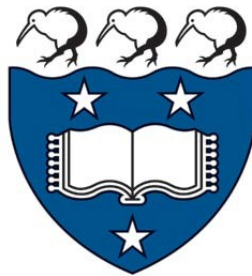


# Text Analytics

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 glean 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

# Contents

<b>Listings</b>	<b>4</b>
<b>Tables</b>	<b>5</b>
<b>Figures</b>	<b>6</b>
<b>1 Introduction</b>	<b>8</b>
1.1 Intention . . . . .	8
1.2 Brief Background: Text Analytics . . . . .	8
1.2.1 Existing Systems . . . . .	9
1.3 Background: inZight . . . . .	9
1.4 Literature Review . . . . .	10
1.5 Scope of work . . . . .	11
<b>2 Text Analytics Prolusion</b>	<b>13</b>
2.1 Overview . . . . .	13
2.2 Terms . . . . .	14
2.3 Framework . . . . .	15
2.3.1 Processing . . . . .	15
2.3.2 Insight . . . . .	16
2.3.3 Visualisation . . . . .	16
<b>3 Program Structure &amp; Development</b>	<b>18</b>
3.0.1 Why R . . . . .	18
3.0.2 Why Shiny . . . . .	18
3.0.3 Why Tidyverse . . . . .	19
3.0.4 Why Functional . . . . .	19
3.0.5 Why lossless data . . . . .	20
3.0.6 Why Git . . . . .	21
3.0.7 Future Developments . . . . .	21
3.1 Program Architecture . . . . .	22
3.1.1 Package and package creation . . . . .	22
3.2 Preparation . . . . .	23

3.2.1	Importing . . . . .	23
3.2.2	Object Preparation . . . . .	25
3.2.3	Filtering . . . . .	26
3.2.4	Lemmatisation . . . . .	26
3.2.5	Stemming . . . . .	26
3.2.6	Stopwords . . . . .	27
3.2.7	Formatting . . . . .	27
3.2.8	Sectioning . . . . .	28
3.2.9	Grouping . . . . .	28
3.3	Insight . . . . .	28
3.3.1	Term Insight . . . . .	29
3.3.2	Aggregate Insight . . . . .	32
3.3.3	Wrapper . . . . .	34
3.4	Visualisation . . . . .	34
3.4.1	Score . . . . .	34
3.4.2	Distribution . . . . .	37
3.4.3	Structure . . . . .	38
3.4.4	Wrapper . . . . .	40
3.5	Application . . . . .	40
<b>4</b>	<b>Conclusion</b>	<b>42</b>
4.1	Summary . . . . .	42
4.1.1	summarise successes . . . . .	42
4.1.2	summarise failures . . . . .	42
4.1.3	general thoughts on the topic . . . . .	42
4.2	Recommendations . . . . .	42
4.2.1	educational potential of text analytics . . . . .	42
4.2.2	what else remains . . . . .	42
<b>A</b>	<b>Appendix</b>	<b>43</b>
	<b>Glossary</b>	<b>88</b>
	<b>Bibliography</b>	<b>89</b>

# List of Listings

1	Installation of package and prototypical deployment of app .	21
2	Import txt . . . . .	43
3	Import csv . . . . .	43
4	Import excel . . . . .	44
5	Import files . . . . .	44
6	Prepare text . . . . .	44
7	Manage stopwords . . . . .	45
8	Format data . . . . .	46
9	Detect and add sections . . . . .	47
10	Determine Term Frequencies . . . . .	47
11	Get n-grams . . . . .	48
12	Get n-grams frequencies . . . . .	48
13	Determine Key Words with Textrank . . . . .	49
14	Determine Term Sentiments . . . . .	49
15	Determine the Moving Average Term Sentiment . . . . .	50
16	Determine the term count over some aggregate . . . . .	50
17	Determine the Key Sections . . . . .	51
18	Determine the Aggregate Sentiments . . . . .	51
19	Insight functions wrapper . . . . .	52
20	Bind aggregate terms . . . . .	52
21	Create Bar Plot . . . . .	53
22	Create Word Cloud . . . . .	54
23	Create Histogram . . . . .	54
24	Create Kernel Density Estimate Plot . . . . .	54
25	Create Time Series Plot . . . . .	55
26	Create Page View Plot . . . . .	55
27	Create Visualisation . . . . .	56
28	UI and Server of Shiny Application . . . . .	61

