WikiPediaPedia

Cian Johnston

B.A. (Mod.) Computer Science, Linguistics and a Language

Final Year Project, April 2010

Supervisor: Carl Vogel

Second Reader: Martin Emms
Declaration

I hereby declare that this thesis is entirely my own work and that it has not been submitted as an exercise for a degree at any other university.

________________________________________  April 7, 2010
Cian Johnston
Permission to Lend

I agree that the Library and other agents of the College may lend or copy this thesis upon request.

_________________________________________ April 7, 2010

Cian Johnston
Acknowledgements

I would first and foremost like to thank Dr. Carl Vogel, without whom this project would not be possible. His patient guidance, interesting ideas, support and sound advice have aided me through my entire time spent at Trinity College.

A very special thanks also goes towards the brilliant minds who created the tools and resources used in this project, in particular, the developers of Python, NLTK, Enju, and COMLEX.

Thanks are also due to Aengus Walton, for his insight into the issues relating to the wikipedia corpus, and for his advice in completing this project.

Last but not least, I would like to thank my family for supporting me throughout my college years - both emotionally and financially.
Contents

1 Introduction .................................................. 1
   1.1 Motivation .............................................. 1
   1.2 Aims ....................................................... 2

2 Resources and Tools ......................................... 3
   2.1 Software .................................................. 3
      2.1.1 Python ............................................. 3
      2.1.2 R .................................................... 7
      2.1.3 TreeTagger .......................................... 7
      2.1.4 Enju HPSG Parser ................................ 7
   2.2 Resources ................................................ 8
      2.2.1 Wikipedia Corpus ................................. 8
      2.2.2 Project Gutenberg Encyclopedia .................. 9
      2.2.3 COMLEX ............................................ 9

3 Resource Preparation ........................................ 10
   3.1 Wikipedia Corpus ....................................... 10
      3.1.1 The Original Corpus ............................. 10
      3.1.2 Extending the Corpus ............................. 12
A Code Listings

A.1 unxml.py ................................. 49
A.2 lemmas.py ............................... 51
A.3 parse.sh ............................... 53
A.4 util.py ................................. 53
A.5 analyze.py .............................. 57
A.6 style_calc.py ............................ 58
A.7 naive_arity.py ......................... 59
A.8 parsed_arity.py ......................... 64
A.9 zipf_deviance.py ....................... 66
A.10 zipf.py ............................... 67
A.11 readability.py ......................... 69
A.12 tree_depth.py ......................... 73
A.13 parse_complex.py ...................... 75
List of Figures

4.1 Incorrect parse for the sentence *The dog the man walks drinks water.* ............... 27

4.2 Incorrect parse for a sentence from the Gutenberg Encyclopedia. ........................................ 29

5.1 Naive Arity Estimate versus Parsed Measurement per Article ........................................ 36

5.2 Naive Arity Estimate versus Parsed Measurement per Sentence ..................................... 37

5.3 Parsed Arity Measurement versus Production-based measure of left-branching per article ................................. 39

5.4 Parsed Arity Measurement versus Production-based measure of left-branching per sentence ........................................ 40

5.5 Production-based measure of left branching versus sentence length ........................................ 43

5.6 Sentential verb arity versus sentence length ......................................................... 44
List of Tables

5.1  Average Values per Article across Subcorpora . . . . . 34
5.2  Correlations of verb arity and left-branching with var-
     ious readability measures . . . . . . . . . . . . . . . 41
Abstract

What makes a text easier or harder to read? There exist many readability measures (e.g. FOG, SMOG, Flesch-Kinckaid Reading Ease / Grade Level, etc.) designed to judge just this. However, these are all rather linguistically naive, and, as such, do not explicitly take account of either the structural complexity or the relational complexity of a text.

This paper considers verb arity as being a possible measure of relational complexity in a text. Measuring the numbers of arguments taken by verbs is a costly process however, usually requiring either a human or a parser with a broad-coverage grammar to make any reliable judgements on the argument-adjunct distinction.

Is it possible to naively estimate verb arity? In this thesis, the Wikipedia Corpus created by Walton (2009) is taken, and parsed using the Enju HPSG parser. The COMLEX verb subcategorization dictionary is used to naively measure the arities of the verbs in the Wikipedia corpus, which are then compared with the arities obtained by the parser. The freely available first volume of the Project Gutenberg Encyclopedia is used for comparison.

However, it remains to be seen whether verb arity is a useful measure of complexity. To this end, the measure of raw production-based depth detailed in Sampson (1997) is applied to the corpora and compared with the measured verb arities on a per-article and a per-sentence basis.
Chapter 1

Introduction

A previous project completed by Walton (2009) was concerned with creating a corpus using a subset of the English Wikipedia, on which a statistical analysis of stylistics and homogeneity was performed using various measures. This project is, in many respects, a continuation, focusing on verb arity as a measure of relational complexity.

1.1 Motivation

The complexity of a text (i.e. how easy or hard a text is to read) is generally determined by textual complexity measures. Amazon’s Text Stats, for example, allows customers to view various readability indices for a text they may wish to purchase (currently Gunning fog Index, Flesch Index and Flesch-Kincaid Index are provided). These measures are predominantly textual in that they take measures such as sentence length, word length and word frequency into account.
These measures, while well-known and very useful, however, ignore more detailed linguistic information about the text, such as the structural or relational complexity. It would seem expedient to consider such complexity measures in tandem to the more conventional textual complexity measures when evaluating the readability of a text.

1.2 Aims

Firstly, the question as to whether it is possible to estimate verb arity naively needed to be answered. To that end, the extremely detailed and complete COMLEX verb subcategorisation dictionary was used to naively estimate verb arities.

Secondly, to gauge the reliability of this naive measure of verb arity, the Wikipedia corpus was parsed with the Enju HPSG parser, and the arities were compared on a per-article basis and a per-sentence basis.

Finally, the measure of tree depth discussed by Sampson (1995) is compared on both a per-article and a per-sentence basis to the verb arity measures, with a view to ascertaining whether it is a meaningful measure of complexity.

As a side effect of this project, a large parsed corpus comprised of both Wikipedia and the Project Gutenberg Encyclopedia is now available, and should be a valuable resource for anyone requiring a freely available corpus on which to run experiments relating to corpus linguistics.
Chapter 2

Resources and Tools

Most, if not all, of the software used for this project is open-source. Of the resources, COMLEX is the only one not freely available for academic use.

2.1 Software

2.1.1 Python

Python is a dynamically typed, interpreted, multi-paradigm general-purpose programming language. It includes modules for almost any conceivable task, and thousands of other modules are freely available at the [Python Package Index](https://pypi.org).

Python was chosen as the primary programming language for this project due to its:

- Ease and speed of development
• Minimal and uncluttered syntax

• Interactive mode

• Wealth of useful modules (NLTK, csv, lxml)

Python may not be the fastest programming language available in terms of speed of execution, as it is not compiled to machine code, but interpreted on-the-fly. However, it offers a major speed advantage in terms of getting programs written. Additionally, its abovementioned interactive mode allows the user to quickly prototype functions and generally “play around” with the language in a much more informal fashion than with other languages such as C or Java.

Python source code listings will appear as shown below:

```
# Cian Johnston <johnstc@tcd.ie>
# These are comments

def sample_code(arg1, arg2):
    """This is a docstring""

    for i in range(10):
        print("%d bottles of beer on the wall..." %i)
```

Names of Python scripts (or scripts used in any other language) are in monospaced font. Unless stated otherwise, the scripts are included as code listings in the Appendix.

A number of Python modules were used in the project; the most important of them are detailed below.
NLTK

NLTK (Natural Language ToolKit) is a Python module containing a number of useful tools for Natural Language Processing, including (but not limited to):

• Corpora
  – Brown Corpus
  – Penn Treebank
  – EUROPARL Corpus

• Statistical Tools

• Grammar Development Tools

• Parsers

• Tokenizers

• Taggers

Basically, it is an essential item in any Python-using computational linguist’s toolbox.

csv

Python has native support for reading and writing in the CSV (Comma-Separated Value) format. This file format\footnote{There is no formal standard for CSV; the general consensus is that fields are separated by commas and that commas inside double-quoted strings are ignored.} was used due to its
widespread support (if not supported, it is trivial to hack due to its minimal syntax), and its excellent Python integration. An example is given below in Listing 2.2.

