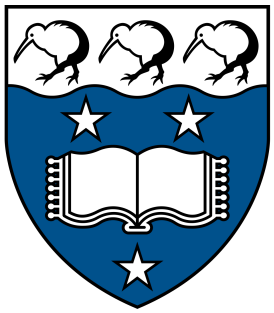
# Enabling Text Analytics

Jason Peter Cairns

Supervised by Chris Wild



Bachelor of Science (Honours)
Department of Statistics
The University of Auckland
New Zealand

**Abstract**

Text Analytics serves to reveal insights from a body of text. Within the broad category of text analytics, we seek to answer questions about what the text is communicating, what is felt about it, and how this information is structured. This report describes the creation of a user-friendly program to perform text analytics functions, using modern `R` with the Shiny web application framework. A literate style illustrates top-down the structure of such a program, as well as the data structures and computational processes that have established their value for it.

# Acknowledgements

I would like to thank my supervisor, Chris Wild, for his constant encouragement and enthusiasm for the project and my work.

I also thank my family (Bryan, Wendy, Ian, and Caroline) for their support and understanding during my study.

My thanks also go to those who provided technical advice, especially Daniel Barnett for advice on Shiny.

# Contents

# List of Listings

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Intention

The intention of this project is to develop a user friendly program that that allows for the performance of common text analyses. It is intended to be capable of integrating with iNZight, with a similar user base. The application is to be flexible enough to deal with a range of text formats, and produce attractive output quickly. The application makes a wide range of text analyses easily accessible to beginners, while still providing significant practical capabilities in text analytics.

## 1.2 Text Analytics Preliminaries

Text Analytics is comprised of a variety of processes and techniques to extract information from text and provide a high-level overview. The text almost always requires some initial processing. Some of the following functionalities have proven utility, and are placed in more context in Chapter 2;

**Sentiment** In order to answer what emotions are conveyed in a text, sentiment analysis is commonly performed. The technique yields some measure of what is represented in an emotional sense by the text, with a range different methods and their associated outputs allowing for different forms of the analysis. Sentiment analysis won't pick up the subtle nuances that a human reader would, but generally gives a reasonable impression over the extent of a text.

**Associated Words** The meaning of a text is largely dependent on the various structures within and between words. Looking at how words are associated, through correlation, common sequences, visualisation of sections, etc., allow for high-level investigation of the associations

between words. The higher level not only saves individual efforts, but can demonstrate any emergent properties inherent to a text, in a way that a direct reading won't necessarily reveal.

**Summarisation** Automation of an executive summary, or a list of key words, typically falls under the purview of "summarisation". The primary aim is to rank and select the most "representative" words, combinations of words, or sentences from a text. A few major techniques dominate, being somewhat complex in nature. The results seem to be surprisingly representative of a text.

**Feature Counts** The simplest quantitative measure is very often the most informative; from simple word counts, to selective counts of sentences within groups, counting features can reveal how much written weighting is given to various elements, aiding insight into content, structure, and sentiment simultaneously.

## 1.3 Existing Systems

There are several existing systems in the field of Text Analytics. The field was initially nurtured as a sub-field of Computer Science, being computationally-dependent in nature. More recently, there has been increasing statistical interest. The existing systems reflect this; most older text analytics programs were Artificial Intelligence focused, being experimental in nature and typically composed in lisp. There is also a closely related subfield of linguistics called "corpus linguistics", which is served by a small set of R packages and single-purpose applications performing text analytics tasks[1].

More recently, major statistical programs have been incorporating text analytic features, with a few smaller text analytics-specific programs appearing. `SAS`[2], `SPSS`[3], and `R`[4] are all examples of major statistical processing systems, with recent additions of text analytics capabilities. An example of a text analytics-specific application is `txtminer`, a commercial web app designed for analysing text at a deep level, with a linguistic focus, over multiple languages, for an "educated citizen researcher"[5]. A variety of R packages aiding in text analytics were assessed and experimented with, as detailed in Section 1.5.

## 1.4 Background: iNZight

iNZight is a statistical analysis suite developed at the University of Auckland[6]. It provides rapid analysis and visualisation of data in a user-friendly format. It is used by data journalists, students, and researchers, among others.

Our program will form part of the suite of modules extending iNZight. It provides a simple GUI interface to rapidly perform common text analyses. The primary audience for this text module are those learning the fundamentals and potential of text analysis, which could include students of the traditional text analytics fields of Statistics and Computer Science, but can and should include students of Linguistics, Communications, Law, History, and any other text-based field. Beyond the educational aspects of the program, it is fully functional for practical use for general text analysis. Of particular interest in applied statistics is the exploration of data from free-response items in surveys.

## 1.5 Literature Review

The initial review of existing literature was informed by the scope desired for the program, with `R`-related sources forming the bulk of the literature.

Two textbooks were read to begin with, both being related to R: "*Text mining with R: A tidy approach*"[7], and "*An Introduction to Text Processing and Analysis with R*"[8]. *Text Mining with R* was the prompt to use a "tidy" approach to text analytics, which has proved to be useful in interactive analysis, though inappropriate for development (discussed further in Section 3.1.3).

The Natural Language Processing (NLP) task view on CRAN maintains references to high quality packages relating to text analytics[9]. The following packages were consulted and experimented with:

`tidytext`**[10]** is a text-mining package using tidy principles, providing excellent interactivity with the tidyverse, as documented in the book *Text mining with R: A tidy approach*[7].

`tm`**[11]** is a text-mining framework that was the main package for text mining in `R`, but appears to have been made redundant by `tidytext` and `quanteda` of late.

`quanteda`**[12]** sits alone next to `qdap` in the Pragmatics section of the NLP task view, and offers a similar capability to `tidytext`, though from a more object-oriented paradigm, revolving around "corpus" objects. It also has extensions such as offering readability scores, something that may be worth implementing in the future.

`qdap`**[13]** is a "quantitative discourse analysis package", with an extremely rich set of tools for the analysis of discourse in text, which may arise from plays, scripts, interviews etc. It includes output on length of discourse for conversational agents, turn-taking, and sentiment within passages of speech.

`sentimentr`[**14**] is a rich sentiment analysis and tokenising package, with features including handling negation, amplification, etc. in multi-sentence level analysis. An interesting feature is the ability to output text with sentences highlighted according to their inferred sentiment.

`dygraphs`[**15**] is a time-series visualisation package capable of outputting very clear interactive time-series graphics, useful for any time-series in the text analysis module.

`gganimate`[**16**] produces animations on top of the `ggplot2` package.

`textrank`[**17**] implements the "TextRank" algorithm to extract keywords automatically from a text.

`ggpage`[**18**] produces impressive page-view charts with word highlighting, allowing for a clear overview of a text and it's structure.

`udpipe`[**19**] performs tokenisation, parts of speech tagging (which aids textrank), and more, based on the well-recognised `C++ udpipe`, using the "Universal Treebank".

`BTM`[**20**] performs "Biterm Topic Modelling", which is useful for "finding topics in short texts (as occurs in short survey answers or twitter data)". It uses a somewhat complex sampling procedure, and like LDA topic modelling, requires a corpus for comparison. The `BTM R` package is based on the `C++ BTM`.

`crfsuite`[**21**] provides a complete modelling framework, which is outside our scope, but could be useful later.

`humaniformat`[**22**] is useful in the analysis or removal of personal names, by providing a dictionary and properties of thousands of names.

Packages for obtaining text:

`gutenbergr`[**23**] can search and download texts from *Project Gutenberg* directly. *Project Gutenberg* is the largest public online repository of public domain books.

`rtweet`[**24**] can directly attain tweets from the social media site *twitter*. It requires an API key from *twitter*, which is an arduous process to obtain.

`WikipediaR`[**25**] can download *Wikipedia* pages and associated metadata directly.

## 1.6 Scope of work

The total scope possible for text analytics is near endless. As such, it is essential that limits were placed the scope of this program. There are two primary areas of limitation: text type, and analysis type.

By limiting the forms of text worked with, less effort was required to be spent on consideration of every single possible import and transformation case, and more time on the actual design of analysis. The simplest means with which to create the limitation exists in allowing only import of particular text-file formats — in this case, import caters only for flat `txt` files, as well as tabular `csv` and `xlsx` files. What is not provided is access in-program to common text sources through their API, such as Twitter or Project Gutenberg.

Through focusing on dictionary-based, rather than model-based analyses, we have avoided much of the associated complexities of modelling. An example of this is that it is common to categorise words based on their grammatical category, then use models that take the categories into account. By avoiding that, there has been far more functionality implemented in a shorter amount of time, with the analyses still performing soundly. Additionally, the primary focus is on a general audience, and it is typically only more advanced, linguistically-trained users who would make intelligent use of very complex analyses.

# Chapter 2

# Text Analytics Background

Text analytics is often considered a sub-field of data science, which is the extraction of insight from structured and unstructured data[26].

Text analytics is less commonly associated with statistics proper, given that much of the methodology is computer science heavy. The program developed in this project encourages a statistically-based analysis for text, giving emphasis to summary statistics of analyses, as well as distributions of results.

Many text analytics analytics solutions are in fact computer science based, rather than statistical. In this sense, text analytics mirrors data science closely, with much of the development occurring in computer science departments, and some breakthrough models introduced by statisticians. For example, the "bag of words" model was expounded upon by computer scientists (in association with linguists), along with most other models, to the point where "text mining", the name more commonly associated with text analytics in the past, was effectively treated as a sub-field of computer science[27]. However, many important models are very statistical in nature, such as Latent Dirichlet allocation, which uses complex Bayesian inference, and was developed in conjunction with statisticians[28].

## 2.1 Overview

Having as it's object of interest textual communication, text analytics borrows much of the vocabulary of linguistics. Some important concepts allow us to speak the language of text analytics:

**word** The smallest element uttered in isolation with objective meaning. Not necessarily separated by spaces, which is an orthographic convention.

**term** "a word or expression that has a precise meaning in some uses or is peculiar to a science, art, profession, or subject" [29] — here text analysts have capitalised on the generalisation of "term" to include sub-components or aggregations of words.

**token** A concrete instance of a word in text. The phrase "The cat on the mat" has 5 tokens, as the first "the" is considered separately to the second.

**text** A written communication, in text analytics typically excluding "textuality" elements such as a reader's interpretation. Examples include books, YouTube comments, articles, etc.

**stem** The reduced form of a word, created by removing inflections. For example, "readily" may have the "ly" removed to become "readi"; in linguistic terms, the process of morphological (units of meaning) reduction of a word.

**lemma** The dictionary form of a word. Typically without tense. The lemma of "readily" is "ready", and the lemma of "is" is "be". In linguistic terms, a lemma is the conventional form of a lexeme (the basic meaning behind related words).

**lexicon** A language's inventory of lexemes (lemmas). In text analytics a lexicon is typically a dictionary data structure containing a list of words and possibly some values for each word.

**corpus** A collection ("body") of texts, with analysis usually within and between the texts.

## 2.2 Terms

At the centre of text analytics are words. Different models of text analytics typically differ over the basis by which they treat words. For example, the "bag of words" model only considers the list of unique words in a document, and how many each of these occur. The framework used in this project is based on the "tidy approach", wherein each word is considered a distinct element of a vector, which may, as a whole, represent a text. This can be aligned with other information, in the form of a data frame. The tidy approach preserves ordering, unlike the bag of words model.

A significant benefit of maintaining the ordering of the words of a document is that structural information is preserved, allowing for analysis of the emergent structures within a text. If additional information on sentences, chapters, or other textual groupings are preserved or inferred (and this program is capable of both), then these structures can be analysed in their

own right. Common structures include sentences, chapters, documents, and n-grams:

n-gram A series of words that occur in a direct n-length sequence. For example, in the phrase, "The quick brown dog", the following 2-grams (bi-grams) exist: "The quick", "quick brown", "brown dog". This is generalised to any number of sequential words. They are useful in text analytics to determine word sequences, as well as common adverb-verb and adjective-noun pairs.

## 2.3 Framework

Text analytics follows much of the same process (Fig. 2.1) as many other data science analyses. We have identified three main sections of a standard text analysis, being processing, "insights", and visualisation. "Insight" here is a word we have specialised the meaning of to capture the concept of producing statistical summaries or sets of summaries from processed text. The three sections identified are similar to the data science process, however due to the lack of structure present in most text data formats, extra processing is typically required to infer or associate structure. The various processes of attaining insight typically follow a similar functional process with a standard input and standard output, with some notable exceptions. Visualisation is often similar to other statistical analyses, though the nature of text presents some unique challenges and opportunities

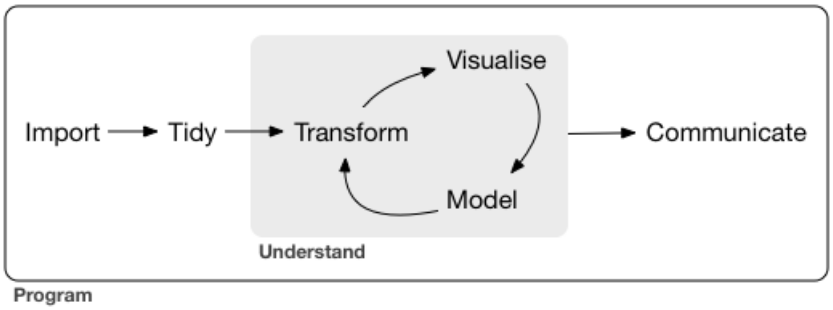



Figure 2.1: Standard data science process[30]

### 2.3.1 Processing

Processing is the first component of every text analysis, commonly returned to and modified within an analysis. Processing refers to the import, cleaning and transformation of text data. It is performed in order to attain a standard structure for text data, which is notoriously unstructured. The data is generally very messy, requiring significant effort to clean and "tidy".

Importation of text data is fairly simple, being no different to the import of any other data. However, words need to be separated, a process known as "tokenisation". Once words are separated and kept separate through some data structure, words that are unnecessary for analysis are removed. Text analytics is opposite to most statistical analyses in this respect; unusual data points may sometimes be removed in statistical analyses as outliers, but common words that render no insight, such as articles like "the", are commonly removed in a text analysis. These removed words are known as "stopwords". The remaining words may be transformed into lemmas or stemmed. As text files often have no metadata indicating internal structure such as chapters, these can be inferred through a process created by this project, which we call "sectioning". Finally, the processed data is pulled together into some standard data structure, and fed through to the next part of the text analytics process.

### 2.3.2 Insight

After the text data has been made malleable enough through processing, it is ready for analysis to produce some "insight measure", wherein various statistics and scores can be computed from the processed text.

In order to attain useful information about the text, "insight" functions can be applied to the data, giving metrics relating to various properties of the text which can capture important information about the text.

Most of our insight functions will compute some score or statistic for texts. These may be applied at the individual word level, or at some aggregate level. Key functions include the frequency count of words, n-grams and n-gram frequency, summary scores, and correlations between words. Sentiment is a common score that is computed to map the emotional charge of a particular word to some numeric score, or emotional category. The program implements all of these functions, as well as some unique and experimental variations on them. Other common analyses include determining the frequency of words in several individual documents relative to their overall frequency, known as "term frequency — inverse document frequency", and inferring common topics in a text, known as "topic modelling"

### 2.3.3 Visualisation

A clear way to display and communicate the result of an analysis or exploration is through visualisation. Visualisations are applied to "insights". Typically a variety of different visualisations are available to be applied any particular insight measure.

When visualising text, most insight measures produce a numeric output. These numeric scores or statistics can be visualised through common stat-

istical graphics, such as bar plots, histograms, density plots, etc. When each word of text is considered a step forward through time in a text, the scores of each word may be used to plot a time series. Uniquely to the visualisation of text, the property belonging to each word or group of words can also be visualised as an overlay on the original page of text — known as a "page view". The words themselves may be used as the central piece of a visualisation beyond the bounds of a page, with "word clouds" sizing words according to their insight score, and laying them randomly about a plot.

Alternative visualisations were explored, with much experimentation regarding different displays of sentiment insights through scatterplots. However, these were found to be difficult to interpret. The only visualisations maintained are those satisfying the constraints of clarity and ease of interpretation.

# Chapter 3

# Program Structure & Development

## 3.1 Preliminaries

The structure and development of the program has been based mainly on compatibility with the existing system of iNZight Lite. The current form of iNZight Lite uses `R` to perform analyses and handle data, with the interface using Shiny, an `R` package allowing for declarative UI and server definitions to compose a web app, in a framework similar to JavaScript's react.js[31]. Much of the program has been written in a functional form, which is idiomatic of R. This has extended to make use of a constellation of packages known as the "Tidyverse"[32], including "ggplot2"[33], for graphic visualisations. This text analytics program followed that lead, using both `R` and Shiny, as well as the functional paradigm with the tidyverse.

### 3.1.1 Why R?

Aside from compatibility, `R` is a language well suited to text analytics in it's own right, with the vectorised semantics mapping well to the problem space of long sections of text. The package ecosystem is also very well equipped to handle text analytics, with the official package repository, CRAN, being audited to maintain a high level of quality that other repositories such as npm or pip, don't come close to achieving.

### 3.1.2 Why Shiny?

Shiny is the default choice for creating web-friendly interfaces in R. Funded and developed by RStudio, it is typically made use of for dashboard creation, though it has even been put to use as a chat client. Shiny makes use of

the react style of interface and server declaration, in which state is heavily de-emphasised in the semantics of the code, thus keeping the paradigm as declarative as reasonably possible for a dynamic application. This has some drawbacks, where state may be useful if an object is to have transformations iteratively performed on it.

### 3.1.3 Why Tidyverse?

The Tidyverse is "an opinionated collection of packages designed for data science", which is primarily used in this project for data manipulation with the included package "dplyr"[34], and visualisation with "ggplot2". The packages in the collection have mostly consistent syntax and style, and are built around the underlying principle of "tidy data", wherein the central working object is a dataframe or dataframe derivative, with a "long" format, where every column is a variable, and every row is an observation[35]. This allows for great consistency and ease of manipulation, as there is only one datatype to manage. My text analytics package was designed with this philosophy in mind, maintaining tidy data and using tidyverse functionality at all times, which has greatly aided in consistency and standardisation of development.

However, not all data is modelled well by a tidy dataframe, especially the hierarchical data inherent in much of text analytics. In these cases, adherence to tidyverse principles has become more of a hindrance than a help. The over-emphasis on the tibble (dataframe) class, encouraged by the tidyverse, has been an unnecessary constraint for development. All of the more recent functionality in the program has made decreasing use of the tidyverse aside from ggplot2, which has the unparalleled ability to facet plots easily based on some grouping variable — demonstrated in our application through Fig. 3.1.

