools.*
- Chapter 6: System Design
  *An overview of how the modules are combined into a single system.*
- Chapter 7: Conclusion and Future Work
  *An overview of our work, and a look at how it could be extended.*

## 1.5 Overview of Existing Corpora

We give here a brief overview of some existing corpora. These are of interest either because they relate to South African English or because they describe the methodology used in the creation of language-specific Web corpora. Of the former, only very sparse and incomplete work has been done, but of the latter, there are several large and well-documented examples. The examination of the tools and algorithms that the researchers

used to create these corpora is left for the later relevant sections, but we outline here a description of each, stating their sizes and to which of our sub-goals their research pertained most closely.

### 1.5.1 South African Corpora

Although there is no current electronic corpus of South African English, related work has been done in the creation of such a corpus, notably by Pienaar & De Klerk (2009) who created a small corpus specifically of Indian South African English (ISAE); De Klerk (2002), who outlined the design for a full corpus of Xhosa English and Black South African English; and Jeffery (2003), who aimed to create a South African corpus as a component of the International Corpus of English. All of these attempts at South African corpora focused on a subset of South African English (such as "Indian South African English" or "Black South African English"), and all used manually-transcribed audio recordings to some extent, and were therefore very limited in size.

### 1.5.2 Language-specific Corpora

**Web Corpus for NLP** Liu & Curran (2006) built a 10 billion word corpus of English in order to help train systems for natural language processing tasks, including context-sensitive spelling correction. This corpus was created using Web spiders, based on a topic-diverse set of seed Universal Resource Locators (URLs). It used a machine-learning based approach for cleaning content and a rule-based approach for filtering out non-useful text. We later examine their methods used for both gathering data and boilerplate removal.

**Web as Corpus (WaCky)** The *WaCky* corpus is a collection of the three language-specific corpora—built simultaneously using the same method—"ukWaC", "deWaC" and "itWaC", respectively for British English, German, and Italian (Baroni *et al.*, 2009). This collection consisted of over 1 TB of data, with 350–400 GB of raw data per corpus. However, after document filtering (including discarding all documents that were not of mime type text/html, smaller than 5 KB or larger than 200 KB) this fell to 60 GB, and after near-duplicate removal, to 35 GB. Over 4.5 billion tokens were extracted from the approximately 6 million remaining documents (Baroni *et al.*, 2009). In later sections, we take special interest in the methods used for cleaning and deduplication of these corpora,

and also more briefly examine the Heritrix Web crawler used by the WaCky researches for gathering the content.

**NoWaC** (Guevara, 2010) closely followed the research of (Baroni *et al.*, 2009) in order to build a Web corpus of Norwegian. It has a final size of about half of *WaCky's*, with a raw crawl size of 550 GB, and a final result of 690 million tokens and 0.85 million documents. We examine the methodology of this corpus, as for that of (Baroni *et al.*, 2009), as it pertains to cleaning, deduplication, and data gathering.

**Reference Corpus of Contemporary Portuguese (CRPC)** The CRPC does not explicitly state statistics on number of documents, but 312 million tokens were extracted after cleaning and deduplication (Généreux *et al.*, 2012). No details are given on how data was gathered, and it seems that the researchers were not concerned about duplicate content, but they did complete an evaluation of cleaning tools which is of interest to us.

**RSS Feed Evolving Corpus** Prabowo & Thelwall (2006) created an evolving Web corpus from Rich Site Summary (RSS) feeds. Over 19 000 feeds, sourced from Google and Blogstreet[1] were monitored on an hourly or daily basis, with new data being continuously added to the corpus. It was used to identify significant topics and trends in newspapers and blogs, and therefore differed in purpose and goals to the corpora previously described. However, it remains of interest to us for its methods of using RSS feeds to gather data.

## 1.6 Summary

There is an unfulfilled need for an evolving language specific Web corpus of South African English in order to allow language researchers to draw insight into how our language is being used. Limited attempts to build corpora specific to South African English have been made, but these have all been small and specific to sub-categories of South African English such as Black South African English or Indian South African English. Other language-specific Web corpora have been created, and these have followed different methods for the gathering, storing, cleaning, deduplication, and analysis of data. Because there is no obviously optimum way to achieve any of these tasks, the different methods need to be compared and evaluated.

[1]The authors source this as http://www.blogstreet.com, but this domain is currently parked and available for sale

## 1.7 Research Question and Goals

Can we create an evolving text corpus of South African English, with content sourced automatically from the World Wide Web, either evaluating and using or creating tools for the necessary sub-goals of data collection and storage, cleaning, deduplication, and language analysis? We examine prior work in creating language specific corpora, and where there exists no *de facto* best method for achieving a specific goal, we identify and evaluate different tools and algorithms. The research will facilitate the current efforts of the DSAE and the Rhodes University Linguistics Department, while extending research possibilities that involve South African English, and aims to furthermore provide a set of guidelines for any future researchers who attempt to build evolving, language-specific Web corpora.

# Chapter 2

# Gathering and Storing Data

The first step in building a Web corpus, before cleaning, deduplication, or analysis can take place, is to source, gather, and store data. There are several ways to achieve this. Although some of the corpora examined provide details in how gathering data was achieved, there is little information about what storage systems were used. In this chapter, we look at three ways in which data can be gathered. These are: 1) crawling the Web by starting at a specific page or set of pages, and recursively gathering links from these; 2) gathering data through Application Programming Interfaces (APIs); and 3) using Rich Site Summary (RSS) feeds to continually collect data shortly after it is first published. For data storage, we identify two alternatives, looking at traditional Relational Database Management Systems (RDBMSs) and NoSQL data stores, specifically MongoDB.

## 2.1 Data Gathering

Data gathering can be divided into two subcomponents, referred to here as 'backward crawling' and 'forward crawling'. The first indicates gathering existing content, while the second indicates a system that gathers articles as they are published. Backwards crawling is accomplished either through crawling the World Wide Web, that is following URLs recursively for a specific set of start URLs, or by using Application Programming Interfaces (APIs) for services such as archive.org, which stores snapshots of Web pages as they appeared at specific points in time. Forward crawling can be achieved by using Rich Site Summary (RSS) feeds, which alert consumers to new articles as they are published.

The corpus systems already mentioned used either a customized Web spider approach—in which spiders were seeded with selected URLs, such as the home pages of academic institutions, and Web pages were scraped recursively from these, by visiting each page linked to by the seed page, and each page linked to by these, to some predefined depth—or a search engine Application Programming Interface (API) approach—in which search terms were selected and pages fetched programmatically through commercial search engines. Another way to gather data is by watching Real Site Syndication (RSS) feeds, and downloading new content as it is published. A relatively new challenge in scraping Web content is found in an ever-increasing amount of data being dynamically generated, based on a user's interactions with a page. For example, comments on some online news sites are only loaded through an AJAX call made by a Web browser when the user scrolls to the bottom of the article page, or when the user clicks on a specific 'load comments' button. With JavaScript communication to the server changing the data on the page, a simple HTTP request using a URL cannot be relied upon to return the correct and complete data, and retrieving this dynamic data automatically is therefore a complicated task.

In this chapter, we describe the possible methods we have identified for gathering data, followed by our results showing that using APIs for backwards crawling and for gathering dynamic data can be preferable, and the methods we chose to use. We then look at the advantages and disadvantages of NoSQL systems, stating why we decided to use MongoDB to store all data for the corpus of South African English.

### 2.1.1 Crawling the Web

Web crawlers, also known as Web spiders, recursively gather data based on URLs found on each page that they visit. Given a start URL, a crawler will visit the page, extract all URLs from this page, add these to a queue, and recursively crawl each URL in this queue, saving each page as it progresses. A rudimentary crawler can easily be created in several lines of code. However some more refinement is usually desired, such as functionality to avoid visiting pages more than once, some degree of parallelization or other means of improving performance, and the ability to follow "rules", such as selecting which data formats are relevant and which to ignore, following rules found in a site's robots.txt file, and ignoring or prioritizing pages based on keywords in the URL. Customizable crawlers and Web scraping frameworks exist which can provide this functionality, but there exists no *de facto* best way of achieving this. Pomikálek (2011, p. 16) states, "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 developed by Internet Archive." Liu & Curran (2006) also used a spidering approach to gather data, using seed URLs from the Open Directory[1], which provides a list of URLs organized by category.

**Heritrix Crawler** The Heritrix crawler was designed to fit as many use cases as possible, and is therefore useful in broad crawling (prioritizing amount of data), focused crawling (where quality of content is a priority), continuous crawling (looking for changes on already-crawled sites) and experimental crawling (everything else) (Mohr *et al.*, 2004, p. 3). It was developed by the Internet Archive, with primary goals being extensibility and openness. It is therefore released under the GNU Lesser General Public License (LGPL) and written in object oriented, modularized and documented Java. The Heritrix crawler also takes advantage of multi threading to speed up retrieval of Web pages (Mohr *et al.*, 2004). This crawler was used by Baroni *et al.* (2009) in building the NoWaC corpus collection, as well as by Guevara (2010) in the creation of the corpus of Norwegian.

**Scrapy** Scrapy is another scraping framework, written in Python, which also focuses on being extensible and is open-source. Scrapy is built on the Python Twisted library, which makes use of asynchronous network calls, supposedly resulting in much faster Web page retrieval than using, for example, the standard Python urllib2 library. Scrapy has functionality to easily parse HTML and Extensible Markup Language (XML), and also allows for easy filter creation to facilitate the cleaning and sanitization of crawled data. It also has support for handling rules found in the robots.txt Web page found on many Web sites, and for custom rules which can, for example, differentiate between pages based on keyword matching in the URL. Although Scrapy is still in development (currently at version 0.22) and has not been used in creating any of the corpora looked at above, there are several companies which acknowledge that they use Scrapy, with uses ranging from scraping and crawling to testing and pre-caching Web sites and Web applications (Scrapy, n.d.).

**The Wayback Machine** The Web site archive.org provides a free service called The Wayback Machine, accessible through Web APIs. Archive.org takes frequent 'snapshots' of a range of pre-identified pages from all over the Web, and makes these snapshots available through this Wayback Machine's API. These snapshots are represented simply by the HTML of the page as it appeared at the time of the snapshot. This mitigates

[1] http://www.dmoz.org

many of the challenges involved in crawling the Web, as the API makes it trivial to make sure we visit all data sequentially and exactly once.

### 2.1.2 RSS Feeds

While the previous data gathering methods described all rely on retrieving data that have already been published on the World Wide Web, a less common way of gathering data for a corpus is by monitoring pre-identified sources for *new* data. One way to achieve this is by using Rich Site Summary (RSS) feeds. RSS feeds are presented using XML documents. They are designed to alert consumers when new content is published, and they usually give a summary of the new content with a URL to the page at which the content is hosted. These feeds exist for all the major online publications and many smaller ones as well. Prabowo & Thelwall (2006) describe using RSS feeds for live trend analysis by constructing "evolving" corpora, that is corpora which are constantly updated from the RSS feeds of various publications, and which frequently occurring terms can be extracted to analyse current trends in close to real time. Fairon (2006) describe using a similar technique for building specialized corpora and (Fairon *et al.*, 2008) describes building a linguistic search engine, *glossanet*, using two tools, named *corporator* and *Unitex*. Unfortunately these last two papers are badly translated, and the tools and resources they refer to have either moved, no longer exist, or seem to be badly maintained. Nonetheless, the idea of using this method to slowly but constantly build up a corpus remains appealing as using RSS feeds close to guarantees that the retrieved content is relevant (feeds do not link to irrelevant pages such as word lists, error pages, or adverts) and this content is also presented in a standardized format. Furthermore, useful meta-data such as that concerning the author and data can often be extracted more easily and reliably from a feed than from the Web page where the content is hosted. It is to be noted, however, that RSS normally provides only a summary of the main article within the feed itself, along with a link to the host page of the article. Therefore, while some information can be extracted directly from the feed, the main Web page still has to be retrieved and scraped as well.

### 2.1.3 Dynamic Data

The most common data that are generated dynamically, as described above, that is also relevant and desirable for corpus inclusion, is the comments found on many online newspapers and blogs. These comments are a very good representation of how English is being

used by the general language users (that is, those who are not necessarily trained reporters or authors), but due to the fact that there can be many thousands of user comments on a single article, these comments are not usually all loaded as static data when the URL for the article is requested. Instead, JavaScript is activated either as the viewer scrolls to the bottom of the page, firing an event to fetch more comments, or when the viewer clicks on a "load more comments" button or equivalent. This makes fetching them using a standard Web spider challenging. There are several ways to solve this problem. The first is to use a browser automation tool. Such a tool actually opens a Web browser such as Mozilla Firefox, and can simulate mouse movements and clicks programmatically. Another way is to use a crawler specifically designed to look for and retrieve dynamic content. One example of this is Crawljax, which is described below. Finally, depending on the way comments have been implemented, it may be possible to retrieve the data via a Web service API call. Currently, the most commonly used comment system is Disqus, which does provide an API, albeit one with large limitations on what data can be accessed and how many requests can be made.

**Selenium** Selenium is a browser automation tool, focused on allowing users to fully and automatically test Web sites which make heavy use of AJAX or other dynamic content. It uses JavaScript to send commands to a Web page, and as it is able to simulate mouse clicks and other user actions, as well as able to fire JavaScript events on the Web page, it can replicate a Web page viewer's actions exactly, and access exactly the same content (Gheorghiu, 2005). This is different from traditional Web crawlers which generally send an HTTP request for a given URL and can access only the static content returned by that request. Although Selenium is aimed at Web application creators who need to automate tests for their own Web applications, its functionality is useful in crawling dynamic sites as well, as it is able to initiate the scroll or click events need to retrieve the comments, as described above. The problem, however, is that the actions needed are unlikely to be the same over multiple Web sites, and thus work would need to be done in order to determine exactly which actions need to be sent in order to retrieve the desired content.

**Crawljax** Another solution to the problem of dynamic data is found in the open source tool Crawljax. Mesbah *et al.* (2008, p. 1) set out the problem which Crawljax aims to solve, stating "AJAX techniques shatter the metaphor of a Web 'page' upon which general search crawlers are based". Crawljax loads a dynamic Web site and builds a state-flow graph, which uses the browser's Document Object Model (DOM) tree and "captures the states of the user interface, and the possible transitions between them". Crawljax

identifies relevant clickable elements of the DOM and then exercises actual clicks, creating all possible states, and converting these states into static HTML, which can then be stored. Although Crawljax focuses on finding clickable elements in the DOM tree, the authors state that "other event types can be used just as well to analyze the effects on the DOM in the same manner" (Mesbah *et al.*, 2008, p. 4), which indicates that the behaviour seen on many news sites which results in comments being loaded on an 'on-scroll' event should not be a problem.