Listing 2.2: Writing some simple values to a CSV file

```python
import csv
def csv_example():
    f = open("filename.csv", "w")
    writer = csv.writer(f, quoting=csv.QUOTE_ALL)
    header = ["i", "i squared"]
    writer.writerow(header)
    nums = range(100)
    for n in nums:
        writer.writerow([n, n ** 2])
    f.close()
```

lxml

lxml is “a Pythonic binding for the libxml2 and libxslt libraries”. It is the most feature rich and easy to use library available for working with XML in Python. Most importantly, it is fast, which was an important consideration given that a large portion of the project involved processing XML. The Enju HPSG parser can output parse trees in XML format; and during the course of parsing the Wikipedia Corpus, roughly 6 gigabytes of XML were generated. Additionally, lxml supports XPath, a query language that allows one to perform searches on XML element trees. For example, the following Python code returns all verb token elements in an Enju parse tree:

Listing 2.3: lxml XPath example

```python
def verb_tokens(element_tree):
    ```
2.1.2 R

The R Software Environment is a free alternative to programs such as SPSS (Statistical Package for the Social Sciences). It provides a wide variety of statistical techniques (linear and nonlinear modelling, statistical tests, classification, clustering...), and includes an integrated set of software utilities for data manipulation, calculation and graphical display of data.

2.1.3 TreeTagger

TreeTagger [8] is a probabilistic (read: fast), freely available, extensible, reliable, and cross-platform Part-Of-Speech (POS) tagger, lemmatizer and (optionally) chunker, developed by Helmut Schmid at the Institute for Natural Language Processing at the University of Stuttgart, Germany. Initial exploratory statistics in the Wikipedia Corpus called for, at the very least, POS tagging, lemmatization and chunking; TreeTagger was the best resource available for that purpose.

2.1.4 Enju HPSG Parser

Enju is a syntactic parser for the English language, jointly developed by the Tsuji laboratory and the Department of Computer Science at the University of Tokyo. It comprises a wide-coverage probabilistic
HPSG grammar and an efficient parsing algorithm, and can efficiently analyse the syntactic and semantic structures of English sentences. It is freely available for academic usage, and is supported on multiple platforms (source code is also available).

2.2 Resources

2.2.1 Wikipedia Corpus

The Wikipedia Corpus used was compiled by Walton (2009) \cite{walton2009}. It comprises some 7,000 articles from the English Wikipedia, and includes three subcorpora:

**BA-Corpus** (Best Articles) Articles that would be selected for an “offline release version” of Wikipedia.

**LS-Corpus** (Lacking Sources) Articles that have been marked as requiring more sources and/or citations.

**SE-Corpus** (Style Editing) Articles that do not conform to Wikipedia’s manual of style, and require substantial amounts of re-writing.

Rather than choosing articles at random from the English Wikipedia, Walton chose to create these subcorpora on a non-random basis to ensure a good representative sample of the entire Wikipedia. One point worthy of note is that, due to the dynamic and rapidly changing nature of Wikipedia, the corpus is most likely slightly obsolete, as it was compiled in late 2008. The corpus and Walton’s original supporting code are all available to explore online.
2.2.2 Project Gutenberg Encyclopedia

For a reference text, it seemed that an encyclopedia would be the best candidate text to consider. The Project Gutenberg Encyclopedia filled this role by virtue of being the only candidate easily available. It is essentially the first volume of the 1911 Encyclopedia Brittanica, scanned and proofread.

2.2.3 COMLEX

In order to naively estimate verb arity, information on the number of arguments taken by verbs in the English language was required. One option would have been to take a large parsed corpus and to compute arity values for verbs. However, such information sources already exist, one of which is COMLEX. It contains roughly 38,000 head words, along with extremely detailed morphological and syntactic information about each head word. A sample COMLEX entry for the verb \textit{abase} is given below:

\begin{verbatim}
(VERB :ORTH "abase" :SUBC ((NP)) :FEATURES ((VVERYVING :PASTPART T)))
\end{verbatim}

The only information immediately relevant to the naive estimation of verb arity used was the \texttt{:SUBC} feature, which details the type and number of arguments that the verb can take. In the example given, the verb \textit{abase} only ever has one argument, a noun phrase. All features and values for head words in COMLEX are specified in the COMLEX syntax reference manual \cite{3}.
Chapter 3

Resource Preparation

Many of the resources mentioned in the previous chapter required some measure of preprocessing in order to be truly useful. This chapter details the preprocessing steps taken prior to the analysis stage.

3.1 Wikipedia Corpus

3.1.1 The Original Corpus

The original Wikipedia Corpus contained the following files for each article:

Wiki Mark-up (*.markup.txt)

MediaWiki, the Wiki used by Wikipedia, utilises a markup language called wiki mark-up. A user would write an article in wiki mark-up, and the MediaWiki server then compiles the mark-up into HTML. There is no formally defined standard for wiki mark-up, which poses an immediate problem for anyone wish-
ing to access information in this format. Walton (2009) used a
toolset written by Evan Jones that utilises the MediaWiki parser
and a modified version of the BeautifulSoup XML/HTML pars-
ing library to convert the mark-up to pseudo-XML. These files
contain the original wiki mark-up for the article.

Edit History (*.hist)
These files contain the edit history for the article in CSV format.
They contain information about the Wikipedia username of the
editor, the number of edits made by this editor, the date of the
first edit, and the date of the last edit.\footnote{The timeframe of the edits is between 2000, a year before Wikipedia was established, and October 8th, 2008, which was the date of the snapshot used by Walton (2009)}

Article Content (*.txt)
The content of the article once the wiki mark-up had been re-
moved, in a pseudo-XML format.

POS-tagged text (*.pos)
These files contain the POS-tagged text of the article, tagged by
Walton (2009) with the default maximum entropy POS tagger
in NLTK. Words and tags are separated with slashes (/), one
entry per line.

Open Category POS-tagged text (*.opencategories.pos)
These files contain a subset of the information in the POS-
tagged text files, except that “closed category words” are omit-
Walton (2009) also computed the word frequency distributions for each article. These are in CSV format.

Each of these articles is contained within two subdirectories named the first and second bytes of the md5sum hexadecimal digest of the article title. \footnote{E.g. for the article Montreal, the hexadecimal digest of the article name's md5sum is 0004b45d97463070fbfd2c26207c427d, so the directory structure is 00/04/}.

### 3.1.2 Extending the Corpus

At the time of writing, a number of changes and additions have been made by the author to his local copy of the Wikipedia corpus. These changes are detailed below.

#### Minor changes

Purely due to aesthetic reasons, the pseudo-XML files were renamed from \texttt{*.txt} to \texttt{*.xml}. Additionally, unnecessary \texttt{*.txt} filename extensions were removed from other files.

#### Extraction of the article text from the pseudo-XML

The script \texttt{unxml.py} was written to parse the pseudo-XML of the articles and extract the contents of the pseudo-XML's \texttt{text} element.
BeautifulSoup was used as an XML parser in this case, as lxml refused to parse the pseudo-XML.\(^3\) `unxml.py` also takes care of sentence tokenization, detailed later.

**Addition of information from TreeTagger (\*.tagged, \*.chunked)**

As previously mentioned, TreeTagger is able to output part-of-speech tag, lemma, and chunking information. For each article in the corpus, the raw text was fed into the tool and output into files named \*.tagged and \*.chunked. The former contain only word, tag and lemma information output by TreeTagger, separated by tab characters. The latter contain information about chunks present in the article, such as noun chunks, verb chunks and preposition chunks.

The chunk information had been computed with an aim to implementing an approach to computing verb arity similar to that of Manning (1993) [4]. Due to time constraints, however, this approach was not pursued.

The Corpus was tagged and chunked with the following command (assuming the Corpus is located in the current directory, and assuming TreeTagger is installed locally with the appropriate environment variables set):

```
Listing 3.1: Command to tag and chunk the Corpus

user@host:~/wp_corpus$ for i in `find . -name *.txt`; do
cat $i | tree-tagger-english > `dirname $i`/`basename $i .txt`.tagged;

```

\(^3\)BeautifulSoup attempts to be as forgiving as possible with malformed XML/HTML input, and will “guess” the correct document structure.
Compilation of Lemma Frequency Distributions (*.lemmas)

Lemma frequency distributions were computed with the script `lemmas.py`. This script reads the lemma information present in *.tagged files and outputs a frequency distribution of the article’s lemmas in descending order of frequency. These files end with the extension *.lemmas. These lemma frequency distributions were used to determine article quality.

Sentence tokenization

Sentence tokenization is the process of splitting up a text into individual sentences. One of the most popular methods for sentence boundary detection is the Punkt algorithm developed by Tibor Kiss and Jan Strunk. It is an unsupervised algorithm that has been shown to work with many European languages, and is therefore most attractive to computational linguists.

The articles in the Wikipedia corpus were not already tokenized into sentences. Walton (2009) performed sentence tokenization as a separate step in the generation of statistics, but the tokenized sentences were then discarded. As the Enju HPSG parser requires its input to have one sentence per line, it was decided that the articles should be sententially tokenized and saved in that form.

NLTK includes a `PunktSentenceTokenizer` class, complete with
a serialized already-trained instance of a `PunktSentenceTokenizer`. A text can be tokenized simply with the following Python code:

```python
>>> import nltk
>>> text = open(input_filename).read()
>>> sents = nltk.sent_tokenize(text)
>>> # Or, using the serialized object:
>>> tokenizer = nltk.data.load("/tokenizers/punkt/english.pickle")
>>> sents = tokenizer.sent_tokenize(sentences)
```

The tokenization process was done in tandem to the pseudo-XML text extraction process. Not all sentences were tokenized with 100% accuracy, though, causing problems which will be detailed in the next section.

**Running the Enju HPSG Parser on the Corpus (*.parsed.xml)**

In order to have a reference measure for the reliability of the naive approach to verb arity, it was decided to run a broad-coverage parser on the corpus. The Enju HPSG parser was selected on recommendation from a past student.

The Enju deliverable package contains two parsers: `enju` and `mogura`. `enju` is a high-accuracy parser that takes roughly 500 milliseconds to parse a sentence (dependant on sentence length). `mogura` is a lower-accuracy parser, but it is an order of magnitude faster than the `enju` parser, requiring roughly 50 ms to parse a sentence. Both the `enju` and `mogura` parsers can output the parse information in several different formats, such as that used by the Penn Treebank, and

---

4That is, the object and its internal state has been saved to disk for re-use.
XML. XML was chosen as the preferred output format due to the inherent ease in processing it.

Additionally, both parsers can be run as a local web service - that is, a sentence can be sent to the parser in a specially constructed URL, and the parse information will be returned. The author had intended to use this particular method to perform parsing. However, the parser tended to crash after a certain number of sentences had been processed.

The Wikipedia corpus contains roughly 7,000 articles, the mean article length being 1,825 words. The average sentence length is 22 words. Walton (2009) gives the total word count for the corpus as 12,656,396 words, and the total sentence count as being 540,107. At 500 ms per sentence (on average), parsing the corpus with the Enju parser would have taken approximately 3 days.

As explained previously, not all sentences were perfectly tokenized. This is to be expected in any corpus-based work. When an incorrectly tokenized sentence was fed to the Enju parser, it tended to either crash or take a large amount of time to process the sentence(s). This made the method of using the previously described local web service unsuitable.

Instead, a bash script (parse.sh) was written to iterate through the article .txt files, passing them to the parser via stdin, and redirecting the parser’s stdout to the proper filename. However, this approach meant starting the parser for every article. Given that both enju and mogura take about 20 seconds to begin accepting input, this approach took between one and two days longer to complete.
The parser was allotted however much time it required to complete its task.

3.1.3 Issues relating to the Corpus

As mentioned previously, the Wikipedia articles were converted from so-called wiki mark-up, a meta-language for which no formally defined schema exists, to the author’s knowledge. This conversion process was far from perfect; the tool used to extract the text from the mark-up “seems to go into an infinite loop on some articles with complex wiki mark-up” [9]. These articles were simply aborted; in any case, whatever the wiki mark-up parser managed to glean from the text remains.

Additionally, the wiki mark-up parser used was not always 100% effective. A glaring example is the History article in the BA-Corpus. A runaway ref tag caused the parser to generate invalid XML, and the process used in unxml.py to extract the text from the pseudo-XML failed in this case. It is unknown how many similar “glitches” have occurred. To alleviate this problem, the articles were filtered according to how closely their lemma-frequency distributions were approximated by a Zipfian distribution. This measure is detailed later.
3.2 COMLEX

The main issue with using COMLEX as an information source was its slightly obtuse format. COMLEX comes bundled with utilities, but these utilities require a proprietary version of Lisp (Allegro Common Lisp). Parsing the COMLEX dictionary without the aid of these utilities in order to map it to another representation (such as XML) proved a time-consuming task.

It turned out that another person had already solved the issue for a similar resource. His program, although tailored to work with the NOMLEX information source, required only trivial modifications to work with the COMLEX dictionary. Sadly, his implementation was discovered too late to be of any real use.

3.2.1 Parsing COMLEX

In order to extract the relevant subcategorization information present in COMLEX, a Python script `comlex.py` was written. It iteratively parses a COMLEX typed feature structure and converts it into a Python dictionary (essentially a hash map). This allows simple and immediate access to the verb subcategorization information. The complete information source is several thousand lines long, and cannot be distributed due to copyright issues.
### 3.2.2 Verb Frames and Framegroups

In COMLEX, a *verb frame* is a set of arguments that a verb accepts, in order of appearance. The simplest verb frame is possibly the **INTRANS** verb frame, indicating the verb is intransitive, or only takes one argument (the subject). The verb frames that a verb accepts are listed as values in the :SUBC feature of a verb’s entry in COMLEX. Each verb frame has a unique identifier such as **NP**, **INTRANS**, or other more esoteric labels such as **NP-P-ING-OC**. Almost all verb frames are specified in detail in the COMLEX reference manual [3].

To capture phenomena such as *dative shift* or any other permutations allowed with a verb’s arguments, a concept called a *framegroup* is used. Essentially, a framegroup consists of a number of verb frames whose identifiers are preceded with an asterisk (*). If a verb specifies a framegroup as a possible argument, any of the verb frames contained in the framegroup are permissible. An example framegroup along with its constituent verb frames is given below, extracted from the COMLEX reference manual.

#### Listing 3.2: Sample COMLEX Framegroup

```
(vp-frame *np-pp-pp :cs ((np 2) (pp 3 :pval ("")) (pp 4 :pval ("")))
  :gs (:subject 1 :obj 4 :obj2 2 :obj3 3)
  :ex "They carried the documents from the embassy to the airport")
(vp-frame *pp-pp-np :cs ((pp 2 :pval ("")) (pp 3 :pval ("")) (np 4))
  :gs (:subject 1 :obj 2 :obj2 3 :obj3 4)
  :ex "They carried from the embassy to the airport the documents ratified by the premier")
```

These verb frames and framegroups are specified in this format.
They were manually converted into a form understandable by Python, and are saved in the file `comlex_frames.py`. The arity was taken in all cases to be the number of items present in the `:gs` feature of the verb frame. This Python module has an internal facility to check whether all the verb frames referenced in COMLEX are known, to ensure that no typographical errors took place in the manual transcription process. At the time of writing, four verb frames are not specified in the COMLEX manual:

- Frame `AS-S-INF-OMIT` of the verbs “vote” and “react”
- Frame `PART-WH-S` of the verb “dream”
- Frame `P-NADVP-TIME` of the verb “begin”
- Frame `NUNITP-FROM-RANGE` of the verb “yield”

### 3.3 Project Gutenberg Encyclopedia

The Project Gutenberg Encyclopedia is simply a large text file. The first volume of this encyclopedia was used as a reference; other volumes have yet to be scanned and proofread - a non-trivial task! Some 1,700 articles are contained in this volume.

Lunney (2007) had previously performed a project that used Gutenberg as a corpus; one of the main issues encountered was automatically removing the header added by the Project Gutenberg

---

5 This file is not included in the appendix as it is rather large.

6 However, the frame *NUNITP-FROM-RANGE* exists; possibly a typographical error?
This header was simply removed by hand here, as there was only one large file to process.

### 3.3.1 Segmenting the Encyclopedia

The larger issue was segmenting this text file into the individual articles. The only obvious demarcation of where an article began and ended were two or more newline characters, and then the article title in capital letters. The interactive Python interpreter was used to:

- Construct a regular expression that split the text based on the above criteria without consuming any of the article title.
- Ensure that the regular expression had not been over-zealous in splitting the text into articles. Where this was the case, a simple ordering measure was used: if an apparent article title was contextually out of order, it was deemed to be an error, and simply appended to the previous article.
- As the apparent title of the article was not easy to discern due to irregularities in formatting, the articles were simply written to numbered files.

Once this task had been completed, similar steps were performed as on the Wikipedia Corpus:

- Sentence Tokenization

---

7 The header generally contains copyright (or lack thereof) and disclaimer information.
• TreeTagger tag and chunk information

• Lemma frequency distribution information

• Parsing the article text with the Enju parser

One interesting issue that arose with regard to sentence tokenization was that the Punkt sentence tokenizer had difficulty with quite a few features of the Project Gutenberg Encyclopedia’s text. For example, it found citations such as (Oxford, 1888), vol. i. p. 4. (P. GI.) extremely difficult to parse correctly. Additionally, abbreviations such as q.v., were not recognised correctly. As a fallback measure, another regular expression was used to split the article into individual sentences. A sentence boundary was considered to be a full stop preceding one uppercase letter followed by one or more lowercase letters. It does not, however, handle cases where a full stop inside quotation marks ends a sentence.

A more reliable method could possibly be to hand-segment a subset of the articles and to train the a Punkt sentence tokenizer on the hand-segmented data. This approach was deemed too time-consuming, and was therefore set aside in favour of the “quick and dirty” approach.
Chapter 4

Analysis Measures

4.1 Verb Arity as a measure of Relational Complexity

One can consider a verb as a relation between entities. A sentence can therefore be seen as a number of relations between various entities. One would think, then, that the number of relations in a sentence would have an effect on the overall complexity of a sentence. Below is given (partly) an example sentence from the Desert article of the Wikipedia Corpus.

Deserts can be defined as areas that receive an average annual precipitation of less than 250 mm.

In this sentence, there are a number of relations, both first-order and higher-order:

• can(deserts, be)
• be(deserts, areas)

• define(unknown, deserts)

• receive(areas, an average annual...)

Intuitively, the average verb arity for this sentence is 2, and the total verb arity is 8. The passivised version of the verb *define* is considered here as having two arguments, one of which is unspecified.

### 4.1.1 Naive Arity

The naive arity is defined here as the average number of arguments a verb takes. More formally:

\[
A_{naive}(V) = \frac{\sum_i^n A_{naive}(F_i)}{n}
\]

Where:

• \( V \) is an arbitrary verb,

• \( F \) is either a verb frame or a framegroup,

• \( n \) is the number of frames or framegroups accepted by \( V \),

• \( A_{naive}(F) = length(F) \) if \( F \) is a verb frame,

• \( A_{naive}(F) = \frac{\sum_i^n A_{naive}(F_i)}{n} \) if \( F \) is a framegroup.

That is, the naive arity metric takes the average of the numbers of items in each verb frame appearing in the subcategorisation list of the verb. Where an item in the subcategorisation list is a framegroup, the average arity of the framegroup is considered to be the average of
the number of items in each of the verb frames contained within the framegroup.

Take, for example, the verb walk. The :SUBC list for this verb in COMLEX is given below as:

Listing 4.1: COMLEX Entry for the verb walk