The use of non-standard evaluation is also set up to work well for interactive data analysis, though for development purposes, the need for quoting and unquoting symbols and names has led to a large amount of unnecessary overhead relative to simply using string-based selection.

### 3.1.4 Why Functional?

Functional Programming is a programming paradigm placing emphasis on pure functions, with no state in the program. Every function has a surjective mapping from input to output, so for the same input, no matter what else has happened in the program, the same output should be expected. This is often contrasted with object-oriented programming, wherein objects are created that change state as the program progresses. Assignment of the form, `x <- x + 1`, would not exist in a functional program, as the

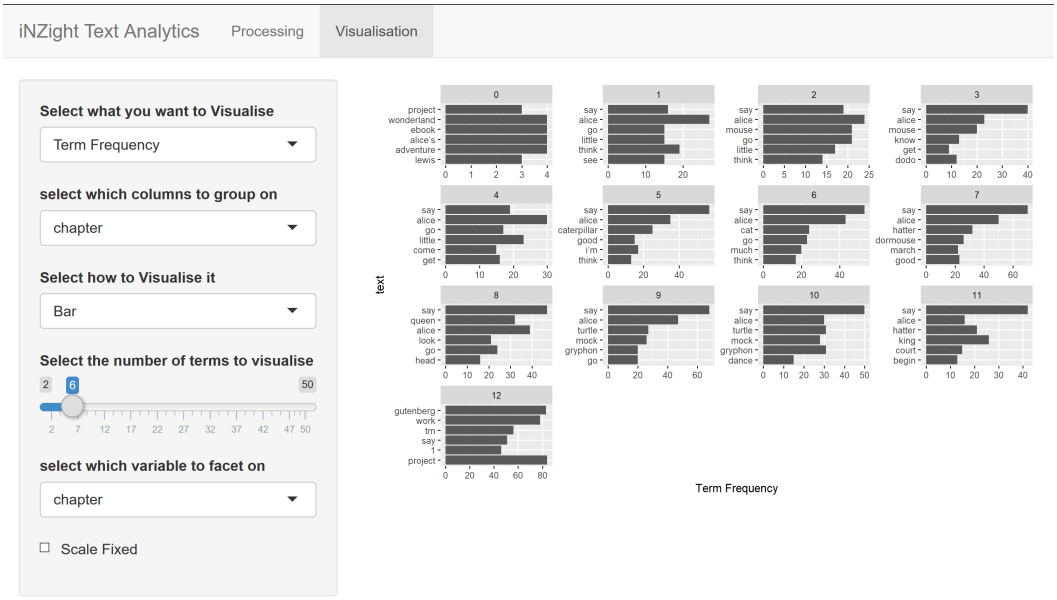



Figure 3.1: Visualisation of facetting

same variable `x` changes reference at different points. Thus, for- and while-loops are typically lacking, as they depend on mutating variables with each iteration. The text analytics program is nearly entirely functional, with the exception of simple I/O, and some functions that would be obfuscated in meaning if made functional, such as `get_ngram()`. Functional programming has brought benefits of easy debugging, as state is no longer a consideration, and an ability to reason about functions in mathematical terms.

### 3.1.5 Why lossless data?

The largest issue with keeping the program functional (Section 3.1.4) and tidy (Section 3.1.3) has been the issue of maintaining the history of the text data. Because there is no state, history can be difficult to maintain in a memory-efficient manner, and for transformations on text, history is essential to have at hand. A notable example is the issue of stopword removal; if stopwords are removed, with nothing indicating their original location, and the subsequent text is plotted as a page view, the actual structure of the original text is missing, with no means to indicate where stopwords are. Thus the central data structure had to be designed to be as lossless as possible. This was achieved in the form of a dataframe, with a column keeping the original text, and a new column holding the text to be analysed, equivalent

to the original text column, but possibly transformed with lemmatisation, and stopwords "removed" by replacing them with NA values. Every row corresponds to a single word from the original text, thus the data structure is congruent with tidyverse functions.

### 3.1.6 Why Git?

Git was used from start to finish of development for version control. Git was used instead of other version control systems such as subversion or mercurial due to the networking effect, as collaboration is made easier with all being familiar with the use of GitHub as a centralised repository. In addition, the R "remotes" package and Shiny itself allow for the direct download and installation of the entirety of the text analytics package and web application, as in Listing 1.

```
1 install.packages(c("remotes", "shiny"))
2 remotes::install_gitlab("jasoncairns/inzightta")
3 shiny::runGitHub("jcai849/inzightta-shiny")
```

Listing 1: Installation of package and prototypical deployment of app

## 3.2 Program Architecture

The program is composed of an R text analytics package, developed over the course of this project, and a web application, developed towards the end. The package is composed of three separate sections, being Preparation, Insight, and Visualisation. This serves to provide functionality from the start to the finish of a text analysis. The web application is composed of a UI (user interface) and a server, reliant on the package. The interface is largely generated server-side depending on the form of the analysis. The server does much of the coordination for the application, sending options specified by the user in the UI over to the functions defined in the package, and returning the results, acting much like the "controller" in the model-view-controller framework of software engineering. A diagram of the program as a system is shown in Fig. 3.2.

The program has been structured in this manner to make it as modular as possible, to the point where, with some minor further development on the package, it could be released on CRAN as a powerful text analytics R package in it's own right.

### 3.2.1 Package Description and Creation

The package, given the working title "iNZightTA" (iNZight Text Analytics), serves to provide all of the background functionality for the program. In all,

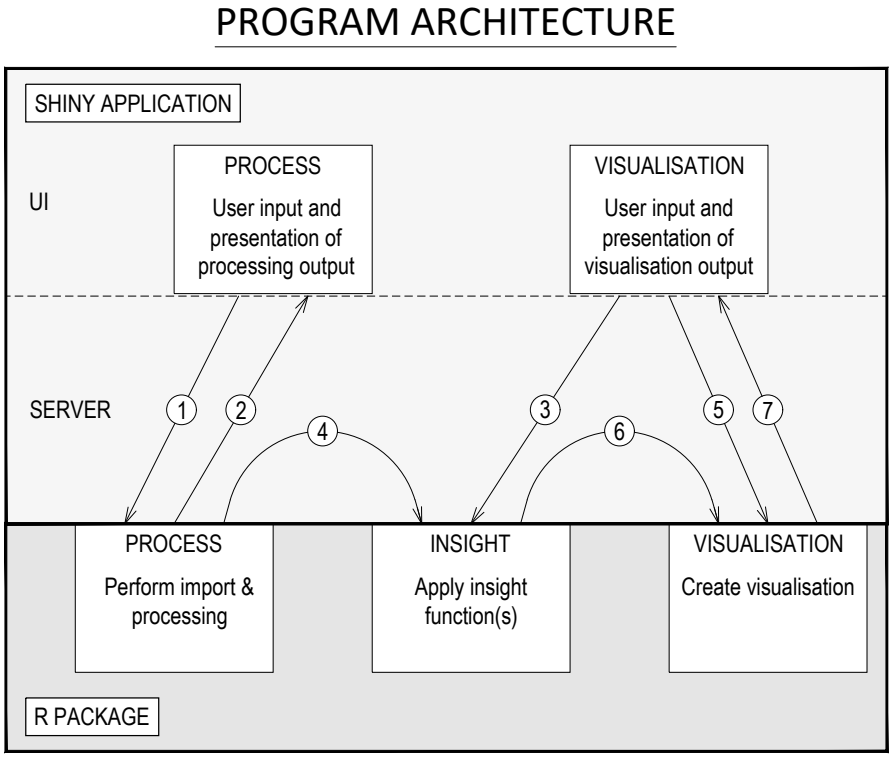



Figure 3.2: Program architecture

there are over 50 major functions defined, as well as numerous utility and anonymous functions.

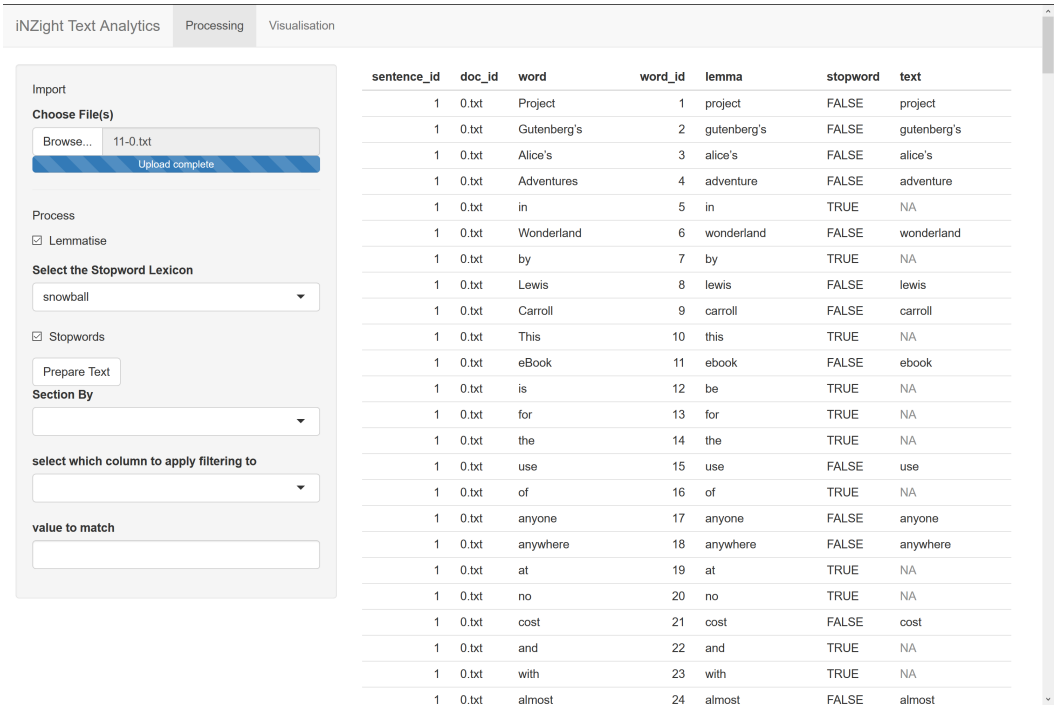
Package development mostly followed a "best practices" procedure, with file structure following standard `R` patterns. The documentation made use of `Roxygen2`[36], and a 30 page manual was automatically generated from inline function comment documentation. Testing was performed on a variety of text types, including free-form survey responses in the form of `csv`'s, and novels from *Project Gutenberg* in the form of `txt` files.

Dependencies are a necessary evil, and care was taken to include only highly regarded packages. The `tidytext` package was made extensive use of due to it's advocacy by the author of the textbook, *Text mining with R* (who is also the author of the package). Unfortunately, despite the online acclaim for `tidytext`, the author's poor development practices lead to several updates of the `tidytext` package having backwards incompatible changes, including functions outputting a completely separate class of object to what was output previously, and the documentation not being updated to reflect these changes. Upon deeper inspection of the package source code, investigating several issues, it was found that most of the functions are wrappers around equivalent functions provided by other packages. This has created a consistent interface, but on measure, has not provided enough of a benefit to justify the issues encountered via the very leaky abstraction.

## 3.3 Preparation

The first step in all text analysis is to import the text data and wrangle it into a data structure suitable for statistical analysis. In this case, the data structure for all filetypes is designed to be lossless, in the form of a dataframe that retains the original text. This was a key design decision, to avoid "destructive edits", which is the practice where the original input can't be recovered after the transformation. It is non-injective, and non-invertible. Thus, when certain changes are required, an earlier state is needed. `Tidytext` has made the decision to encourage destructive edits, which is acceptable when the user is in an interactive `R` session with full control over every possible variable assignment, but not for development for a GUI user. Hence, we have made the explicit decision to have non-destructive transformations only, after hitting repeated roadblocks related to Destructive edits. Memory is cheap for computers, and summarisation functions can always be delayed, to retain as much information as possible. The concept of nondestructive edits is not new or unique, with much of graphic design relying upon it[37].

In the application, the first tab encountered by a user is preparation, which includes import and processing (Fig. 3.3). At present, a representation of




| sentence_id | doc_id | word | word_id | lemma | stopword | text |
|---|---|---|---|---|---|---|
| 1 | 0.txt | Project | 1 | project | FALSE | project |
| 1 | 0.txt | Gutenberg's | 2 | gutenberg's | FALSE | gutenberg's |
| 1 | 0.txt | Alice's | 3 | alice's | FALSE | alice's |
| 1 | 0.txt | Adventures | 4 | adventure | FALSE | adventure |
| 1 | 0.txt | in | 5 | in | TRUE | NA |
| 1 | 0.txt | Wonderland | 6 | wonderland | FALSE | wonderland |
| 1 | 0.txt | by | 7 | by | TRUE | NA |
| 1 | 0.txt | Lewis | 8 | lewis | FALSE | lewis |
| 1 | 0.txt | Carroll | 9 | carroll | FALSE | carroll |
| 1 | 0.txt | This | 10 | this | TRUE | NA |
| 1 | 0.txt | eBook | 11 | ebook | FALSE | ebook |
| 1 | 0.txt | is | 12 | be | TRUE | NA |
| 1 | 0.txt | for | 13 | for | TRUE | NA |
| 1 | 0.txt | the | 14 | the | TRUE | NA |
| 1 | 0.txt | use | 15 | use | FALSE | use |
| 1 | 0.txt | of | 16 | of | TRUE | NA |
| 1 | 0.txt | anyone | 17 | anyone | FALSE | anyone |
| 1 | 0.txt | anywhere | 18 | anywhere | FALSE | anywhere |
| 1 | 0.txt | at | 19 | at | TRUE | NA |
| 1 | 0.txt | no | 20 | no | TRUE | NA |
| 1 | 0.txt | cost | 21 | cost | FALSE | cost |
| 1 | 0.txt | and | 22 | and | TRUE | NA |
| 1 | 0.txt | with | 23 | with | TRUE | NA |
| 1 | 0.txt | almost | 24 | almost | FALSE | almost |

Figure 3.3: Processing screen

the created dataframe is shown.

### 3.3.1 Importing

Text must first be brought in from an outside source to be useful for the program. The import functions are such that all text from different files exist in dataframes of equivalent structure. The primary differences are that each row of an imported `txt` file corresponds to a single line, whereas each row of an imported tabular file corresponds to the row of the tabular file. Importantly for tabular files, the column of the text intended for analysis must currently be given the header of "text" prior to import. This condition will be relaxed later. Of note is that importing of files is a mostly solved problem in R, and most of the import functions in this package are simple wrappers that serve the main role of standardising the output. In the web app, importing occurs first, immediately prior to processing, with a file browse button and a loading bar (Fig. 3.4)

#### Import `txt`

The simple function `import_txt()` (Listing 2) used to import `txt` files simply reads lines from a file, and converts the data into a tibble[38], with a single column named "text" and single row containing the document as a

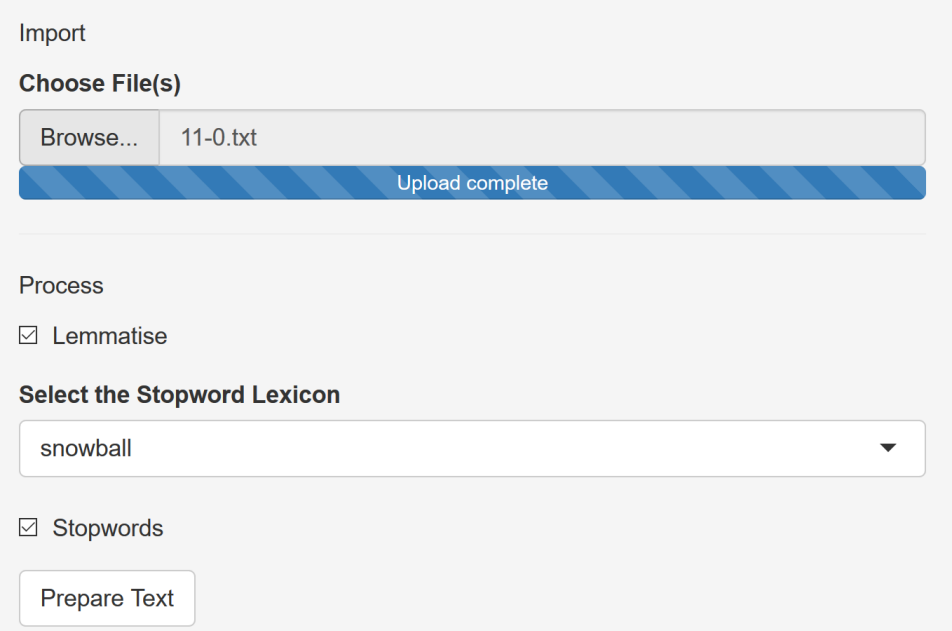



Figure 3.4: Importing a document

long string.

### Import `csv`

CSV is a plaintext tabular format, with columns typically delimited by commas, and rows by new lines. A particular point of difference in the importation of tabular data and regular plaintext is that the text of interest for the analysis should be (as per tidy principles) in one column, with the rest being additional information that can be used for grouping or filtering. Thus, additional user input is required, namely the specification of which column is the text column of interest. The function `import_csv()` (Listing 3) is a simple wrapper around `readr::read_csv()`[39], created only to maintain consistent naming conventions.

### Import Excel

Much data exists in the Microsoft Excel format, and this must be catered for. As tabular data, it is treated equivalently to `csv`, with the function `import_excel()` (Listing 4) being a wrapper around `readr::read_excel()`.

### Import Wrapper for Arbitrary Number of Files

To have just one function required to import files, two sub-functions (Listing 5) are defined; `import_base_file()` imports any file, and `import_files()`

makes use of the aforementioned, to import multiple files. The import of multiple files is no trivial task; the program must shape them in such a way that they retain identification, and fit into the same data structure together. Different filetypes can be combined in the same analysis.

The base wrapper function takes in the filename, and other relevant information, handling the importation process. It also stamps in the name of the document as a column.

The base file import is generalised to multiple files with a multiple import function: this will be our sole import function.