# List of Tables

3.1	Primary data structure format . . . . .	26
3.2	Formatting Logic for Stopwords and Lemmatisation . . . . .	27
3.3	Bigram behaviour with missing values indicated by “X” . . . . .	30



# List of Figures

2.1	Standard data science process[29]	15
3.1	Visualisation of facetting	20
3.2	Processing screen	24
3.3	Importing a document	24
3.4	Section types	28
3.5	n-gram visualisation with facetting by chapter	30
3.6	Moving average term sentiment visualised with Time Series	32
3.7	Visualisation screen	35
3.8	Options for n-gram	36
3.9	Key words visualised with Word Cloud	37
3.10	Term Sentiment visualised with histograms, faceted by chapter	38
3.11	Aggregate Term Count over sentences visualised as a density	39
3.12	Aggregate Sentiment visualised with Page View	40

# 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 student user base. The application is to be flexible enough to deal with a range of text formats, and produce attractive output quickly.

### 1.2 Brief Background: Text Analytics

Text Analytics is comprised of a variety of processes and techniques to extract information from text. The text almost always requires some initial processing. Some of the following functions 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 reasonable output over the extent of a text.
- **Associated Words:** The meaning of a text is dependent on the structure between and within words. Looking at how words are associated, through correlation, common sequences, visualisation of sections, etc., allow for a clear high-level assessment 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 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.

### 1.2.1 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, typically composed in lisp.

More recently, major statistical programs have been incorporating text analytic features, with a few smaller text-analytic specific programs appearing. SAS[1], SPSS[2], and R[3] 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 web app for analysing text at a deep level (with something of a linguistic focus) over multiple languages, for an “educated citizen researcher”[4]. A variety of R packages aiding in text analytics were assessed and experimented with, detailed in section 1.4.

## 1.3 Background: inZight

iNZight is a statistical analysis suite developed at the University of Auckland[5]. It provides rapid visualisation of data in a user-friendly format. It is used by students at high school and higher education.

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 are those learning the fundamentals and potential of text analysis and statistics, 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.

## 1.4 Literature Review

The initial review of existing literature was informed by the constraints of 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”[6], and “An Introduction to Text Processing and Analysis with R”[7]. *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 3.0.3).

The Natural Language Processing (NLP) task view on CRAN contains a lot of high quality packages with some relation to text analytics[8]. The following packages were consulted and experimented with:

- **tidytext**[9] 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”[6]
- **tm**[[10]] 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**[11] 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.
- **qdap**[12] 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. Includes output on length of discourse for agents, turn-taking, and sentiment within passages of speech.
- **sentimentr**[13] 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**[14] 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**[15] produces animations on top of the ggplot2 package,
- **textrank**[16] implements the textrank algorithm to extract keywords automatically from a text
- Packages for obtaining text:
  - **gutenbergr**[17] can search and download texts from Project Gutenberg directly. Project Gutenberg is the largest public on-line repository of public domain books.
  - **rtweet**[18] can directly attain tweets from the social media site twitter. Requires an API key from twitter, which is an arduous process to obtain.
  - **WikipediaR**[19] can download Wikipedia pages and associated metadata directly.
- **ggpage**[20] produces impressive page-view charts with word highlighting, allowing for a clear overview of a text and it's structure.
- **udpipe**[21] performs tokenisation, parts of speech tagging (which aids textrank), and more, based on the well-recognised C++ udpipe, using the Universal Treebank
- **BTM**[22] 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. Based on C++ BTM
- **crfsuite**[23] provides a complete modelling framework, which is outside scope, but could be useful later.
- **humaniformat**[24] is useful in the analysis or removal of personal names, by providing a dictionary and properties of thousands of names.