```
1 :SUBC (
    (ADVP−PP :PVAL ("to" "from"))
    (NP−ADVP)
    (PP−PP :PVAL ("up" "from" "into" "towards" "toward" "to"))
    (PART−PP :PVAL ("toward" "to" "on") :ADVAL ("in")))
    (NP)
    (INTRANS)
    (NP−PP :PVAL ("p−dir"))
    (PP :PVAL ("p−dir"))
)
```

Using the frame and framegroup arities computed from the information in the COMLEX reference manual, the average arity of the verb walk is therefore:

\[ A_{naive}(walk) = \frac{3 + 2 + 3 + 3 + 2 + 1 + \frac{3+3}{2} + 2}{8} = 2.375 \]

This process is implemented in the Python module `naive arity.py`. For the sake of speed, the arities for all the verbs present in COMLEX were precomputed and saved in the file `naive verb arity.py`, which, at the time of writing, contains 5186 verbs.

When analyzing articles, the average naive arity, standard deviation from the mean, and the number of verbs are recorded. When analyzing individual sentences, the average arity, the total arity of
the sentence (i.e. the total number of verb relations present), and the number of verbs are recorded.

4.1.2 Parsed Arity

This particular measure of verb arity is much more straightforward to compute - the parser has already made the argument-adjunct distinction. In the parse trees generated, all verbs are marked as tok elements with a category attribute of ‘V’. Each verb token has a number of argn attributes, n being counted from 1 to the maximum number of arguments admitted by the verb. The Python module parsed arity.py computes verb arity by executing the get() method of the verb element for increasing values of n, until None is returned. However, this measure of verb arity is only as good as the parser used.

Parsing Errors

Enju seems to have trouble with certain constructions in English. In particular, sentences with centre-embedding tend to be parsed incorrectly. Consider, for example, the sentence The dog the man walks drinks water. The parse tree for this sentence, shown in Figure 4.1, is incorrect - the verb drinks is incorrectly analyzed as a plural noun.

Even more worrying are the issues that crop up when analyzing real sentences. One would intuitively consider the style of writing of an
Figure 4.1: Incorrect parse for the sentence *The dog the man walks drinks water.*
encyclopedia published in 1911 to be more complex than the toy sentences generally employed by linguists as examples. The first sentence of the Gutenberg Encyclopedia article on the letter A reads:

\textit{This letter of ours corresponds to the first symbol in the Phoenician alphabet and in almost all its descendants.}

Again, the parse tree for this sentence is shown in Figure 4.2. The whole sentence is categorised by Enju as a noun phrase, and the main verb \textit{corresponds} is analyzed, again, as being a plural noun. This is especially worrying as the underlying grammatical construction is rather simple. In the sentential analysis, this problem is sidestepped by discarding any sentences that have no tokens categorised as verbs. In the per-article analysis, the parsing errors are hopefully balanced out by properly parsed sentences. Whether this is due to the usage of the lower-accuracy \textit{mogura} parser instead of the \textit{enju} parser is not known.
Figure 4.2: Incorrect parse for a sentence from the Gutenberg Encyclopedia.
4.2 Syntactic Complexity Measure

Sampson (1997), while investigating Yngve’s hypothesis about the depth of left-branching in the English language, devised a number of measures of left branching, or depth. For each of these measures of “depth”, the respective measure was computed against the SUSANNE Corpus.

Sampson’s production-based measure of depth was the only measure that, when computed for the SUSANNE Corpus, showed no correlation with sentence length. This measure of depth can be viewed as a measure of syntactic sentence complexity. The other measures defined were a depth-based measure of left-branching, and a realization-based measure of left-branching, and the weighted versions of the three measures.

Sampson (1997) defines the production-based measure of depth as

\[ \text{production-based measure of depth} = \frac{1}{n} \sum_{i=1}^{n} \left( \frac{d_i}{D} \right) \]

where \(d_i\) is the depth of the node, \(D\) is the depth of the tree, and \(n\) is the number of nodes.

\[ \text{production-based measure of depth} = \frac{1}{n} \sum_{i=1}^{n} \left( \frac{d_i}{D} \right) \]

\[ \text{production-based measure of depth} = \frac{1}{n} \sum_{i=1}^{n} \left( \frac{d_i}{D} \right) \]

In the domain of computer science, the depth of a terminal node in a tree would generally be considered as the distance of the node (in arcs) from the root node.

In the terminology used by Yngve (1960) and adopted by Sampson, the depth of a terminal node refers to the total amount of left-branching contained in the path linking the terminal node to the root node of a grammatical tree structure.
This measure of depth is implemented in the Python module `tree_depth.py`.

### 4.3 Article Quality Measure

In order to automatically filter out articles that were badly parsed from the wiki mark-up, an easily computed measure of article quality was required. Zipf’s Law is a well-known empirical law, formulated by the linguist George Kingsley Zipf (1935, 1949) [10] [11]. Zipf’s Law states that, given some corpus of natural language utterances, the frequency of any word in the corpus is inversely proportional to its rank in the frequency table. Put simply, in a corpus of natural language, there will be a few words that appear very frequently, and many words that appear very rarely. Given this fact, it was decided to use a Zipfian distribution as a means of determining an article’s quality. Quality, in this case, means that an article has been well-parsed.

Zipf’s Law is generally empirically demonstrated with word frequencies, but also holds true for lemma frequencies. When producing an idealized Zipfian distribution for an article, it is also sensitive to the length of the article. This is desirable, as abnormally short articles are definitely possible candidates for being marked as badly parsed.

The Python modules `zipf.py` and `zipf_deviance.py` were implemented to compute this quality metric. The `zipf.py` module has functionality to generate an idealised Zipfian distribution for a population of arbitrary size. The coefficient is assumed to be 1; it can be
changed with the optional \texttt{exp} argument to the \texttt{zipf\_dist()} function.

The Zipfian Deviance of a text is defined here as the sum of the absolute values of the differences between the expected (according to the idealised Zipfian distribution) and the observed frequencies, divided by the population size.

### 4.4 Textual Readability Tests

The readability tests documented and implemented in Walton (2009) were used as an additional measure of comparison. The tests implemented are:

- Flesch Reading Ease
- Automated Reading Index
- Flesch-Kincaid Grade Level
- Coleman Liau
- Gunning FOG Index
- SMOG (Simple Measure of Gobbledygook)
- Laesbarheds Index
- Linsear Write

Many of these tests require a text of 100 or more words on which to operate. As such, they cannot be included in the sentential comparison analysis stage.
Chapter 5

Results

5.1 Summary of Results Compared across Corpora

Average results per article for the various measures shown are summarised in table 5.1 below. In this table, BA stands for the Best Articles subcorpus, LS stands for the Lacking Sources subcorpus, SE stands for the Style Editing subcorpus, and PG stands for the Project Gutenberg Encyclopedia subcorpus. This is after filtering out results with a Zipfian deviation greater than 0.5.

Interesting to note is that, overwhelmingly, the verbs in all corpora tend to have two arguments. The naive estimation of arity, on average, matches the parsed measure rather well. The Zipfian measure of article quality seems to point towards the BA corpus being more similar to natural language. Interestingly, the measure of left-branching seems to point towards both the BA corpus and the PG
Table 5.1: Average Values per Article across Subcorpora

<table>
<thead>
<tr>
<th></th>
<th>BA</th>
<th>LS</th>
<th>SE</th>
<th>PG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Naive Arity</td>
<td>1.994</td>
<td>1.999</td>
<td>2.012</td>
<td>1.988</td>
</tr>
<tr>
<td>... Standard Deviation</td>
<td>0.2633</td>
<td>0.20525</td>
<td>0.2378</td>
<td>0.2418</td>
</tr>
<tr>
<td>Average Parsed Arity</td>
<td>1.917</td>
<td>1.916</td>
<td>1.911</td>
<td>1.917</td>
</tr>
<tr>
<td>... Standard Deviation</td>
<td>0.3906</td>
<td>0.3579</td>
<td>0.3792</td>
<td>0.3627</td>
</tr>
<tr>
<td>Average Left-Branching</td>
<td>0.19773</td>
<td>0.132</td>
<td>0.1586</td>
<td>0.2243</td>
</tr>
<tr>
<td>... Standard Deviation</td>
<td>0.07300</td>
<td>0.10195</td>
<td>0.0957</td>
<td>0.0162</td>
</tr>
<tr>
<td>Number of Verbs</td>
<td>588.6</td>
<td>62</td>
<td>122</td>
<td>106.1</td>
</tr>
<tr>
<td>Zipf Deviance</td>
<td>0.2635</td>
<td>0.3821</td>
<td>0.3568</td>
<td>0.3370</td>
</tr>
<tr>
<td>Article Length</td>
<td>4130</td>
<td>411.7</td>
<td>822.6</td>
<td>813.4</td>
</tr>
</tbody>
</table>

corpus being syntactically more complex.

At such an abstract level, however, one cannot make any significant judgements on these results. The main questions posed in the paper will be discussed presently.

5.2 Naive Approximation of Verb Arity

Firstly, the question as to whether verb arity can reliably be naively estimated needs to be addressed.

A scatterplot of the naive estimation of verb arity against the parsed measurement is shown in Figure 5.1. If the naive estimate of verb arity perfectly matched that of the parsed measurement, one
would expect to see a perfectly straight line. This is obviously not the case: the cross-shaped pattern on the graph centred at the point (2.0, 2.0) shows a broad area of disagreement between the parsed and naive measures.

Figure 5.2 shows a scatterplot of the naive estimation of verb arity against the parsed measurement per sentence. Again, broad disagreement is shown between the naive and parsed measurements. The large number of 0 values for the naive arity estimation can be explained by the decision to return 0 as the arity of a verb not present in COMLEX. Ignoring this artifact, however, still does not change the fact that there is no apparent linear agreement between the naive and parsed measures of arity. The null hypothesis, that there is no correlation between the naively obtained arities and the parsed values, cannot be rejected. Pearson’s product-moment correlation test shows small significant ($p < 2.2 \times 10^{-16}$) correlations for both per-article and per-sentence analyses, but these seem to be a reaction to the extreme non-linearity of the data.

5.3 Correlation between verb arity and left-branching

Another area to be explored is whether the reliable arity measurement has any relation to the measurement of left-branching. Otherwise stated: whether there is any correlation between relational complexity and syntactic complexity.
Figure 5.1: Naive Arity Estimate versus Parsed Measurement per Article
Figure 5.2: Naive Arity Estimate versus Parsed Measurement per Sentence
Figure 5.3 shows a scatterplot of the averages per article of verb arity versus the measure of left-branching, and figure 5.4 shows the same measures, but per sentence.

No linear relationships are immediately visible from either figure 5.3 or 5.4. Pearson’s product-moment correlation fails to provide a statistically significant result for the per-article analysis data. However, performing the correlation on the per-sentence values yields a significant result of there being no correlation between the two variables. It would seem that, according to these measures, syntactic and relational complexity are disjunct. The statement made carries only as much force as the reliability of the production-based measure of left-branching as a measure of syntactic complexity.

5.4 Correlations with textual readability tests

Pearson correlations were run between the textual readability tests and both parsed arity and left-branching measures. All statistically significant results are given in table 5.2. Based on these readings, it appears that the link between textual complexity and both syntactic and relational complexity is quite tenuous, if not nonexistent. However, too many of the correlation tests were insignificant to license any stronger statements on the matter.
Figure 5.3: Parsed Arity Measurement versus Production-based measure of left-branching per article
Figure 5.4: Parsed Arity Measurement versus Production-based measure of left-branching per sentence
Table 5.2: Correlations of verb arity and left-branching with various readability measures

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arity / FRE</td>
<td>0.0213</td>
<td>-0.0257</td>
</tr>
<tr>
<td>Left-branching / ARI</td>
<td>1.383 × 10^{-05}</td>
<td>0.048</td>
</tr>
<tr>
<td>Left-branching / FOG</td>
<td>1.829 × 10^{-06}</td>
<td>-0.0532</td>
</tr>
<tr>
<td>Left-branching / FRE</td>
<td>9.66 × 10^{-08}</td>
<td>0.059</td>
</tr>
</tbody>
</table>

5.5 Correlation between left-branching measure and sentence length

The reason Sampson’s raw production-based measure of depth was used as a measure of syntactic complexity was because it was shown to be invariant on sentence length \[5\]. However, to err on the side of caution, the values recorded for the measure of left-branching were correlated against sentence length. This proved to be a valuable, if somewhat disturbing, safety check. Both figure 5.5 and Pearson’s product-moment coefficient test show a medium-size \(r = 0.426\) positive correlation \(p < 2.2 \times 10^{-16}\) between the measure of left-branching and sentence length.

The figure shows a sentence length cutoff of 100 – the parser places a hard limit on the length of a sentence. Additionally, to better represent the data, only values between 0.1 and 0.25 for the left-branching measure were graphed. The relationship seems to be
more nonlinear than linear.

There are a number of possible explanations for this anomaly. The first is that the left-branching measure was simply implemented incorrectly. Sampson’s explanation of the raw production-based measure of left-branching was rather brief; possibly allowing for misinterpretation.

Another possibility is that this particular measure of left-branching was not suitable for the parse trees generated by Enju, which are overwhelmingly binary branching. However, Sampson does state:

\[
\ldots \text{“I do not myself believe that grammatical branching is always binary. I am proposing that we count word depth in a way that gives the same results whether that is so or not.”} \] 5

Without launching into a rigorous mathematical proof of Sampson’s production-based measure of left-branching being agnostic to binary branching, it would seem extremely unlikely that Sampson would make such a statement, and then define a measure that weights binary branching tree structures more than co-ordinations in tree structures.

It is possible, that, due to internal mechanisms of the Enju parser that are not immediately apparent, longer sentences necessitate a greater degree of left-branching in the parsed output. Such an investigation lies far outside the scope of this project, in any case.
Figure 5.5: Production-based measure of left branching versus sentence length
5.6 Correlation between verb arity and sentence length

Figure 5.6 shows sentential average verb arity plotted against sentence length. The plot and a Pearson correlation coefficient of 0.038 with

Figure 5.6: Sentential verb arity versus sentence length
\( p < 2.2 \times 10^{-16} \) both point to there being no linear correlation between the average number of arguments taken by verbs and the sentence length.
Chapter 6

Conclusions

Many of the basic questions of this paper remain unanswered, sadly. Some surprising results (and some much less so) emerged however, not least of which was the correlation between sentence length and the measure of syntactic complexity meant to be sentence length agnostic. Whether the implementation of Sampson’s production-based measure of tree depth can still be considered a valid measure of syntactic complexity for trees produced by the Enju HPSG parser remains yet to be seen.

The result of syntactic and relational complexity being possibly disjunct is also surprising. That there is no correlation between the length of a sentence and the average verb arity is also not surprising, as are the lack of correlations between relational and syntactic complexity.

Additionally, it was by no means unexpected that naively esti-

---

1I say possibly, as the reliability of the syntactic complexity measure has fallen into question for reasons stated earlier.
mating verb arity proved to be problematic and inaccurate. The examples of “bomb” sentences that can cause parsing errors cause immediate concern for alarm, as this fact directly undermines the quality metric used to evaluate the naive measurement of verb arity, and, additionally, the ultimate measure of relational complexity. A successive iteration of a similar project would be advised to use a higher quality parser, or, better yet, a hand-tagged corpus.

Finally, the result of “about 2” for the overall average arity of a corpus consisting of over 14 million words is mildly surprising, but not completely unexpected when one considers that the most common grammatical construction in English is *Subject Verb Object*. 
Appendices
Appendix A

Code Listings

Resources and supporting source code are available online at the following address:

http://ducss.cs.tcd.ie/~johnstc/FYP/

A.1 unxml.py

This script reads a pseudo-XML file as its first command-line argument, and writes the sententially tokenized text to a file named after its second argument.