**Disqus** A final solution worth noting is that of retrieving dynamic data through an API instead of attempting to retrieve it from the page on which it appears. The comment platform *Disqus* deserves special attention, as many online publications, including *Mail and Guardian* and *IOL*, are now using it. Furthermore, it provides a free and well-documented API, albeit with some quite heavy usage restrictions, including restrictions on the number of calls made in a specified time, and restrictions on what content can be accessed on sites for which one does not have ownership. Given some unique pieces of information, such as a so-called 'forum-name' and 'article-url' (not always the actual URL of the article), API calls can be made to return comments for that article in JavaScript Object Notation (JSON) format. These JSON responses follow a strict format and contain a plethora of meta-data, much of which could be useful for language analysis (Disqus, 2007). Unfortunately, a private key needs to be provided, as only the owner of a given site has access to the comments via the API.

## 2.2 Results

For backwards crawling, we tested a basic implementation of a Scrapy crawler, and also gathered data through the Wayback Machine API. Gathering data through the API was more accurate and using this allowed us to overcome many of the challenges faced when using Scrapy. We therefore decided that all backwards crawling for our corpus should be done through the API, and did not run tests for the Heritrix crawler. For forward crawling, we used the Python package Feedparser to monitor and gather data from RSS feeds. For gathering dynamic data, we ran preliminary tests using both Selenium and Crawljax, but decided to rely on the Disqus API instead. This was for similar reasons to the backwards crawling, as we found that the data gathered through the API were better structured and easier to obtain.

### 2.2.1 Data Sources

To create the initial corpus of South African English, we gathered data from 11 online South African newspapers. These 11 publications were a sub-set of a larger list of identified South African publications, consisting of online newspapers and magazines. The full list can be found in Appendix A. The 11 selected ones were used to build the initial corpus, were selected to include large publications such as *Independent Online* (IOL) as well as smaller publications such as the student newspaper of Rhodes University, *The Oppidan Press*.

### 2.2.2 Scrapy

We created a basic spider in Scrapy and set it to crawl recursively using the home pages of the 11 initial publications described above as seed URLs. We limited the search to pages on the .co.za domain and saved the HTML of all retrieved pages to a database. The data collected over a 24 hour period was not ideal. It consisted of 274 000 HTML pages, about 54 GB, but a lot of this data was uninteresting, consisting of, for example, accommodation listings, forgotten password pages, terms of service pages, and e-commerce Web sites. Furthermore, we encountered issues regarding available memory when implementing large-scale crawls. Scrapy adds all the URLs it finds on all pages it crawls to a queue, and then crawls every URL in the queue. However, because each crawled page is likely to contain dozens of URLs, this queue tends to grow throughout the duration of the crawl. Because Scrapy keeps the queue in memory, we had to limit the size and duration of the crawl.

We created a second spider using Scrapy, limiting it to the IOL domain (iol.co.za), and only fetching data from pages which contained the word 'news' in the URL. IOL's URLs are based on a category hierarchy with all of their articles beginning with a 'news' category, followed by sub-categories, such as 'crime' or 'sport'. The data-set created by this crawl was far more accurate than the one described above, and a majority of the pages returned were news articles. However, the crawl for the entire IOL site finished in a few hours, and consisted of only 18 000 articles and under 5 GB of data. This was smaller than expected, as the IOL Web site contains articles dating back to 1999. We realized that the search was limited by the fact that pages on IOL tend to only link to recent articles, as the sidebar and main navigation page of the bar remain constant over all articles. That is, even if a page from 1999 is manually visited, it does not link to other articles from 1999, but rather to articles published on the day that the page is viewed. Therefore, a recursive scraping algorithm, such as the one Scrapy uses, is not ideal.

### 2.2.3 The Wayback Machine

Using the API provided by archive.org's Wayback Machine[2], we were able to achieve a more thorough crawl of many Web sites than we were through using Web spiders. This is because the Wayback Machine provides an API which returns a 'snapshot' of the URL requested at a specific point in time. Using this, there is no longer a reliance on older articles being linked to from newer ones, as we can request the snapshots of the homepage for a specific date, and use the links from the snapshot to find articles from that period. Unfortunately archive.org's archives are not completely exhaustive, but all 11 of our initial publications were at least partially represented. Larger publications had regular snapshots going back for many years. For example, the first snapshot for *Mail & Guardian* (mg.co.za) is from 1997, where the page was still nothing more than an advertising snippet for a hosting company. Smaller publications had fewer snapshots. A search for *Dispatch Live* (dispatchlive.co.za), for example, only returned snapshots from 2014, even though a date-range limited Google search showed that articles from this publication existed from as early as 2010.

We ran experimental crawls using the Wayback Machine on three of the publications, namely *IOL*, *Mail & Guardian*, and *Grocott's Mail*. Using the WayBack machine is fairly straightforward. An API call can be made, giving a URL and a desired date. A response is then received from the API containing a URL and an available date, which represents the snapshot hosted by archive.org made closest to the requested date. To implement a backwards crawl of a domain, we requested the snapshot of a publication's homepage for a recent date, and then iteratively decreased this date, retrieving older and older snapshots until there were no more available. For each snapshot, we retrieved all URLs, a majority of which were links to articles by that publication for the period of just before the snapshot, and downloaded the HTML of each, saving it to a database. For each API call, if the the closest snapshot returned was one we had already processed, we subtracted one day from our date and made another call. If a new snapshot was returned, we crawled this for URLs and saved the HTML of each, before subtracting one day from the date and making a new request.

A small modification was introduced due to the fact that even if an archive.org snapshot was available for the homepage of a publication for a specific date, it was not guaranteed that the snapshot would encapsulate the pages for all URLs of the homepage on that date. Therefore, for every URL found for a given snapshot, we would first attempt to access

[2]see: http://archive.org/Web/

that URL also through the Wayback Machine. If it was not available, we would attempt to retrieve the relevant page directly. There were still some pages that were not accessible through either of these methods, as they were not contained by the Wayback Machine snapshot and were no longer available directly on the publication's servers. Pseudocode of algorithm used, with this modification, is seen in Algorithm 1.

```
desiredDate = today();
lastDate = NULL;
while do
    closestSnapshot = archiveorg.getsnapshot(desiredDate);
    if if not closestSnapshot['date'] == lastDate then
        lastDate = closestSnapshot['date'];
        links = crawl(closestSnapshot['url']);
        for link in links do
            if wayback link available then
                save HTML;
            else
                if direct link available then
                    save HTML;
                end
            end
        end
        desiredDate = lastDate - 1;
    else
        desiredDate -= 1;
    end
end
```

**Caveats of the WayBack Machine** Using this, we were able to achieve relevant datasets for specific publications. The caveats were the sparsity of the snapshots for some publications, as well as the page retrieval speed, which was many factors slower than using Scrapy, due to the overhead introduced by the API, the fact that an additional request for available snapshots had to be made for each day before retrieving the actual snapshot, that we needed to look in two places (archive.org and the original location) for some pages, and the slow response time of archive.org's servers[3]. Finally, even though

[3]We suspect that archive.org may be purposefully limiting the rate of requests, as the service is free

the data was far more relevant than that crawled with Scrapy, there were still many irrelevant pages that were crawled. An example of this was seen in the dating site that is also run by IOL, which is linked to from their main news site. Because each dating profile is reached through a unique URL, and because these are linked to form the IOL news site, the crawler fetched hundreds of dating profiles which contained no content of any linguistic interest.

### 2.2.4 RSS Feeds

All of the 11 publications that we chose to create the initial corpus provided RSS feeds. Many of these publications published a single RSS feed, which was updated with new articles either daily or hourly. Some publications, such as *IOL*, published dozens of RSS feeds, with different feeds for different categories. In total, we sourced 158 feeds for the 11 publications, and all of these were added to a watch-list. Our system checked all 158 feeds every hour, and fetched all new articles, saving the HTML to a database along with metadata. Between April and October of 2014 we retrieved about 117 000 articles using these feeds, amounting to 10 GB of data. Because RSS feeds are designed to alert readers to new content, these feeds do not link to uninteresting pages such as the dating profiles and login pages that were inadvertently retrieved when using the data gathering methods described above. The only unwanted data retrieved through these feeds were pages containing mainly images or videos, and little or no written content, but these are fairly easily identified. Furthermore, the feeds contain well-structured meta-data, using Extensible Markup Language (XML), about each article. This meta-data includes information such as the original publication date of the article, the headline, and, for some feeds, the author of the article. These meta-data are all useful, and without the assistance of RSS feeds these data have to be extracted from the HTML directly, a process which is non-trivial to achieve automatically as there exists no single meta-data formatting convention which is followed by all publications.

**Feedparser** To easily parse the feeds, we used Feedparser, a Python package that is able to not only read all of the RSS versions commonly in use (RSS 0.9x, RSS 1.0 , RSS 2.0)but also other feed formats (CDF, Atom 0.3 and Atom 1.0). It is also fully open-source and documented (Pilgrim, 2010). Although RSS feeds are in essence just XML, and can be parsed manually or with an XML parser, the extra abstraction layer provided by Feedparser removes the need to deal with differences in feed conventions. For example,

some feeds may use 'pubDate' to specify the date of the article, while others may use 'updated'. Once a feed has been read by Feedparser, all of the data can be accessed using attributes which are consistently named.

We used the Linux Cron scheduler to run a Python script on an hourly basis that checked each of the feeds in our watch list, and looked for URLs that were not yet in our database. If the URL was identified as new, the HTML was fetched using the Python urllib2 module and saved to the database.

### 2.2.5 Dynamic Data

To retrieve dynamic data, such as comments on online news articles, we attempted solutions using the aforementioned Crawljax, Selenium, and the Disqus API. No generalized solution could be found using Crawljax and Selenium, as the actions needed to load the dynamic data varies from publication to publication. Although Crawljax is able to analyse the DOM tree and save every possible state of the Web page to a different static file, it still proved highly problematic to identify and extract the comments from within these files. We were able to develop site-specific solutions using Selenium, but these involved hard-coding commands to click specific JavaScript buttons based on their IDs. We discarded this possibility. Not only was it not general enough, but it also risked becoming non-functional based on even minor updates to the publication's Web sites.

**Disqus API** We had more success using the Disqus API. Although not all publications use the Disqus comment system, there are many that do, including most of the larger publications, such as *IOL* and *Mail & Guardian*. The comments from these publications would have made up the majority of the comment data in any case, as a popular article on one of these could generate more than 1000 comments, while the smaller publications, such as *Dispatch Live*, tended to generate no more than three or four comments for popular articles, and many articles were entirely without comments. Furthermore, comments retrieved using the Disqus API included valuable meta-data, such as the date and the username of the poster. The comments also required no cleaning of HTML tags and other boilerplate, as the API returned the full comment in plain text as part of its response. Unfortunately, the API is designed to be used by site owners, in order to easily run analysis on comments generated by their site. Therefore, the public API returns only limited data about a comment thread, and a private key, supposedly specific to each site, needs to be passed along with a request to retrieve the full comments of a given article.

However, Web browsers retrieve the comments using the same API. We used Burp Proxy to inspect the HTTP headers sent by our browser when loading comments on various online news sites, and were surprised to find that there was a constant global key which allowed full access to any Disqus comments feed. This key is not included in any of the Disqus documentation, and we therefore cannot be assured that it will continue to work indefinitely, but because the data returned by the API was ideally formatted for our needs, we decided to use this system anyway.

One potential challenge in retrieving comments in conjunction with our RSS system was that our RSS feed monitor collected articles as soon as they were published. Because readers tend to comment on articles hours, days or even weeks after the article is published, we could not retrieve the comments at the same time as the article. One solution would have been to retrieve the comments on a regular basis for each article, saving only the ones that we had not saved before. However, part of the API response that is received from Disqus when requesting the comments for a specific article is a 'comments_closed' flag, which indicates whether the comment section for that article is still open. We decided therefore to only retrieve comments once the response indicated that the comments section was closed. To this end, we ran a second Cron-scheduled Python script, which retrieved the beginning of the Disqus response for every article that did not have comments yet. If the API indicated that the comment section was now closed, we would retrieve the articles, and mark the article in the database to indicate that we did not need to retrieve comments again. If not, we would leave the collection of comments for a later date.

## 2.3 Data Storage and Database Systems

The classic way to store data is in a Relational Database Management System (RDBMS), while a more recent trend in data-storage is the use of so-called NoSQL or Not Only SQL[4] systems. Each paradigm has advantages and disadvantages, so it is worth taking some time to examine each in choosing what database system to use. Below, we look at the differences between RDBMS and NoSQL systems using the work of Hecht & Jablonski (2011) and Cattell (2011) who give an analysis, comparison and evaluation of various NoSQL database systems. The focus here is on so called Document Stores, a variant of NoSQL database systems which seems particularly suited to storing corpus data, as the flexible schema typically used by these systems allow for rapid development and easy storage and searching of text documents.

[4]This definition is a bit controversial. See, for example, Cattell (2011)

**Scalability** One of the more often cited advantages of most NoSQL databases is their ability to efficiently scale horizontally (Cattell, 2011, Hecht & Jablonski, 2011). This means that instead of having to invest in a powerful database server upfront, which may not be used to capacity for many years, one can use low-cost hardware to host the database at first, and add machines (whether physical or virtual) as data grow and more resources are required. This advantage fits particularly well with the RSS-fed corpus described above, as a corpus built in this way will start off needing to deal with a very small amount of data but will grow constantly and potentially indefinitely.

**Flexibility** NoSQL database systems also tend to be more flexible than RDBMSs (Hecht & Jablonski, 2011). RDBMSs need to have a rigid schema, defined before the insertion of data, and entities stored need to strictly conform to a preset definition, with specified parts of the data in named columns. To make changes to the schema, all old data need to be converted into the format of the new schema. NoSQL databases, in contrast, and specifically so-called Document Stores and Key Value Stores, allow for flexibility both at a schema and at a data level. Hecht & Jablonski (2011, p. 337) emphasize this by explaining that new attributes can be added to new documents at runtime and Cattell (2011, p. 24) supports this claim, saying that NoSQL databases are advantageous for applications which "require a flexible schema, allowing each object in a collection to have different attributes."

**Rapid Development** While RDBMSs tend to require a substantial amount of time to be put into their design, set-up and maintenance, NoSQL database systems arguably require less effort. Medium to large applications which use an RDBMS will often have database administrators in addition to developers, but Hecht & Jablonski (2011, p. 337) emphasize that Document Stores stand in contrast to this, stating, "Storing data in interpretable JSON documents have the additional advantage of supporting data types, which makes document stores very developer-friendly." Cattell (2011, p.24) emphasizes that the simplicity of flexible data is often easier to understand and work with than that found in an RDBMS, and states "Likewise for a document store on a simple application: you only pay the learning curve for the level of complexity that you require". Considering that the corpus system for South African English needs to built within several months by a single developer, rapid development advantages have a necessarily high weighting.