### 3.3.2 Object Preparation

From the imported files, their representations are transformed into a lossless and efficient data structure that any analysis can make use of, with a single function, `text_prep()` (Listing 6). Our solution to the essential constraint of losslessness is to separate and identify rows by each word in a dataframe. The identification includes the line ID, the sentence ID, and the word ID, producing a dataframe that takes the form of Table 3.1.

| line_id | sentence_id | word_id | word |
|---|---|---|---|
| 1 | 1 | 1 | the |
| 1 | 1 | 2 | quick |
| 2 | 1 | 3 | brown |

Table 3.1: Primary data structure format

The reason for the ID columns is the preservation of the structure of the text; If required, the original text, or chunks of it, can be reconstructed in entirety, sans minor punctuation differences. The options for the function in-app are given as checkboxes and dropdowns.

### 3.3.3 Filtering

Filtering of text is implemented directly with the `dplyr::filter()` function, directly in the server of the Shiny app. Filtering can take place multiple times throughout an analysis. The program is flexible enough such that after some initial analytics have been done in the insight layer, preparation can be returned to and the text can be filtered based on features seen in the analytics.

### 3.3.4 Lemmatisation

Lemmatisation is the process of transforming words into their dictionary form. It is a very complex, stochastic procedure, as natural languages don't

follow consistent and clear rules all the time. Hence, models have to be used. Despite the burden, it is generally worthwhile to lemmatise words for analytics, as there are many cases of words not being considered significant, purely due to taking so many different forms relative to others. Additionally, stopwords work better when considering just the lemmatised form, rather than attempting to exhaustively cover every possible form of a word.

`textstem` is an `R` package allowing for easy lemmatisation, with it's function `textstem::lemmatize_words()` transforming a vector of words into their lemmatised forms (thus being compatible with `dplyr::mutate()` straight out of the box)[40]. The lemmatisation in this program is managed entirely by this single function in the server end of the Shiny app. The package udpipe was another option, but it requires downloading model files, and performs far more in-depth linguistic determinations such as parts-of-speech tagging, that at this point are excessive. It is worth noting that, as with stopwords, there are different dictionaries available for the lemmatisation process, but we only use the default, as testing has shown it to be the simplest to set up and just as reliable as the rest.

### 3.3.5 Stemming

Stemming is the removal of word endings, and is far simpler than lemmatisation. For example, lemmatisation may change the word "happiness" to "happy", while stemming would change it to "happi", removing the "ness". This doesn't require as complex a model, as it is fairly deterministic. Stemming is not always as effective as lemmatisation, as the base word ending is not concatenated back on at the tail, so all that remains are "word stumps". However, it may sometimes be useful when the lemmatisation model isn't working effectively, and `textstem` provides the capability with `textstem::stem_words()`. We have not implemented this yet, as it is not as essential to an analysis when lemmatisation is already available.

### 3.3.6 Stopwords

The package makes use of dictionary-form stopwords, allowing for the input of both developed lexicons as well as user input. Two functions (Listing 7) compose stopwords: `get_sw()`, which gathers user input, queries the selected lexicon and combines the two, and `determine_stopwords()` which adds a boolean `TRUE | FALSE` column to the input dataframe. The stopwords functions operate on the original text, but if lemmatisation has been performed, then the functions operate on the lemmatised forms.

### 3.3.7 Formatting

The final component in preparation is to format the prepared object with the correct attributes which allow formatting to be automated. A wrapper named `format_data()` (Listing 8) is defined that takes all combinations of stopwords and lemmatisation options and intelligently connects them to create the "insight column" in a dataframe, when insight functions are applied later on in the pipeline. For the purpose of standard interoperability, e.g., with the `ggpage` package, this column is named "text".

At the heart of this function is an `ifexp()` that encodes the following logic involving the interaction of stopwords and lemmatisation, to enable the correct output text based on stopword and lemmatisation options, as given in Table 3.2.

| | Stopwords True | Stopwords False |
|---|---|---|
| Lemmatise True | Lemmatise, determine stopwords on lemmatisation, perform insight on lemmas sans stopwords | Lemmatise, perform insight on lemmas |
| Lemmatise False | Determine stopwords on original words (no lemmatisation), perform insight on words sans stopwords | Perform insight on original words |

Table 3.2: Formatting Logic for Stopwords and Lemmatisation

Based on the combination, stopword filtering and lemmatisation take place inside the function.

### 3.3.8 Sectioning

Plain text, as might exist as a Gutenberg download, differs from more complex representations in many ways, including a lack of sectioning — for example, chapters require a specific search in order to jump to them. A closure, `get_search()` (Listing 9) is composed that searches text based on a regular expression intended to capture a particular section. Several search functions have been created from that closure, such as `get_cantos()`, `get_sections()`, etc. The sections are added to the dataframe with `section()`. These have been given as a drop-down selection list in the app (date, advanced users could be given the option to compose their own regular expressions for sectioning.

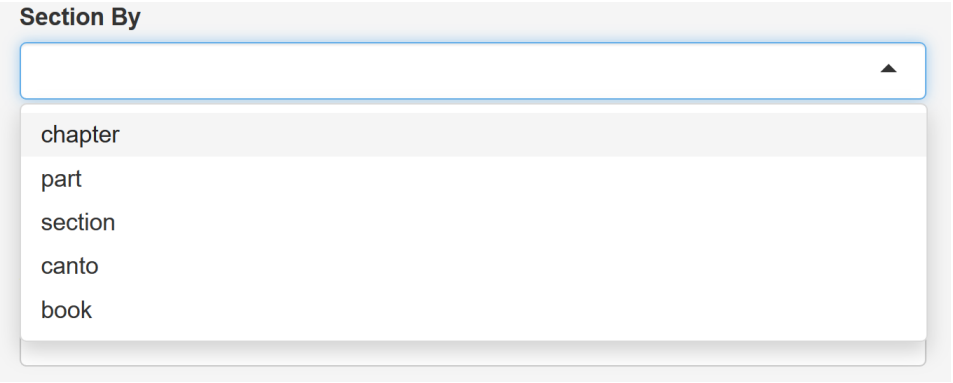



Figure 3.5: Section types

### 3.3.9 Grouping

Grouping is an essential, killer feature of the app. The implementation makes use of a `dplyr::group_by()` command in the Shiny server on the prepared object, over user-specified groups, and all further insights and visualisations are performed groupwise. This allows for immediate and clear comparisons between groups.

Like filtering, after some initial analytics have been done in the insight layer, preparation can be returned to, and a new variable constructed based upon the initial analytic results, so the the text can be grouped-on based on the the features seen in the analysis.

## 3.4 Insight

After processing has yielded an appropriate data structure to operate upon, analytic functions may be performed, under the banner of "insight". The bulk of this package is composed of the insight functions, divided into term insights, and higher level (aggregate) insights. Higher level insights exist where units of analysis may be n-grams, sentences, chapters, documents, etc. Additional functionality is very easy to implement, just requiring a named function that can operate on the standard data structure. The output of an insight measure is added as an additional column to the dataframe resulting from processing the text.

### 3.4.1 Term Insight

Individual words form the basis of any analysis, and reveal a great amount of information about a text. Term insights all have the same expected form of input an output, with the output vector always matching the size of the input vector, with elements being a direct element-wise mapping.

| list | list shifted back | bigram |
|---|---|---|
| | 1 | |
| 1 | 2 | 1 2 |
| 2 | X | 2 4 |
| X | 4 | X |
| 4 | 5 | 4 5 |
| 5 | X | 5 7 |
| X | 7 | X |
| 7 | X | X |
| X | | X |

Table 3.3: Bigram behaviour with missing values indicated by "X"

### Term Frequency

Frequencies of words are useful in getting an understanding of what terms are common in a text. This is one insight in particular that usually requires stopwords to have been previously removed, otherwise the top words will always be syntactical glue, such as articles. A function named `term_freq()` (Listing 10) is defined to perform this analysis. Running this function with stopwords left in is a good comparative diagnostic for checking whether the stopword lexicon is removing words that aren't intended to be removed.

### n-Grams

One difficulty that presents itself in the determination of n-grams is the treatment of missing words. Stopwords have to be considered, lest the original text be altered destructively. An example of this difficulty is through the two sentences, "I went to the shops", and, "I went to the car". Most stopword lexicons would remove all words but "went", "shops", and "car". This leaves a grey area — should the n-grams make reference to the missing values, or skip over them?

This was answered in this package by the slightly more difficult to implement skipping-over of missing values, in the n-gram insight function `get_ngram()` (Listing 11). This way, the results of the analysis would not be polluted with "NA", as is the very intention of stopword removal. An example of the expected output of this procedure is given in Table 3.3; each element of the bigram column corresponds to the element in the list column of the same row. n-grams are stored in this way to maintain the data-frame structure.

The function is one of the only internally non-pure functions (through changing state through for- and while- loops) in the entire package, as searching sequentially along a list is far easier to conceive of in imperative terms.

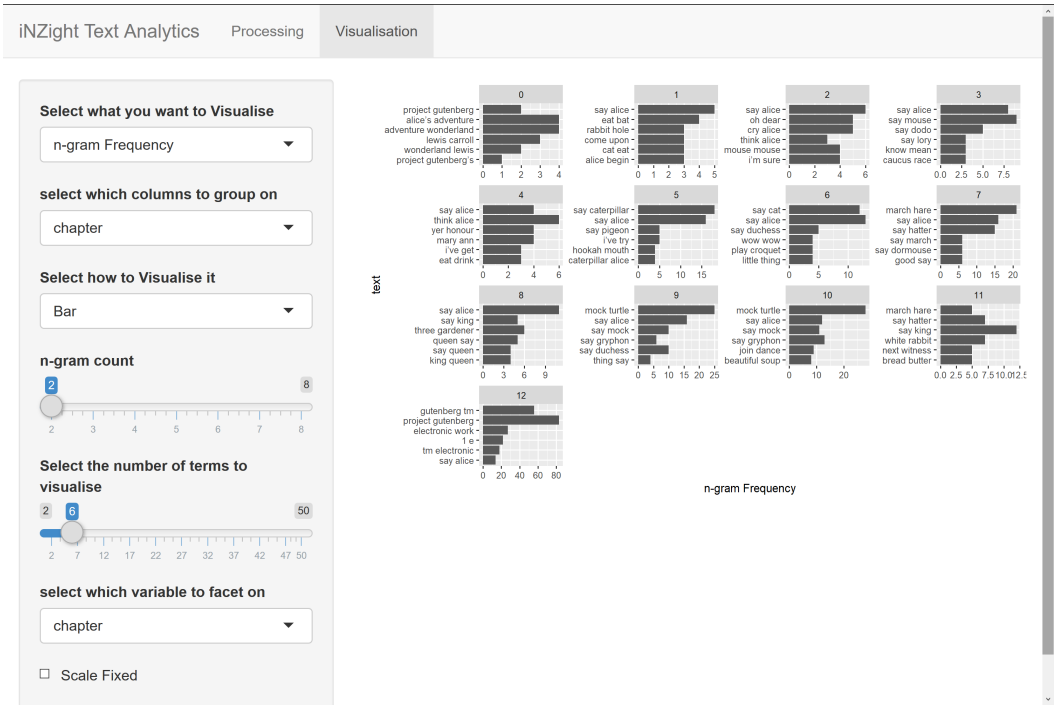



Figure 3.6: n-gram visualisation with facetting by chapter

The frequency of a particular n-gram is given by applying the `term_frequency()` function to the newly determined n-gram column, in the `ngram_freq()` wrapper (Listing 12). A typical visualisation of n-grams is demonstrated in Fig. 3.6.

### Key Words

Determination of key words is a useful analytic function that differs from finding the most frequent words. Key word algorithms are generally based on some derivative of the TextRank algorithm. This algorithm constructs an adjacency matrix of a directed graph with words or sentences as the nodes, and the arc weights reflecting some measure of similarity between words, typically a function of co-occurrence within the same sentence. The PageRank algorithm is run on this graph, picking up the most representative words that may not necessarily be the most frequent. The words with the highest PageRank scores are then selected to produce a set of key words. A function named `keywords_tr()` (Listing 13) implements this, on the shoulders of the `textRank` package[17].

Of note is that all words other than stopwords are treated as relevent, but the standard algorithm works better on data that has had parts of speech tagging, typically assessing only nouns and adjectives. This function doesn't

make use of such tagging, as the processing burden for Parts of Speech tagging is enormous and slow, undercutting the necessity for the interactive app to be quick and responsive. A faster, simplified version may be implemented in the future.

### Term Sentiment

Term sentiment is one of the most common analyses, especially in the automated processing of free-form survey responses. It gives some measure of the sentiment of the terms in a text, giving insight into the emotional charge of a text and it's structure.

Sentiment analysis can either make use of dictionaries, where each word has an associated score, or models, where complexities such as negation or amplification are considered. The function defined in this package to determine term sentiment is named `term_sentiment()` (Listing 14). It currently makes use of the simpler and less processing-intensive dictionary form. There are numerous dictionaries available, giving numeric scores or categorisations of sentiment type. As numeric scores are the easiest to visualise, these have been the intended implementation of the function. Well-regarded dictionaries ("lexicons") include "AFINN", which gives a numeric score centred at zero, "bing", which gives a classification of the emotional category, and "ISO". The default of this program is the lexicon "AFINN".

A problem in dictionary methods for sentiment analysis is that the same word may or may not carry a perticular emotional charge, depending on the subject area and type of writing. Specialised dictionaries exist for some specialist subject areas, which aim to remedy this. For example, in mathematics, a "problem" has very different emotional connotations to a "problem" in the diagnosis of a hospitalised patient.

The function currently possesses a strong dependency on the `tidytext` package. This has come at a cost, with `tidytext` having a constantly changing and unreliable API — future development will move away from such a dependency. At present (though not at the time of original inclusion), the `tidytext` package requires download of sentiment dictionaries through the command line at first use, which is an annoying impediment in a GUI application.

### Moving Average Term Sentiment

A statistical summary of a sentiment in time gives a more contextual picture of how sentiment is given in a text. The function defined as `ma_term_sentiment()` (Listing 15) performs such a task and is flexible enough to take any statistic or lag, defaulting to be a moving average of 10. The first `lag` terms will be `NA`, and the number of `lag` is inclusive, matching the semantics of

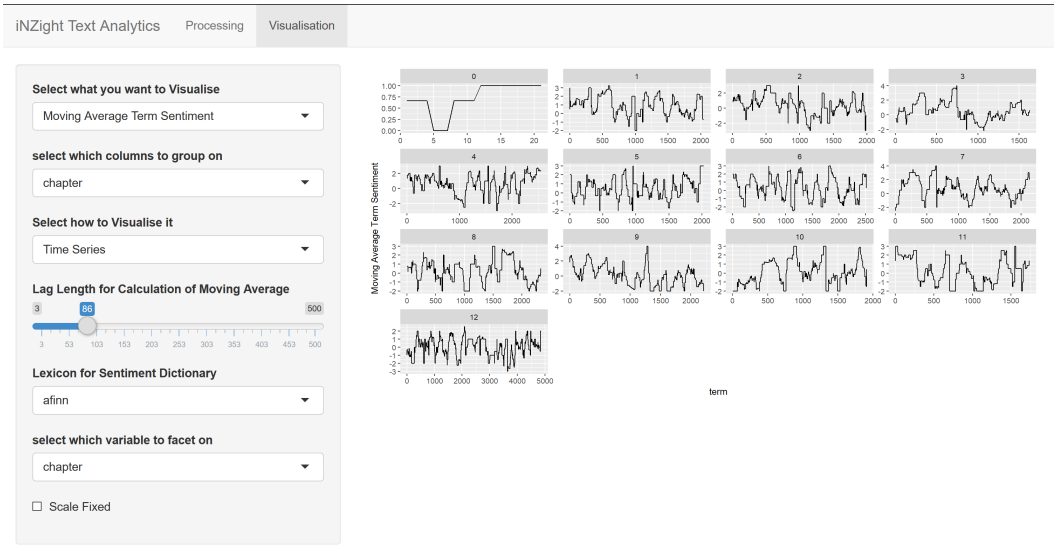



Figure 3.7: Moving average term sentiment visualised with Time Series

R's indexing starting at 1. Moving Average Term Sentiment finds a natural visualisation as a time series plot, demonstrated in Fig. 3.7.

### 3.4.2 Aggregate Insight

Analysis is not just restricted to words alone, and can consider the emergent forms composed through collections of words, such as sentences and documents, even tweets or responses. With a `csv` input file, aggregates can be also be defined in columns corresponding to the text column, and made use of through the following functions. All functions perform similarly to the term-level insight functions, with vectors as input and vectors as output. The aggregate functions also require another vector defining aggregate groupings, in a similar manner to the `INDEX` argument of `tapply()`.

#### Term Count

Word count on some aggregate group follows the pattern where the simpler a function is, the more analytical power it seems to give. A function named `term_count()` (Listing 16) is defined to count terms per grouping. The typical usage for this would have an aggregation over sentence ID, giving the word count in each sentence. This can be used in conjunction with grouping, allowing for a hierarchical measure of the distribution of, for example,

sentence lengths per chapter within a document. If these markedly differ between one chapter and the rest, there is likely a different author or style in that one chapter.

Note in the function the near canonical example of split-apply-combine, or MapReduce style. This will allow for major performance gains if parallelised, ideal for the large datasets typically made use of in text analytics.

### Key Sections

Often keywords aren't very explanatory on their own; patterns only really develop in the aggregate. Determining key sections aggregating over sentence reveals the most representative sentences in a document. As such, it is a form of automated summarisation. A function named `key_aggregates()` is given to compute such an analysis (Listing 17). LexRank is used as the default algorithm for finding key-sentences, as testing has demonstrated that textrank takes too long, though LexRank still takes a long time for processing[41]. In testing this function, it was run over the original paper describing LexRank, with the results being a very efficient summary of the paper, outlining the key notions behind the algorithm.

LexRank is essentially the same as TextRank, however it uses cosine similarity of Tf-Idf vectors as it's measure of similarity. LexRank is better at working across multiple texts, due to the inclusion of a heuristic known as "Cross-Sentence Information Subsumption (CSIS)".

Testing shows a performance of around 3–4 minutes for $\approx 30,000$ words of text aggregated over $\approx 3000$ sentences. This is not terrible performance for a graph-traversing algorithm, but a warning is definitely required at the user end.

### Aggregate Sentiment

Like the added context that key sentences bring over key words, a similar situation is true of sentiment. A function named `aggregate_sentiment()` (Listing 18) is given to determine a statistic of sentiment over some group; most commonly, the mean sentiment of each sentence or "response" in a text. This function is higher-order, and can deliver any statistic of a group's sentiment; mean, median, variance etc. Importantly, this function will only work with numeric sentiment lexicons, with AFINN as the default.