## 1.5 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 files — in this case, import is only defined 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 Prolusion

Text analytics is often considered a sub-field of data science, which is the extraction of insight from structured and unstructured data[25].

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 simple statistics-based analysis for text, giving emphasis to summary statistics of analyses, as well as distributions of results.

Many text 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, with 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[26]. 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[27].

### 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” [28] — 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 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 (both of which this program is capable of), then these structures can be analysed in their own right. Common structures include:



- **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.
- **Sentences**
- **Chapters**
- **Documents**

## 2.3 Framework

Text analytics follows much of the same process (fig. 2.1) as many other data science analyses. I have identified three main sections of a standard text analysis, being processing, insights, and visualisation. These 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

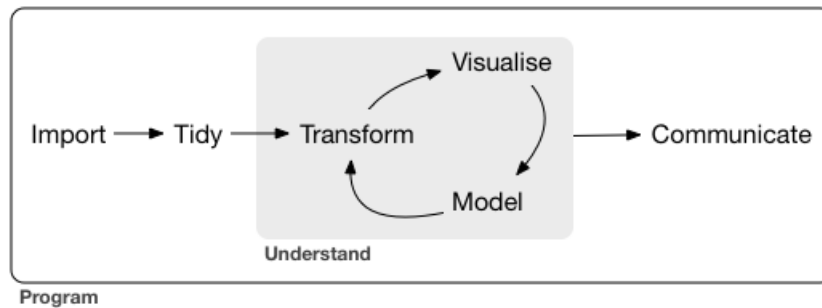


Figure 2.1: Standard data science process[29]

### 2.3.1 Processing

Processing is the first component of every text analysis, commonly returned to 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 will 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 are 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, *sectioning*. Finally, the processed data is pulled together in some standard data structure, and fed through to the next part of the text analytics process.

### 2.3.2 Insight

After text data has been made malleable enough through processing, it is ready for analysis to attain insight from it, where various statistics and scores can be computed from it.

In order to attain useful information about a text, “insight” functions can be performed on the data, where metrics relating to various properties of the text can reveal important information about a text.

Most insight functions will compute some score or statistic for texts. These may be performed at the individual word level, or at some aggregation. Key functions include the frequency count of words, n-grams and n-gram frequency, summary scores, 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 communicate the result of an analysis or exploration is through visualisation.

When visualising text, most analyses arrive at some numeric end product. Thus, these numeric scores or statistics can be visualised through common statistical 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 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 display 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

The structure and development of the program has been based mainly on compatibility with the existing system of iNZight. The current form of iNZight 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 react.js[30]. 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”[31], including “ggplot2”[32], for graphic visualisations. This text analytics program followed that lead, using both R and shiny, as well as the functional paradigm with the tidyverse.

### 3.0.1 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.

### 3.0.2 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.

### 3.0.3 Why Tidyverse

The Tidyverse is “an opinionated collection of packages designed for data science”, which is primarily used for data manipulation with the included package “dplyr”[33], and visualisation with “ggplot2”. The packages in the collection have mostly consistent syntax and style, and are built around the underlying primality 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[34]. This allows for great consistency and ease of manipulation, as there is only one datatype to manage. The 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. Emphasising just one single object (with occasional consideration of vectors) has been an unnecessary constraint for development, and all of the more recent functionality 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.

### 3.0.4 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,  $x \leftarrow x + 1$ , would not exist in a functional program, as the 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