**Disadvantages of NoSQL** There are however some disadvantages in using the NoSQL paradigm. Although this remains a bit of a generalization, and many factors need to be

taken into account, NoSQL systems tend to be slower and to offer fewer of the ACID (atomicity, consistency, isolation, and durability) guarantees, which are a focal point of RDBMSs. The Document Store MongoDB, for example, offers only very basic locking, and therefore inconsistencies can appear, especially if multiple threads or clients are performing operations on the database simultaneously. Cattell (2011, pp. 16-17) says, "Like other NoSQL systems, the document stores do not provide ACID transactional properties". Cattell (2011, p. 24) also points out that RDBMS are 'tried and tested', and have over thirty years of development and refinement behind them. While our research was ongoing, new figures were published which compared Postgres and MongoDB. This research suggested that MongoDB used 33 percent more disk space than Postgres, took three times longer for insert operations, and took 2.5 times longer for data selection (Linster, 2014).

**Deciding on a Database System** With the above in mind, we decided that the advantages of MongoDB outweighed its potential issues. We have used MongoDB through the Python wrapper Pymongo to store all corpus data, including articles, comments, and meta-data, as well as system data, such as lists of publications and feed URLs. At the start of this research, we used MongoDB version 2.4, but later updated to version 2.6. The newer version included major updates in functionality and stability for the 'text search' feature, bringing it out of beta, and we migrated all of the search features of our corpus over to use this functionality. This feature of MongoDB allows for efficient searching of textual data, similar to that found in search engines. That is, search terms are automatically stemmed so that results containing the same word in a different form are still matched. For example, searching for 'blueberry' will return results containing not only 'blueberry' but also 'blueberries'. The text indices which MongoDB automatically creates in order to provide this functionality can be quite large – for our initial corpus of 90 000 articles, the text index was 745 MB. But a noticeable speed-up in the search functionality of the corpus was seen.

## 2.4 Summary

In the first half of this chapter, we examined different methods to gather and to store data. For the former, we looked at both backwards crawling, or gathering data that have already been published, as well as forward crawling, or monitoring publications for data that is yet to be published. We noted the challenges inherent to backwards crawling, which included differentiating relevant data from noise, such as login pages and dating site

listings, and showed how some of these could be overcome by using API services instead of recursive crawlers. We also examined RSS feeds, and how these can be used to easily monitor specific sites for new data. The challenges presented by dynamic data, loaded based on a user's interactions with a Web page, were examined, and again we showed how these challenges could be overcome by using relevant APIs instead of crawling or scraping.

Secondly, we looked at storing data, and gave a comparison of traditional SQL RDBMSs and NoSQL systems, with a focus on MongoDB. We motivated our decision to use MongoDB, but also showed some of its disadvantages.

# Chapter 3

# Boilerplate Removal

Data taken from the World Wide Web is not in a format ideal for a corpus. Not only are there usually extraneous data found in headers, footers, and sidebars of Web pages, but even the main content of the page is "dirty"—that is, it contains HTML tags, and non-alphabetic characters are encoded into HTML named entities such such as "–". Therefore so-called "cleaning" of collected data is necessary. Evert (2008, p. 3489) emphasizes this issue and states:

> Web pages are even messier than other text sources, though, and interesting linguistic regularities may easily be lost among the countless duplicates, index and directory pages, Web spam, open or disguised advertising, and boilerplate. Therefore, thorough preprocessing and cleaning of Web corpora is crucial in order to obtain reliable frequency data.

In this chapter we examine and evaluate different methods for boilerplate removal. Although evaluations of different cleaning methods have appeared in previous literature, there exists no definitive published test which compares all of the possible cleaning methods that we identified. We therefore created our own evaluation system, in which we manually cleaned dozens of articles and then checked the results of the various cleaning methods against our manual 'proof' texts. While two of the systems ('Boilerpipe' and 'Reporter') performed far better than the others, each of these displayed different weaknesses. We therefore included functionality in the system to allow both systems to be used.

**Machine Learning** In the corpus built for natural language processing (Liu & Curran, 2006), a four-step approach was used for cleaning. First, encoding was translated to remove the HTML named entities and replace these with single characters. Second, sections of the data were identified as "sentences". A sentence could either be a grammatical language sentence, or, for example, a mathematical equation. Machine learning was used to create these sentence boundaries using a Maximum Entropy classifier. Tokenization of words was included as a third step. The fourth step was to filter out unwanted text, which was achieved through rule-based analysis, ensuring that sentences and documents contained a high enough ratio of alphabetic characters and dictionary words to other tokens.

## 3.1 Cleaning Tools

While one approach is to build a customized solution, as above, another is to use a "cleaning tool". Cleaning tools have been designed and published not only by corpus researches, but also by, for example, companies who rely on Web-sourced data for trend analysis and research. We identified four such tools: NCleaner, BTE, Boilerpipe, and Reporter. These are described below.

### 3.1.1 NCleaner

This is the approach used by Généreux *et al.* (2012) in building CRPC, the large Portuguese corpus. The tool used was NCleaner (originally called StupidOS) (Evert, 2008, 2007). It is written in Perl and uses a four-step approach which differs from that used by Liu & Curran (2006). For the first step, it uses regular expressions to remove some HTML tags, such as images and comments. Secondly, it uses the Web browser Lynx to convert the HTML to plain-text, and standardize encoding to UTF-8. It then uses more regular expressions to delete some boilerplate, such as easily recognizable headers and sidebars. Finally, it uses n-gram models to evaluate each section of the text and either reject that section as boilerplate or to include it in the clean text output. NCleaner competed in the text-cleaning competition, CleanEval (Baroni *et al.*, 2008), which started in 2007, and although Evert suggests that NCleaner performs comparatively well, the reliance on regular expressions is cause for concern, as the issues which arise in attempting to parse HTML,

both in terms of efficiency and accuracy, with regular expressions are well-known.[1]

### 3.1.2 BTE

The WaCky corpora collection used an algorithm for cleaning content based on a Python script called BTE (Body Text Extraction). While NCleaner tried to identify and remove boilerplate, BTE takes the opposite approach, attempting to identify the 'main text' of a Web page, and discarding everything else. (Baroni *et al.*, 2009). Finn (2010) explains this algorithm as follows

> BTE extracts the main body of text from a Web page. It does this by tokenizing the document and performing some shallow processing. The HTML document is tokenized and represented as a binary string where a 0 represents a tag token and a 1 represents a text token. If we graph cumulative total tokens on the x-axis and cumulative tag tokens on the y-axis we get a graph something like that shown [in Figure 3.1].

The idea is that the main article contains a lower ratio of tag tokens to text tokens, and is usually found between two sections which contain a relatively high ratio of tag tokens to text tokens, as most Web pages have headers and footers which are tag-rich. Thus where the graph flattens out in middle is representative of where the main text of the article is likely to be, and this is extracted and kept while the sections of the article represented by steeper slopes on the graph are discarded.

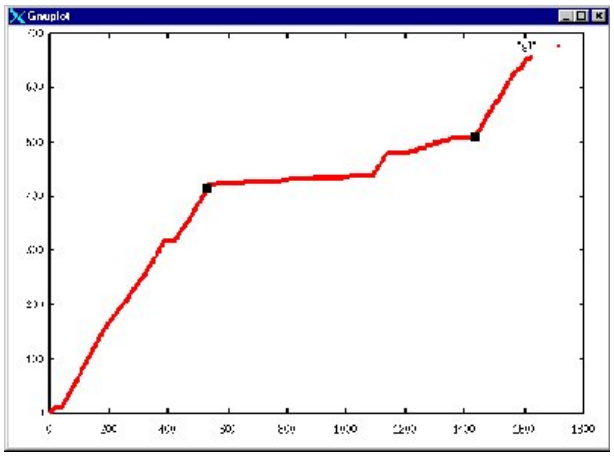



Figure 3.1: Cumulative graph of a Web page as seen by BTE (Finn, 2010)

[1]See the famous first answer to this stackoverflow post for a lighthearted explanation of why this could be a problem: http://stackoverflow.com/questions/1732348/regex-match-open-tags-except-xhtml-self-contained-tags

The BTE tool was also used by Pomikálek (2011), who found it to outperform all of the entries of the CleanEval competition.

### 3.1.3 Boilerpipe

Boilerpipe is written in Java and was developed by (Kohlschütter *et al.*, 2010). It uses so-called shallow text features of a Web page to classify text elements. The authors claim that they 'compare the approach [used by Boilerpipe] to complex state-of-the-art techniques and show that competitive accuracy can be achieved, at almost no cost. The shallow features looked at to distinguish boilerplate from text are broken down into several categories. 'Structural features' involve the placement and frequency of HTML tags, of which Boilerpipe looks only at <p>, , <a> and <h1> to <h6> tags. 'Shallow text features' involve average word and sentence length, as well as the relevant position of content. Longer words and sentences are more likely to indicate non-boilerplate content, while content appearing next to boilerplate is more likely to be boilerplate itself. 'Densitometric features' looks at text density, which Boilerpipe bases on calculations used by 'pixel-density Computer Vision-based approaches'. Using these categories, Boilerpipe categorizes all text as either 'long' or 'short', and, taking into account some other heuristics, removes short text as boilerplate and leaves long text as the main text.

### 3.1.4 Reporter

Yet another cleaning tool is the Python package *Reporter*. Similarly to BTE but different from NCleaner Reporter attempts to identify the "main text" of a Web page, discarding everything but this. It does this by building a DOM tree and traversing this in reverse order, starting at the leaf nodes. It classifies each tag set as either a paragraph or container, and then uses heuristic rules, including word count, to determine the score of the tag. The highest scoring tag is then returned as the main text of the article (Janssens, 2012). Because Reporter was developed internally by Visual Revenue, it has not received much attention, and there are no published results comparing it to other cleaning tools.

### 3.1.5 Reliability of CleanEval's Methods

The CleanEval competition is cited by many corpus researches, but Kohlschütter *et al.* (2010), call their evaluation methods into question. While testing and evaluating Boiler-

pipe using CleanEval's methods and data, they noted that erring on the side of keeping too much text (at the cost of having some boilerplate in one's results) always achieved a high score, and that keeping *all* text resulted in "only marginally worse" results than BTE's best solution (BTE). This is consistent with our findings, discussed later, that BTE included far too much boilerplate in its results to be of any use.

### 3.1.6 Comparative Results

With the evaluation methods used by the CleanEval competition having been called into doubt, and because there is no published comparison which evaluates all of the above tools, we created our own evaluation system to produce comparative test results. We discovered that Boilerpipe and Reporter far outperformed BTE, but we were unable to run NCleaner, as it is written in Perl and has not been updated to run easily on 64-bit architecture. Boilerpipe and BTE have settings to run variations on their algorithms, such as to be optimized specifically for certain types of text, so for our tests we ran a total of six algorithms. These consisted of Boilerpipe (three different variations), BTE (two variations) and Reporter.

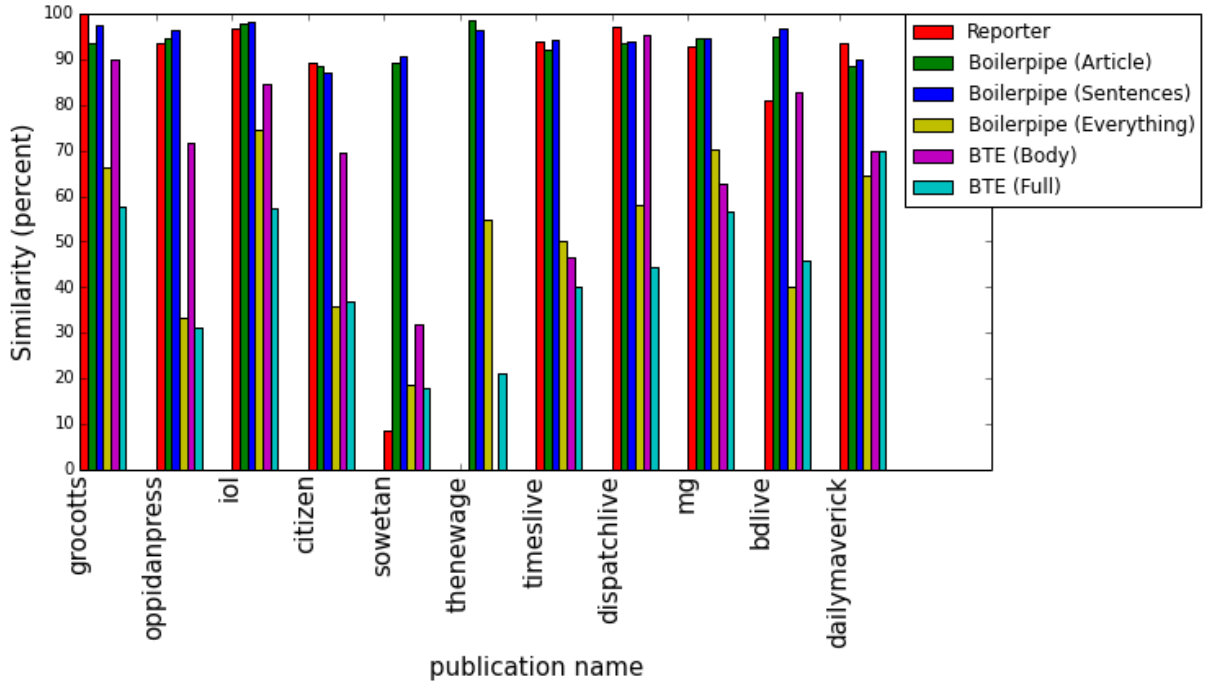
To build our evaluation system we manually removed the boilerplate from a number of news articles and then used a TF-IDF distance algorithm to compare the output of each of the tools against our manual proofs. We then used the Linux tool wdiff[2] to do a finer-grained comparison, checking whether the tools were erring by including boilerplate in their results, or by misidentifying text as boilerplate and removing this. Instead of simply comparing articles on their length, wdiff actually completes a full matching algorithm, calculating which sections of text were added and which removed.

Our manually selected texts aimed to give insight into which tools worked best over a range of different publications, and also which ones performed consistently well within a single publication. We therefore cleaned one text manually from each of the 11 publications which made up our initial corpus as well as a further 30 texts from *IOL*, which is the publication from which a majority of our initial data was sourced. After the statistical analysis, we also manually inspected the results of each algorithm, aiming to discover insights into what kinds of input caused each to misidentify boilerplate.

**Statistical Results** We evaluated the results on 1) overall similarity to our manually cleaned texts, 2) how much extra text they included, and 3) how much text they deleted.