```
#!/usr/bin/env python
#
# coding: utf-8
#
import os
import sys
import BeautifulSoup
import codecs
import nltk.tokenize

/home/johnstc/Documents/FYP/wikipedia/bin/unxml.py
```

49
def unxml(stuff):
    """
    Returns the contents of the <text> tag in stuff
    """
    soup = BeautifulSoup.BeautifulSoup(stuff)
    return "\n".join(tok.tokenize(soup.text.contents[0]))

def unxml_file(filename):
    stuff = open(filename).read()
    return unxml(stuff)

def main():
    if len(sys.argv) < 3:
        print("Usage: unxml.py <infile.xml> <outfile.txt>"
    else:
        infilename, outfilename = sys.argv[1:3]
        f = codecs.open(infilename, "r", "utf-8")
        stuff = f.read()
        f.close()
        try:
            text = unxml(stuff)
            f = codecs.open(outfilename, "w+", "utf-8")
            f.write(text)
            f.close()
        except BeautifulSoup.HTMLParseError:
            print("Something went wrong with file %s" % infilename)

if __name__ == "__main__":
    main()
A.2 lemmas.py

This Python script reads in TreeTagger output from a .tagged file and outputs a lemma frequency distribution for the file.

```
#!/usr/bin/env python
#coding:utf8
import util
import nltk
import sys

def preprocess_article(article_filename):
    article_file = open(article_filename)
    tuples = []
    for line in article_file:
        tuples.append(tuple(line.strip().split()))
    return tuples

def extract_vocab(tuples):
    lemmas = []
    for tup in tuples:
        try:
            (word, tag, lemma) = tup
            if lemma != "<unknown>":
                lemmas.append(lemma)
        except ValueError:
            continue
    return nltk.FreqDist(lemmas)

def write_results(freqs, output_filename):
    outfile = open(output_filename, "w+")
    for sample in freqs:  # Sorted by default, thanks nltk.thultr.
        outfile.write("%s,%s\n"%(sample, freqs[sample]))
    outfile.close()
```
def make_outfilename(input_filename, ext):
    base = input_filename[:len(ext)]
    return base + " . lemmas"

def main():
    # No arguments, batch mode
    if len(sys.argv) == 1:
        ext = " . tagged"
        print("Gathering files ...")
        filenames = util.files_with_suffix(util.CORPUSPATH, ext)
        n = len(filenames)
        for (i, filename) in enumerate(filenames):
            print("PROCESSING (%d/%d): %s" %(i+1, n, filename))
            tuples = preprocess_article(filename)
            freqs = extract_vocab(tuples)
            outfilename = make_outfilename(filename, ext)
            write_results(freqs, outfilename)
            print("WRITING (%d/%d) : %s" %(i+1, n, outfilename))
    elif len(sys.argv) == 2:
        # Batch mode on a directory
        ext = " . tagged"
        print("Gathering files ...")
        filenames = util.files_with_suffix(sys.argv[1], ext)
        n = len(filenames)
        for (i, filename) in enumerate(filenames):
            print("PROCESSING (%d/%d): %s" %(i+1, n, filename))
            tuples = preprocess_article(filename)
            freqs = extract_vocab(tuples)
            outfilename = make_outfilename(filename, ext)
            write_results(freqs, outfilename)
    elif len(sys.argv) == 3:
        print("Working ... "),
        input_filename = sys.argv[1]
        output_filename = sys.argv[2]
        tuples = preprocess_article(input_filename)
        freqs = extract_vocab(tuples)
        write_results(freqs, output_filename)
        print("done.")
A.3 parse.sh

This simple bash script was ultimately used to parse the articles one-by-one.