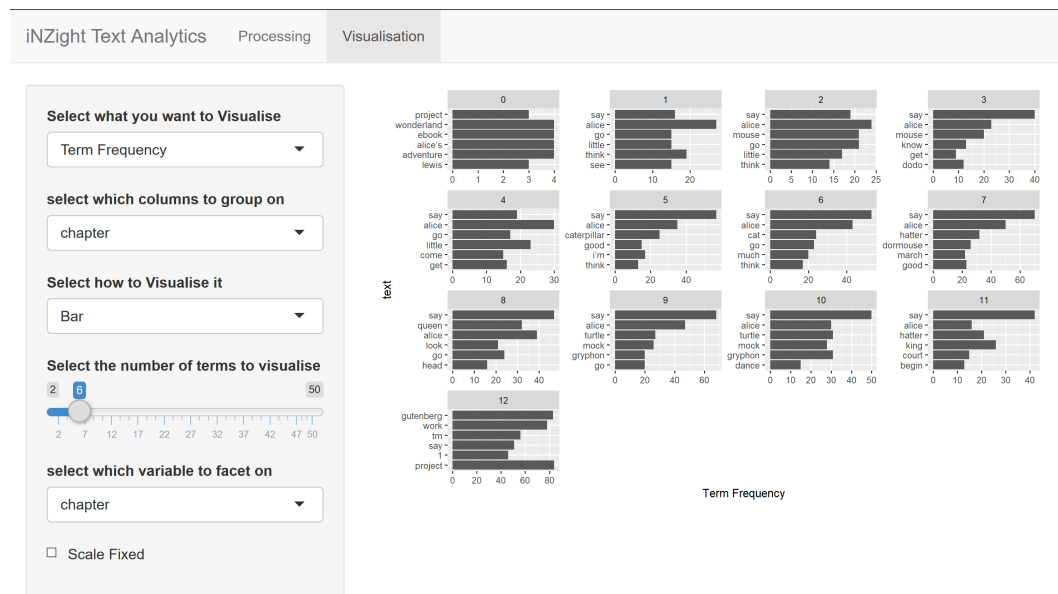


Figure 3.1: Visualisation of facetting

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.

### 3.0.5 Why lossless data

The largest issue with keeping the program functional (section 3.0.4) and tidy (section 3.0.3) 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 stopwords 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, with stopwords removed. Every row corresponds to a single word from the original text, thus the data structure is congruent with tidyverse functions.

### 3.0.6 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

### 3.0.7 Future Developments

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 3.0.3, 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 forbids the creation of other classes, as Haskell aptly demonstrates[35]. This move away from a tidy paradigm would also involve the jettisoning of the tidytext dependency, which would require working directly with the external packages wrapped by tidytext.

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 Gutenberg and Twitter. 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.

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.

There are also an enormous range of analyses that can be performed on text. With the existing, highly flexible design of this program, adding further analyses is a simple matter of adding functions to the package, and providing options in the interface for them.

## 3.1 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 user interface and a server, reliant on the package. The interface is largely generated server-side depending on the form of the analysis.

The program has been structured in this manner to make it as modular as possible. With some minor further development on the package, it could be released on CRAN as a powerful text analytics package in it's own right. Beyond this, as the web application uses the shiny framework, the UI and server exist in different functions by default — development has just ensured that the UI is a greater reflection of the server state.

### 3.1.1 Package and package creation

The package, given the working title “inzightta” (iNZight Text Analytics), serves to provide all of the background functionality for the program. In all, there are over 50 major functions defined, excluding numerous utility and anonymous functions.

Package development mostly followed a “best practices” procedure, with file structure following standard R patterns. Documentation made use of Roxygen2[36], where 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. “tidytext” 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 the “tidytext” package, poor development practices lead to several updates of the package having backwards incompatible changes, including functions outputting a completely separate class to what was output previously, with the documentation not updating to reflect these changes. Upon deep inspection of the package source following 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.



## 3.2 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 the related to preparation, which includes import and processing (fig. 3.2). At present, a representation of the created dataframe is shown.

### 3.2.1 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 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. 3.3)

#### Import txt

The simple function (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 long string.

iNZight Text Analytics
Processing
Visualisation

Import

Choose File(s)

Browse...
11-0.txt

Upload complete

Process

☒ Lemmatise

Select the Stopword Lexicon

snowball

☒ Stopwords

Prepare Text

Section By

select which column to apply filtering to

value to match

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 3.2: Processing screen

Import

Choose File(s)

Browse...
11-0.txt

Upload complete

Process

☒ Lemmatise

Select the Stopword Lexicon

snowball

☒ Stopwords

Prepare Text

Figure 3.3: Importing a document

### 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, in the specification of which column is the text column of interest. The function (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 this function (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; one that imports any file, and one making use of it 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.