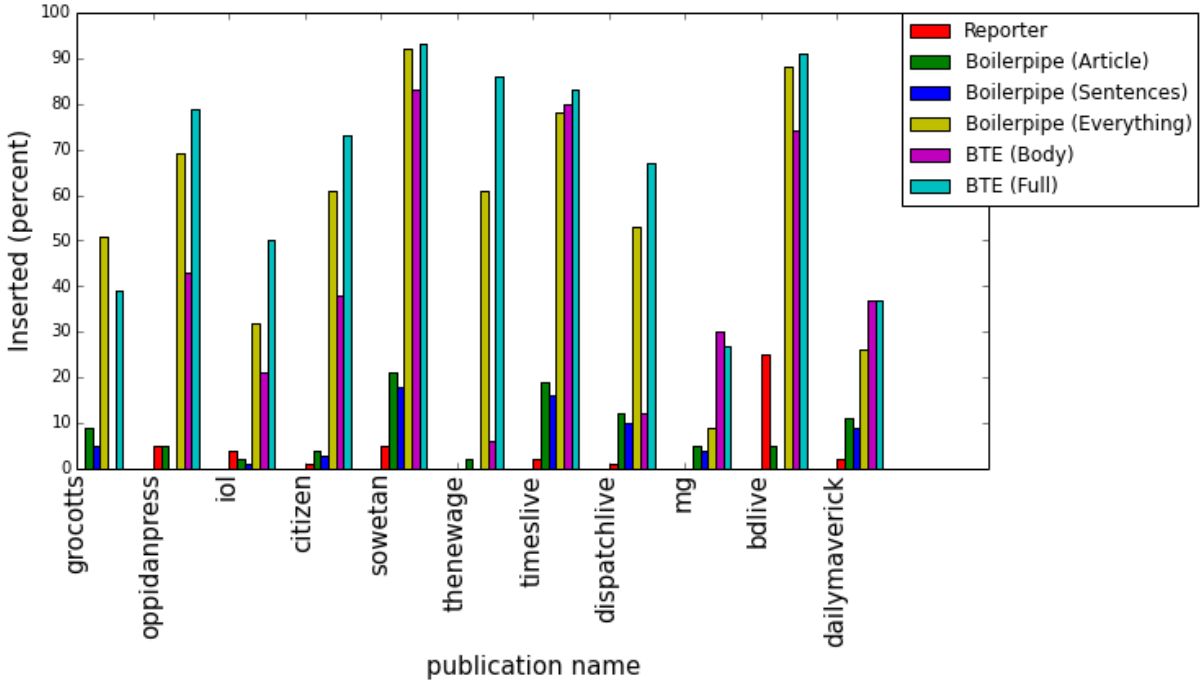
[2]see http://www.gnu.org/software/wdiff/

The graphs below reflect similarity, inserted, and deleted text for our data-set which covered all 11 publications. We can see in Table 3.3 that BTE includes far too much text in almost all cases, and inserted even more text than Boilerpipe's 'keep everything' option. This is because it failed to exclude CSS and other invisible content, which Boilerpipe successfully removed. In Table 3.2 we can see that Reporter achieved the highest overall accuracy in four of the eleven cases, but achieved scores of below 10 percent in two cases. Boilerpipe achieved consistently high scores over all 11 publications.



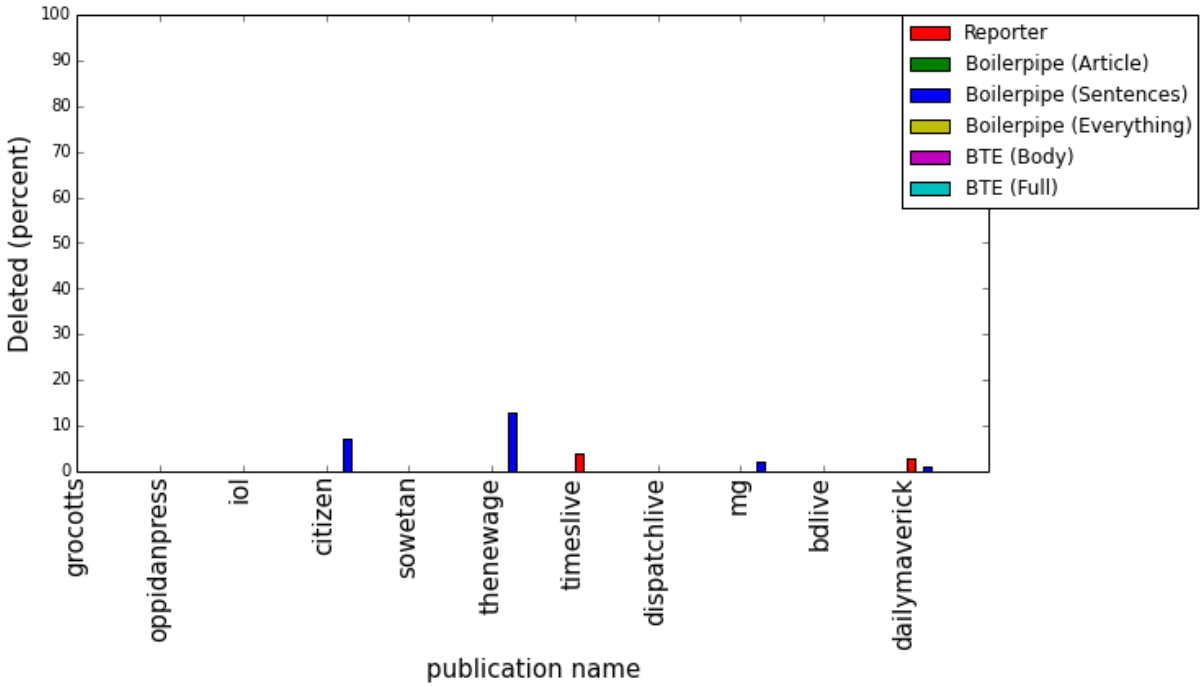

(a)

Figure 3.2: Similarity of Cleaning Tools over all Publications




(a)

Figure 3.3: Inserted text of Cleaning Tools over all Publications




(a)

Figure 3.4: Deleted text of cleaning algorithms

With Reporter and Boilerpipe being evidently the two best-performing algorithms for our data-set, we discuss these in more detail below.

**Reporter Results** Although the Reporter algorithm performed very well in most cases, we can see in Figure 3.2 that it did very badly with articles from *Sowetan Live* ("sowetan") and *The New Age* ("thenewage"). Running the algorithm on other articles from those publications showed that Reporter did consistently badly for these publications, often mis-identifying a large chunk of CSS as the main text for The New Age, and often selecting only a single paragraph of the main text from *Sowetan Live*. Looking at Figure 3.2 again, we can see that Reporter performed fairly badly with articles from *Business Day Live* ("bdlive"). These articles include a 'Kindle version' in the source, which was a copy of the entire article, set to be hidden when viewed from a normal browser and shown on a Kindle e-reading device, but this led the Reporter algorithm to effectively include the main text of each article twice in its results. Furthermore, the BDLive articles use an HTML formatting style in which no actual white-space is used between  tags and the surrounding text. Reporter therefore joins the last word of each paragraph with the first word of the next, resulting in a paragraph such as "...is seen in town. In other news..." being extracted as "...is seen in town.In other news...", with "town.In" being tokenized as a single word.

**Boilerpipe Results** Boilerpipe performed consistently well over all the publications, but not always better than Reporter. When set to 'Sentences', the algorithm only keeps whole sentences, which is largely useful for most publications. However this setting did result in some sections of articles being incorrectly removed. This happens for cases such as quoted speech, where full sentences may not be used. When set to 'Article', Boilerpipe no longer incorrectly removed pieces of the text, but as we can see from the the graph in Figure 3.3, this sometimes resulted in larger amounts of boilerplate being incorrectly included as part of the identified text.

**IOL Data-set** We chose *IOL* to test the accuracy of these algorithms over many articles from a single publication. The full results for each of the algorithms on the 30 IOL articles we manually cleaned can be found in Appendix B. Average and worst cases are tabulated below (with worst cases being the minimum for overall similarity, and the maximum for extra text inserted and deleted) and we can see in Table 3.1 that although the Boilerpipe algorithm set to extract sentences performs best overall, and achieves the most accurate

results in terms of not including boilerplate incorrectly, it does incorrectly delete the most text (see Table 3.3), and incorrectly deleted as much as 19 percent of one of the articles. In Table 3.2 we can see that Reporter and Boilerpipe set to the Article setting performed very similarly, and although Boilerpipe did better still when set to the Sentences setting, this must be contrasted against the parallel bad performance concerning deleting text.

Table 3.1: Similarity accuracy of specific clean evaluation

| **Similarity (Percent)** | | |
|---|---|---|
| | Mean | Minimum |
| Reporter | 92.49 | 78.05 |
| Boilerpipe (Article) | 92.30 | 77.38 |
| Boilerpipe (Sentences) | 93.61 | 79.19 |
| Boilerpipe (Everything) | 61.95 | 37.15 |
| BTE (Body) | 67.57 | 33.51 |
| BTE (Full) | 43.29 | 25.79 |

Table 3.2: Inserted text results of specific clean evaluation

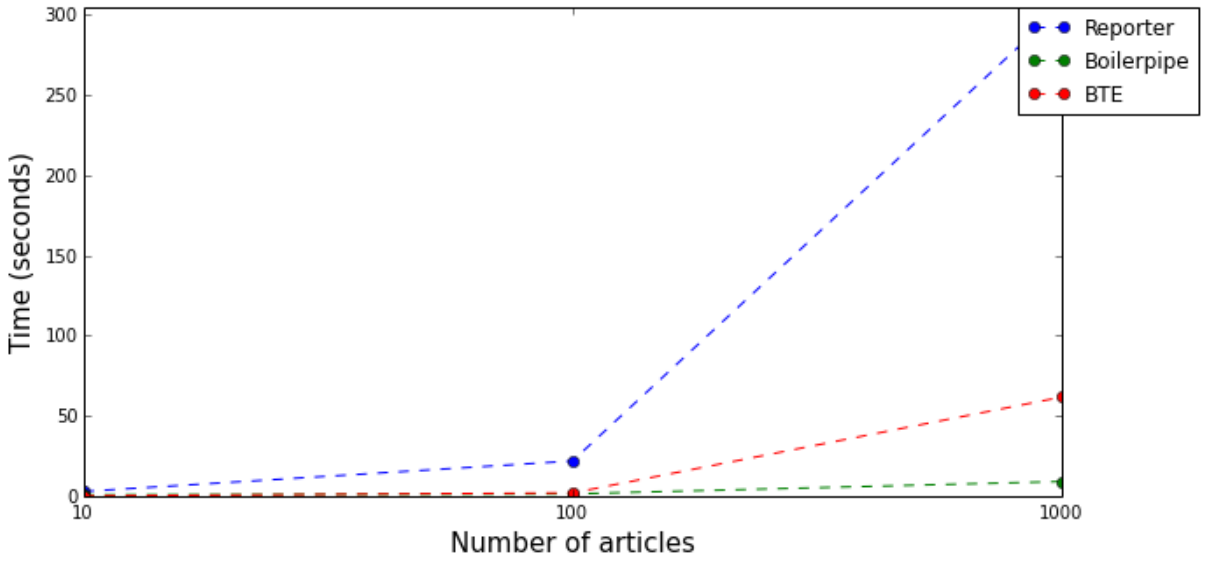
| **Inserted (Percent)** | | |
|---|---|---|
| | Mean | Maximum |
| Reporter | 7.13 | 24.00 |
| Boilerpipe (Article) | 8.27 | 23.00 |
| Boilerpipe (Sentences) | 4.17 | 13.00 |
| Boilerpipe (Everything) | 38.87 | 73.00 |
| BTE (Body) | 33.53 | 64.00 |
| BTE (Full) | 57.17 | 86.00 |

Table 3.3: Deleted text results of specific clean evaluation

| **Deleted (Percent)** | | |
|---|---|---|
| | Mean | Maximum |
| Reporter | 0.00 | 0.00 |
| Boilerpipe (Article) | 0.07 | 2.00 |
| Boilerpipe (Sentences) | 1.67 | 19.00 |
| Boilerpipe (Everything) | 0.17 | 1.00 |
| BTE (Body) | 0.03 | 1.00 |
| BTE (Full) | 0.03 | 1.00 |

**Large-scale Comparative Manual Tests** As a final comparative check, we ran the Boilerpipe and Reporter algorithms on our initial corpus of 90 000 articles. It was impractical to create proof results for a large number of articles, and we had no perfect algorithm to create such proofs automatically, but we still managed to identify problematic articles by looking at the difference of *output length* produced by each algorithm. We manually inspected dozens of outputs where the length of the output of Reporter differed from the length of the output of Boilerpipe by more than 1000 characters, and discovered that Boilerpipe failed to remove large amounts of boilerplate from some IOL articles. This boilerplate was found in the form of dating adverts, and it was not uncommon to find upwards of 100 sentences of the form 'I'm a 44 year old man looking to meet women between the ages of 35 and 45' in a single IOL article. These were successfully removed by Reporter, but not by Boilerpipe. This was the case even when Boilerpipe was set to 'Sentences' mode, as the adverts were categorized as full sentences.

**Timing** Although cleaning of articles is usually done shortly after crawl time, and therefore on only a few articles at a time, there may occasionally be the need to clean many articles at once. This might be either because a large new source of HTML data is to be added to the corpus, or if problems arise with the cleaning of a specific publication. In the second case, 'bad' cleaning may happen over a long period of time, leaving many badly cleaned articles in the corpus, which then need to be re-cleaned *en masse*. In either case, the speed of the cleaning algorithm is important, and we therefore evaluated the time efficiency of Reporter, Boilerpipe and BTE. We can see in Figure **??** that Reporter is significantly slower than the other algorithms, while Boilerpipe is the most efficient.




(a)

Figure 3.5: Timing of cleaning algorithms

**Choosing an Algorithm** With the two best-performing algorithms, Boilerpipe and Reporter, having mixed strengths and weaknesses, we decided not to tie our corpus creation system to a single algorithm. Instead, we decided to use Boilerpipe set to Article mode by default, but to allow each publication to be associated with a specific algorithm. Having achieved acceptable results using cleaning tools, we did not implement and test a machine-learning system such as that used by (Liu & Curran, 2006). While the Reporter algorithm is available as a Python package, Boilerpipe was written in Java, and therefore more work was required in order to integrate it into our System, which is almost completely implemented in Python. We used a Python wrapper for Boilerpipe, python-boilerpipe, and the Python-Java bridge JPype. Although this added the requirement of the Java Runtime Environment (JRE) to our system, the speed and accuracy advantages afforded by Boilerpipe made this worthwhile.

## 3.2 Summary

Having identified and described several ways in which Web-sourced content can be cleaned of boilerplate, we set up an evaluation system to test some of these possibilities. We found that two algorithms, Boilerpipe and Reporter, performed better than the others, and implemented both of these in our corpus system.

# Chapter 4

# Deduplication

One of the major problems in creating a Web corpus is that of duplication. Many URLs can resolve to a single Web page, and Web pages can easily be duplicated and hosted somewhere else. This can badly skew analysis, especially that of the frequency of certain terms appearing together. Pomikálek *et al.* (2009) say:

> Duplicate documents in text corpora do damage to corpus derived statistics. Much corpus use is based around identifying patterns which are much more common that one would expect by chance.