### Word Correlation

Word correlation provides a measure of relation between two words, typically based on their existence in sentences together. This is the word-level insight that is the most difficult-to-implement in an efficient manner, due

to the requirements that the dataframe remains tidy and lossless. It would be impractical to have as the columns every word with every correlation score to every other word. The solution conceived of for this is by taking input from the user to specify words, giving a vector of correlations between every other word in the text and the specified word, with element-wise correspondence between words in the text and the correlation coefficients. The best form of visualisation would be individual words with their scores, a correlation matrix for some words, or a table and search. `widyr` implements a `widyr::pairwise_cor()` function, however it appears to determine the correlations of every single word with each other, creating a far larger object than what is needed[42]. Implementation will likely have to be performed from scratch in the future, unless an appropriate package is found.

### 3.4.3 Wrapper

The insights of choice can all be combined into a wrapper function `get_insight()` (Listing 19), taking the forms and arguments of the insights and applying those chosen. Two separate functions were created for the careful handling of term and aggregate insight. Initially these functions made use of an `eval(parse())` style of evaluation, however this proved to be too unclear in communicating the function capabilities, and was changed to a table of functions that could be called. The loss in power was justified by the gain in clarity.

The losslessness of the data structure created finds it's strengths in the function `bind_aggregate()` (Listing 20), which recreates the original text structure based on some aggregation, such as sentence ID.

## 3.5 Visualisation

Visualisation is extremely useful in the communication of text analysis, and often forms the end-point of much of the automated analysis. Development has grouped visualisations by their output intention, rather than by their implementation, as part of the single-responsibility principle, to be ends-focused rather than means-focused. All of the visualisations in this package make heavy use of `ggplot2`, due to the ease of facetting plots based on grouping variables. The "Grammar of Graphics", which ggplot is based upon, has not necessarily proved any more useful than the base `R` plots for development purposes, with ease of facetting and the excellent compatibility with Shiny being the principal reasons for use.

Visualisation in the app consists of a full tab including options and output (Fig. 3.8). Options vary by type of insight selected, for example n-grams having unique options related, as in Fig. 3.9.

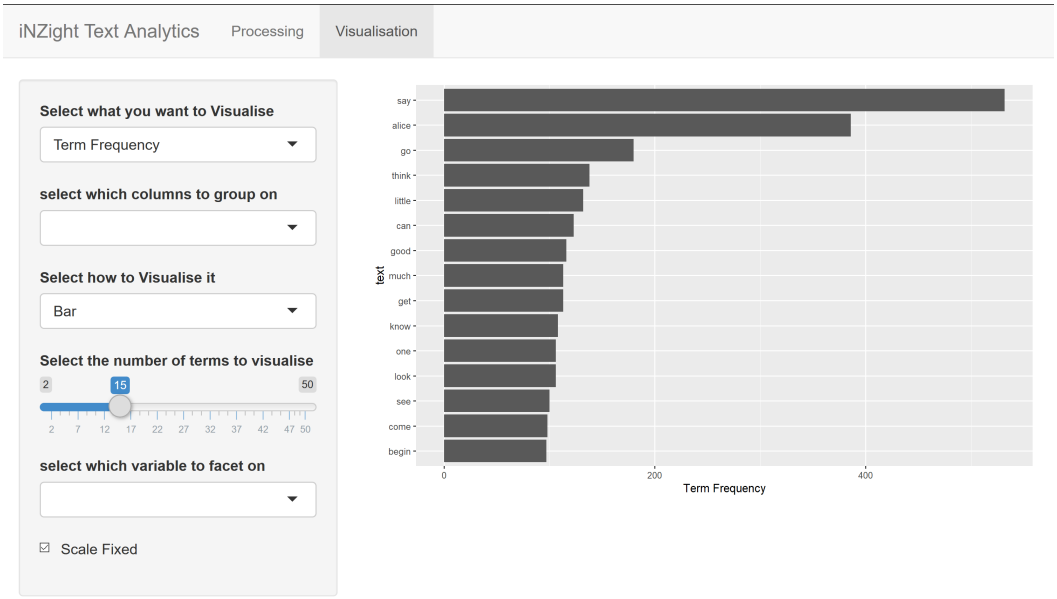



Figure 3.8: Visualisation screen

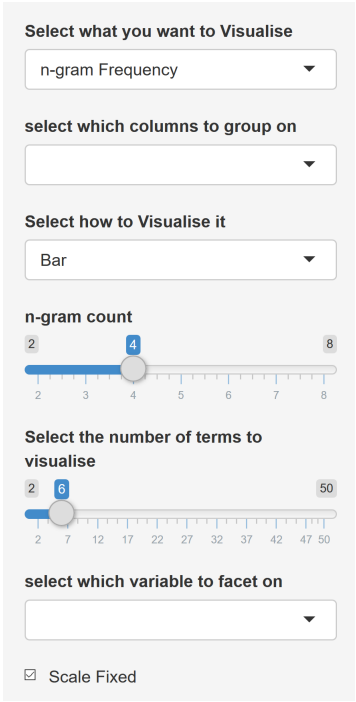



Figure 3.9: Options shown when n-gram frequency is selected.

### 3.5.1 Score

Most insight functions return some numeric measure, typically used as a score. Visualisations that illustrate the score directly fit into the "score" category. Such visualisations are typically the most useful in attaining clear analytic results.

#### Bar Plot

Bar plots give the most direct demonstration of scoring for text, allowing for clear comparisons between terms. A function, `score_barplot()` (Listing 21) has been defined to create bar plots for the data type resulting from the insight functions. An issue with the implementation of `ggplot2::geom_bar()` is the reliance on factor levels to provide ordering, which may provide incorrect ordering when facetting, as factor levels are fixed at a global level for a vector. `tidytext::reorder_within()` purports to solve this problem, though it has not been used due to the nonsensical coupling of such a function with the package, and the need to remove dependencies on the `tidytext` package.

#### Word Cloud

Word clouds depict a random layout of words of varying sizes reflecting their scores, and a function named `score_wordcloud()` (Listing 22) has been created to implement them. They are only about as useful as pie charts for putting across any more than a rough idea of what the highest scoring items are. Despite this, they are an oft-requested form of visualisation, as they really do look good. Building this functionality in was a guilty pleasure, and the package used to implement wordclouds, ggwordcloud, is exceedingly well made[43]. An in-app example is given in Fig. 3.10.

The option is given for a variety of shapes for the word cloud to take. Future development will allow for a bright range of colours in the text of the word cloud, to be more aesthetically pleasing.

### 3.5.2 Distribution

The distribution of scores are interesting for their own sake in text analysis, providing an informative summary of scores over the entirety of a text. Distributions are particularly enlightening when comparing between groups, when facetted. Over a large dataset that free-form survey responses may provide, it is useful to calculate the mean sentiment for each response, and assess the resulting distribution — this program provides such capabilities.

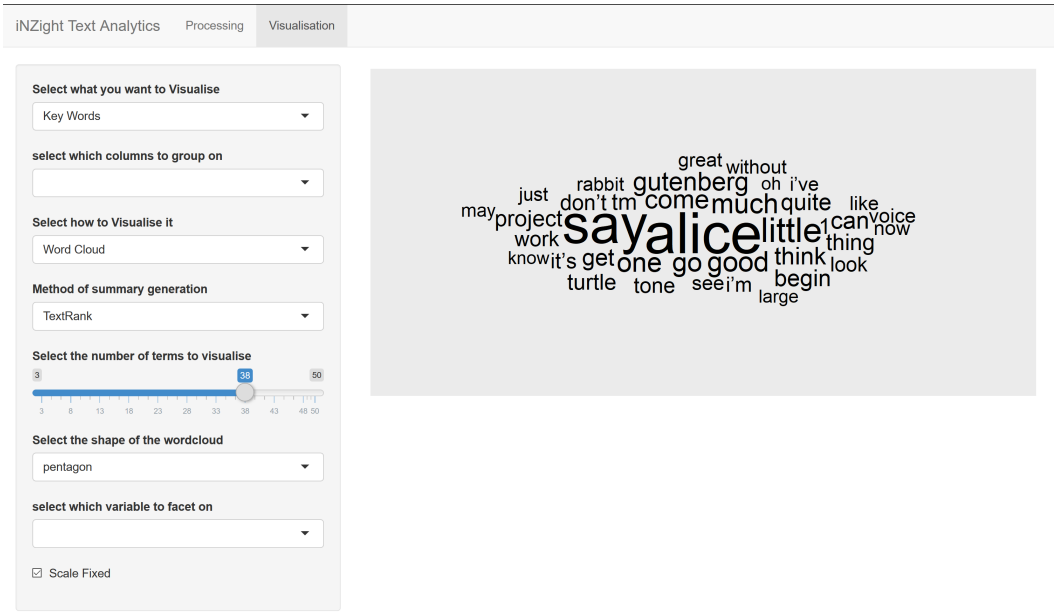



Figure 3.10: Key words visualised with Word Cloud

### Histogram

Histograms show a discrete distribution of some score. They are particularly useful for discrete scores, such as the kind produced by an AFINN sentiment analysis (Section 3.4.1 and Fig. 3.11). A function, `dist_hist()` (Listing 23), was created for visualisation in the form of a histogram.

### Density

To visualise the density estimation as a continuous distribution for some score, a function named `dist_density()` (Listing 24) was defined in a similar form to the histogram visualisation function. Density visualisations work particularly well for the statistics given by aggregate sentiment analyses (Section 3.4.2). In the application, they are treated as any other visualisation, just requiring selection from a drop-down menu, as in Fig. 3.12.

## 3.5.3 Structure

The structure of the text holds information that no individual piece of text holds by itself. The evolution of certain characteristics over time, as well as patterns in the text, can be brought to light with visualisations of the structure.


Figure 3.11: Term Sentiment visualised with histograms, facetted by chapter

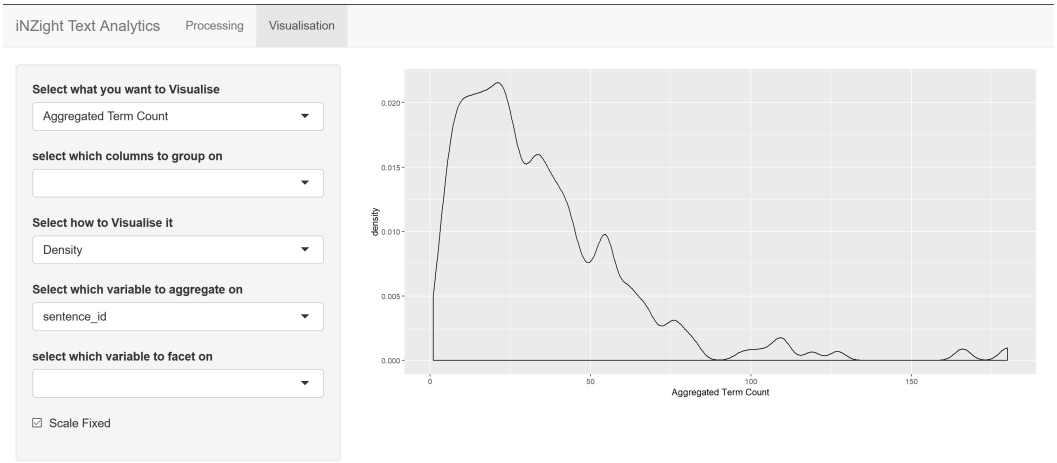



Figure 3.12: Aggregate Term Count over sentences visualised as a density

#### Time Series

Treating text as a time series, with each term representing a unit of time, allows for a running summary of the text from beginning to end. A function named `struct_time_series()` (Listing 25) was composed to capture this form of visualisation. Time series plots pair particularly well with a moving-average term-sentiment, as described in Section 3.4.1 (Listing 15).

#### Page View

The concept and function implementation (`struct_pageview()` (Listing 26) of page view is based on the visualisation format of the package `ggpage`. The intention is for each word in the original text to be represented as a rectangle on a page, with a width proportional to the word length. The fill of the rectangle may reflect scores, or even some category. Future development will have the original text superimposed over the word placeholders, with the ability to search through the text. Page view is set as the default for visualising aggregate and term sentiment in the application (Fig. 3.13).

Different colour palettes will also bring clarity to the visualisation, with only sequential colour palettes being currently implemented.

The `R` package `ggpage` is set up for making immediate plots, but by using the package's constructor `ggpage_build()` and `ggpage_plot()`, complex functions can be formed in the immediate representation from build before plotting[18]. Facetting is an issue, with the standard implementation lacking clear support.

Creating this function was the initial impetus for having a lossless data structure; once stopwords were removed, they were still desired to be represented in the structure of the text, but with no analytics performed on them. This was achieved in the program, with stopwords taking space, but having no coloured fill.

### 3.5.4 Wrapper

Visualisation is performed through a wrapper function `get_vis()` (Listing 27), that can perform the necessary facetting and changes to scaling. The special case of page view requires additional facetting considerations.

## 3.6 Application

The Shiny web app (Listing 28) is the frontpiece of the whole program. In about 300 lines of code, it defines both the interface and the server functions, which call my R package. As with all shiny applications, it consists of a UI

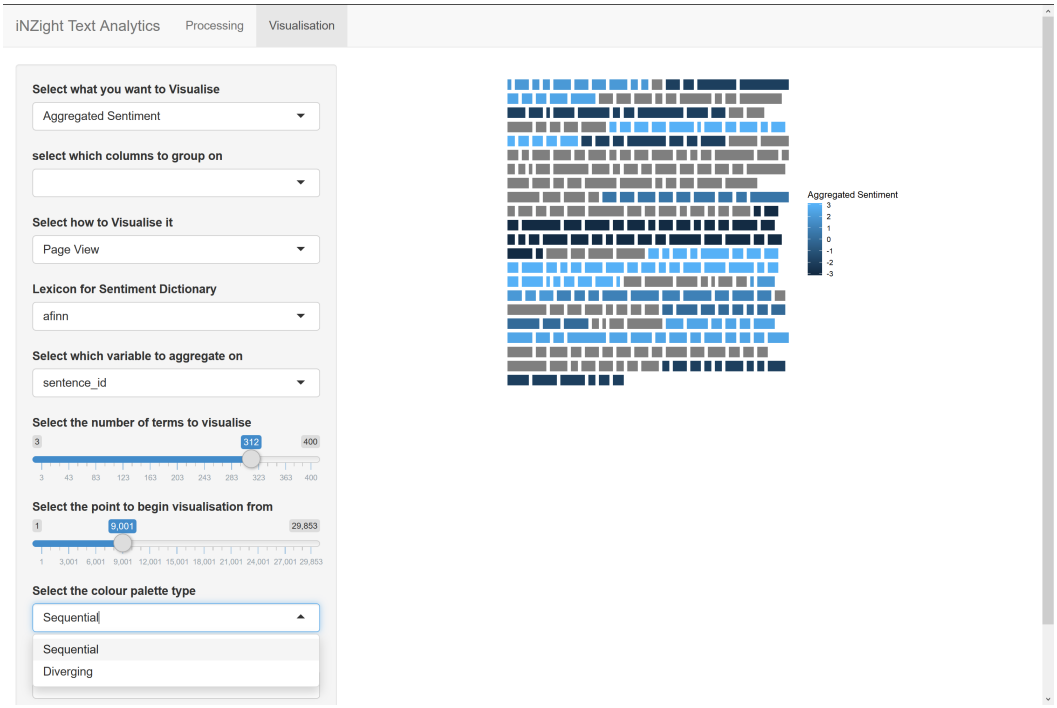



Figure 3.13: Aggregate Sentiment visualised with Page View

front-end that is rendered in a web browser for the user to interact with, and a server back-end that performs the analytic work.

The app consists of two main tabs: processing, and visualisation. The processing tab is the first screen encountered by a user, allowing them all of the processing functions offered by the package, in a GUI interface. Once one or more text files have been imported, lemmatised, sectioned, etc. in the processing tab, the user moves to visualisation, presented immediately with a default plot from a default analysis. The user may select a different analysis from a drop-down list of analyses, which call from the insight functions in the package, automatically and intelligently. Each analysis has a default set of options and a visualisation associated with it, so that as soon as a choice has been made to visualise some text-construct, a plot instantly appears. Alternative visualisations can be selected with a drop down menu.

Additional options, for the control of both the analysis and visualisation, dynamically appear, depending on the type of insight and associated visualisation. After an insight is selected, the server generates options related to the insight for the UI, as shown in Table 3.4. Similarly, when a visualisation is selected, the server generates options related to the visualisation for the UI, as shown in Table 3.5.

| Insight | Insight Options | Default Options | Default Visualisation |
|---|---|---|---|
| Term frequency | — | | Bar plot |
| n-gram frequency | n-gram count | 2 (bigram) | Bar plot |
| Key words | Method of summary generation | TextRank | Bar plot |
| Term sentiment | Sentiment lexicon | AFINN | Page view |
| Moving average term sentiment | Moving average lag | 50 | Time series |
| | Sentiment lexicon | AFINN | |
| Aggregated term count | Variable to aggregate on | Generated from input text | Bar plot |
| Key sections | Method of summary generation | LexRank | Bar plot |
| | Variable to aggregate on | Generated from input text | |
| Aggregated sentiment | Sentiment lexicon | AFINN | Page view |
| | Variable to aggregate on | Generated from input text | |

Table 3.4: Dynamically generated UI for insight options

| Visualisation | Visualisation Options | Default Options |
|---|---|---|
| Word Cloud | Number of terms to visualise | 15 |
| | Shape of word cloud | Circle |
| Page Views | Number of terms to visualise | 100 |
| | Point in text to begin visualisation from | 1 |
| | Colour Palette type | Sequential |
| Bar plot | Number of terms | 15 |
| Time Series | — | |
| Density | — | |
| Histogram | — | |

Table 3.5: Visualisations and their options

All visualisations are additionally affected by the option of facetting by some variable from the input text.

The application has undergone a rebuild over the course of the project, starting with an app mirroring the logic of the underlying text analytics package closely, before moving to a more user-centric design for interaction.

Originally, the app required a user to perform the preparation stages, then select what insight functions to perform, before moving on to visualisation. This was a programmer-centric manner of interaction, being imperative in the sense that the user had to specify how to perform the analysis.