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

### 3.2.2 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 (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.

The reason for the ID columns is the preservation of the structure of the text; If required, the original text can be reconstructed in entirety, sans minor punctuation differences. The following function automatically formats any data of the format returned by the initial import functions. The options are given as checkboxes and dropdowns in the app itself

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

### 3.2.3 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 on based on features seen in the analytics.

### 3.2.4 Lemmatisation

Lemmatisation is effectively the process of getting words into 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 its 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, like 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.

### 3.2.5 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. Stem-

ming 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.

### 3.2.6 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.

### 3.2.7 Formatting

The final component in preparation is to format the prepared object with the correct attributes to have formatting automated. A wrapper (listing 8) is defined that takes all combinations of stopwords and lemmatisation options and intelligently connects them for the “insight column” in a dataframe, which the insight is performed upon. 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.

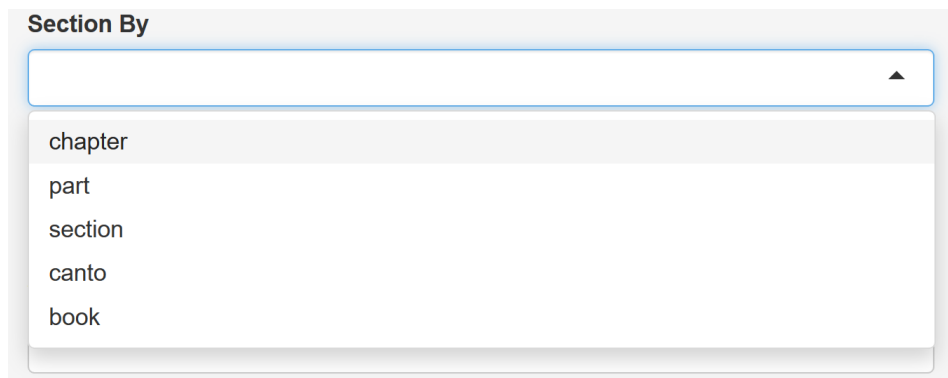


Figure 3.4: Section types

### 3.2.8 Sectioning

Plaintext, 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 (listing 9) is composed that searches and sections text based on a regular expression intended to capture a particular section. Several search functions have been created from that closure. These have been given as a drop-down selection list in the app (fig. 3.4). At a later date, advanced users could be given the option to compose their own regular expressions for sectioning.

### 3.2.9 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 the text can be grouped on based on the analytics.

## 3.3 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 insight, and higher level (aggregate) insight. Higher level insights exist where units of analysis may be sentences, chapters, documents, etc. Additional

functionality is very easy to implement, just requiring a named function that can operate on the standard data structure.

### 3.3.1 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 and output, with the output vector always matching the size of the input vector, with elements being a direct element-wise mapping.

#### 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 requires stopwords to have been previously removed, otherwise the top words will always be syntactical glue, such as articles. A function (listing 10) is defined to perform this analysis.

#### 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 project by the slightly more difficult to implement skipping-over of missing values, in the n-gram insight function (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.

The function is one of the only internally non-pure functions in the entire package, as searching sequentially along a list is far easier to conceive of in imperative terms.

The frequency of a particular bigram is given by applying the `term_frequency()` function to the newly determined n-gram column. A typical visualisation of n-grams is demonstrated in fig. 3.5.

#### 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

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”