Although it is trivial to compare two documents and calculate a percentage-based similarity rating, with a large database pairwise comparison of all documents becomes highly impractical, even with parallelization. Completely identical documents can be removed by storing a hash value for each, and removing duplicates by noting collisions (in Order(1) time), but the problem remains for so-called 'near-duplicates'. These are documents which have been slightly edited (or even simply had a small change in formatting), and hence produce a completely different hash value to those produced by *almost* identical documents. Ideally, even documents which have undergone moderate to major editing should be removed.

In this chapter, we look at both exact- and near deduplication. For exact deduplication we discuss at which stage of the data collection process this step is best carried out. For near deduplication, we look at four possible algorithms. These consist of: a full pair-wise comparison, in which every article is compared against every other; a TF-IDF vector based algorithm, in which all articles are loaded into memory and a full similarity matrix

is calculated; a deduplication tool written in C which uses Judy Arrays, and a custom algorithm which uses hashes of article sentences to increase the efficiency of a pairwise comparison. We show that the last of these is the most suitable for our needs.

## 4.1 Exact Deduplication

Exact duplication is easily detectable through hashing, but there are still some considerations to be taken into account. Exact duplication of HTML is possible, and often occurs when a single Web site has more than one URL leading to a single page. More common is exact duplication of the main text of an article, as two different sites may publish the same article. In this case, headers, footers, and other site-specific HTML is different, and the exact duplication can only be detected at a textual level, that is, after the boilerplate-removal step described in the previous chapter has been completed.

**Hashing and Fingerprints** Most common hashing algorithms, such as MD5 and SHA* focus on being secure and non-reversible, and this can lead to lower efficiency. Fingerprinting is similar to hashing, but focuses on low-probability collision instead of security, and is therefore supposedly better suited to deduplication. Heydon & Najork (1999, p. 222) state:

> Fingerprints offer provably strong probabilistic guarantees that two different strings will not have the same fingerprint. Other checksum algorithms, such as MD5 and SHA, do not offer such provable guarantees, and are also more expensive to compute than fingerprints.

It is not completely clear what Heydon & Najork (1999) mean by "provable guarantees", but some basic calculations show that the probability of an MD5 hash collision (where two different inputs produce the same output) is approximately $1/2^{128}$, which is arguably low enough for the risk to be insignificant[1] Furthermore, 'expensive' is a vague term. Using standard MD5 hashing, we created hashes for the HTML of all 90 000 articles in our initial corpus in about six minutes and for the cleaned text representation of each article (which are substantially shorter than the HTML and therefore slightly faster to hash) in about four minutes.

[1]See some discussion on the subject here: http://stackoverflow.com/questions/201705/how-many-random-elements-before-md5-produces-collisions

Based on the above, we decided that MD5 hashes were suitable for our requirements, and we therefore did not investigate fingerprinting possibilities. Some exact deduplication is already automatically done at the data gathering stage, as our database has an index on article URLs and it ignores any URL which is already present in the database. Based on our non-deduplicated initial set of 90 000 articles, only 196 produced hash collisions based on the HTML alone, whereas hashing the cleaned text of the articles produced 12681 hash collisions. We therefore decided not to implement any exact deduplication at an HTML level, relying on the database URL index and the hashes of cleaned articles to remove exact duplicate documents.

## 4.2 Near Deduplication

Many deduplication algorithms make some use of N-grams. Cavnar *et al.* (1994, p. 2) defines an N-gram as "an N-character slice of a longer string". For deduplication a "gram" can instead be taken to be a word (see, for example, (Pomikálek *et al.*, 2009), and (Baroni *et al.*, 2009), who both talk of n-grams as a sequence of n words). One possibility is to regard any documents that share the same non-trivial n-gram as near-duplicates. Even relatively small n-grams can strongly indicate a large degree of duplication, and because n-gram representations of many documents can be kept in memory simultaneously (as n-grams do not take up much space), efficiency is greatly improved compared to a naive pairwise comparison. Baroni *et al.* (2009) used a set of 25 5-gram samples of a document and identified near-duplicate documents as those sharing two or more 5-grams. They state:

> The [two 5-gram] threshold might sound low, yet there are very low chances that, after boilerplate stripping, two unrelated documents will share two sequences of five content words. A quick sanity check conducted on a sample of 20 pairs of documents sharing two 5-grams confirmed that they all had substantial overlapping text.

This same approach was used by Guevara (2010) in building the NoWaC Norwegian corpus, except for the fact that on detection of duplicates using this method, the creator inexplicably chose to discard *both* copies of duplicate data. He states:

> This is a very drastic strategy leading to a huge reduction in the number of kept documents: any two documents sharing more than 1/25 5-grams were

> considered duplicates, and both documents were discarded. The overall number of documents in the archive went down from 11.40 to 1.17 million after duplicate removal.

Although it might be argued that duplicate data are more likely to be completely irrelevant, such as error messages or other computer generated text, 'drastic' seems an understatement for this strategy, as there is likely to be a a significant amount of data which are interesting but still duplicated, such as different newspapers publishing the same article. Therefore, the approach used by Pomikálek *et al.* (2009) seems preferable. They used a slightly more sophisticated algorithm, in which the n-grams were externally sorted, and full comparison of all documents containing duplicate 10-grams was undertaken, with only those having a higher than 50 percent similarity being discarded.

Pomikálek *et al.* (2012, p.503) have since developed a tool called 'OnIOn' (One Instance Only), which can supposedly remove documents with a customizable level of similarity in a single pass of the corpus. Their description of the algorithm is as follows:

> For each document, all n-grams of words are extracted (10-grams by default) and compared with the set of previously seen n-grams (union of the n-grams of previously seen documents). This identifies the parts of the document which already exist in the currently processed part of the corpus. If the proportion of text within these duplicated parts is above a predefined threshold, the document is discarded. Otherwise, the document is preserved and its n-grams are added to the set of previously seen n-grams.

The authors note that the major disadvantage of this is that all n-grams still need to be held in memory simultaneously, and although n-grams are small compared to the complete articles, this can still be prohibitive for large corpora.

### 4.2.1 Sentence-level Near Deduplication

We developed a custom deduplication algorithm based on the number of sentences shared between any given pair of articles. Keeping in mind the impracticalities of full pair-wise comparison of every article in a corpus, but also wanting to achieve a very high level of accuracy, we based our algorithm on the idea that no articles can ever be duplicate or even similar unless they have at least one sentence in common. Using this, we managed

to achieve accuracies comparable with full pair-wise comparison, while processing only a fraction of the articles. To this end, we used the inherent efficiency of hash-tables so that for any given article, we could retrieve a list of possible duplicates of that article (articles containing at least one sentence in common) in linear time. We could then do a full pair-wise comparison on this small set of articles, and decide whether the article of interest was to be regarded as a near duplicate.

The pseudo-code for our deduplication algorithm runs as follows:

```
Data: A corpus of articles, containing duplicates and near duplicates
Result: A corpus of articles, with all duplicates and near duplicates removed
for article in corpus do
    fingerprints = getFingerprints(article);
    for fingerprint in fingerprints do
        get list of articles containing this fingerprint;
        add this list to set of possible duplicates;
        if fingerprint in database then
            update fingerprint in database to link current article ID;
        else
            insert fingerprint in database, linked to current article ID;
        end
    end
    for possibleduplicate in possibleduplicates do
        get pairwise comparison of article and possible duplicate;
        if similaritymeasure >threshold then
            Remove article and continue to next article;
        end
    end
end
```

**Algorithm 1:** Sentence Level deduplication

**Fingerprints** We used a set of MD5 hashes, one per sentence, to represent the 'fingerprints' of an article. These were calculated as a set, so if an article contained the same sentence twice or more, only a single hash was stored. Each sentence-hash was stored as an entry in the database, along with an associated list of articles which contained that sentence. We decided that the possibility of a hash collision (two different sentences resulting in the same hash value), was low enough to be negligible. This was partly because

of the fact that while collisions are possible, they remain highly improbable, with a 50 percent probability being reached only after $2^{64}$ sentences hashed, and partly because collisions would not actually break the algorithm, provided they were fairly low in number. The hashes are only used to find *possible* duplicates, and a collision would simply add an extra possible duplicate to the list of articles to check. But as the pairwise comparison is done on non-hashed sentences (see below), these false positives would be disregarded.

**Similarity Measure** We used a very simple similarity measure which, like the rest of the algorithm, also relied on common sentences between two articles. To compare two articles, we tokenize each as a list of sentences. These lists were represented as Python sets, and we could therefore easily calculate the intersection and union of these lists. The ratio of the number of terms in the intersection (the sentences common to both articles) to the number in the union (all the sentences from either article) gives us a number between 0 and 1 which represents the similarity of the two articles. Unlike other similarity algorithms, relatively low matching percentages indicate highly similar articles, as it is very unlikely for two articles to share more than a sentence or two by sheer chance. Preliminary results showed that a 20 percent or higher match using this algorithm indicated very similar articles, often those which showed 65+ percent similarity using other algorithms.

**Encoding and Style Challenges** One issue that became apparent with sentence-level deduplication is that caused by the lack of standardization of text encoding. While we stored all data encoded as unicode, different publications may well encode their text using different standards. Text encoding standards, or often the lack thereof, are not always the most pleasant things with which to work (Spolsky, 2003), and often differ from publication to publication. Thus, one publication may choose to use 'Smart quotes', those stylized so that opening and closing quotation marks appear different, while others may use standard ASCII quotation marks in which there is no difference between those at the start and end of a quoted phrase. On top of this, the style guides of different publications may differ as to where single quotation marks are used, and where double. The issue then arises that two identical sentences may produce completely different hash results, because a different encoding is used, or a different punctuation style. In some cases we identified, a similarity match as low as 15 percent was calculated for two identical articles, which differed only in style and encoding. To fix this, we used a simplified representation of every sentence, in which we converted the entire article to lowercase, and then ignored all characters that were not part of the lowercase English alphabet.

**Optimization** Our algorithm is mainly linear or Order(n) as, because we manage to drastically limit the work to do per article to be deduplicated, the time is largely based on finding possible duplicates using an index of sentence hashes. But inefficiency appears when there are many possible duplicates found. This occurs when an article contains common sentences which it is more likely to share with completely unrelated articles. For example, a phrase such as 'more to follow' may be found in many breaking news articles no matter their content. To optimize the algorithm, we introduced two customizable parameters. First, the *minimum length* of sentences to consider, and, second, the *maximum matches* which a sentence could return in order to be considered. For *minimum length*, we used a character count with a default of 20. This is because shorter sentences are less likely to indicate duplicate content than longer ones, for the longer a sentence is the less likely it is to appear twice by sheer chance. While a minimum length of 20 characters is likely to be a good cut off point for any sized corpus, the *maximum matches* should be based on the size of the corpus, as the number of times any given sentence appears is strongly related to this. We were unable to definitively discover the best weighting to use for this variable, but some figures on how they affected accuracy and timings for our initial corpus are given below in Section 4.2.2.

**Modification** Although for our purposes it was sufficient to discard any article which was deemed a near duplicate, higher accuracy could be desirable. More specifically, given a pair of near duplicate articles, it might be better to choose which one to keep and which to discard, instead of simply discarding the first one as we have done. A small modification to our sentence-based algorithm can be made to allow for this. Instead of discarding an article the moment a near-duplicate is detected and moving on to the next, instead the similarity measure between the two articles can be saved to the database, and the rest of the 'possible duplicates' processed, with a similarity measure of each against the current article also being saved. Using this modification, a full similarity matrix, similar to the TF-IDF one described below, can easily be created. An intelligent selection on which near-duplicates to keep and which to discard than be be made a on second run through the database.

### 4.2.2 Results

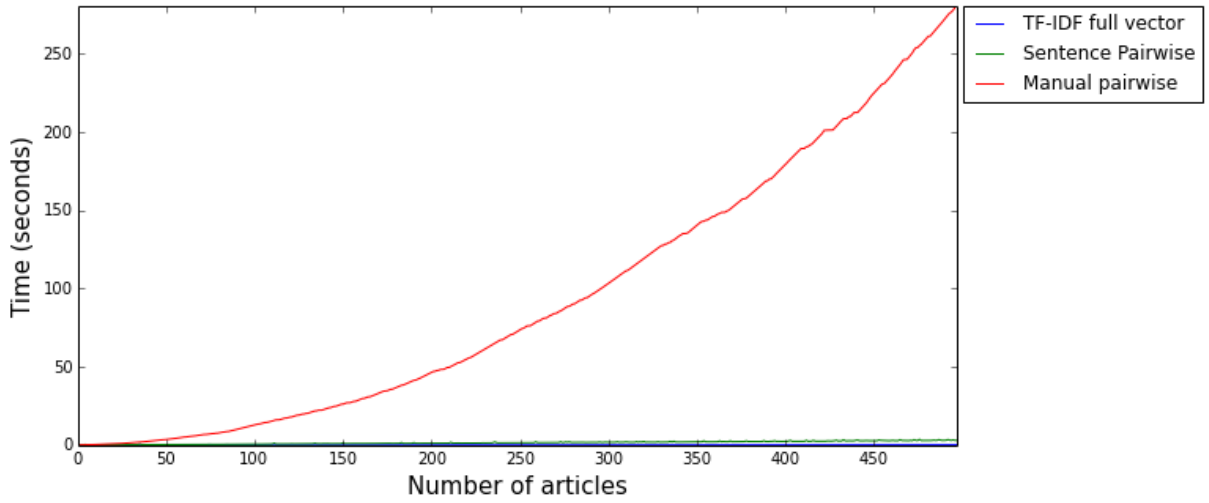
We tested our algorithm in terms of accuracy and efficiency against a full pair-wise similarity algorithm (in which each article is checked against each other), and against a Vector

Space Model which relies on a Term Frequency–Inverse Document Frequency (TF-IDF) model using a cosine distance calculation. We shall not describe this process in depth, but it allows us to calculate an N x N similarity matrix for N articles, from which the similarity between any two documents can be extracted. In summary, we calculate the *term frequency* (TF) for all terms in our articles, and the *inverse document frequency* (IDF), or the ratio of the number of articles containing the TF to the total number of article, to represent each as a number. Vectors are derived from each article, and by calculating the cosine distance of each vector, we can calculate the similarity of the documents. To achieve this, we used the Python library sklearn, which includes a TfidfVectorizer. This allowed us to compute a full similarity matrix for a list of *articles* with:

```
tfidf = TfidfVectorizer().fit_transform(articles)
pairwise_similarity = (tfidf * tfidf.T).A
```

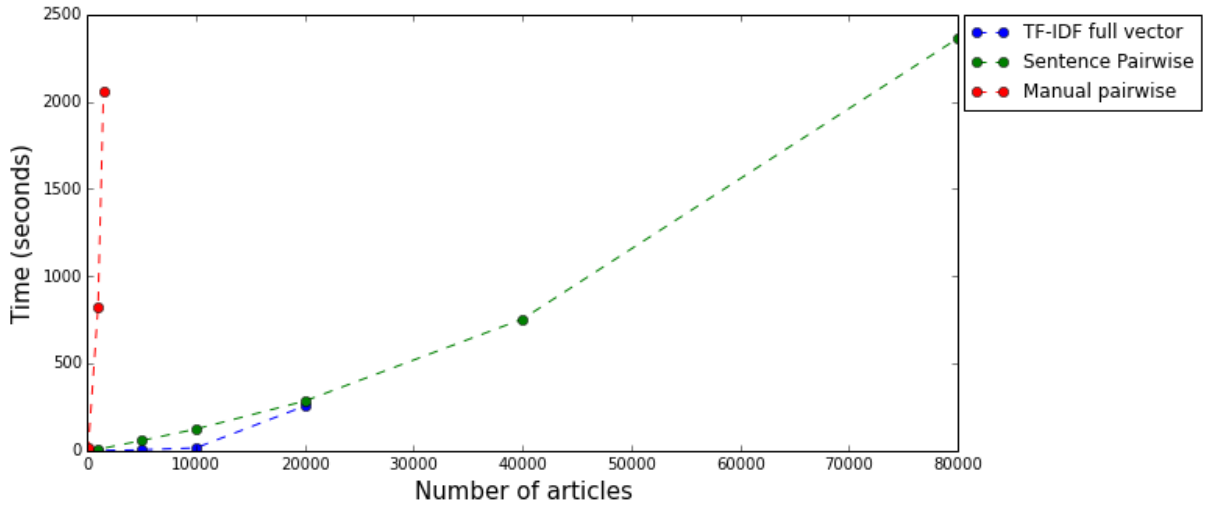
The problem with this is that, although efficient for small amounts of articles, it requires keeping all articles in memory simultaneously, and therefore cannot be run on a full corpus. However, it provides a good benchmark against which we can test our own algorithm. Another benchmarking standard is a full pairwise check, in which every article is checked against every other. This is inefficient and is clearly $Order(n^2)$, but again provides a benchmark. We also attempted to use the OnIOn, discussed before, but as this is designed to work with corpora stored as plain text files in a standard file system, instead of on documents in a database, we were unable to adapt it for our needs.