This was changed dramatically to be more declarative: The user performs whatever preparation is necessary, then states what analysis they want to see, and possibly how they want to see it, and the app determines what has to happen in the background to deliver on this request. This is effectively automatic visualisation based on the insight selection, which is closer to the behaviour and philosophy of the main iNZight program.

Using a reactive framework in the creation of the app has led to several difficulties, most notably in the lack of state held by the central object. To have some capacity for iterative analysis, going from preparation to visualisation, several successive variables have been used to pass one logically prior output to the next, with the final variable being used for the display.

# Chapter 4

# Conclusion

## 4.1 Recommendations

### 4.1.1 Future Development

The next step for this project is to add several additional insight functions, such as tf-idf scoring, to facilitate "topic modelling". Following this, integration into the main iNZight application can take place, which won't be as arduous as project integration typically is, as this program and iNZight both make use of the same Shiny web app framework.

This project has brought to light many workflows and technologies that if made use of, would improve the application substantially. As hinted at in Section 3.1.3, ideally development would be based around the creation of several carefully considered classes using R's S4 object oriented system, with a few basic generic functions for the classes. The functional paradigm has proved remarkably useful in providing a program that is easy to reason about, and immutability aids this especially. There is nothing about this paradigm that precludes the creation of other classes, as Haskell aptly demonstrates[44]. This move away from a tidy paradigm would also involve the jettisoning of the residual dependencies on the `tidytext` package, and working directly with the external packages wrapped by `tidytext` instead.

An object-oriented approach will also allow for further losslessness in the data. For instance, the behaviour of filtering results in a subset of the input dataframe, while an object may have methods defined that simply "mask" whatever meets the filtering predicates — hiding, instead of deleting, thereby allowing for retrieval later if needed.

A realistic development that will be easy to implement given some time is support for in-program access to online text sources such as Project Guten-

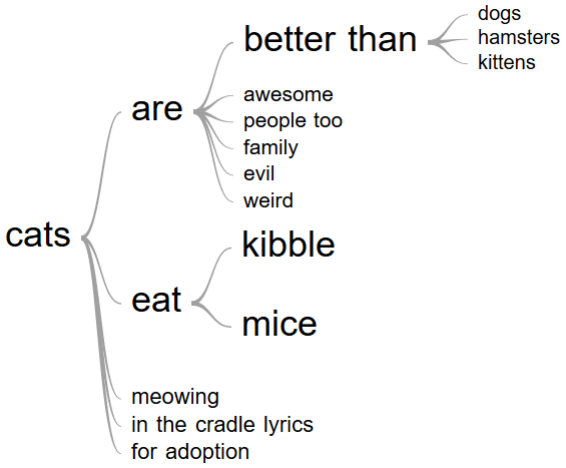



Figure 4.1: Common word orderings of sentences visualised with word trees through the googleVis package

berg and Twitter, as forms of general online text extraction. These would be included as more text import formats are allowed. In support of the additional text complexity, options for more complex model-based analyses will be easy to swap in for the current dictionary-based analyses, using additional arguments to existing functions that trigger in-function calls to the model-based analyses. Included in this bundle of work would be more processing options, including those for existing import formats; for example, allowing user specification of the text column in a `csv` file.

Greater user control over components of lexicons would be a worthwhile pursuit, especially in the specification of stopwords and sentiment. User-added stopwords and sentiments would provide great utility, alongside user-specified deletions of words from the base lexicons. This should be performed in a way that makes it as simple as possible; tools to aid the user could be a clickable table of common words, to delete or add to the lexicon.

More visualisation of words in context would be well-received by users. The ability to search for words and visualise that search in some form would aid this. A specific implementation of this could make use of word trees provided by Google Charts, through the googleVis[45] package, which produces network diagrams of common word orderings (an example depicted in Fig. 4.1). The network diagrams show the common orderings of words around the searched word, sizing and arranging according to frequency. Another option is to make use of "concordance" tables, which are made heavy use of in corpus linguistics (Fig. 4.2). Concordance tables find every sentence that the searched word or phrase appears in, creating a table of these sentences, aligned with the search at the centre of the table.

88 5 Applied Character String Processing

Table 5.1: A sample concordance of *alphabetic* and *alphabetical* in the BNC

| # | left context | word | right context |
|---|---|---|---|
| 1 | specify the equation press the end key which is between the | alphabetic | and the numeric key pads , that will then submit that request |
| 2 | an integer between 0 and 9 , and A is an | alphabetic | character . # NISS No issue has been specified for the referenced |
| 3 | particular the notion that a word is a contiguous sequence of | alphabetic | characters . Amsler shows that this notion ignores important |
| 4 | ... It was only in the days of the first widespread | alphabetic | culture that the idea of ' logic ' appears to have arisen |
| 5 | records than for signs . In most languages with writing systems | alphabetic | fingerspelling has been available for over two hundred years . |
| 6 | of icebreakers when she noticed that the control panel had an | alphabetic | keyboard. Can't hurt to try , she thought , unjacking |
| 7 | for the widespread development of the particular form of | alphabetic | literacy evident in Greece must clearly be sought in the social |
| 8 | Retrieval from a conventional filing system relies on either | alphabetic | or numeric indexation but the use of a computer enables data to |
| 9 | on . Fingerspelling as it exists today consists of a direct | alphabetic | representation of the language as it would be written down . In |
| 10 | describe . According to Goody and Watt , the development of | alphabetic | script and its wide diffusion throughout society – the two |
| 11 | individuals or items which can be symbolized by a number or | alphabetical | abbreviation . For example , 929 Schil Biography of Schiller 820 |
| 12 | Filofax – but you probably do need a college diary or | alphabetical | address book with space for telephone numbers , etc . Finally , |
| 13 | A to Z OF FISH HEALTH # JERZY GAWOR continues his | alphabetical | advice on fish health problems . # Top : Tablet food forms |
| 14 | this context . Abish 's use of arbitrary formal limits in | Alphabetical | Africa ( 1974 ) ' becomes the vehicle for an adventurous plot |
| 15 | index to post code directories. # 5. # Telephone directories – | alphabetical | and classified . # 6. # Telephone Dialling Codes – an essential |
| 16 | in order to arrange it . A classified rather than an | alphabetical | approach was necessary in the index because an internationally |
| 17 | jazz titles from early this and late last year , in | alphabetical | artist order . Old faces mix with the new but all have |
| 18 | when I sell it again I can put the type in | alphabetical | ascending order (SP:PS0H9) Mm (SP:PS0H9) so it 'll be |
| 19 | . These six should ideally be selected at random from an | alphabetical | class list . The observer must ensure that the six named in |
| 20 | brother , Master Richard of Stainby , author of a revised | alphabetical | concordance to the Bible , and the great scholar , Alexander of |

Figure 4.2: Concordance table of a search for "alphabetic"[1]

Another future possibility for development is parallelisation; with the large size of some text documents, processing time can be dramatically improved through transforming the text in parallel. The functional style aids this, as threading is not required to be consciously determined, as packages exist to transform existing *apply functions into parallel form.

### 4.1.2 Educational Potential

The application developed has enormous educational potential for aiding in learning the basics of text analytics for a very broad range of students and disciplines. This is primarily due to the structured approach to text analytics captured in the preparation-insight-visualisation cycle.

This cycle occurs in a rapid procession in-app, thereby not getting in the way between learning the theory and seeing the results of an analysis. The rapid speed of results also allows for experimentation, which encourages curiosity and reinforces an iterative approach to analysis.

Finally, while the GUI interface has the natural limitation of non-scriptability by the end-user, it is far more intuitive for someone trying to learn text analytics without the cognitive overhead of having to learn text analytics package API's, the `R` language, or concepts of programming in order to perform such analyses.

## 4.2 Summary

This project conceptualised and delivered a user-friendly application with which to perform text analytics, providing powerful processing, insight, and visualisation functionality. The process of creating the application led to the development of an `R` package, serving as the backend to the applications Shiny GUI. The program is intended to integrate with the existing iNZight application, and should be a significant educational asset.

### 4.2.1 Closing Remarks

This project has taught me an enormous amount.

Naturally, I have learnt a great deal about text analytics, a field that I knew nothing about prior to commencing this project. Considering text analytics from a statistical perspective has led to new insights for me in the application of statistical methodology and procedures.

My programming abilities have improved greatly from seeing through a highly non-trivial application end-to-end. The added constraint of following a functional paradigm has challenged me and forced me to think more creatively in providing solutions. Working with the number of libraries I have has taught me some (sometimes painful) lessons in dependency management and version control.

Communication was essential to the success of this project, and I also learnt a lot about effectively communicating technical information to people with varying levels of field-specific knowledge.

# Appendices

# Appendix A

# Listings

This appendix chapter includes all of the source code needed to build the package and application. There are live links in the pdf file connecting the main discussion to individual listings.

Listing 2: Import `txt`

```
#' Import text file
#'
#' @param filepath a string indicating the relative or absolute
#'     filepath of the file to import
#'
#' @return a [tibble][tibble::tibble-package] of each row
#'   corrresponding to a line of the text file, with the column named
#'   "text"
import_txt <- function(filepath){
  paste(readLines(filepath), collapse = "\n") %>%
    tibble::tibble(text=.)
}
```

Listing 3: Import `csv`

```
#' Import csv file
#'
#' @param filepath a string indicating the relative or absolute
#'     filepath of the file to import
#'
#' @return a [tibble][tibble::tibble-package] of each row
#'   corrresponding to a line of the text file, with the column named
#'   "text"
import_csv <- function(filepath){
  readr::read_csv(filepath) ## %>%
    ## table_textcol()
}
```

Listing 4: Import excel

```
#' Import excel file
#'
#' @param filepath a string indicating the relative or absolute
```

```
#'     filepath of the file to import
#'
#' @return a [tibble][tibble::tibble-package] of each row
#'     corrresponding to a line of the text file, with the column
#'     named "text"
import_excel <- function(filepath){
  readxl::read_excel(filepath) ## %>%
    ## table_textcol()
}
```

Listing 5: Import files

```
#' Base case for file import
#'
#' @param filepath string filepath of file for import
#'
#' @return imported file with document id
import_base_file <- function(filepath){
  filetype <- get_filetype(filepath)
  filename <- basename(filepath)
  if (filetype == "csv"){
    imported <- import_csv(filepath)
  } else if (filetype == "xlsx" | filetype == "xls") {
    imported <- import_excel(filepath)
  } else {
    imported <- import_txt(filepath)
  }
  imported %>%
    dplyr::mutate(doc_id = filename)
}

#' Import any number of files
#'
#' @param filepaths char vector of filepaths
#'
#' @return a [tibble][tibble::tibble-package] imported files with
#'   document id
#'
#' @export
import_files <- function(filepaths){
  filepaths %>%
    purrr::map(import_base_file) %>%
    dplyr::bind_rows()
}
```

Listing 6: Prepare text

```
text_prep <- function(data){
    data %>%
      tidytext::unnest_tokens(output = sentence, input = text,
                              token = "sentences", to_lower = FALSE) %>%
      dplyr::mutate(sentence_id = dplyr::row_number()) %>%
      dplyr::group_by(sentence_id, add=TRUE) %>%
      dplyr::group_modify(~ {
          .x %>%
              tidytext::unnest_tokens(output = word, input = sentence,
                                      token = "words", to_lower=FALSE) %>%
              dplyr::mutate(word_id = dplyr::row_number())
      }) %>%
      ungroup_by("sentence_id")
}
```

Listing 7: Manage stopwords

```
#' Gets stopwords from a default list and user-provided list
#'
#' @param lexicon a string name of a stopword list, one of "smart",
#'     "snowball", or "onix"
#'
#' @param addl user defined character vector of additional stopwords,
#'     each element being a stopword
#'
#' @return a [tibble][tibble::tibble-package] with one column named "word"
get_sw <- function(lexicon = "snowball", addl = NA){
  addl_char <- as.character(addl)
  tidytext::get_stopwords(source = lexicon) %>%
    dplyr::select(word) %>%
    dplyr::bind_rows(., tibble::tibble(word = addl_char)) %>%
    stats::na.omit() %>%
    purrr::as_vector() %>%
    tolower() %>%
    as.character()
}

#' determine stopword status
#'
#' @param .data vector of words
#'
#' @param ... arguments of get_sw
#'
#' @return a [tibble][tibble::tibble-package] equivalent to the input
#'   dataframe, with an additional stopword column
#'
#' @export
determine_stopwords <- function(.data, ...){
  sw_list <- get_sw(...)
  .data %in% sw_list
}
```

Listing 8: Format data

```
#' takes imported one-line-per-row data and prepares it for later analysis
#'
#' @param .data tibble with one line of text per row
#'
#' @param lemmatize boolean, whether to lemmatize or not
#'
#' @param stopwords boolean, whether to remove stopwords or not
#'
#' @param sw_lexicon string, lexicon with which to remove stopwords
#'
#' @param addl_stopwords char vector of user-supplied stopwords
#'
#' @return a [tibble][tibble::tibble-package] with one token per line,
#'   stopwords removed leaving NA values, column for analysis named
#'   "text"
#'
#' @export
format_data <- function(.data, lemmatize=TRUE, stopwords=TRUE,
                        sw_lexicon="snowball", addl_stopwords=NA){
  formatted <- .data %>%
    text_prep()

  text <- ifexp(lemmatize,
```

```
                ifexp(stopwords,
                      dplyr::mutate(formatted,
                                    lemma = tolower(textstem::lemmatize_words(word)),
                                    stopword = determine_stopwords(lemma,
                                                                   sw_lexicon,
                                                                   addl_stopwords),
                                    text = dplyr::if_else(stopword,
                                                          as.character(NA),
                                                          lemma)),
                      dplyr::mutate(formatted,
                                    lemma = tolower(textstem::lemmatize_words(word)),
                                    text = lemma)),
                ifexp(stopwords,
                      dplyr::mutate(formatted,
                                    stopword = determine_stopwords(word,
                                                                   sw_lexicon,
                                                                   addl_stopwords),
                                    text = dplyr::if_else(stopword,
                                                          as.character(NA),
                                                          word)),
                      dplyr::mutate(formatted, text = word)))
  return(text)
}
```

Listing 9: Detect and add sections

```
#' creates a search closure to section text
#'
#' @param search a string regexp for the term to seperate on, e.g. "Chapter"
#'
#' @return closure over search expression
get_search <- function(search){
  function(.data){
    .data %>%
      stringr::str_detect(search) %>%
      purrr::accumulate(sum, na.rm=TRUE)
    }
}

#' sections text based on chapters
#'
#' @param .data vector to section
#'
#' @return vector of same length as .data with chapter numbers
#'
#' @export
get_chapters <- get_search("^[\\s]*[Cc][Hh][Aa]?[Pp][Tt]([Ee][Rr])?")

#' sections text based on parts
#'
#' @param .data vector to section
#'
#' @return vector of same length as .data with part numbers
#'
#' @export
get_parts <- get_search("^[\\s]*[Pp]([Aa][Rr])?[Tt]")

#' sections text based on sections
#'
#' @param .data vector to section
#'
#' @return vector of same length as .data with section numbers
```

```
#'
#' @export
get_sections <- get_search("^[\\s]*([Ss][Ss])|([Ss][Ee][Cc][Tt][Ii][Oo][Nn])")

#' sections text based on cantos
#'
#' @param .data vector to section
#'
#' @return vector of same length as .data with canto numbers
#'
#' @export
get_cantos <- get_search("(?i)canto (XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$")

#' sections text based on book
#'
#' @param .data vector to section
#'
#' @return vector of same length as .data with book numbers
#'
#' @export
get_books <- get_search("(?i)book$")

#' Adds section column to dataframe
#'
#' @param .data dataframe formatted as per output of prep process
#'
#' @param section_by character name of what to section over
#'
#' @return input dataframe with additional section column
#'
#' @export
section <- function(.data, section_by){
    sec_table <- list("chapter" = get_chapters,
                      "part" = get_parts,
                      "section" = get_sections,
                      "canto" = get_cantos,
                      "book" = get_books)
    .data %>%
        dplyr::mutate(!! section_by := sec_table[[section_by]](word))
}
```

Listing 10: Determine Term Frequencies

```
#' Determine term frequency
#'
#' @param .data character vector of terms
#'
#' @return numeric vector of term frequencies
#'
#' @export
term_freq <- function(.data){
  .data %>%
    tibble::enframe() %>%
  dplyr::add_count(value) %>%
  dplyr::mutate(n = dplyr::if_else(is.na(value),
                      as.integer(NA),
                      n))  %>%
  dplyr::pull(n)
}
```

Listing 11: Get n-grams

```r
#' Returns the n-grams, skipping NA values
#'
#' @param .data vector to get n-grams from
#'
#' @param n number of n-grams to attain
#'
#' @return n-gram vector without NA values
#'
#' @export
get_ngram <- function(.data, n){
    main_n <- n
    ngrams <- rep(NA_character_, length(.data))
    for (i in seq_along(.data)){
        if (i - 1 >  length(.data) - n) break
        if (is.na(.data[i])){
            next
        } else {
            ngram <- .data[i]
        }
        j <- i + 1
        while(n > 1){
            if (j >  length(.data)){
                ngram <- NA_character_
                break
            }
            if (is.na(.data[j])){
                j <- j + 1
            } else {
                ngram <- paste(ngram, .data[j])
                n <- n - 1
                j <- j + 1
            }
        }
        ngrams[i] <- ngram
        n <- main_n
    }
    return(ngrams)
}
```

Listing 12: Get n-grams frequencies

```r
#' NOT FOR PRODUCTION - STILL IN TESING. Returns the count of n-grams, skipping NA values
#'
#' @param .data vector to get n-grams from
#'
#' @param n number of n-grams to attain
#'
#' @return count of each associated n-gram
#'
#' @export
ngram_freq <- function(.data, n){
  term_freq(get_ngram(.data, n))
}
```

Listing 13: Determine Key Words with Textrank

```r
#' Determine textrank score for vector of words
#'
#' @param .data character vector of words
#'
```

```
#' @param summ_method method to use for summarisation: textrank or
#'     lexrank. Doesn't do anything yet
#'
#' @return vector of scores for each word
#'
#' @export
keywords_tr <- function(.data, summ_method){
  relevent <- !is.na(.data)
  tr <- textrank::textrank_keywords(.data, relevent, p=+Inf)
  score <- tr$pagerank$vector %>% tibble::enframe()
  data <- .data %>% tibble::enframe("number", "name")
  dplyr::full_join(data, score, by="name") %>%
    dplyr::pull(value)
}
```

