Building a Corpus of South African English: Literature Review

Submitted in partial fulfilment
of the requirements of the degree of

Bachelor of Arts (Honours)

of Rhodes University

Gareth Dwyer - garethdwyer@gmail.com
Supervisor: James Connan

Grahamstown, South Africa
May 2014

1 Introduction

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". This review looks into past work done in corpus creation, with the aim to create a corpus specific to SAE, using similar methods. A corpus of SAE is needed by the Dictionary Unit for South African English (DSAE), the publisher 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, 2014b,a). 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 of analysis which can be run.

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. Although detailed work has been done in building language-specific corpora, there exists no large and accessible corpus of South African English. This review looks not only at the methods used in building these corpora, but also examines work done in the related fields of data storage, web crawling, and deduplication, as it is possible that with the fast-changing state of the World Wide Web (especially with an increase of dynamically created content), and with advances in some of these related fields, that better methods than those used in building existing corpora could be developed. The structure of this chapter is as follows:

- Section 2: Overview of existing corpora
- Section 3: Gathering data

- Section 4: Data storage
- Section 5: Cleaning data
- Section 6: Deduplication
- Section 7: Tagging and analysis

Section two gives only an overview of each corpus, with more details of each examined under the later relevant sections.

2 Existing Corpora

Only very sparse and incomplete work has been done in building corpora specific to South African English, but large projects for other general corpora and for language-specific corpora exist. Below is a look at some of the more notable examples of each.

2.1 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.

2.2 General Web Corpora

Liu & Curran (2006) built a 10 billion word corpus of English in order to help train systems for natural language processing tasks. Unlike many corpora built before this one, the authors used custom web spiders to gather content instead of using the search engine APIs for Google, Yahoo, or Bing. Custom algorithms were also developed for cleaning

the gathered text, instead of using existing tools. Because the customized methods and algorithms for each of these steps is described in detail, this work is especially useful for those who aim to build a similar system.

2.3 Language-specific Corpora

There exist several language-specific corpora whose authors have published detailed information on the methods used in their creation. This review examines three such examples. The first is "WaCky", which is in fact 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). The second is "NoWaC", a corpus for Norwegian, which used methods overlapping in part with the WaCky collection (Guevara, 2010), and finally "CRPC", a large corpus of contemporary Portuguese (Généreux *et al.*, 2012).

WaCky The *WaCky* corpus 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).

NoWaC The Norwegian corpus, *NoWac*, followed a very similar methodology to *WaCky* in creation, but 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 (Guevara, 2010).

CRPC The Reference Corpus of Contemporary Portuguese does not explicitly state statistics on size and number of documents, but 312 million tokens were extracted after cleaning and deduplication.

3 Gathering Data

There exist several ways to automatically gather data for a corpus. The corpus systems already mentioned used either a customized web spider approach –in which spiders were

seeded with selected Uniform Resource Locators (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 (Asynchronous JavaScript and XML) call when the user scrolls down far enough on the page, or clicks on a specific 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. Each of these points is examined below.

3.1 Crawling the Web

Creating a web crawler or web spider from scratch is possible. Given a start URL, it simply needs to 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. 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."

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 Guevara (2010) in building the NoWaC corpus of Norwegian.

Scrapy Scrapy is another scraping framework, written in Python, which also focuses on being extensible and is also 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 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 websites, 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 almost 50 companies which acknowledge that they use Scrapy, with uses ranging from scraping and crawling to testing and pre-caching web sites and apps (Scrapy, n.d.).

3.2 RSS feeds

A less common way of gathering data for a corpus is by using Rich Site Summary (RSS) feeds. RSS feeds are presented using Extensible Markup Language (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 newspapers, and even many of the smaller ones such as *Grocott's Mail* and *The Oppidan Press*, and may be found for other categories of online content too, such as online magazines. Prabowo & Thelwall (2006) describes 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 analyze 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 data such as word lists, error pages, or adverts) and this content is also presented in a standardized format. Furthermore, useful metadata 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 well.

feedparser Feedparser is 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).