In Figure 4.1 we can see the timing results for a test sub-corpus of just 500 articles, and we note that both our sentence algorithm and the TF-IDF vector algorithm perform almost instantly for any number of articles below 500. In Figure 4.2 we can our sentence algorithm is linearly bounded, although it is not quite linear. Although the TF-IDF vector algorithm is also efficient, we were unable to deduplicate more than 20 000 articles using this method, receiving a memory error or segmentation fault for higher numbers of articles.




(a)

Figure 4.1: Efficiency of deduplication algorithms on small data-set




(a)

Figure 4.2: Efficiency of deduplication algorithms on large data-set

To test for accuracy, we compared the results of our algorithm against those of a full pair-wise check for a sub-corpus of 1000 articles. Each identified exactly the same pairs of articles as duplicate or near duplicate, although as our algorithm did not necessarily fetch the articles in a sequential order, the pairs were sometimes reversed, meaning a different document got marked for deletion.

For the optimized algorithm, we maintained high accuracy rates for near duplicates when using a limit of 100 matches per sentence, and manually examining all the sentences that

produced more than 100 matches showed that none of these were real sentences, and were mainly the result of incorrectly cleaned text. Examples of these included sentences such as 'Click here to subscribe to our newsletter'. The results for some variants of the 'maximum matches' limit described above can be seen in Table 4.1, which shows the 'maximum matches limit' along with the number of near duplicates detected, the number of exact duplicates detected, and the total running time to process all 90000 articles. We were unable to run either the full pairwise deduplication algorithm or the TF-IDF vector algorithm on the entire corpus, but based on a tests of the sub-corpus containing 1000 articles, we are confident that the first result, with no limit to the maximum matches each sentence could return, is highly accurate. It is interesting that reducing the maximum matches a sentence could return to 5 hardly reduced the accuracy of near duplicates at all, while more than halving the running time. The exact duplicates accuracy is seen to get drastically worse as we increase the limit, but this is not of concern as these duplicates will be removed by detecting hash collisions before the near-duplicate algorithm is run. A final point to note is that although we detected 12 681 exact duplicates using hash collisions, we found only 11022 of these using the slowest and most accurate version of our sentence algorithm. We assume that the extra duplicates consisted of 'articles' which contained only a few words of main text. These articles are included in the corpus as some of the RSS feeds include links to videos or image galleries, and once these have gone through cleaning, often a single short sentence is all the 'main text' that a cleaning algorithm can find to extract. These would not have been detected using our sentence algorithm if the total article text was shorter than 20 characters, or did not contain an identifiable sentence.

Table 4.1: Deduplication accuracy and timings with optimization

| Max Matches | Near | Exact | Time Taken (m) |
|---|---|---|---|
| 99999 | 13443 | 11022 | 68 |
| 1000 | 13387 | 4256 | 40 |
| 5 | 13299 | 3588 | 28 |

## 4.3 Implementation

Our algorithm runs efficiently enough to deduplicate a medium sized corpus which has not undergone any near deduplication. This efficiency is carried through to the deduplication

of a single article, as when we add a new article to a corpus which has gone through deduplication, we still only need to check its similarity against a handful of possible matches. Our implementation strategy was therefore to run the deduplication algorithm on a regular basis, also using the Linux Cron scheduler, as with the data gathering and data cleaning steps. Instead of running through every article, it only needs to look at articles which have not yet undergone the deduplication process, comparing these to any article in the corpus which contains common sentences.

## 4.4 Summary

Having looked at several different algorithms to carry out exact and near deduplication, we evaluated two possibilities for exact deduplication (detecting hash collisions at either an HTML level or a cleaned text level) and three for near deduplication (full pairwise deduplication, TF-IDF vector deduplication, and a custom-created sentence-based pairwise deduplication). We showed that detecting duplicates at an HTML level was not effective, but that exact duplicates could easily be detected on a cleaned text level. For near deduplication, we showed that our algorithm was far faster than a full pairwise deduplication algorithm, and far more memory efficient than a TF-IDF vector algorithm, while still achieving comparable accuracy.

# Chapter 5

# Language Analysis Tools

Having a well-built, thoroughly deduplicated corpus, built from boilerplate-free articles is not enough. Without analysis tools, no meaningful information can be extracted. The final system therefore included a variety of functionality which aimed to facilitate linguistic research. This functionality includes frequency analysis, keyword in context search, and collocation analysis, as well as the ability produce Parts of Speech tagged text. Each of these is described below, where we examine first the theory behind the functionality and then our implementation.

## 5.1 POS Tagging

Parts of Speech, (POS) tagging describes the method of identifying the grammatical function of a word in context, differentiating not only between nouns, verbs, adjectives, etc, but also sub-categories of these. For example, "Adjective", "Adjective, comparative", and "Adjective, superlative" would be assigned separate tags (Santorini, 1990). (Garside *et al.*, 1987) describe the CLAWS word-tagging system as:

> [A] system for tagging English-language texts: that is, for assigning to each word in a text an unambiguous indication of the grammatical class to which this word belongs in this context.

One system which has the functionality to parse and annotate text with POS elements is the so-called Penn Treebank project, a modification of which was used by Liu & Curran

(2006). A similar and alternative POS tagger is the CLAWS word-tagging system (Rayson & Garside, 1998). This has over 30 years of continuous development behind it and was used to tag the British National Corpus. Each tagger requires a license for full use, but 10 percent of the Penn Treebank system is made freely available, and the CLAWS tagger has a Web interface which allows a user to submit limited submissions of a limited length. These submissions are then fully tagged and returned to the user.

### 5.1.1 NLTK

The NLTK (Natural Language Toolkit) is a Python platform, designed to facilitate programs to process natural—that is, human—languages. It provides advanced POS tagging functionality and is free and open source, while also being well-documented (Bird *et al.*, 2009). The NLTK includes the freely available 10 percent of the Penn Treebank can use this to easily tag English sentences. The NLTK can also be set up to use the full Penn Treebank if given external access to this.

### 5.1.2 Implementation

We used the NLTK default Penn tagger. Unfortunately, this is not perfect, and is especially inaccurate when it comes to parsing specifically South African terms, which are of the most interest to our corpus. For example, one article contained the sentence "There would be new Nestea and maas products, and a new yoghurt production facility", which was tagged as "('There', 'EX'), ('would', 'MD'), ('be', 'VB'), ('new', 'JJ'), ('Nestea', 'NNP'), ('and', 'CC'), ('maas', 'VBZ'), ('products', 'NNS')". The South African word "maas" has been tagged as 'VBZ', a present third person singular verb, where it should have been tagged as 'NN', a singular noun. To fix this, the tagger would have to be trained on a correctly tagged set containing South African terms, but as no such reliable corpus exists, the problem becomes one of the chicken and the egg. Ideally, a smaller training corpus would have to be tagged by manually, and the tagger could thus be bootstrapped to perform well on larger data-sets. However, as we had no manually tagged South African text, this was necessarily left for future work.

We ran the tagging functionality on a scheduled Cron task which gathered articles which had been cleaned and deduplicated, but which had not yet been tagged. The cleaned text of these articles was tagged, with the tags saved to the database, associated with their article.

## 5.2 Keyword in Context (KWIC)

'Keyword in Context' or 'KWIC' refers to how the concordance displays results of a specific search term. The name is a bit of a misnomer, as the 'Keyword' can be several words, or a partial word. For a search term $t$, the KWIC functionality searches the entire corpus, and displays any $t$ found, along with a specified amount of context on the left and right. For example, searching the corpus for 'Zuma' produces results as can be seen below:

**KWIC**

**Node:**

| Hit | KWIC | Source |
|---|---|---|
| 1 | d not want to legitimise the ANC's plan to shield Zuma from accounting for the scandalFormer ANC chief w | link |
| 2 | commendations, said it was absurd to suggest that Zuma should pay back the cost of the non-security upgr | link |
| 3 | whip, said there was no evidence suggesting that Zuma had acted illegally.Motshekga said there was no l | link |
| 4 | dent Xi Jinping and South African President Jacob Zuma join their hands at a group photo session during | link |
| 5 | the SARB." On Thursday last week, President Jacob Zuma extended his gratitude to Marcus. "We wish to tha | link |
| 6 | vernor for her excellent service and leadership," Zuma said in a statement at the time. "She has steered | link |
| 7 | ed the achievement and maintenance of stability." Zuma said government valued Marcus's contribution and | link |
| 8 | wished her all the best in her future endeavours. Zuma's office said he would announce the new governor | link |
| 9 | inst a background of reports that President Jacob Zuma's daughter Gugu Zuma is working on a replacement | link |
| 10 | reports that President Jacob Zuma's daughter Gugu Zuma is working on a replacement production for top So | link |
| 11 | atings high. Last week Sowetan reported that Gugu Zuma was working on Uzalo, a replacement production wi | link |
| 12 | earance is still to be confirmed. President Jacob Zuma appointed the commission in 2011 to investigate a | link |
| 13 | President Jacob Zuma is considering concerns from various disability o | link |
| 14 | the new national executive last month, President Zuma reconfigured the ministry of women, children and | link |
| 15 | the department of social development. "President Zuma reiterates that government recognises and address | link |
| 16 | e Willie Seriti, was appointed by President Jacob Zuma three years ago to investigate alleged corruption | link |
| 17 | s final report on the upgrades to President Jacob Zuma's homestead. In an application to the Pietermarit | link |
| 18 | R155 million civil claim relating to work done at Zuma's KwaZulu-Natal homestead. SIU spokesman Boy Ndal | link |

(a)

Figure 5.1: KWIC results for search 'Zuma'

To implement this functionality, we used MongoDB's text search to get all articles matching the search term. We then searched each of these for the exact match (which may appear many times in a single article, meaning that one article may produce several results, which should each be displayed separately). We retrieved these matches as a list of tuples, which each tuple containing (left-context, keyword, right-context), which we could then return to the user with HTML font tags to emphasise the keyword within the context.

## 5.3 Collocations

Stubbs (1995) describes collocations as "a relationship of habitual co-occurrence between

words (lemmas or word-forms)". For example, Stubbs shows that the word 'cause' habitually appears close to words such as 'harm' and 'destruction'. Thus the group (cause, harm) is an interesting *collocation*, while similarly 'cause' is a *collocate* of 'harm' and vice-versa. Although any two words which appear close to each other in any given text can be called a collocation, we are specifically interested in being able to identify *significantly common* collocations within a corpus. For example, the word 'the' might well appear very frequently with the word 'cause', but this is simply because both words are relatively common and is uninteresting. The collocation 'cause trouble' is far more interesting, as it occurs with a significantly high frequency, even when taking into account the frequency of each word.

Collocations can be used for various linguistic research. Stubbs (1995) for example uses collocations to analyse which words have mainly 'negative' collocates (such as 'cause' which almost always appears close to a negative word such as 'trouble' or 'destruction').

We give a brief overview of Stubb's work, with the goal of replicating the functionality of AntConc, a popular, free concordancer which uses Stubb's equations to calculate collocations (**?**).

There is some controversy as to the best way of calculating the significance of a collocation, but one common way is to look at the frequency of the collocation compared to the frequency of each of the words which make up the collocation and compare this statistic to how often the collocation would be *expected* in a completely randomly generated corpus. Stubbs points out that the idea of a completely random corpus is somewhat absurd, but that this method is still useful to rank collocations against each other, even if the absolute 'score' of each is arguably inaccurate. This score is calculated based on the ratio of the *observed* frequency of the collocation to the *expected* frequency. To calculate the expected frequency of a two word collocation, we will talk about one of the two words which make up the collocation as the 'node' word, and the second the 'collocate'. The expected frequency of the collocation occurring (in a random corpus) is calculated by multiplying the frequency of each, relative to the corpus size. Therefore taking:

$n = node$
$c = collocate$
$N = sizeOfCorpus$

Our expected collocation frequency is calculated by:

$frequency(n)/N \times f(c)/N$

Which is the same as:

$$frequency(n) \times frequency(c)/N^2 \tag{5.1}$$

While the *observed* frequency is simply the frequency of the two words appearing together divided by the size of the corpus, or:

$$frequency(n, c)/N \tag{5.2}$$

The significance of the word is determined by the ratio of the observed frequency to the expected frequency, which we calculate, from equations 5.1 and 5.2, as:

$$[frequency(n, c)/N]/[frequency(n) \times frequency(c)/N^2] \tag{5.3}$$

But any natural language is far from random, and these calculations are based on the hypothetical ideal of a perfectly random corpus. This problem is explained by Stubbs (1995), who states: "A problem with this calculation of O/E is that almost any observed co-occurrence is hundreds of times more likely than by chance.", and later shows that significance scores drastically increase for larger corpora. To normalize these scores, we take the base 2 log, so the final equation, referred to as the 'mutual information' or 'MI' score, is:

$$log_2[frequency(n, c) \times N]/frequency(n) \times frequency(c) \tag{5.4}$$

Because the actual score is not used, but only the relative *rank* of the scores, this normalization does not cause any issues. Stubbs (1995) says, "The logarithmic value is used for historical reasons, which are hardly relevant here. Its only effect is to reduce, and therefore possibly to disguise, the difference between scores on different collocates."