Listing 14: Determine Term Sentiments

```
#' Determine sentiment of terms
#'
#' @param .data vector of terms
#'
#' @param lexicon sentiment lexicon to use, based on the corpus
#'   provided by tidytext
#'
#' @return vector with sentiment score of each word in the vector
#'
#' @export
term_sentiment <- function(.data, lexicon="afinn"){
  data <- tibble::enframe(.data, "number", "word")
  tidytext::get_sentiments(lexicon) %>%
    dplyr::select(word, value) %>%
    dplyr::right_join(data, by="word") %>%
    dplyr::pull(value)
}
```

Listing 15: Determine the Moving Average Term Sentiment

```
#' Determine the lagged sentiment of terms
#'
#' @param .data vector of terms
#'
#' @param lexicon sentiment lexicon to use, based on the corpus
#'     provided by tidytext
#'
#' @param lag how many (inclusive) terms to compute statistic over
#'
#' @param statistic base statistic used to summarise the data, capable
#'     of taking an na.rm argument
#'
#' @return vector with lagged sentiment score of each term in the input vector
#'
#' @export
ma_term_sentiment <- function(.data, lexicon="afinn", lag = 10, statistic = mean){
    sents <- term_sentiment(.data, lexicon)
    ## lagged_sents <- rep(NA, length(sents))
    ## for (i in seq(lag, length(sents))){
    ##     lagged_sents[i] <- statistic(sents[(seq(i - lag + 1, i))], na.rm = TRUE)
    ## }
    ## lagged_sents
    c(rep(NA, lag - 1),
      sapply(seq(lag, length(sents)),
```

```
            function(i){x <- statistic(sents[(seq(i - lag + 1, i))],
                                       na.rm = TRUE)
                                       ifelse(is.nan(x),
                                              NA,
                                              x)
            }))
}
```

Listing 16: Determine the term count over some aggregate

```
#' Determine the number of terms at each aggregate level
#'
#' @param .data character vector of terms
#'
#' @param aggregate_on vector to split .data on for insight
#'
#' @return vector of number of terms for each aggregate level, same
#'   length as .data
#'
#' @export
term_count <- function(.data, aggregate_on){
  split(.data, aggregate_on) %>%
    purrr::map(function(x){rep(length(x), length(x))}) %>%
    dplyr::combine()
}
```

Listing 17: Determine the Key Sections

```
#' get score for key sentences as per Lexrank
#'
#' @param .data character vector of words
#'
#' @param summ_method method to use for summarisation: textrank or
#'     lexrank. Doesn't do anything yet
#'
#' @param aggregate_on vector to aggregate .data over; ideally, sentence_id
#'
#' @return lexrank scores of aggregates
#'
#' @export
key_aggregates <- function(.data, aggregate_on, summ_method){
  ## prepare .data for lexrank
  base <-  tibble::tibble(word = !! .data, aggregate = aggregate_on)
  aggregated <- base %>%
    dplyr::group_by(aggregate) %>%
    stats::na.omit() %>%
    dplyr::summarise(sentence = paste(word, collapse = " ")) %>%
    dplyr::mutate(sentence = paste0(sentence, "."))
  ## lexrank
  lr <- aggregated %>%
    dplyr::pull(sentence) %>%
    lexRankr::lexRank(., n=length(.),removePunc = FALSE, returnTies = FALSE,
            removeNum = FALSE, toLower = FALSE, stemWords = FALSE,
            rmStopWords = FALSE, Verbose = TRUE)
  ## match lexrank output to .data
  lr %>%
    dplyr::distinct(sentence, .keep_all = TRUE) %>%
    dplyr::full_join(aggregated, by="sentence") %>%
    dplyr::full_join(base, by="aggregate") %>%
    dplyr::arrange(aggregate) %>%
    dplyr::pull(value)
```

```
}
```

Listing 18: Determine the Aggregate Sentiments

```
#' Get statistics for sentiment over some group, such as sentence.
#'
#' @param .data character vector of words
#'
#' @param aggregate_on vector to aggregate .data over; ideally,
#'   sentence_id, but could be chapter, document, etc.
#'
#' @param lexicon as per term sentiment
#'
#' @param statistic function that accepts na.rm argument; e.g. mean,
#'   median, sd.
#'
#' @return sentiment of same length as input vector aggregated over the aggregate_on vector
#'
#' @export
aggregate_sentiment <- function(.data, aggregate_on, lexicon = "afinn", statistic = mean){
  tibble::enframe(.data, "nil1", "word") %>%
    dplyr::bind_cols(tibble::enframe(aggregate_on, "nil2", "aggregate")) %>%
    dplyr::select(word, aggregate) %>%
    dplyr::mutate(sentiment = term_sentiment(word, lexicon)) %>%
    dplyr::group_by(aggregate) %>%
    dplyr::mutate(aggregate_sentiment =
                    (function(.x){
                      rep(statistic(.x, na.rm = TRUE), length(.x))
                    })(sentiment)) %>%
    dplyr::pull(aggregate_sentiment)
}
```

Listing 19: Insight functions wrapper

```
#' perform group-aware term operations on the data
#'
#' @param .data dataframe of terms as per output of format_data
#'
#' @param operations character vector of term operations to perform
#'
#' @param ... additional arguments to the operation - only sensible for singular operations
#'
#' @return .data with operation columns added
#'
#' @export
get_term_insight <- function(.data, operations, ...){
    opstable <- list("Term Frequency" = term_freq,
                     "n-gram Frequency" = ngram_freq,
                     "n-grams" = get_ngram,
                     "Key Words" = keywords_tr,
                     "Term Sentiment" = term_sentiment,
                     "Moving Average Term Sentiment" = ma_term_sentiment)
    ops <- opstable[operations]
    lapply(seq(length(ops)),
           function(x){
               name <- dplyr::sym(names(ops[x]))
               operation <- ops[x][[1]]
               df <- dplyr::mutate(.data,
                                   !!name := operation(text, ...))
               df[names(ops[x])]
           }) %>%
```

```
        dplyr::bind_cols(.data, .)
}

#' perform group-aware aggregate operations on the data
#'
#' @param .data dataframe of terms as per output of format_data
#'
#' @param operations character vector of operations to perform
#'
#' @param aggregate_on character name of the column to perform aggregate operations on
#'
#' @param ... additional arguments to the operation - only sensible for singular operations
#'
#' @return .data with operation columns added
#'
#' @export
get_aggregate_insight <- function(.data, operations, aggregate_on, ...){
    opstable <- list("Aggregated Term Count" = term_count,
                     "Key Sections" = key_aggregates,
                     "Aggregated Sentiment" = aggregate_sentiment,
                     "Bound Aggregates" = bind_aggregation)
    ops <- opstable[operations]
    lapply(seq(length(ops)),
           function(x){
               name <- dplyr::sym(names(ops[x]))
               operation <- ops[x][[1]]
               agg_on <- dplyr::sym(aggregate_on)
               df <- if (names(ops[x]) == "Bound Aggregates"){
                         dplyr::mutate(.data,
                                       !!name := operation(word, !! agg_on))
                     } else {
                         dplyr::mutate(.data,
                                       !!name := operation(text, !! agg_on, ...))
                     }
               df[names(ops[x])]
           }) %>%
        dplyr::bind_cols(.data, .)
}
```

Listing 20: Bind aggregate terms

```
#' bind aggregate terms together
#'
#' @param data vector of terms
#'
#' @param aggregate_on vector of aggregations
#'
#' @return data with every aggregation bound, as in a sentence
#'
#'
bind_aggregation <- function(data, aggregate_on){
    tibble::tibble(data, agg = aggregate_on) %>%
        dplyr::group_by(agg) %>%
        dplyr::mutate(bound = paste(data, collapse = " ")) %>%
        dplyr::pull(bound)
}
```

Listing 21: Create Bar Plot

```
#' output a ggplot column graph of the top texts from some insight function
#'
```

```
#' @param .data a dataframe containing "text" and insight columns as
#'     per the output of the get_(term|aggregate)_insight wrapper
#'     function
#'
#' @param y symbol name of the column insight was
#'     outputted to
#'
#' @param x symbol name of column for insight labels
#'
#' @param n number of bars to display
#'
#' @param desc bool: show bars in descending order
#'
#' @export
score_barplot <- function(.data, y, n = 15,
                                x = text, desc = FALSE){
    wrap <- 50
    text <- dplyr::enquo(x)
    insight_col <- dplyr::enquo(y)
    .data %>%
        dplyr::distinct(!! text, .keep_all=TRUE) %>%
        dplyr::arrange(dplyr::desc(!! insight_col)) %>%
        dplyr::group_modify(~{.x %>% head(n)}) %>%
        dplyr::ungroup() %>%
        dplyr::mutate(text = forcats::fct_reorder(shorten(!! text, wrap),
                                                  !! insight_col,
                                                  .desc = desc)) %>%
        ggplot2::ggplot(ggplot2::aes(x = text)) +
        ggplot2::geom_col(ggplot2::aes(y = !! insight_col)) +
        ggplot2::coord_flip()
}

#' Shorten some text up to n characters
#'
#' @param .data character vector
#'
#' @param n wrap length of text
#'
#' @return shortened form of .data
#'
shorten <- function(.data, n){
    ifelse(nchar(.data) > n,
           paste(substr(.data, 1, n), "...", sep = ""),
           .data)
}
```

Listing 22: Create Word Cloud

```
#' output a ggplot wordcloud graph of the top texts from some insight function
#'
#' @param .data a dataframe containing "text" and insight columns as
#'     per the output of the get_(term|aggregate)_insight wrapper
#'     function
#'
#' @param y symbol name of the column insight was
#'     outputted to
#'
#' @param x symbol name of column for insight labels
#'
#' @param n number of words to display
#'
#' @param shape character: shape of the wordcloud
```

```
#'
#' @export
score_wordcloud <- function(.data, y, n = 15,
                                x = text, shape = "circle"){
    text <- dplyr::enquo(x)
    insight_col <- dplyr::enquo(y)
    .data %>%
        dplyr::distinct(!! text, .keep_all=TRUE) %>%
        dplyr::arrange(dplyr::desc(!! insight_col)) %>%
        dplyr::group_modify(~{.x %>% head(n)}) %>%
        dplyr::ungroup() %>%
        ggplot2::ggplot(ggplot2::aes(label = stringr::str_wrap(!! text, 30), size = !! insight_col)) +
        ggwordcloud::geom_text_wordcloud(shape = shape, rm_outside = TRUE) +
        ggplot2::scale_size_area(max_size = 24)
}
```

Listing 23: Create Histogram

```
#' output a histogram of the distribution of some function of words
#'
#' @param .data the standard dataframe, modified so the last column
#'     is the output of some insight function (eg. output from
#'     term_freq)
#'
#' @param col_name symbol name of the column insight was
#'     performed on
dist_hist <- function(.data, col_name){
    q_col_name <- dplyr::enquo(col_name)
    .data %>%
        ggplot2::ggplot(ggplot2::aes(x = !! q_col_name)) +
        ggplot2::geom_histogram()
}
```

Listing 24: Create Kernel Density Estimate Plot

```
#' output a histogram of the distribution of some function of words
#'
#' @param .data the standard dataframe, modified so the last column
#'     is the output of some insight function (eg. output from
#'     term_freq)
#'
#' @param col_name symbol name of the column insight was
#'     performed on
dist_density <- function(.data, col_name){
    q_col_name <- dplyr::enquo(col_name)
    .data %>%
        ggplot2::ggplot(ggplot2::aes(x = !! q_col_name)) +
        ggplot2::geom_density()
}
```

Listing 25: Create Time Series Plot

```
#' output a ggplot time series plot of some insight function
#'
#' @param .data a dataframe containing "text" and insight columns as
#'     per the output of the get_(term|aggregate)_insight wrapper
#'     function
#'
#' @param y symbol name of the column insight was
#'     outputted to
#'
```

```
#' @export
  struct_time_series <- function(.data, y){
    q_y <- dplyr::enquo(y)
    .data %>%
        dplyr::filter(!is.na(!! q_y)) %>%
        dplyr::mutate(term = seq_along(!! q_y)) %>%
        ggplot2::ggplot(ggplot2::aes(term, !! q_y)) +
        ggplot2::geom_line(na.rm = TRUE)
}
```

Listing 26: Create Page View Plot

```
#' Colours a ggpage based on an insight function
#'
#' @param .data a dataframe containing "word" and insight columns as
#'     per the output of the get_(term|aggregate)_insight wrapper
#'     function
#'
#' @param col_name symbol name of the insight column intended to
#'     colour plot
#'
#' @param num_terms the number of terms to visualise
#'
#' @param term_index which term to start the visualisation from
#'
#' @param palette determine coloration of palette (not yet implemented)
#'
#' @return ggplot object as per ggpage
#'
#' @export
struct_pageview <- function(.data, col_name, num_terms, term_index, palette){
    end <- min(nrow(.data), term_index + num_terms)
    q_col_name <- dplyr::enquo(col_name)
    .data[seq(term_index, end),] %>%
        dplyr::pull(word) %>%
        ggpage::ggpage_build() %>%
        dplyr::bind_cols(.data[seq(term_index, end),]) %>%
        ggpage::ggpage_plot(ggplot2::aes(fill = !! q_col_name)) ## +
}
```

Listing 27: Create Visualisation

```
#' create a group-aware visualisation
#'
#' @param .data the standard dataframe, modified so the last column
#'     is the output of some insight function (eg. output from
#'     term_freq)
#'
#' @param vis character name of visualisation function
#'
#' @param col character name of the column to get insight from
#'
#' @param facet_by character name of the column to facet by
#'
#' @param scale_fixed force scales to be fixed in a facet
#'
#' @param ... additional arguments to the visualisation
#'
#' @export
get_vis <- function(.data, vis, col, facet_by="", scale_fixed = TRUE, ...){
    vistable <- list("Page View" = struct_pageview,
```

```
                    "Time Series" = struct_time_series,
                    "Bar" = score_barplot,
                    "Density" = dist_density,
                    "Histogram" = dist_hist,
                    "Word Cloud" = score_wordcloud)
    y <- dplyr::sym(col)
    chart <- vistable[[vis]](.data, !! y, ...)
    if (shiny::isTruthy(facet_by)){
        facet_name <- dplyr::sym(facet_by)
        q_facet_name <- dplyr::enquo(facet_name)
        return(chart + ggplot2::facet_wrap(ggplot2::vars(!! q_facet_name),
                                           scales = ifelse(vis == "struct_pageview" | scale_fixed,
                                                           "fixed",
                                                           "free")))

    } else {
        return(chart)
    }
}
```