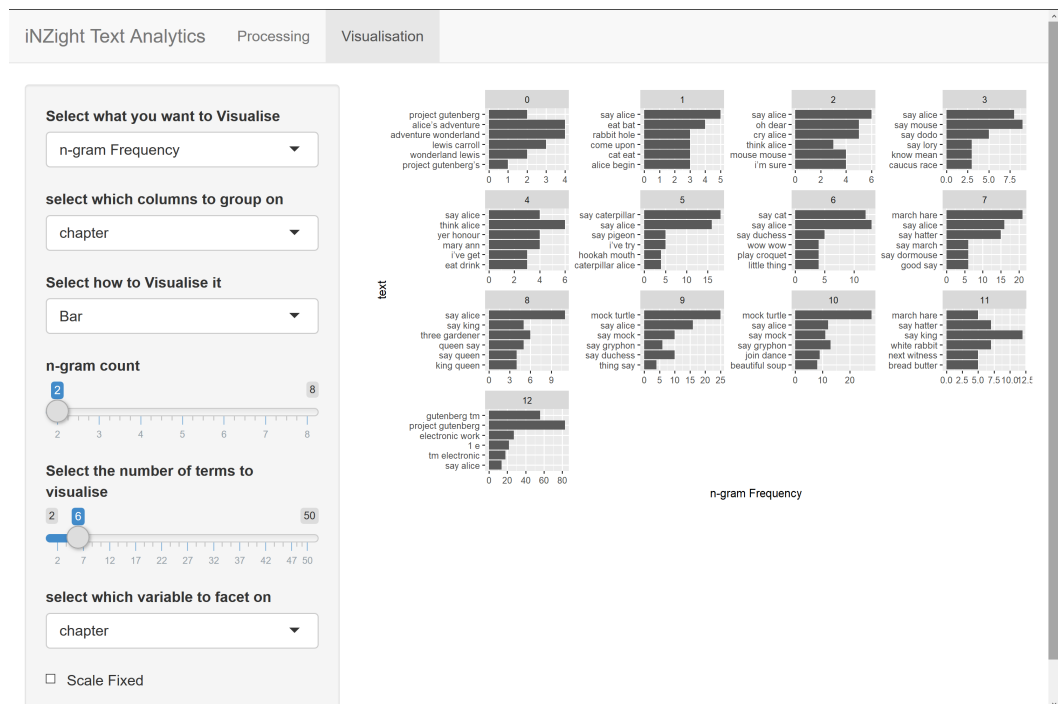


Figure 3.5: n-gram visualisation with facetting by chapter



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 (listing 13) is defined in order to implement this, on the shoulders of the `textRank` package[16].

Of note is that all words other than stopwords are treated as relevant, 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, though a 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 its 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. This function 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”.

This function (listing 14) also 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 package requires download of sentiment dictionaries through the command line at first use, making it completely inappropriate for 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. This function (listing 15) performs such a task and is flexible enough to take any statistic or lag, defaulting to be a

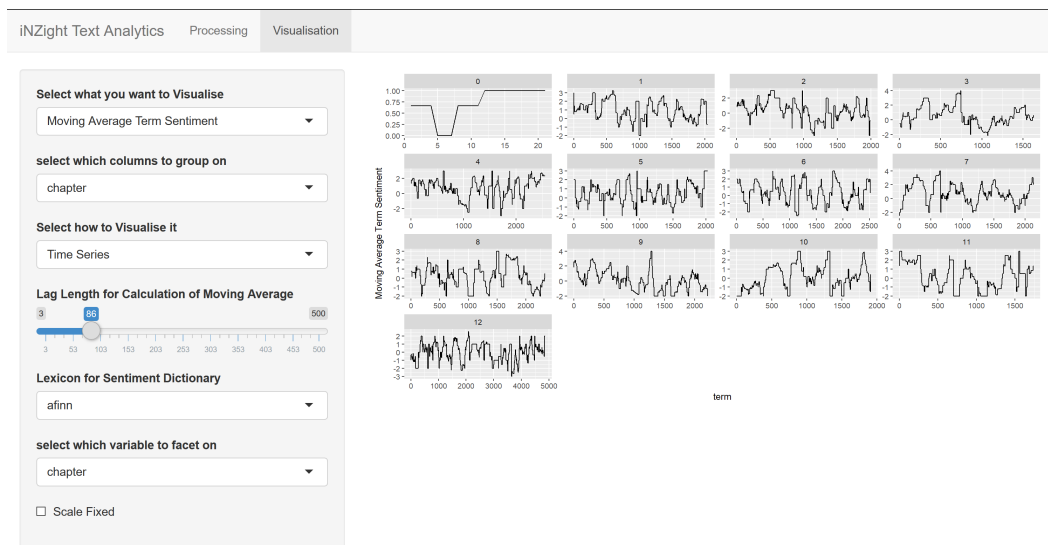


Figure 3.6: Moving average term sentiment visualised with Time Series

moving average of 10. The first `lag` terms will be `NA`, and the number of `lag` is inclusive, matching the semantics of R's indexing starting at 1. Moving Average Term Sentiment finds a natural visualisation as a time series plot, demonstrated in fig. 3.6.