3.3 Dynamic Data

The most common data that is 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 some 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 websites 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 stating, solve "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 of 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, but informal testing suggested that this restriction is circumventable.

4 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[1] systems. Each paradigm has advantages and disadvantages, so it is worth taking some time to examine each in choosing what database system to use. This review looks 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 flavour of NoSQL database which seems at first blush to suit the needs of corpus creation.

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 grows 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 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 needs 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

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

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.

5 Cleaning Data

Data taken from the World Wide Web is not in a format ideal for a corpus. Not only is 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 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, but discussion of this will be left for section 7. 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.

cleaning tools While one approach is to build a customized solution, as above, another is to use a "cleaning tool". 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[2] 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

[2]See section 6 for more on n-grams

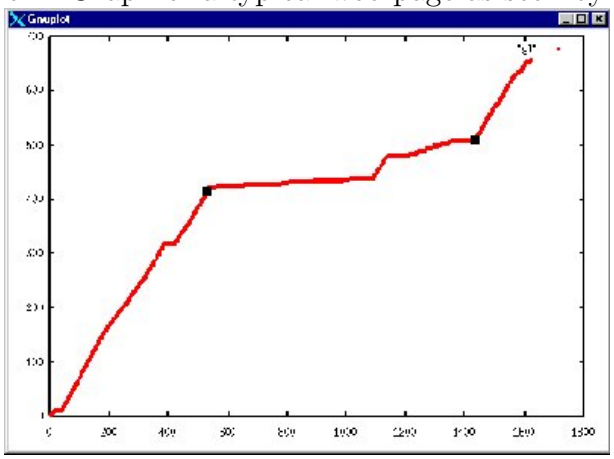
which arise in attempting to parse HTML, both in terms of efficiency and accuracy, with regular expressions are well-known.[3]

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 below.

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.

Figure 1: Graph of a typical web page as seen by BTE



(Finn, 2010)

[3]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.

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). Although there is no literature on a comparison of these two tools, a quick test suggested that Reporter is more accurate than BTE.

6 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, p. ?) says:

> 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 O(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, or at least linked in someway to near-duplicates.

N-grams Most 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 is 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 is 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% similarity being discarded.

Pomikálek *et al.* (2012, p.503) have since developed a tool called 'onion', 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 extremely large corpora.

hashing and fingerprints Although we have focused on the more problematic task of near deduplication, detection of exact duplicates is still necessary, and although this can be done using straightforward hashing, commonly-used hashing algorithms take pains to be 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 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 calculations show that the probability of an MD5 hash collision (where two different inputs produce the same output) is approximately $1/2^{128}$, which are arguably low enough for the risk to be insignificant[4].

7 Tagging and Analysis

After data has been collected, it needs to be tagged and analyzed, in order for lexicographers and researches to make full use of it. There exist various so-called 'taggers', which facilitate the grammatical analysis of sections of text. These are summarized below.

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

POS tagging POS (Parts of Speech) 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.

Penn Treebank tagger The so-called Penn Treebank project is one such POS tagger, and was used by (Liu & Curran, 2006). The tagger was modified in order to correctly identify tokens typically found in web pages such as email addresses.

CLAWS word-tagging system Another 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.

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 10 percent of the Penn Treebank (as much as is freely available) and can use this to tag English sentences. The NLTK can also be set up to use the full Penn Treebank if given external access to this.

8 Conclusion

This review looked at prior work done in building web corpora and related fields, with the goal of incorporating this work into the creation a corpus of South African English. This prior work includes research done in building language-specific corpora, and in fields which relate to the various sub-problems, including scraping content from the Internet (both static and dynamic), data storage and database systems, cleaning data (removing HTML tags and other irrelevant data or 'boilerplate') and deduplication (for both exact- and near-duplicates) as well as various tools developed to assist in each of these areas.

References

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`.

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`.

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

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 28 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`.

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 on 20 May 2014.