Listing 28: UI and Server of Shiny Application

```
library(shiny)
library(inzightta)
library(rlang)

ui <- navbarPage("iNZight Text Analytics",
                 tabPanel("Processing",
                          sidebarLayout(
                              sidebarPanel(
                                  tags$p("Import"),
                                  fileInput("file1", "Choose File(s)",
                                            multiple = TRUE,
                                            accept = c("text/csv",
                                                       "text/comma-separated-values,text/plain",
                                                       ".csv", ".xlsx", ".xls")),
                                  tags$hr(),
                                  tags$p("Process"),
                                  checkboxInput("lemmatise", "Lemmatise"),
                                  uiOutput("sw_lexicon"),
                                  checkboxInput("stopwords", "Stopwords"),
                                  actionButton("prep_button", "Prepare Text"),
                                  selectInput("section_by", "Section By",
                                              list("", "chapter", "part", "section", "canto", "book")),
                                  uiOutput("vars_to_filter"),
                                  textInput("filter_pred", "value to match", "")),
                              mainPanel(
                                  tableOutput("table")))),
                 tabPanel("Visualisation",
                          sidebarLayout(
                              sidebarPanel(selectInput("what_vis",
                                                       "Select what you want to Visualise",
                                                       list("Term Frequency",
                                                            "n-gram Frequency",
                                                            "Key Words",
                                                            "Term Sentiment",
                                                            "Moving Average Term Sentiment",
                                                            "Aggregated Term Count",
                                                            "Key Sections",
                                                            "Aggregated Sentiment")),
                                           uiOutput("group_by"),
                                           uiOutput("insight_options"),
```

```
                                        uiOutput("vis_options"),
                                        uiOutput("vis_facet_by")),
                                mainPanel(
                                    plotOutput("plot"))))
                  )

server <- function(input, output) {
    imported <- reactive({
        inzightta::import_files(input$file1$datapath)})
    prepped <- eventReactive(input$prep_button, {
        imported() %>%
            format_data(input$lemmatise, input$stopwords, input$sw_lexicon, NA)})
    sectioned <- reactive({
        data <- prepped()
        if (isTruthy(input$section_by)){
            data <- data %>%
                section(input$section_by)}
        data})
    filtered <- reactive({
        data <- sectioned()
        if (isTruthy(input$filter_var) &
            isTruthy(input$filter_pred)){
            data <- data %>%
                dplyr::filter(!! dplyr::sym(input$filter_var) == input$filter_pred)}
        data})
    grouped <- reactive({
        data <- filtered()
        if (isTruthy(input$group_var)){
            data <- data %>%
                dplyr::group_by(!! dplyr::sym(input$group_var))}
        data})
    output$table <- renderTable({
        filtered() %>% head(300)})
    output$sw_lexicon <- renderUI(selectInput("sw_lexicon", "Select the Stopword Lexicon",
                                              stopwords::stopwords_getsources()))
    output$vars_to_filter <- renderUI(selectInput("filter_var",
                                                  "select which column to apply filtering to",
                                                  c("", names(sectioned())) %||% c("")))
    output$group_by <- renderUI(selectInput("group_var",
                                            "select which columns to group on",
                                            c("", names(filtered())) %||% c("")))
    output$insight_options <- renderUI({
        switch(input$what_vis,
               "Term Frequency" = selectInput("vis_type",
                                              "Select how to Visualise it",
                                              list("Bar",
                                                   "Word Cloud",
                                                   "Page View",
                                                   "Time Series",
                                                   "Density",
                                                   "Histogram")),
               "n-gram Frequency" = tagList(selectInput("vis_type",
                                                        "Select how to Visualise it",
                                                        list("Bar",
                                                             "Word Cloud",
                                                             "Page View",
                                                             "Time Series",
                                                             "Density",
                                                             "Histogram")),
                                            sliderInput("n_gram",
                                                        "n-gram count",
                                                        2, 8, 2)),
```

```
                "Key Words" = tagList(selectInput("vis_type",
                                                  "Select how to Visualise it",
                                                  list("Bar",
                                                       "Word Cloud",
                                                       "Page View",
                                                       "Time Series",
                                                       "Density",
                                                       "Histogram")),
                                      selectInput("summ_method",
                                                  "Method of summary generation",
                                                  list("TextRank", "LexRank"))),
                "Term Sentiment" = tagList(selectInput("vis_type",
                                                       "Select how to Visualise it",
                                                       list("Page View",
                                                            "Word Cloud",
                                                            "Time Series",
                                                            "Bar",
                                                            "Density",
                                                            "Histogram")),
                                           selectInput("sent_lex",
                                                       "Lexicon for Sentiment Dictionary",
                                                       list("afinn", "bing",
                                                            "loughran", "nrc"))),
                "Moving Average Term Sentiment" = tagList(selectInput("vis_type",
                                                                      "Select how to Visualise it",
                                                                      list("Time Series",
                                                                           "Word Cloud",
                                                                           "Page View",
                                                                           "Bar",
                                                                           "Density",
                                                                           "Histogram")),
                                                          sliderInput("term_sent_lag",
                                                                      "Lag Length for Calculation of Moving Averag
                                                                      3,500,50),
                                                          selectInput("sent_lex",
                                                                      "Lexicon for Sentiment Dictionary",
                                                                      list("afinn", "bing",
                                                                           "loughran", "nrc"))),
                "Aggregated Term Count" = tagList(selectInput("vis_type",
                                                              "Select how to Visualise it",
                                                              list("Bar",
                                                                   "Word Cloud",
                                                                   "Page View",
                                                                   "Time Series",
                                                                   "Density",
                                                                   "Histogram")),
                                                  selectInput("agg_var",
                                                              "Select which variable to aggregate on",
                                                              c("", names(grouped())) %||% c(""))),
                "Key Sections" = tagList(selectInput("vis_type",
                                                     "Select how to Visualise it",
                                                     list("Bar",
                                                          "Word Cloud",
                                                          "Page View",
                                                          "Time Series",
                                                          "Density",
                                                          "Histogram")),
                                         selectInput("summ_method",
                                                     "Method of summary generation",
                                                     list("TextRank", "LexRank")),
                                         selectInput("agg_var",
                                                     "Select which variable to aggregate on",
```

```
                                                c("", names(grouped())) %||% c(""))),
                "Aggregated Sentiment" = tagList(selectInput("vis_type",
                                                             "Select how to Visualise it",
                                                             list("Page View",
                                                                  "Word Cloud",
                                                                  "Time Series",
                                                                  "Bar",
                                                                  "Density",
                                                                  "Histogram")),
                                                 selectInput("sent_lex",
                                                             "Lexicon for Sentiment Dictionary",
                                                             list("afinn", "bing",
                                                                  "loughran", "nrc")),
                                                 selectInput("agg_var",
                                                             "Select which variable to aggregate on",
                                                             c("", names(grouped())) %||% c(""))))})
    insighted <- reactive({
        switch(input$what_vis,
                "Term Frequency" = get_term_insight(grouped(),
                                                    input$what_vis),
                "n-gram Frequency" = get_term_insight(grouped(),
                                                      c("n-grams", "n-gram Frequency"),
                                                      input$n_gram),
                "Key Words" = get_term_insight(grouped(),
                                               input$what_vis,
                                               input$summ_method),
                "Term Sentiment" = get_term_insight(grouped(),
                                                    input$what_vis,
                                                    input$sent_lex),
                "Moving Average Term Sentiment" = get_term_insight(grouped(),
                                                                   input$what_vis,
                                                                   input$sent_lex,
                                                                   input$term_sent_lag),
                "Aggregated Term Count" = get_aggregate_insight(grouped(),
                                                                c("Bound Aggregates", input$what_vis),
                                                                input$agg_var),
                "Key Sections" = get_aggregate_insight(grouped(),
                                                       c("Bound Aggregates", input$what_vis),
                                                       input$agg_var,
                                                       input$summ_method),
                "Aggregated Sentiment" = get_aggregate_insight(grouped(),
                                                               c("Bound Aggregates", input$what_vis),
                                                               input$agg_var,
                                                               input$sent_lex))})
    output$vis_options <- renderUI({
        switch(input$vis_type,
                "Word Cloud" = tagList(sliderInput("num_terms",
                                                   "Select the number of terms to visualise",
                                                   3, 50, 15),
                                       selectInput("wordcloud_shape",
                                                   "Select the shape of the wordcloud",
                                                   list("circle",
                                                        "cardioid",
                                                        "diamond",
                                                        "square",
                                                        "triangle-forward",
                                                        "triangle-upright",
                                                        "pentagon",
                                                        "star"))),
                "Page View" = tagList(sliderInput("num_terms",
                                                  "Select the number of terms to visualise",
                                                  3, 400, 100),
```

```
                                    sliderInput("term_index",
                                                "Select the point to begin visualisation from",
                                                1, nrow(insighted()), 1),
                                    selectInput("palette",
                                                "Select the colour palette type",
                                                list("Sequential", "Diverging"))),
                "Bar" = tagList(sliderInput("num_terms", "Select the number of terms to visualise",
                                            2,50,15))})
    output$vis_facet_by <- renderUI(tagList(selectInput("vis_facet",
                                                        "select which variable to facet on",
                                                        c("", names(grouped())) %||% c("")),
                                            checkboxInput("scale_fixed", "Scale Fixed", value=TRUE)))
    visualisation <- reactive({
        switch(input$vis_type,
                "Word Cloud" = switch(input$what_vis,
                                    "n-gram Frequency" = get_vis(
                                        insighted(),
                                        input$vis_type,
                                        input$what_vis,
                                        input$vis_facet,
                                        input$scale_fixed,
                                        input$num_terms,
                                        x = `n-grams`,
                                        shape = input$wordcloud_shape),
                                    "Aggregated Term Count" =,
                                    "Key Sections" =,
                                    "Aggregated Sentiment" = get_vis(
                                        insighted(),
                                        input$vis_type,
                                        input$what_vis,
                                        input$vis_facet,
                                        input$scale_fixed,
                                        input$num_terms,
                                        x = `Bound Aggregates`,
                                        shape = input$wordcloud_shape),
                                    get_vis(insighted(),
                                            input$vis_type,
                                            input$what_vis,
                                            input$vis_facet,
                                            input$scale_fixed,
                                            input$num_terms,
                                            shape = input$wordcloud_shape)),
                "Page View" = get_vis(insighted(), input$vis_type,
                                    input$what_vis,
                                    input$vis_facet,
                                    input$scale_fixed,
                                    input$num_terms,
                                    input$term_index,
                                    palette = input$palette),
                "Time Series" = get_vis(insighted(), input$vis_type,
                                      input$what_vis,
                                      input$vis_facet,
                                      input$scale_fixed),
                "Bar" = switch(input$what_vis,
                               "n-gram Frequency" = get_vis(insighted(),
                                                            input$vis_type,
                                                            input$what_vis,
                                                            input$vis_facet,
                                                            input$scale_fixed,
                                                            input$num_terms,
                                                            x = `n-grams`),
                               "Aggregated Term Count" =,
```

```
289                                 "Key Sections" =,
290                                 "Aggregated Sentiment" = get_vis(insighted(),
291                                                                  input$vis_type,
292                                                                  input$what_vis,
293                                                                  input$vis_facet,
294                                                                  input$scale_fixed,
295                                                                  input$num_terms,
296                                                                  x = `Bound Aggregates`),
297                                 get_vis(insighted(), input$vis_type,
298                                         input$what_vis,
299                                         input$vis_facet,
300                                         input$scale_fixed,
301                                         input$num_terms)),
302                 "Density" = get_vis(insighted(), input$vis_type,
303                                     input$what_vis, input$vis_facet,
304                                     input$scale_fixed),
305                 "Histogram" = get_vis(insighted(), input$vis_type,
306                                       input$what_vis,
307                                       input$vis_facet,
308                                       input$scale_fixed))})
309     output$plot <- renderPlot({
310         visualisation()})
311 }
312
313 # Create Shiny app ----
314 shinyApp(ui, server)
```

# Appendix B

# Package Manual

This appendix chapter consists of a copy of the documentation for the `R` package, which was automatically generated through the Roxygen2 system from in-source comments.

# Package 'inzightta'

September 24, 2019

**Title** iNZight Text Analytics

**Version** 0.0.0.9000

**Description** Provides text analytics functions for the importation, analysis, and visualisation of text. This package is designed specifically for output in the shiny program, with the analytical functions all working well with dplyr tools.

**License** GPL-3

**Encoding** UTF-8

**LazyData** true

**Imports** readr, tibble, stringr, dplyr, readxl, purrr, tidytext, textstem, magrittr, stats, textrank, lexRankr, ggpage, ggplot2, forcats, shiny, ggwordcloud

**RoxygenNote** 6.1.1

**NeedsCompilation** no

**Author** Jason Cairns [aut, cre]

**Maintainer** Jason Cairns <jcai849@aucklanduni.ac.nz>

# R topics documented:

## aggregate_sentiment *Get statistics for sentiment over some group, such as sentence.*

### Description

Get statistics for sentiment over some group, such as sentence.

### Usage

```
aggregate_sentiment(.data, aggregate_on, lexicon = "afinn",
  statistic = mean)
```

### Arguments

| | |
|---|---|
| .data | character vector of words |
| aggregate_on | vector to aggregate .data over; ideally, sentence_id, but could be chapter, document, etc. |
| lexicon | as per term sentiment |
| statistic | function that accepts na.rm argument; e.g. mean, median, sd. |

### Value

sentiment of same length as input vector aggregated over the aggregate_on vector

## bind_aggregation *bind aggregate terms together*

### Description

bind aggregate terms together

### Usage

```
bind_aggregation(data, aggregate_on)
```

### Arguments

| | |
|---|---|
| data | vector of terms |
| aggregate_on | vector of aggregations |

### Value

data with every aggregation bound, as in a sentence

## concat_walk *concat list 1 and 2, moving past NA values*

### Description

concat list 1 and 2, moving past NA values

### Usage

```
concat_walk(list1, list2)
```

### Arguments

| | |
|---|---|
| list1 | list or vector for first bigram token |
| list2 | list or vector for second bigram token |

### Value

paste of list1 and list2, skipping NA's

## concat_walk_i *concat list 1 and 2 at index, skipping NA values*

### Description

concat list 1 and 2 at index, skipping NA values

### Usage

```
concat_walk_i(i, list1, list2)
```

### Arguments

| | |
|---|---|
| i | numeric index to assess index at |
| list1 | list or vector for first token |
| list2 | list or vector for second token |

### Value

paste of list1 and list2 at index i, skipping NA's

## determine_stopwords *determine stopword status*

### Description

determine stopword status

### Usage

```
determine_stopwords(.data, ...)
```

### Arguments

| | |
|---|---|
| .data | vector of words |
| ... | arguments of get_sw |

### Value

a [tibble][tibble::tibble-package] equivalent to the input dataframe, with an additional stopword column

## dist_density *output a histogram of the distribution of some function of words*

### Description

output a histogram of the distribution of some function of words

### Usage

```
dist_density(.data, col_name)
```

### Arguments

| | |
|---|---|
| .data | the standard dataframe, modified so the last column is the output of some insight function (eg. output from term_freq) |
| col_name | symbol name of the column insight was performed on |

## dist_hist *output a histogram of the distribution of some function of words*

### Description

output a histogram of the distribution of some function of words

### Usage

```
dist_hist(.data, col_name)
```

### Arguments

| | |
|---|---|
| .data | the standard dataframe, modified so the last column is the output of some insight function (eg. output from term_freq) |
| col_name | symbol name of the column insight was performed on |

## format_data *takes imported one-line-per-row data and prepares it for later analysis*

### Description

takes imported one-line-per-row data and prepares it for later analysis

### Usage

```
format_data(.data, lemmatize = TRUE, stopwords = TRUE,
  sw_lexicon = "snowball", addl_stopwords = NA)
```

### Arguments

| | |
|---|---|
| .data | tibble with one line of text per row |
| lemmatize | boolean, whether to lemmatize or not |
| stopwords | boolean, whether to remove stopwords or not |
| sw_lexicon | string, lexicon with which to remove stopwords |
| addl_stopwords | char vector of user-supplied stopwords |

### Value

a [tibble][tibble::tibble-package] with one token per line, stopwords removed leaving NA values, column for analysis named "text"

## get_aggregate_insight *perform group-aware aggregate operations on the data*

### Description

perform group-aware aggregate operations on the data

### Usage

```
get_aggregate_insight(.data, operations, aggregate_on, ...)
```

### Arguments

| | |
|---|---|
| .data | dataframe of terms as per output of format_data |
| operations | character vector of operations to perform |
| aggregate_on | character name of the column to perform aggregate operations on |
| ... | additional arguments to the operation - only sensible for singular operations |

### Value

.data with operation columns added

## get_bigram *Determine bigrams*

### Description

Determine bigrams

### Usage

```
get_bigram(.data)
```

### Arguments

| | |
|---|---|
| .data | character vector of words |

### Value

character vector of bigrams

## get_books *sections text based on book*

### Description

sections text based on book

### Usage

```
get_books(.data)
```

### Arguments

| | |
|---|---|
| .data | vector to section |

### Value

vector of same length as .data with book numbers

## get_cantos *sections text based on cantos*

### Description

sections text based on cantos

### Usage

```
get_cantos(.data)
```

### Arguments

| | |
|---|---|
| .data | vector to section |

### Value

vector of same length as .data with canto numbers

## get_chapters *sections text based on chapters*

### Description

sections text based on chapters

### Usage

```
get_chapters(.data)
```

### Arguments

| | |
|---|---|
| .data | vector to section |

### Value

vector of same length as .data with chapter numbers

## get_filetype *Get filetype*

### Description

Get filetype

### Usage

```
get_filetype(filepath)
```

### Arguments

| | |
|---|---|
| filepath | string filepath of document |

### Value

filetype (string) - NA if no extension

## get_ngram *Returns the n-grams, skipping NA values*

### Description

Returns the n-grams, skipping NA values

### Usage

```
get_ngram(.data, n)
```

### Arguments

| | |
|---|---|
| .data | vector to get n-grams from |
| n | number of n-grams to attain |

### Value

n-gram vector without NA values

## get_parts *sections text based on parts*

### Description

sections text based on parts

### Usage

```
get_parts(.data)
```

### Arguments

| | |
|---|---|
| .data | vector to section |

### Value

vector of same length as .data with part numbers

## get_search *creates a search closure to section text*

### Description

creates a search closure to section text

### Usage

```
get_search(search)
```

### Arguments

| | |
|---|---|
| search | a string regexp for the term to seperate on, e.g. "Chapter" |

### Value

closure over search expression

## get_sections *sections text based on sections*

### Description

sections text based on sections

### Usage

```
get_sections(.data)
```

### Arguments

| | |
|---|---|
| .data | vector to section |

### Value

vector of same length as .data with section numbers

## get_sw *Gets stopwords from a default list and user-provided list*

### Description

Gets stopwords from a default list and user-provided list

### Usage

```
get_sw(lexicon = "snowball", addl = NA)
```

### Arguments

| | |
|---|---|
| lexicon | a string name of a stopword list, one of "smart", "snowball", or "onix" |
| addl | user defined character vector of additional stopwords, each element being a stopword |

### Value

a [tibble][tibble::tibble-package] with one column named "word"

## get_term_insight *perform group-aware term operations on the data*

### Description

perform group-aware term operations on the data

### Usage

```
get_term_insight(.data, operations, ...)
```

### Arguments

| | |
|---|---|
| .data | dataframe of terms as per output of format_data |
| operations | character vector of term operations to perform |
| ... | additional arguments to the operation - only sensible for singular operations |

### Value

.data with operation columns added

## get_valid_input *helper function to get valid input (recursively)*

### Description

helper function to get valid input (recursively)

### Usage

```
get_valid_input(options, init = TRUE)
```

### Arguments

| | |
|---|---|
| options | vector of options that valid input should be drawn from |
| init | whether this is the initial attempt, used only as recursive information |

### Value

readline output that exists in the vector of options

## get_vis *create a group-aware visualisation*

### Description

create a group-aware visualisation

### Usage

```
get_vis(.data, vis, col, facet_by = "", scale_fixed = TRUE, ...)
```

### Arguments

| | |
|---|---|
| .data | the standard dataframe, modified so the last column is the output of some insight function (eg. output from term_freq) |
| vis | character name of visualisation function |
| col | character name of the column to get insight from |
| facet_by | character name of the column to facet by |
| scale_fixed | force scales to be fixed in a facet |
| ... | additional arguments to the visualisation |

`ifexp` *scheme-like if expression, without restriction of returning same-size table of .test, as ifelse() does*

## Description

scheme-like if expression, without restriction of returning same-size table of .test, as ifelse() does

## Usage

```
ifexp(.test, true, false)
```

## Arguments

| | |
|---|---|
| `.test` | predicate to test |
| `true` | expression to return if .test evals to TRUE |
| `false` | expression to return if .test evals to TRUE |

## Value

either true or false

`import_base_file` *Base case for file import*

## Description

Base case for file import

## Usage

```
import_base_file(filepath)
```

## Arguments

| | |
|---|---|
| `filepath` | string filepath of file for import |

## Value

imported file with document id

## import_csv *Import csv file*

### Description

Import csv file

### Usage

```
import_csv(filepath)
```

### Arguments

| | |
|---|---|
| filepath | a string indicating the relative or absolute filepath of the file to import |

### Value

a [tibble][tibble::tibble-package] of each row corrresponding to a line of the text file, with the column named "text"

## import_excel *Import excel file*

### Description

Import excel file

### Usage

```
import_excel(filepath)
```

### Arguments

| | |
|---|---|
| filepath | a string indicating the relative or absolute filepath of the file to import |

### Value

a [tibble][tibble::tibble-package] of each row corrresponding to a line of the text file, with the column named "text"

## import_files *Import any number of files*

### Description

Import any number of files

### Usage

```
import_files(filepaths)
```

### Arguments

| | |
|---|---|
| filepaths | char vector of filepaths |

### Value

a [tibble][tibble::tibble-package] imported files with document id

## import_txt *Import text file*

### Description

Import text file

### Usage

```
import_txt(filepath)
```

### Arguments

| | |
|---|---|
| filepath | a string indicating the relative or absolute filepath of the file to import |