```bash
#!/usr/bin/env bash

FILES=$(find . -name *.txt | grep -v markup)

for FILE in $FILES; do
  OUTFILE='dirname $FILE ' / ' basename $FILE . txt ' . parsed . xml ;
  if [[ ! -e $OUTFILE ]]; then
    echo PARSING: $FILE ;
    cat $FILE | sed ' /$/d ' | mogura -xml 2>/dev/null > $OUTFILE ;
    echo WROTE: $FILE ;
  else
    echo SKIPPING: $FILE
  fi
done
```

A.4 util.py

This Python module contains methods that the author found were being duplicated unnecessarily and saw fit to consolidate in one place. Its main function is to run the `process_article` method of an instance of a `StyleCalculator` object on every article in the Corpus. Additionally, it serves to automatically save the results obtained.
#!/usr/bin/env python
#
# coding: utf-8
#
""
Helper utilities for arity parsing
Cian Johnston, 2009–2010
""

from __future__ import division
import math
import os
import urllib
import csv

CORPUSPATH = os.getenv("WP_CORPUS")

def files_with_suffix(dirname, suffix, ignore=lambda x: True):
    ""
    Returns a list consisting of all filenames ending in suffix
    contained
    in dirname. Optional argument ignore can be used to filter out
    unwanted results similar to grep -v.
    ""
    walk = os.walk(dirname)
    files = []
    for tup in walk:
        for fname in tup[2]:
            if fname.endswith(suffix):
                files.append(os.path.join(tup[0], fname))
    return filter(ignore, files)

def corpus_dirs(corpuspath):
    ""
    Returns the directories (or subcorpora) residing in the corpus
    root
    directory.
    ""
    return [os.path.join(corpuspath, d) for d in os.listdir(}
def fix_name(article_filename, ext):
    """
    Prettifies the filename of an article, given its filename extension
    """
    >>> fix_name("This is a really long filename.txt", ".txt")
    "This is a (really) long filename"
    ""
    # Remove extension
    string = article_filename.replace(ext,"")
    # Replace underscores
    string = string.replace("_", " ")
    # Convert entities
    string = urllib.unquote(string)
    return string

def default_prompt(prompt, default):
    """
    Prompts the user for a string, returning a default value if nothing is entered.
    """
    string = raw_input(prompt)
    if string:
        return string
    else:
        print "Defaulting to %s" % (default)
        return default

def process_subcorpora(styc, def_fname):
    """
    Runs the style calculator on all files in all subcorpora contained in CORPUSPATH
    """
    corpuspath = default_prompt("Enter path to corpus: ", CORPUSPATH)
ext = styc.preferred_ext
res_file = open(default_prompt("Enter results filename: ",
def fname), \
"w+")
res_wrtr = csv.writer(res_file, csv.QUOTE_ALL)
res_wrtr.writerow(['Subcorpus', 'Article'] + styc.header)
for subcorp in corpus_dirs(corpuspath):
    sc_name = os.path.basename(subcorp)
    print("Current subcorpus: %s" % (sc_name))
    files = files_with_suffix(subcorp, ext)
    total = len(files)
    for num, article_path in enumerate(files):
        article_filename = os.path.basename(article_path)
        article_name = fix_name(article_filename, ext)
        print("Processing (%d of %d): %s" % (num + 1, total, \
              article_filename))
        try:
            # Let the Style Calculator do its thing
            results = styc.process_article(article_path)
            # Some articles don’t have any verbs in them...
            if len(results) > 0:
                res_wrtr.writerow([sc_name, article_name] + results)
            else:
                print("Empty: %s" % (article_path))
        except KeyboardInterrupt:
            break
        except ZeroDivisionError:
            # I guess this means the article is sorta empty
            print("Error: %s" % (article_path))
            continue
res_file.close()
print("Done.")

def process_one(styc, in_filename, out_filename):
    results = styc.process_article(in_filename)
    print("Working...")
    with open(out_filename, "w+") as outfile:
        res_wrtr = csv.writer(outfile)
        res_wrtr.writerow(['Subcorpus', 'Article'] + styc.header)
This Python module acts as a frontend to performing a test on a corpus. It takes the name of a module containing a StyleCalculator subclass to use as an argument. It then attempts to import the module, instantiates the StyleCalculator and runs the StyleCalculator's process_article method on each relevant file in the Corpus.

```
#!/usr/bin/env python
#
#coding: utf-8
#
"""Main frontend for analysing arity.
Cian Johnston, 2009–2010
"""
import os
import sys
import util

def main():
    res_fname = lambda acname : os.path.join(util.CORPUSPATH, "results-%s.csv" %acname)
    if len(sys.argv) < 2:
        print("Usage: %s <arity_calculator_class>") % sys.argv[0])
```
else:
    try:
        arity_calc = __import__(sys.argv[1]).get_style_calculator()
        ac_name = sys.argv[1].split(".")[−1]
        util.process_subcorpora(arity_calc, res fname(ac_name))
    except ImportError:
        print(("Cannot import %s" %sys.argv[1]))

if __name__ == ".main."
    main()
length = len(iterable)
return sum(iterable) / length if length > 0 else 0

def std_dev(self, iterable, avg=None):
    """
    Returns the standard deviation of iterable from the mean
    >>> std_dev([1,2,3], 2.0)
    0.81649658092772603
    """
    if avg is None:
        avg = self.average(iterable)
    return math.sqrt((sum([ (i - avg)**2 for i in iterable ]) / \
                      len(iterable)))

def process_results(self, iterable):
    """
    Returns the average, standard deviation, and N of an iterable
    >>> process_results([1,2,3])
    (2.0, 0.81649658092772603, 3)
    """
    avg = self.average(iterable)
    std = self.std_dev(iterable, avg=avg)
    num = len(iterable)
    return [ avg, std, num ]

A.7 naive arity.py

Naively estimates verb arity using COMLEX as an information source.

/home/johnstc/Documents/FYP/wikipedia/bin/naive arity.py

import comlex
import math
import pprint
import style_calc

class NaiveArityCalculator(style_calc.StyleCalculator):
    def __init__(self, frames=None, verbs=None):
        self.complex = complex.complex
        self.memoizedframes = frames if frames is not None else {}
        self.memoizedverbs = verbs if verbs is not None else {}
        self.preferred_ext = "tagged"
        self.header = ["Naive Average", "Naive Standard Deviation",
                       "Naive N"]

    def arity(self, verb):
        try:
            return self.avg_verb_arity(verb)
        except ValueError:
            return None

    def arities(self, verbs):
        return list(filter(lambda x: x is not None, map(self.arity, verbs)))

    def process_article(self, article_filename):
        article_file = open(article_filename)
        verb_lemmas = self.preprocess(article_file)
        results = self.arities(verb_lemmas)
        return self.process_results(results)

    def preprocess(self, article_file):
        tuples = [tuple(line.strip().split()) for line in article_file]
        tuples = [t for t in tuples if isinstance(t, tuple) and len(t) == 3]
        verb_lemmas = []
        for word, tag, lemma in tuples:
            assert isinstance(tag, str), "tag %s isn’t a string for some reason" % (tag)
            if tag.startswith("V"): 
verb.lemmas.append(lemma)

return verb.lemmas

def avg_frame arity(self, frame):
    """
    Returns the average arity of the given frame.
    Raises a ValueError if the frame is unknown.
    """
    try:
        return self.memoizedframes[frame]
    except KeyError:
        self.memoizedframes[frame] = self._avg_frame arity(frame)
    return self.memoizedframes[frame]

def _avg_frame arity(self, frame):
    if frame in comlex.FRAMEGROUPS:
        # Get the average arity of the framegroup
        arities = []
        for f in comlex.FRAMEGROUPS[frame]:
            arities.append(self.avg_frame arity(f))
        return (sum(arities) / len(arities))
    elif frame in comlex.FRAMES:
        return (comlex.FRAMES[frame]["ARITY"])  
    else:
        raise ValueError("Frame %s not in COMLEX!" % (frame))

def _get frames(self, verb):
    frames = []
    for f in comlex.comlex[verb]["VERB"]["SUBC"]:
        if type(f).__name__ == "str":
            frames.append(f)
        elif type(f).__name__ == "dict":
            frames.append(f["TYPE"])  
    return(frames)

def avg_verb arity(self, verb):
    """
    Returns the average arity of all the possible frames of the
    verb.
    """

    verb.lemmas.append(lemma)
    return verb.lemmas
Raises a ValueError if the verb is unknown.