A final point to note is that of 'window size'. Collocations describe terms which appear 'close' to each other, but this is quite vague. Following Stubbs, we define a customizable window size, which is the number of words left or right from a node to look for collocates. A default size of five words in either direction is used, but this can be changed per search, and need not be the same for left and right.

### 5.3.1 Implementation

To implement the collocation functionality, we implemented the equations described above in Python. We then allowed the user to search for any search term, which we took as the node word. We retrieved all articles containing the node word, and added words to the left and right of every match building up a frequency table of each collocation. This was then displayed, sorted either by significance score or total frequency, based on options the user had selected prior to submitting the search.

We tested the results of our system against a very small test corpus which we loaded into the open source concordancer AntConc as well. Our algorithm returned identical results to those produced by AntConc.

Our system returned the collocation results along with a total frequency count, and a frequency count for 'left' and 'right'. The last two are because the order that the node and collocate appear in more often is sometimes significant, as a given collocate may habitually appear on the left of a node word, but never on the right.

A screen-shot of collocates for node 'brigade' is seen below. This word is of interest because of the collocation 'blue-light brigade'. This had been previously identified by language researchers as a term possibly exclusive to, or at least originating from, South African English, and our initial corpus supported this. We can see in Figure 5.2 that 'blue-light' appears as the 24th most significant collocate for brigade, with the collocation appearing a total of five times in the corpus. A similar search for collocates in American English and United Kingdom English corpora showed that 'blue-light' did not appear at all, with the closest match in the United Kingdom corpus being found in a news article about boat racing which contained the snippet "The spectacular charge of the Light Blue Brigade from stakeboat to finish line confounded the bookies".

Figure 5.3 shows the same search ordered by frequency instead of significance. We can see that these results are largely uninteresting, as the most frequency collocates are usually always among the most common words in the corpus.

# Collocates

## Node: brigade

| Rank | Freq | Freq(L) | Freq(R) | Stat | Collocate |
|---|---|---|---|---|---|
| 1 | 2 | 2 | 0 | 18.1655420592 | misarata |
| 2 | 1 | 0 | 1 | 17.1655420592 | vrolik |
| 3 | 1 | 1 | 0 | 17.1655420592 | misrata-based |
| 4 | 1 | 1 | 0 | 17.1655420592 | meat-and-potato |
| 5 | 1 | 0 | 1 | 17.1655420592 | kernes |
| 6 | 1 | 1 | 0 | 17.1655420592 | givati |
| 7 | 1 | 1 | 0 | 17.1655420592 | bring-the-goal-line-technology |
| 8 | 1 | 1 | 0 | 17.1655420592 | anti-braff |
| 9 | 2 | 2 | 0 | 16.5805795585 | bilinska |
| 10 | 2 | 2 | 0 | 16.1655420592 | zahawi |
| 11 | 2 | 0 | 2 | 16.1655420592 | metroplus |
| 12 | 1 | 1 | 0 | 16.1655420592 | golani |
| 13 | 2 | 2 | 0 | 15.8436139643 | motorized |
| 14 | 1 | 1 | 0 | 15.5805795585 | superkart |
| 15 | 1 | 1 | 0 | 15.5805795585 | qaaqaa |
| 16 | 1 | 0 | 1 | 15.5805795585 | ml-class |
| 17 | 3 | 3 | 0 | 15.5805795585 | gendarmes |
| 18 | 2 | 2 | 0 | 15.5805795585 | estelien |
| 19 | 1 | 1 | 0 | 15.5805795585 | brandy-and-coke |
| 20 | 4 | 4 | 0 | 15.5805795585 | al-attar |
| 21 | 1 | 1 | 0 | 15.1655420592 | four-strong |
| 22 | 1 | 0 | 1 | 14.8436139643 | pappas |
| 23 | 1 | 1 | 0 | 14.8436139643 | misrata-led |
| 24 | 5 | 5 | 0 | 14.6801152321 | blue-light |

(a)

Figure 5.2: Collocate results for search ‘brigade’, sorted by significance

# Collocates

## Node: brigade

| Rank | Freq | Freq(L) | Freq(R) | Stat | Collocate |
|---|---|---|---|---|---|
| 1 | 194 | 127 | 67 | 3.65466002722 | the |
| 2 | 76 | 52 | 24 | 3.82028272942 | a |
| 3 | 56 | 26 | 30 | 3.16358573229 | of |
| 4 | 53 | 24 | 29 | 3.17342037364 | and |
| 5 | 45 | 43 | 2 | 9.97023821142 | fire |
| 6 | 38 | 10 | 28 | 2.44985095523 | to |
| 7 | 29 | 10 | 19 | 2.42929322956 | in |
| 8 | 25 | 11 | 14 | 3.54934704074 | said |
| 9 | 23 | 12 | 11 | 4.34246909307 | from |
| 10 | 21 | 21 | 0 | 10.557155213 | intervention |
| 11 | 18 | 13 | 5 | 3.85345483457 | by |
| 12 | 18 | 6 | 12 | 3.11895800878 | was |
| 13 | 17 | 16 | 1 | 9.22648145796 | un |
| 14 | 16 | 2 | 14 | 2.6458098367 | on |
| 15 | 15 | 0 | 15 | 3.96004405706 | had |
| 16 | 15 | 6 | 9 | 2.83544215551 | is |
| 17 | 15 | 7 | 8 | 2.57767861496 | that |
| 18 | 14 | 9 | 5 | 8.09926062478 | force |
| 19 | 13 | 2 | 11 | 3.13960394251 | at |
| 20 | 12 | 6 | 6 | 2.86392083419 | with |
| 21 | 11 | 1 | 10 | 3.66186310928 | were |
| 22 | 11 | 8 | 3 | 2.9436597616 | as |
| 23 | 10 | 5 | 5 | 2.12134861276 | for |
| 24 | 10 | 3 | 7 | 2.51217633147 | it |

(a)

Figure 5.3: Collocate results for search ‘brigade’, sorted by total frequency

## 5.4 Summary

In this chapter, we gave an overview of POS tagging, 'keyword in context' functionality, and a description of collocations. We also included a description of the equations used to calculate the significance of collocations. Finally we described how we integrated functionality for all of these analysis tools into our corpus, and gave some example results.

# Chapter 6

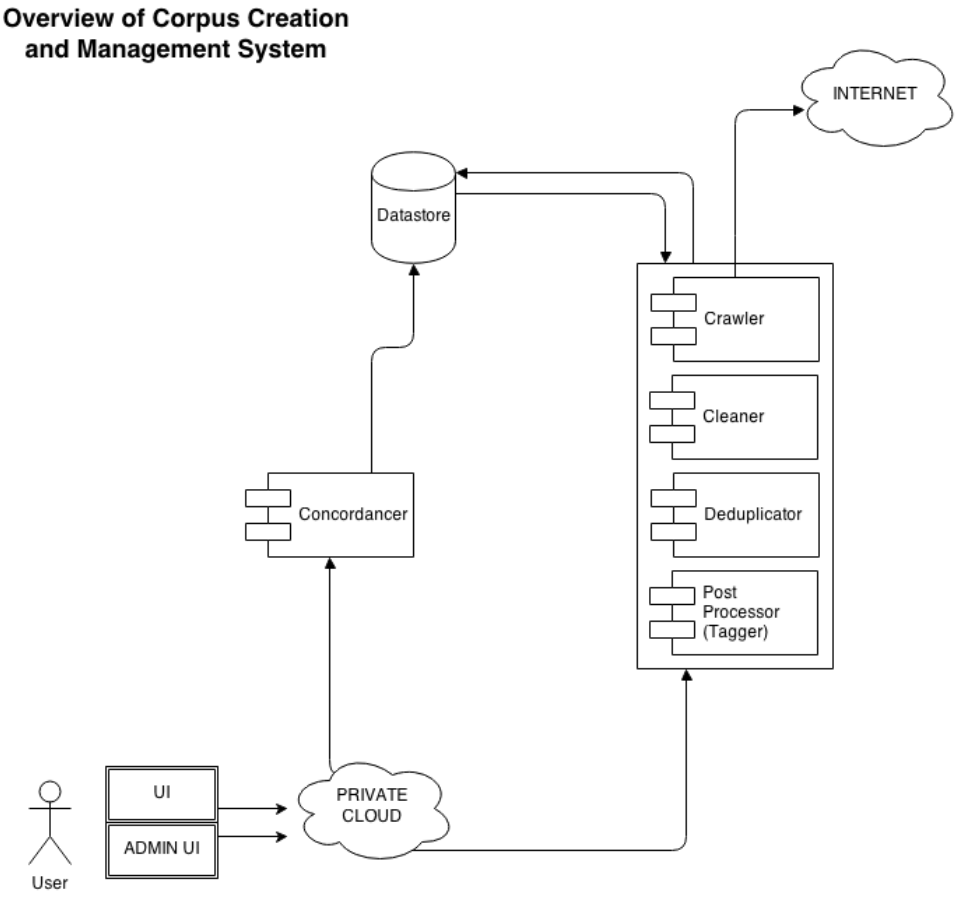
# System Design

We have described how we broke down the sub-goals of corpus creation, and have described and assessed each component individually. In this chapter we briefly describe how we built all the components into a unified system, and give details on the technology used, including details of hardware and software.

## 6.1 Modular Design

Each of the major components of the system were designed to work as independently as possible on the corpus. To achieve this, each component was run as a separate Linux Cron task, and each article contained a number of database flags, set by each component, to ensure that they are processed in the correct order. Thus, the Crawler component would simply crawl HTML and save this to the database with very limited meta-data, such as the crawl date, and the URL, and all other flags would be initialized to False. The Cleaner component would regularly check for new articles which had not yet undergone cleaning, and would save the extracted text to the database, setting the relevant flag to True to show that this had been done. The Deduplicator module then ran on all documents which had already undergone cleaning, and the Tagger only on those articles which had undergone deduplication.

An overview of the system and how the modules interact with the database can be seen in Figure 6.1. At the time of writing, not all functionality had yet been linked up to the user interfaces, and some therefore still had to be implemented at a code level, using the Python modules.




(a)

Figure 6.1: An overview of the entire system

## 6.2 Software Used

Apart from the various code libraries and software packages used to implement the tools and algorithms discussed, we also used the Python framework Flask in conjunction with the Jinja template engine to implement the main and admin user interfaces. This Web application ran on the Web server Apache2 with mod_wsgi enabled to run Python.

## 6.3 Hardware Used

Throughout the research, we ran the corpus system on two machines. The first was a development machine, which consisted of an Intel(R) Core(TM) i5-3570 CPU (Quad Core,

3.40GHz), 8 GB of DDR3 1333 MHz RAM and a 1 TB, 7 200 RPM hard drive with a 64 MB cache. All timings detailed in the research were based on this machine. The second machine was a production machine from which the corpus was accessed and used by language researchers throughout the year, during the development process. This ran on a virtual machine allocated an Intel(R) Xeon(R) CPU E5-2620 v2 (Dual Core, 2.10GHz), 4 GB of virtual RAM and a 45 GB virtual hard drive. Both systems ran Ubuntu, with the development machine running Ubuntu Desktop 14.04 and the production server running Ubuntu Server 12.04.

## 6.4 Current State of the SAE Corpus

At the end of the research period, the corpus had evolved to contain 90 000 articles. From these articles, about 350 000 unique tokens were extracted. Of these, we removed all tokens which contained characters not made up of an permitted character set, which consisted of the standard English alphabet plus the hyphen character. After this filtering process, 224 463 unique word tokens were left. About 13 000 exact duplicates and 13 000 near duplicates were identified in this collection, leaving about 64 000 unique articles.

Language researchers were able to start drawing meaningful analysis from the corpus even while it was still being developed.

# Chapter 7

# Conclusion

In this research, we gave a definition and overview of corpora. We gave a definition for a language-specific evolving Web corpus, and discussed the need for an automated creation and management system for such a corpus which was representative of South African English.

After separating the functionality required for a corpus into separate components, we evaluated each of these in turn, before giving details of how we implemented each. These components included, gathering data, storing data, boilerplate removal, deduplication, and analysis tools.

We discussed various ways to gather data, including backwards crawling and forwards crawling, and we looked at various ways to achieve each. We showed the challenges inherent with using traditional Web crawlers, and discussed how Application Programming Interfaces often provided cleaner and more accurate data for backwards crawling. We examined the challenges of gathering dynamic data, and again showed that these could be at least partially overcome by using APIs. Finally, we looked at RSS feeds, and how they could be used to monitor pre-identified sources for new data, and gather this as it was published.

Several cleaning tools were examined and evaluated. We identified two of these as superior and, after comparing their respective strengths and weaknesses, explained why we needed to use a combination of the two tools for best results.

For deduplication, we examined three candidate algorithms, and showed that the custom algorithm developed by us for this research outperformed the other two.

We then examined the theory of various language analysis tools, including POS tagging, keyword in context and collocations, and discussed our implementation of each of these.

Finally, we gave an overview of the entire system, including descriptions of hardware and software that we used.

We conclude that it is possible to construct an automatic language-specific corpus creator and management system, with a majority of its content sourced from the World Wide Web. We achieved all of our primary research goals, and discuss those which can be improved upon or extended below.

## 7.1 Future Work

No system is ever truly complete, and we give here a brief summary of some sections that we believe can be improved upon, as well as some extensions which could be added to the research.

For the gathering of data, a more general system could be implemented to gather dynamic data. Although we were unable to implement a generic system using the identified tools Crawljax and Selenium, this might still be a possibility given more time and effort. Alternatively, other tools could be discovered or build for such a purpose.

Although our cleaning results were as good as could be expected, we didn't implement and evaluate a machine-learning based system. Such an approach might offer improvements in accuracy if trained for long enough on an accurate data-set.

Deduplication was achieved using a customized algorithm, but we were unable to test this on a large corpus, and can only conjecture on how well this algorithm will perform as the corpus grows. Furthermore, we showed how efficient a TF-IDF vector deduplication algorithm was for even smaller data-sets. Research could be done into building a cluster algorithm based on this, in which full similarity matrices are constructed for subsets of the corpus. Each subset would be individually deduplicated and these subsets would then be merged to result in a corpus free from duplicates.

Finally, our user interface could be extended to be more user-friendly, aesthetically pleasing, and functional.

# References

antconcwebsite Anthony, L. [n.d]. AntConc Collocates Overview. Online. Available from: `http://www.laurenceanthony.net/software/AntConc_Help/Collocates/Overview_(Collocates).htm`. Accessed 25 October 2014.

Baroni, Marco, Chantree, Francis, Kilgarriff, Adam, & Sharoff, Serge. 2008. Cleaneval: a Competition for Cleaning Web Pages. *In: LREC*.