### Value

a [tibble][tibble::tibble-package] of each row corrresponding to a line of the text file, with the column named "text"

## keywords_tr *Determine textrank score for vector of words*

### Description

Determine textrank score for vector of words

### Usage

```
keywords_tr(.data, summ_method)
```

### Arguments

| | |
|---|---|
| .data | character vector of words |
| summ_method | method to use for summarisation: textrank or lexrank. Doesn't do anything yet |

### Value

vector of scores for each word

## key_aggregates *get score for key sentences as per Lexrank*

### Description

get score for key sentences as per Lexrank

### Usage

```
key_aggregates(.data, aggregate_on, summ_method)
```

### Arguments

| | |
|---|---|
| .data | character vector of words |
| aggregate_on | vector to aggregate .data over; ideally, sentence_id |
| summ_method | method to use for summarisation: textrank or lexrank. Doesn't do anything yet |

### Value

lexrank scores of aggregates

## ma_term_sentiment *Determine the lagged sentiment of terms*

### Description

Determine the lagged sentiment of terms

### Usage

```
ma_term_sentiment(.data, lexicon = "afinn", lag = 10,
  statistic = mean)
```

### Arguments

| | |
|---|---|
| .data | vector of terms |
| lexicon | sentiment lexicon to use, based on the corpus provided by tidytext |
| lag | how many (inclusive) terms to compute statistic over |
| statistic | base statistic used to summarise the data, capable of taking an na.rm argument |

### Value

vector with lagged sentiment score of each term in the input vector

## ngram_freq *NOT FOR PRODUCTION - STILL IN TESING. Returns the count of n-grams, skipping NA values*

### Description

NOT FOR PRODUCTION - STILL IN TESING. Returns the count of n-grams, skipping NA values

### Usage

```
ngram_freq(.data, n)
```

### Arguments

| | |
|---|---|
| .data | vector to get n-grams from |
| n | number of n-grams to attain |

### Value

count of each associated n-gram

## score_barplot

*output a ggplot column graph of the top texts from some insight function*

### Description

output a ggplot column graph of the top texts from some insight function

### Usage

```
score_barplot(.data, y, n = 15, x = text, desc = FALSE)
```

### Arguments

| | |
|---|---|
| .data | a dataframe containing "text" and insight columns as per the output of the get_(termlaggregate)_insight wrapper function |
| y | symbol name of the column insight was outputted to |
| n | number of bars to display |
| x | symbol name of column for insight labels |
| desc | bool: show bars in descending order |

## score_wordcloud

*output a ggplot wordcloud graph of the top texts from some insight function*

### Description

output a ggplot wordcloud graph of the top texts from some insight function

### Usage

```
score_wordcloud(.data, y, n = 15, x = text, shape = "circle")
```

### Arguments

| | |
|---|---|
| .data | a dataframe containing "text" and insight columns as per the output of the get_(termlaggregate)_insight wrapper function |
| y | symbol name of the column insight was outputted to |
| n | number of words to display |
| x | symbol name of column for insight labels |
| shape | character: shape of the wordcloud |

## section *Adds section column to dataframe*

### Description

Adds section column to dataframe

### Usage

```
section(.data, section_by)
```

### Arguments

| | |
|---|---|
| .data | dataframe formatted as per output of prep process |
| section_by | character name of what to section over |

### Value

input dataframe with additional section column

## shorten *Shorten some text up to n characters*

### Description

Shorten some text up to n characters

### Usage

```
shorten(.data, n)
```

### Arguments

| | |
|---|---|
| .data | character vector |
| n | wrap length of text |

### Value

shortened form of .data

## struct_pageview *Colours a ggpage based on an insight function*

### Description

Colours a ggpage based on an insight function

### Usage

```
struct_pageview(.data, col_name, num_terms, term_index, palette)
```

### Arguments

| | |
|---|---|
| .data | a dataframe containing "word" and insight columns as per the output of the get_(term\|aggregate)_insight wrapper function |
| col_name | symbol name of the insight column intended to colour plot |
| num_terms | the number of terms to visualise |
| term_index | which term to start the visualisation from |
| palette | determine coloration of palette (not yet implemented) |

### Value

ggplot object as per ggpage

## struct_time_series *output a ggplot time series plot of some insight function*

### Description

output a ggplot time series plot of some insight function

### Usage

```
struct_time_series(.data, y)
```

### Arguments

| | |
|---|---|
| .data | a dataframe containing "text" and insight columns as per the output of the get_(term\|aggregate)_insight wrapper function |
| y | symbol name of the column insight was outputted to |

## table_textcol *Interactively determine and automatically mark the text column of a table*

### Description

Interactively determine and automatically mark the text column of a table

### Usage

```
table_textcol(data)
```

### Arguments

| | |
|---|---|
| data | dataframe with column requiring marking |

### Value

same dataframe with text column renamed to "text"

## term_cooccurance *Determine term cooccurances - extremely slow*

### Description

Determine term cooccurances - extremely slow

### Usage

```
term_cooccurance(.data, term, aggregate_on)
```

### Arguments

| | |
|---|---|
| .data | character vector of terms |
| term | character to find correlations with |
| aggregate_on | vector to aggregate .data over; ideally, sentence_id, but could be chapter, document, etc. |

### Value

numeric vector of term correlations as per phi_coef

## term_corr *Determine term correlations - extremely slow*

### Description

Determine term correlations - extremely slow

### Usage

```
term_corr(.data, term, aggregate_on)
```

### Arguments

| | |
|---|---|
| .data | character vector of terms |
| term | character to find correlations with |
| aggregate_on | vector to aggregate .data over; ideally, sentence_id, but could be chapter, document, etc. |

### Value

numeric vector of term correlations as per phi_coef

## term_count *Determine the number of terms at each aggregate level*

### Description

Determine the number of terms at each aggregate level

### Usage

```
term_count(.data, aggregate_on)
```

### Arguments

| | |
|---|---|
| .data | character vector of terms |
| aggregate_on | vector to split .data on for insight |

### Value

vector of number of terms for each aggregate level, same length as .data

## term_freq *Determine term frequency*

### Description

Determine term frequency

### Usage

```
term_freq(.data)
```

### Arguments

| | |
|---|---|
| .data | character vector of terms |

### Value

numeric vector of term frequencies

## term_sentiment *Determine sentiment of terms*

### Description

Determine sentiment of terms

### Usage

```
term_sentiment(.data, lexicon = "afinn")
```

### Arguments

| | |
|---|---|
| .data | vector of terms |
| lexicon | sentiment lexicon to use, based on the corpus provided by tidytext |

### Value

vector with sentiment score of each word in the vector

ungroup_by *helper function to ungroup for dplyr. functions equivalently to group_by() but with standard (string) evaluation*

## Description

helper function to ungroup for dplyr. functions equivalently to group_by() but with standard (string) evaluation

## Usage

```
ungroup_by(x, ...)
```

## Arguments

| | |
|---|---|
| x | tibble to perform function on |
| ... | string of groups to ungroup on |

## Value

x with ... no longer grouped upon

# Index

# Glossary

**corpus** A collection ("body") of texts, with analysis usually within and between the texts.. 13

**lemma** The dictionary form of a word. Typically without tense. The lemma of "readily" is "ready", and the lemma of "is" is "be". In linguistic terms, a lemma is the conventional form of a lexeme (the basic meaning behind related words).. 13

**lexicon** A language's inventory of lexemes (lemmas). In text analytics a lexicon is typically a dictionary data structure containing a list of words and possibly some values for each word.. 13

**n-gram** A series of words that occur in a direct n-length sequence. For example, in the phrase, "The quick brown dog", the following 2-grams (bigrams) exist: "The quick", "quick brown", "brown dog". This is generalised to any number of sequential words. They are useful in text analytics to determine word sequences, as well as common adverb-verb and adjective-noun pairs.. 14

**stem** The reduced form of a created by removing inflections. For example, "readily" may have the "ly" removed to become "readi". In linguistic terms, the process of morphological (units of meaning) reduction of a word.. 13

**term** "a word or expression that has a precise meaning in some uses or is peculiar to a science, art, profession, or subject" [29] — here text analysts have capitalised on the generalisation of "term" to include subcomponents or aggregations of words. 13

**text** A written communication, in text analytics typically excluding "textuality" elements such as a reader's interpretation. Examples include books, youtube comments, articles, etc.. 13

**token** A concrete instance of a word in text. The phrase "The cat on the mat" has 5 tokens, as the first "the" is considered separately to the second.. 13

**word** The smallest element uttered in isolation with objective meaning. Not necessarily separated by spaces, which is an orthographic conven-

# Bibliography


[1] Guillaume Desagulier. *Corpus Linguistics and Statistics with R. Introduction to Quantitative Methods in Linguistics*. 1st ed. Quantitative Methods in the Humanities and Social Sciences. Springer International Publishing, 2017. XIII, 353. ISBN: 978-3-319-64570-4. DOI: 10.1007/978-3-319-64572-8. URL: https://www.springer.com/gp/book/9783319645704#aboutBook (cit. on pp. 2, 39).

[2] SAS Institute Inc. *SAS Text Miner*. Version 15.1. 2019. URL: https://www.sas.com/software/text-miner.html (cit. on p. 2).

[3] IBM Corp. *IBM SPSS Statistics for Windows*. Version 25.0. 2017 (cit. on p. 2).

[4] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2019. URL: https://www.R-project.org/ (cit. on p. 2).

[5] bnosac. *txtminer - Automated Text Analysis*. 2019. URL: https://bnosac.be/index.php/products/txtminer (cit. on p. 2).

[6] Chris Wild. *iNZight for Data Analysis*. Version 3.4.6. 23rd Sept. 2019. URL: https://www.stat.auckland.ac.nz/~wild/iNZight/index.php (cit. on p. 2).

[7] Julia Silge and David Robinson. *Text mining with R: A tidy approach*. " O'Reilly Media, Inc.", 2017 (cit. on p. 3).

[8] Michael Clark. *An Introduction to Text Processing and Analysis with R*. 9th Sept. 2018. URL: https://m-clark.github.io/text-analysis-with-R/ (cit. on p. 3).

[9] Fridolin Wild, ed. *CRAN Task View: Natural Language Processing*. Version 2019-03-07. Performance Augmentation Lab (PAL, Department of Computing and Communications Technologies, Oxford Brookes University, 7th Mar. 2019. URL: https://CRAN.R-project.org/view=NaturalLanguageProcessing (cit. on p. 3).

[10] Julia Silge and David Robinson. 'tidytext: Text Mining and Analysis Using Tidy Data Principles in R'. In: *The Journal of Open Source Software* 1.3 (11th July 2016), p. 37. ISSN: 2475-9066. DOI: 10.21105/joss.00037. URL: http://dx.doi.org/10.21105/joss.00037 (cit. on p. 3).

[11] Ingo Feinerer and Kurt Hornik. *tm: Text Mining Package*. R package version 0.7-6. 2018. URL: https://CRAN.R-project.org/package=tm (cit. on p. 3).

[12] Kenneth Benoit et al. 'quanteda: An R package for the quantitative analysis of textual data'. In: *Journal of Open Source Software* 3.30 (2018), p. 774. DOI: 10.21105/joss.00774. URL: https://quanteda.io (cit. on p. 3).

[13] Tyler W. Rinker. *qdap: Quantitative Discourse Analysis Package*. 2.3.2. Buffalo, New York, 2019. URL: http://github.com/trinker/qdap (cit. on p. 3).

[14] Tyler W. Rinker. *sentimentr: Calculate Text Polarity Sentiment*. version 2.7.1. Buffalo, New York, 2019. URL: http://github.com/trinker/sentimentr (cit. on p. 4).

[15] Dan Vanderkam et al. *dygraphs: Interface to 'Dygraphs' Interactive Time Series Charting Library*. R package version 1.1.1.6. 2018. URL: https://CRAN.R-project.org/package=dygraphs (cit. on p. 4).

[16] Thomas Lin Pedersen and David Robinson. *gganimate: A Grammar of Animated Graphics*. R package version 1.0.3. 2019. URL: https://CRAN.R-project.org/package=gganimate (cit. on p. 4).

[17] Jan Wijffels. *textrank: Summarize Text by Ranking Sentences and Finding Keywords*. R package version 0.3.0. 2019. URL: https://CRAN.R-project.org/package=textrank (cit. on pp. 4, 24).

[18] Emil Hvitfeldt. *ggpage: Creates Page Layout Visualizations*. R package version 0.2.3. 2019. URL: https://CRAN.R-project.org/package=ggpage (cit. on pp. 4, 33).

[19] Jan Wijffels. *udpipe: Tokenization, Parts of Speech Tagging, Lemmatization and Dependency Parsing with the 'UDPipe' 'NLP' Toolkit*. R package version 0.8.3. 2019. URL: https://CRAN.R-project.org/package=udpipe (cit. on p. 4).

[20] Jan Wijffels. *BTM: Biterm Topic Models for Short Text*. R package version 0.2.1. 2019. URL: https://CRAN.R-project.org/package=BTM (cit. on p. 4).

[21] Jan Wijffels and Naoaki Okazaki. *crfsuite: Conditional Random Fields for Labelling Sequential Data in Natural Language Processing based on CRFsuite: a fast implementation of Conditional Random Fields (CRFs)*. R package version 0.1. BNOSAC, 2007-2018. URL: `https://github.com/bnosac/crfsuite` (cit. on p. 4).

[22] Oliver Keyes. *humaniformat: A Parser for Human Names*. R package version 0.6.0. 2016. URL: `https://CRAN.R-project.org/package=humaniformat` (cit. on p. 4).

[23] David Robinson. *gutenbergr: Download and Process Public Domain Works from Project Gutenberg*. R package version 0.1.5. 2019. URL: `https://CRAN.R-project.org/package=gutenbergr` (cit. on p. 4).

[24] Michael W. Kearney. *rtweet: Collecting Twitter Data*. R package version 0.6.9. 2019. URL: `https://cran.r-project.org/package=rtweet` (cit. on p. 4).

[25] Avner Bar-Hen. *WikipediaR: R-Based Wikipedia Client*. R package version 1.1. 2016. URL: `https://CRAN.R-project.org/package=WikipediaR` (cit. on p. 4).

[26] Martin Rajman and Romaric Besançon. 'Text mining-knowledge extraction from unstructured textual data'. In: *Advances in data science and classification*. Springer, 1998, pp. 473–480 (cit. on p. 6).

[27] Youngjoong Ko. 'A study of term weighting schemes using class information for text classification'. In: *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*. Citeseer. 2012, pp. 1029–1030 (cit. on p. 6).

[28] David M Blei, Andrew Y Ng and Michael I Jordan. 'Latent dirichlet allocation'. In: *Journal of machine Learning research* 3.Jan (2003), pp. 993–1022 (cit. on p. 6).

[29] Merriam-Webster Dictionary, ed. *Term — Definition of Term*. 17th Aug. 2019. URL: `https://www.merriam-webster.com/dictionary/term` (cit. on pp. 7, 88).

[30] Hadley Wickham and Garrett Grolemund. *R for data science: import, tidy, transform, visualize, and model data*. " O'Reilly Media, Inc.", 2016 (cit. on p. 8).

[31] Winston Chang et al. *shiny: Web Application Framework for R*. R package version 1.4.0. 2019. URL: `https://CRAN.R-project.org/package=shiny` (cit. on p. 11).

[32] Hadley Wickham. *tidyverse: Easily Install and Load the 'Tidyverse'*. R package version 1.2.1. 2017. URL: `https://CRAN.R-project.org/package=tidyverse` (cit. on p. 11).

[33] Hadley Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2016. ISBN: 978-3-319-24277-4. URL: https://ggplot2.tidyverse.org (cit. on p. 11).

[34] Hadley Wickham et al. *dplyr: A Grammar of Data Manipulation*. R package version 0.8.3. 2019. URL: https://CRAN.R-project.org/package=dplyr (cit. on p. 12).

[35] Hadley Wickham et al. 'Tidy data'. In: *Journal of Statistical Software* 59.10 (2014), pp. 1–23 (cit. on p. 12).

[36] Hadley Wickham, Peter Danenberg and Manuel Eugster. *roxygen2: In-Line Documentation for R*. R package version 6.1.1. 2018. URL: https://CRAN.R-project.org/package=roxygen2 (cit. on p. 16).

[37] Adobe Inc., ed. *Nondestructive Editing in Photoshop*. 14th Nov. 2018. URL: https://helpx.adobe.com/photoshop/using/nondestructive-editing.html (cit. on p. 16).

[38] Kirill Müller and Hadley Wickham. *tibble: Simple Data Frames*. R package version 2.1.3. 2019. URL: https://CRAN.R-project.org/package=tibble (cit. on p. 17).

[39] Hadley Wickham, Jim Hester and Romain Francois. *readr: Read Rectangular Text Data*. R package version 1.3.1. 2018. URL: https://CRAN.R-project.org/package=readr (cit. on p. 18).

[40] Tyler W. Rinker. *textstem: Tools for stemming and lemmatizing text*. version 0.1.4. Buffalo, New York, 2018. URL: http://github.com/trinker/textstem (cit. on p. 20).

[41] Adam Spannbauer and Bryan White. *lexRankr: Extractive Summarization of Text with the LexRank Algorithm*. R package version 0.5.2. 2019. URL: https://CRAN.R-project.org/package=lexRankr (cit. on p. 27).

[42] David Robinson. *widyr: Widen, Process, then Re-Tidy Data*. R package version 0.1.2. 2019. URL: https://CRAN.R-project.org/package=widyr (cit. on p. 28).

[43] Erwan Le Pennec and Kamil Slowikowski. *ggwordcloud: A Word Cloud Geom for 'ggplot2'*. R package version 0.5.0. 2019. URL: https://CRAN.R-project.org/package=ggwordcloud (cit. on p. 30).

[44] Paul Hudak et al. 'A history of Haskell: Being lazy with class'. In: *In Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages (HOPL-III*. ACM Press, 2007, pp. 1–55 (cit. on p. 37).

[45] Markus Gesmann and Diego de Castillo. 'googleVis: Interface between R and the Google Visualisation API'. In: *The R Journal* 3.2 (Dec. 2011), pp. 40–44. URL: https://journal.r-project.org/archive/2011-2/RJournal_2011-2_Gesmann+de~Castillo.pdf (cit. on p. 38).