```python
try:
    return self.memoizedverbs[verb]
except KeyError:
    self.memoizedverbs[verb] = self._avg_verb arity(verb)
    return self.memoizedverbs[verb]

def _avg_verb arity(self, verb):
    try:
        frames = self._get_frames(verb)
    except KeyError:
        # Verb isn't known
        raise ValueError("Verb not in COMLEX!")
    arities = []
    for frame in frames:
        try:
            arities.append(self.avg_frame arity(frame))
        except ValueError:
            continue
    return (sum(arities) / len(arities))

def _sanity_check(self, verb):
    # First off, check if the verb is actually covered in comlex
    assert self.comlex.comlex.get(verb) is not None, "%s is not
    " recognised \n    as a word in comlex!" %(verb)
    assert self.comlex.comlex[verb].get("VERB") is not None, "%s
    is in \n    comlex, but is not recognised as a verb!" %(verb)
    # Secondly, make sure all the frames comlex says the verb has
    # are actually frames in comlex – workaround for badly parsed
    lisp!
    frames = comlex.comlex[verb]["VERB"][:SUBC]
    for frame in frames:
        assert frame in list(comlex.FRAMES.keys()), "frame %s of
        verb %s is not \n        in comlex.FRAMES!" %(frame, verb)
    assert frame in list(comlex.FRAMEGROUPS.keys()), "frame %
```python
def frames_in_comlex(self, verb):
    frames = comlex.comlex[verb]["VERB"]["SUBC"]
    for frame in frames:
        if frame not in list(comlex.FRAMES.keys()) or frame not in list(comlex.FRAMEGROUPS.keys()):
            return False
    return True

def save_precomputed():
    frames = precompute_naive_frame_arities()
    verbs = precompute_naive_verb_arities()

default_frame_filename = "naive.frame arity.py"
frame_filename = input("Enter location to save frame arities \n((Default: %s): " %(default_frame_filename))
if not frame_filename:
    frame_filename = default_frame_filename

default_verb_filename = "naive.verb arity.py"
verb_filename = input("Enter location to save verb arities \n(DefaultValue: %s): " %(default_verb_filename))
if not verb_filename:
    verb_filename = default_verb_filename

startstr = "#!/usr/bin/env python\n# coding: utf-8 -- all-- \n\n"

framefile = open(default_frame_filename, "w+")
framestr = startstr + pprint.pformat(frames)
framefile.write(framestr)
framefile.close()

verbfile = open(default_verb_filename, "w+")
verbstr = startstr + pprint.pformat(verbs)
verbfile.write(verbstr)
```

63
def precompute_naive_verb_aries():
    verbs = (key for key in list(comlex.comlex.keys()) if comlex.comlex[key].get("VERB") is not None)
    ac = NaiveArityCalculator()
    verb_aries = {}
    for verb in verbs:
        verb_aries[verb] = ac.avg_verb_arity(verb)
    return verb_aries

def precompute_naive_frame_aries():
    ac = NaiveArityCalculator()
    frame_aries = {}
    for frame in list(comlex.FRAMES.keys()):
        try:
            frame_aries[frame] = ac.avg_frame_arity(frame)
        except ValueError as e:
            print(e)
            continue
    for framegroup in list(comlex.FRAMEGROUPS.keys()):
        frame_aries[framegroup] = ac.avg_frame_arity(framegroup)
    return frame_aries

def get_style_calculator():
    return NaiveArityCalculator()

if __name__ == "__main__": save_precomputed()

A.8 parsed_arity.py

Computes the arity of articles parsed by the Enju HPSG parser.

#!/usr/bin/env python
#coding:utf8
from style_calc import StyleCalculator
import lxml.etree as ET
import itertools

import style_calc

class ParsedArityCalculator(StyleCalculator):
    def __init__(self):
        self.preferred_ext = "parsed.xml"
        self.header = ["Parsed Average", "Parsed Standard Deviation",
                       "Parsed Max", "Parsed N"]

        def arity(self, verb_token):
            l = lambda x: x is not None
            g = (verb_token.get("arg%d" % i) for i in itertools.count(1))
            return len(list(itertools.takewhile(l, g)))

        def arities(self, verb_tokens):
            return list(map(self.arity, verb_tokens))

        def process_article(self, article_filename):
            try:
                article_file = open(article_filename)
                article_tree = self.preprocess(article_file)
                verb_tokens = article_tree.findall("//tok[@cat='V']")
                article_file.close()
                results = self.process_results(verb_tokens)
                parsed_avg, parsed_stdev, parsed_max, parsed_n = self.process_results(results)
                parsed_max = max(results)
                return [parsed_avg, parsed_stdev, parsed_max, parsed_n]
            except ET.XMLSyntaxError:
                return []

        def process_sent(self, sent):
            verbs = sent.xpath("//tok[@cat='V']")
            arities = self.arities(verbs)
            return arities

65
A.9  zipf_deviance.py

Computes an idealized Zipfian distribution for an article and measures the magnitude of its deviance from said ideal.

```python
#!/usr/bin/env python
#coding=utf8
import util
import style_calc
import zipf
from operator import itemgetter

class ZipfDevianceCalc(style_calc.StyleCalculator):
    def __init__(self):
        self.header = [ "Zipf Deviance" ]
        self.preferred_ext = ".lemmas"

    def read_freqs(self, lemmafile):
        """
        Returns a list consisting of all the lemmas contained in a .lemmas file
        """
        freqs = []
        for line in lemmafile:
            try:
                # We're only interested in the frequencies
                int_str = line.strip().split(",")[-1]
                freqs.append(int(int_str))
```

except IndexError:
    continue

return freqs

def dist_deviance(self, article_dist):
    """
    Compares the lemma frequency distribution of an article to
    that of an idealized Zipfian distribution
    """

    pop_size = sum(article_dist)
    # TODO: better comparison function?
    fcmp = lambda (x, y): abs(x - y)
    gold_dist = zipf.zipf_dist(pop_size)
    devs = map(fcmp, zip(article_dist, gold_dist))

    try:
        # Weighting by number of words
        result = sum(devs) / pop_size
        return result
    except ZeroDivisionError:
        return None

def process_article(self, article_filename):
    lemmafile = open(article_filename)
    freqs = self.read_freqs(lemmafile)
    result = self.dist_deviance(freqs)
    return [result]

def get_style_calculator():
    return ZipfDevianceCalc()
#!/usr/bin/env python
# coding: utf-8

from memoize import memoized

@memoized
def H(n, m=1):
    """
    Returns the nth generalized harmonic number (m=1 by default)
    Memoized for efficiency.
    """
    return sum([1.0 / k ** m for k in range(1, n+1)])

def zipf_single(rank, pop_size, exp=1):
    """
    Returns the expected number of occurrences for an element of rank
    @rank in
    a population of size @pop_size.
    """
    denom = (rank ** exp) * H(pop_size, exp)
    prob = 1.0 / denom
    return pop_size * prob

def zipf_dist(pop_size, exp=1):
    """
    Returns a zipf distribution for a population of size @pop_size.
    @exp defaults to 1.
    """
    """
A.11 readability.py

This Python module is simply a backported version of part of Walton (2009)’s statistics generator, implementing a number of textual readability measures.

```python
from future import division
import style_calc
import hyphenate
import nltk
import math
import re
from memoize import memoized

class ReadabilityChecker(style_calc.StyleCalculator):
    def __init__(self):
        pass

return [zipf_single(i, pop_size, exp) for i in range(1, pop_size + 1)]
```

/home/johnstc/Documents/FYP/wikipedia/bin/readability.py
```python
self.preferred_ext = ".txt"
self.hyphenator = hyphenate.Hyphenator(hyphenate.patterns, \
hyphenate.exceptions)

def unicode_dammit(self, word):
    return word.decode("utf-8")

def hyphenate(self, word):
    #u_word = self.unicode_dammit(word)
    return self.hyphenator.hyphenate_word(word)

def clean(self, text):
    f = lambda x: x not in "?!;:,() {}\"
    return filter(f, text)

def preprocess(self, article_text):
    # Calculate sentence lengths
    sents = nltk.tokenize.sent_tokenize(article_text)
    sentence_count = len(sents)
    # Count number of words and long words
    words = sum([nltk.word_tokenize(self.clean(sent)) for sent
    in sents], [[]])
    word_count = len(words)
    long_words = len([w for w in words if len(w) > 6])
    # Count number of characters, excluding spaces obviously
    char_count = len(filter(str.isalnum, words))
    # Calculate syllable-related measures
    word_syllables = [self.hyphenate(w) for w in words]
    polysyllable_count = len([w for w in word_syllables if len(w)
    ) > 1])
    syllable_count = len(sum(word_syllables, [[]]))

    return (char_count, syllable_count, word_count,
    sentence_count, \
    polysyllable_count, long_words)
```
```python
def process_article(self, article_filename):
    with open(article_filename) as f:
        text = f.read()
        char_count, syllable_count, word_count, sentence_count, \
        polysyllable_count, long_words = self.preprocess(text)
        ari = self.automated_reading_index(word_count, \
            sentence_count, char_count)
        coli = self.coleman_liu(word_count, sentence_count, \
            char_count)
        fgl = self.flesch_grade_lvl(syllable_count, word_count, \
            sentence_count)
        fre = self.flesch_reading_ease(syllable_count, word_count, \
            sentence_count)
        fog = self.gunning_fog(word_count, sentence_count, \
            polysyllable_count)
        lix = self.laesbarheds_index(word_count, sentence_count, \
            long_words)
        lsw = self.linear_write(word_count, sentence_count, \
            polysyllable_count)
        smog = self.simple_measure_of_gobbledygook(sentence_count, \
            polysyllable_count)
        return [ari, coli, fgl, fre, fog, lix, lsw, smog, \
            char_count, polysyllable_count, word_count, \
            long_words, sentence_count,]

def flesch_reading_ease(self, syllable_count, word_count, 
    sentence_count):
    """Flesch Reading Ease""
    return 206.835 - (syllable_count * 84.6 / (word_count)) - \
        (word_count * 1.015 / (sentence_count))

def automated_reading_index(self, word_count, sentence_count, 
    char_count):
    """Automated Reading Index (ARI)""
```