Baroni, Marco, Bernardini, Silvia, Ferraresi, Adriano, & Zanchetta, Eros. 2009. The WaCky wide web: a collection of very large linguistically processed web-crawled corpora. *Language resources and evaluation*, **43**(3), 209–226.

Bird, S., Klein, E., & Loper, E. 2009. *Natural Language Processing with Python*. O'Reilly Media.

Cattell, Rick. 2011. Scalable SQL and NoSQL data stores. *ACM SIGMOD Record*, **39**(4), 12–27.

Cavnar, William B, Trenkle, John M, *et al.* 1994. N-gram-based text categorization. *Ann Arbor MI*, **48113**(2), 161–175.

De Klerk, Vivian. 2002. Towards a corpus of Black South African English. *Southern African Linguistics and Applied Language Studies*, **20**(1-2), 25–35.

Disqus. 2007. Disqus.com. Online. Available from: `https://disqus.com/api/docs/forums/listPosts/`. Accessed 10 May 2014.

Evert, Stefan. 2007. StupidOS: A high-precision approach to boilerplate removal. *Page 123 of: Building and exploring web corpora: proceedings of the 3rd web as corpus workshop, incorporating cleaneval*, vol. 123.

Evert, Stefan. 2008. A Lightweight and Efficient Tool for Cleaning Web Pages. *In: LREC*.

Fairon, Cédrick. 2006. Corporator: A tool for creating RSS-based specialized corpora. *Pages 43–49 of: Proceedings of the 2nd International Workshop on Web as Corpus*. Association for Computational Linguistics.

Fairon, Cédrick, Macé, Kévin, & Naets, Hubert. 2008. GlossaNet 2: a linguistic search engine for RSS-based corpora. *Pages 34–39 of: Proceedings of the 4th web as corpus workshop (WAC-4)*. Citeseer.

Finn, Aidan. 2010. BTE gets an update. Online. Available from: `http://www.aidanf.net/2010/05/11/bte-gets-an-update.html`. Accessed 14 May 2014.

Garside, R., Sampson, G., & Leech, G.N. 1987. *The Computational analysis of English: a corpus-based approach*. Longman.

Généreux, Michel, Hendrickx, Iris, & Mendes, Amália. 2012. A large Portuguese corpus on-line: cleaning and preprocessing. *Pages 113–120 of: Computational Processing of the Portuguese Language*. Springer.

Gheorghiu, Grig. 2005. A look at selenium. *Better Software*, **7**(8), 38.

Guevara, Emiliano. 2010. NoWaC: A Large Web-based Corpus for Norwegian. *Pages 1–7 of: Proceedings of the NAACL HLT 2010 Sixth Web As Corpus Workshop*. WAC-6 '10. Stroudsburg, PA, USA: Association for Computational Linguistics.

Hecht, Robin, & Jablonski, Stefan. 2011. NoSQL Evaluation.

Heydon, Allan, & Najork, Marc. 1999. Mercator: A scalable, extensible web crawler. *World Wide Web*, **2**(4), 219–229.

Janssens, Jeroen. 2012. Online. Available from: `https://pypi.python.org/pypi/reporter/0.1.2`. Accessed 14 May 2014.

Jeffery, Chris. 2003. On compiling a corpus of South African English. *Southern African Linguistics and Applied Language Studies*, **21**(4), 341–344.

Kohlschütter, Christian, Fankhauser, Peter, & Nejdl, Wolfgang. 2010. Boilerplate detection using shallow text features. *Pages 441–450 of: Proceedings of the third ACM international conference on Web search and data mining*. ACM.

Linster, Marc. 2014. Postgres Outperforms MongoDB and Ushers in New Developer Reality. Online. Available from: `http://blogs.enterprisedb.com/2014/09/24/postgres-outperforms-mongodb-and-ushers-in-new-developer-reality/`. Accessed 25 October 2014.

Liu, Vinci, & Curran, James R. 2006. Web Text Corpus for Natural Language Processing. *In: EACL*.

Mesbah, Ali, Bozdag, Engin, & van Deursen, Arie. 2008. Crawling Ajax by inferring user interface state changes. *Pages 122–134 of: Web Engineering, 2008. ICWE'08. Eighth International Conference on*. IEEE.

Mohr, Gordon, Stack, Michael, Rnitovic, Igor, Avery, Dan, & Kimpton, Michele. 2004. Introduction to heritrix. *In: 4th International Web Archiving Workshop*.

OUP. 2014. About the Oxford English Corpus. Online. Available from: `http://www.oxforddictionaries.com/words/about-the-oxford-english-corpus`. Accessed 14 May 2014.

Pienaar, Leela, & De Klerk, Vivian. 2009. Towards a corpus of South African English: corralling the sub-varieties. *Lexikos*, **19**, 353–371.

Pilgrim, Mark. 2010. Online. Available from: `https://pypi.python.org/pypi/feedparser`. Accessed 14 May 2014.

Pomikálek, Jan. 2011. *Removing boilerplate and duplicate content from web corpora*. Ph.D. thesis, Masaryk University.

Pomikálek, Jan, Rychlỳ, Pavel, Kilgarriff, Adam, *et al.* 2009. Scaling to billion-plus word corpora. *Advances in Computational Linguistics*, **41**, 3–13.

Pomikálek, Jan, Jakubícek, Milos, & Rychlỳ, Pavel. 2012. Building a 70 billion word corpus of English from ClueWeb. *Pages 502–506 of: LREC*.

Prabowo, Rudy, & Thelwall, Mike. 2006. A comparison of feature selection methods for an evolving RSS feed corpus. *Information processing & management*, **42**(6), 1491–1512.

Rayson, Paul, & Garside, Roger. 1998. The claws web tagger. *ICAME Journal*, **22**, 121–123.

RU. 2014a. Corpus of South African English @ Rhodes. Online. Available from: `http://www.ru.ac.za/englishlanguageandlinguistics/research/corpusofsouthafricanenglishrhodes/`. Accessed 28 May 2014.

RU. 2014b. DSAE Publications. Online. Available from: `http://www.ru.ac.za/dsae/publications/`. Accessed 28 May 2014.

Santorini, Beatrice. 1990. Part-of-speech tagging guidelines for the Penn Treebank Project (3rd revision).

Scrapy. Scrapy.org. Online. Available from: `http://scrapy.org/`. Accessed 20 May 2014.

Spolsky, Joel. 2003. Online. Available from: `http://www.joelonsoftware.com/articles/Unicode.html`. Accessed 20 September 2014.

Stubbs, Michael. 1995. Collocations and semantic profiles: on the cause of the trouble with quantitative studies. *Functions of language*, **2**(1), 23–55.

# Appendices

# Appendix A

# South African Publications

The following is a list of identified online South African content sources, including newspapers and magazines. The first list of 11 articles consists of those that were used for research during the initial construction phase of the corpus, while the second list consists of those which will be added to the corpus watch-list after the main development phase is completed.

## A.1 Primary sources

The Independent Online: http://iol.co.za
Grocott's Mail: http://grocotts.co.za
Mail and Guardian: http://mg.co.za
The Oppidan Press: http://oppidanpress.com
The Citizen: http://citizen.co.za
Sowetan Live: http://www.sowetanlive.co.za/
Dispatch Live: http://www.dispatchlive.co.za
The New Age: http://www.thenewage.co.za
Business Day Live: http://bdlive.co.za
Times Live: http://www.timeslive.co.za
Daily Maverick: http://www.dailymaverick.co.za

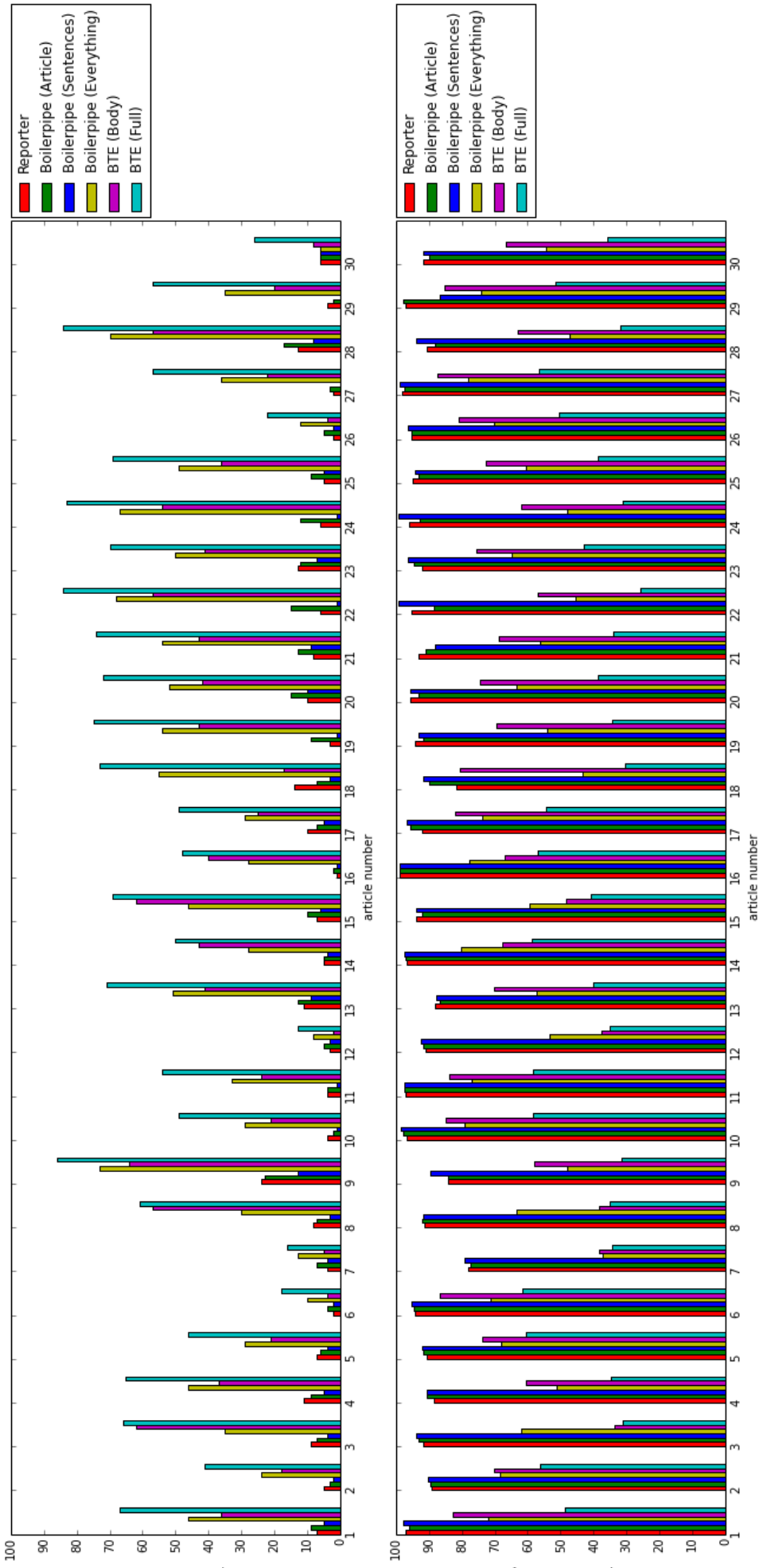
## A.2 Secondary sources

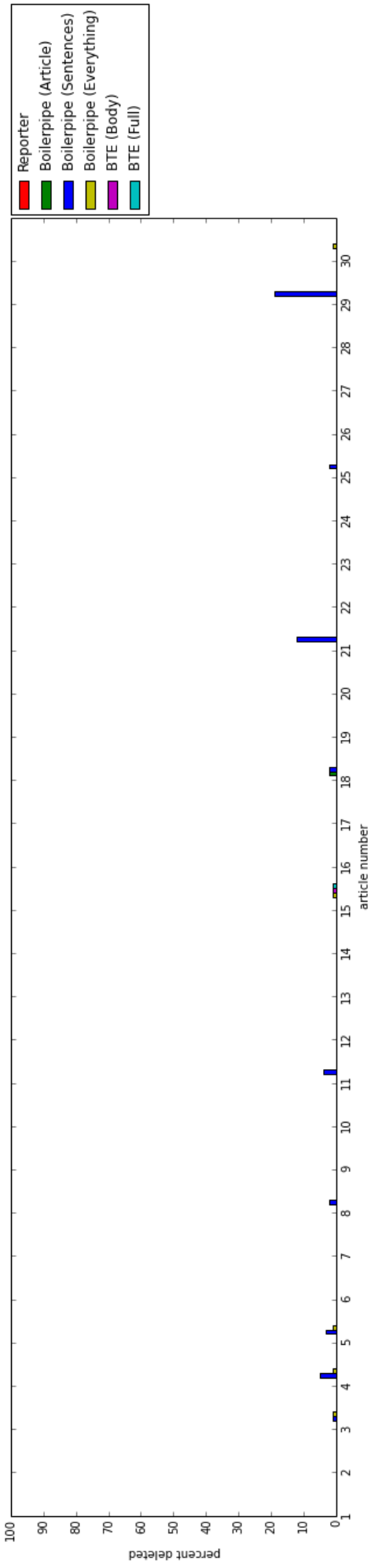
The Mercury: http://www.themercury.co.za
The Witness: http://www.witness.co.za
Daily Sun: http://www.dailysun.co.za
The Herald: http://www.heraldlive.co.za
News24: http://www.news24.com
City Press: http://www.citypress.co.za
Eyewitness News: http://ewn.co.za
Iafrica: http://www.iafrica.com
Finance24: http://www.fin24.com/finweek
You: http://you.co.za
Zigzag: http://www.zigzag.co.za
Drum: http://drum.co.za
Daily Fix: http://dailyfix.co.za
Brainstorm: http://www.brainstormmag.co.za
GQ: http://gq.co.za
Men's Health: http://www.mh.co.za
KickOff: http://www.kickoff.com
Country Life: http://www.countrylife.co.za
Marie Claire: http://www.marieclairvoyant.com
Wine Mag: http://winemag.co.za
Mahala: http://www.mahala.co.za

# Appendix B

# IOL dataset

The graphs below represent the cleaning results for the different algorithms discussed in Chapter 3. Each shows the overall similarity, the amount of inserted text, and the amount of deleted text of each algorithm for each of 30 manually cleaned articles from *IOL*.


Reporter
Boilerpipe (Article)
Boilerpipe (Sentences)
Boilerpipe (Everything)
BTE (Body)
BTE (Full)
article number
Reporter
Boilerpipe (Article)
Boilerpipe (Sentences)
Boilerpipe (Everything)
BTE (Body)
BTE (Full)
article number


Reporter
Boilerpipe (Article)
Boilerpipe (Sentences)
Boilerpipe (Everything)
BTE (Body)
BTE (Full)
percent deleted
article number
0
10
20
30
40
50
60
70
80
90
100
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30