from __future__ import division

def flesch_grade_lvl(self, syllable_count, word_count, sentence_count):
    """
    Flesch-Kincaid Grade Level
    """
    return (0.39 * word_count / float(sentence_count)) + (11.8 * syllable_count / word_count) - 15.59

def coleman_liau(self, word_count, sentence_count, char_count):
    """
    Coleman Liau
    \'A computer readability formula designed for machine scoring\'
    """
    return (5.89 * char_count / float(word_count)) - (0.3 * float(sentence_count) / float(word_count)) - 15.8

def gunning_fog(self, word_count, sentence_count, polysyllable_count):
    """
    Gunning Fog, will change when I find Gunning original
    """
    return 0.4 * ((float(word_count) / float(sentence_count)) + 100.0 * (float(polysyllable_count) / float(word_count)))

def simple_measure_of_gobbledygook(self, sentence_count, polysyllable_count):
    """
    Simple Measure Of Gobbledygook
    """
    return 1.043 * math.sqrt(polysyllable_count * 30.0 / (sentence_count)) + 3.1291
long_words):
    ""
Laesbarhedsindex (Any language)
    ""
    return (word_count/float(sentence_count)) + ((100. *
    long_words) / \
    float(word_count))

def linsear_write(self, word_count, sentence_count, polysyllable_count):
    ""
Linsear Write
    ""
    hardwords = polysyllable_count
    easywords = word_count - hardwords
    hardwords *= 3
    r = hardwords+easywords
    r /= sentence_count
    if r > 20:
        r /= 2
    elif r <= 20:
        r -= 2
        r /= 2
    return r

def get_style_calculator():
    return ReadabilityChecker()

A.12 tree_depth.py

This Python module implements Sampson (1997)’s production-based
measure of tree depth.

/home/johnstc/Documents/FYP/wikipedia/bin/tree_depth.py

#!/usr/bin/env python
#coding: utf8

73
from __future__ import division
import lxml.etree as ET
import style_calc

class ProductionDepthAnalyzer(style_calc.StyleCalculator):
    def __init__(self):
        self.preferred_ext = ".parsed.xml"
        self.header = ["Sentence RPD Average", "Sentence RPD Standard Deviation", "Sentence RPD N"]

    def article_raw_production_depth(self, tree):
        sent = tree.xpath("/article/sentence")
        sent_avgs = []
        for sent in sent:
            sent_avgs.append(self.sent_raw_production_depth(sent))
        avg = self.average(sent_avgs)
        stdev = self.std_dev(sent_avgs, avg=avg)
        return [avg, stdev, len(sent_avgs)]

    def sent_raw_production_depth(self, sent):
        # Cons elements are nonterminals
        nts = sent.xpath(".//cons")
        results = []
        for nt in nts:
            results.append(self.raw_production_depth(nt))
        return self.average(results)

    def raw_production_depth(self, nt):
        dtrs = nt.getchildren()
        assert len(dtrs) != 0, "Node %s has no children!" % (tree.getpath(nt))
        result = 0
        for dtr in dtrs:
            # If the daughter is nonterminal
            if dtr.tag == "cons":  
                # And if the daughter is not the rightmost
                if not self.is_rightmost(dtr):
                    result += 1
        return result / len(dtrs)
A.13 parse_comlex.py

This script was used to parse the COMLEX data file. Additional manual post-editing was necessary to ensure correctness; this does the bulk of the task.
```python
findall: str -> [str...]
Returns all non-overlapping occurrences of sub in string

i = string.find(sub, offset)
while i >= 0:
    listindex.append(i)
    i = string.find(sub, i + 1)
return listindex

def find_bracket_exprs(string):
    """
    findbracketexprs: string -> [(int,int)...]
    Returns the indices of all top-level bracketed expressions in
    string.
    ""
    results = []
i = 0
    # initial sanity check
    # commented out because sanity causes insanity
    # in cases such as "(WORD :ORTH "" :POS "NONE")"
    #if string.count("(") != string.count(")"):
        # raise ValueError("Unmatched brackets in string")
    while i < len(string):
        if string[i] == ":" :
            end = find_matching_bracket(string, i)
            if end is not None:
                results.append((i, end))
                i = end + 1
        i += 1
    return results

def find_matching_bracket(string, start):
    """
    find_matching_bracket: str -> int
    Finds the index of the closing bracket matching the bracket
    specified
    at index start. Ignores brackets enclosed in quoted strings.
    Raises a ValueError if string[start] is not an opening bracket.
    ""
```
brackets = 0
result = None
if string[start] == "(":
    brackets += 1
    i = start + 1
    quotes = False
while((brackets != 0) and i < len(string)):
    c = string[i]
    if c == "\":
        quotes = not quotes
else:
    raise ValueError("No bracket found at index %d" % (start))
return result
def is_bracket_expr(expr):
    ""
    is_bracket_expr: str → bool
    ""
    return (expr.startswith("(") and expr.endswith(")"))
def is_atom(expr):
    ""
    is_atom: str → bool
    ""
    return is_bracket_expr(expr) and expr[1:−1].isalpha()
def is_bracket_list(expr):
    ""
    is_bracket_list: str → bool
```python
    return is_bracket_expr(expr) and not is_tfs(expr)

def is_tfs(expr):
    if is_bracket_expr(expr):
        expr = expr[1:-1]
        stuff = expr.split()
        if stuff[0].isupper() and stuff[1].startswith('"':
            return True
        else:
            return False
    else:
        return False

def tokenize_expr(expr):
    ""
    tokenize_expr: str -> [list...]
    ""
    tokens = []
    expr = expr[1:-1]
    expr = expr.strip()
    bracket_exprs = find_bracket_exprs(expr)
    if bracket_exprs:
        dontskip = [0] + sum([list(t) for t in find_bracket_exprs(
            expr)], [[]] + [-1])
        idxes = list(zip(dontskip[:2], dontskip[1:2], dontskip[:2]))
        # Yet another ugly hack.
        idxes = [idxes[0]] + [(a + 1, b, c) for (a, b, c) in idxes
            [1:]]
        for (start, bracketstart, bracketend) in idxes:
            atoms = expr[start:bracketstart].strip().split()
            for atom in atoms:
                tokens.append(atom.strip(" "))
                # +1 because slices are silly
                bracketexpr = expr[bracketstart:bracketend + 1]
                # omg recursion
                tokens.append(tokenize_expr(bracketexpr))
        return tokens
```

else:
    return [ e.strip("\n\") for e in expr.split() ]
#return expr.split()
return tokens

def expclist_todict(exprlist):
    # we don't want anything like [[FOOBAR]]
    if type(exprlist).__name__ == "str":
        return exprlist
    elif type(exprlist).__name__ == "list" and len(exprlist) == 1:
        return exprlist[0]
    else:
        # If it's a feature structure, do what needs to be done
        if any([x.startswith(":" if hasattr(x, "startswith") else False for x in exprlist]):
            features = [ x for x in exprlist if hasattr(x, "startswith") and x.startswith(":" ) ]
            values = [ exprlist[exprlist.index(f) + 1] for f in features ]
            for i in range(len(values)):
                if type(values[i]).__name__ == "list":
                    values[i] = expclist_todict(values[i])
            return dict([("TYPE", exprlist[0])] + list(zip(features, values)))
    else:
        return [ expclist_todict(expr) for expr in exprlist ]

def read_exprs(f):
    ""
    read_exprs: File -> [str...]
    Reads all s-expressions from file f
    """
IGNORE = "\n\t\r"
# make sure the file is at starting position
f.seek(0)
everything = ".".join([c for c in f.read() if c not in IGNORE])
# leave things as they were
f.seek(0)
expridxs = find_bracket_exprs(everything)
exprs = []
for (start, end) in expridxs:
    exprs.append(everything[start:end + 1])
return exprs

def unify(dicts, d):
    IGNORE_TYPES = [":TAGS", ":ORTH", ":TYPE"]
    orth = d[":ORTH"]
    typename = d[":TYPE"]
    rest = dict(((k,v) for (k,v) in list(d.items()) if k not in IGNORE_TYPES))
    dicts.setdefault(orth, {})
    dicts[orth].setdefault(typename, rest)

def parse_all(infile):
    dicts = {}
    for expr in infile:
        d = exprlist_to_dict(tokenize_expr(expr))
        unify(dicts, d)
    return dicts

def parse_comlexfile():
    try:
        infile = input("Enter filename to open: ")
        if not infile:
            infile = "comlex.txt"
            print("Defaulting to %s" % (infile))
        outfile = input("Enter filename to dump to: ")
        if not outfile:
            outfile = "comlex_dump.py"
            print("Defaulting to %s" % (outfile))
        f = open(infile)
        print("Working...", end=' ')
        dicts = parse_all(f)
        f.close()
        print("done, writing to %s..." % (outfile))
        pp = pprint.PrettyPrinter(indent=4)
        f = open(outfile, "w+")
        f.write(pp.pformat(dicts))
```python
f.close()
return 0
except(KeyboardInterrupt):
    return -1

if __name__ == "__main__":
    parse_comlexfile()
```
Bibliography