### 3.3.2 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 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 (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 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 is given to compute such an analysis (listing 17). Lexrank is used as the algorithm for key-sentences, as 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 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 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 (listing 18) is given to determine a statistic of sentiment over some group. 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 this is by adding columns for each distinct word specified, giving correlations there. 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 far too large an object than what is needed[42]. Implementation will likely have to be performed from scratch in the future, unless an appropriate package is found.

### 3.3.3 Wrapper

The insights of choice can all be combined into a wrapper function (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 with the gain in clarity.

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

## 3.4 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. 3.7). Options vary by type of insight selected, for example n-grams having unique options related, as in fig. 3.8.

### 3.4.1 Score

Most insight functions return some numeric measure, typically used as a score. Visualisations that illustrate the score directly fit into the “score”

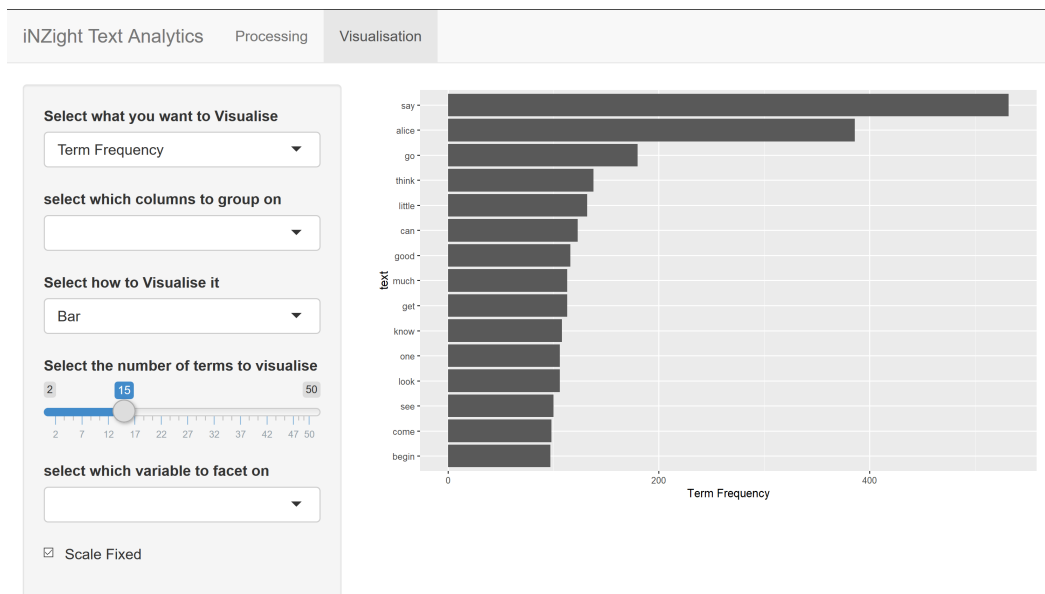


Figure 3.7: Visualisation screen

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 (listing 21) has been defined to create bar plots for the data type created in this package. 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 non-sensical coupling of such a function with the package, and the need to remove dependencies on `tidytext`.

## Word Cloud

Word clouds depict a random layout of words of varying sizes reflecting their scores, and a function has been created to implement them (listing 22). They are 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

**Select what you want to Visualise**

n-gram Frequency ▼

**select which columns to group on**

▼

**Select how to Visualise it**

Bar ▼

**n-gram count**

2 4 8

2 3 4 5 6 7 8

**Select the number of terms to visualise**

2 6 50

2 7 12 17 22 27 32 37 42 47 50

**select which variable to facet on**

▼

☒ Scale Fixed

Figure 3.8: Options for n-gram

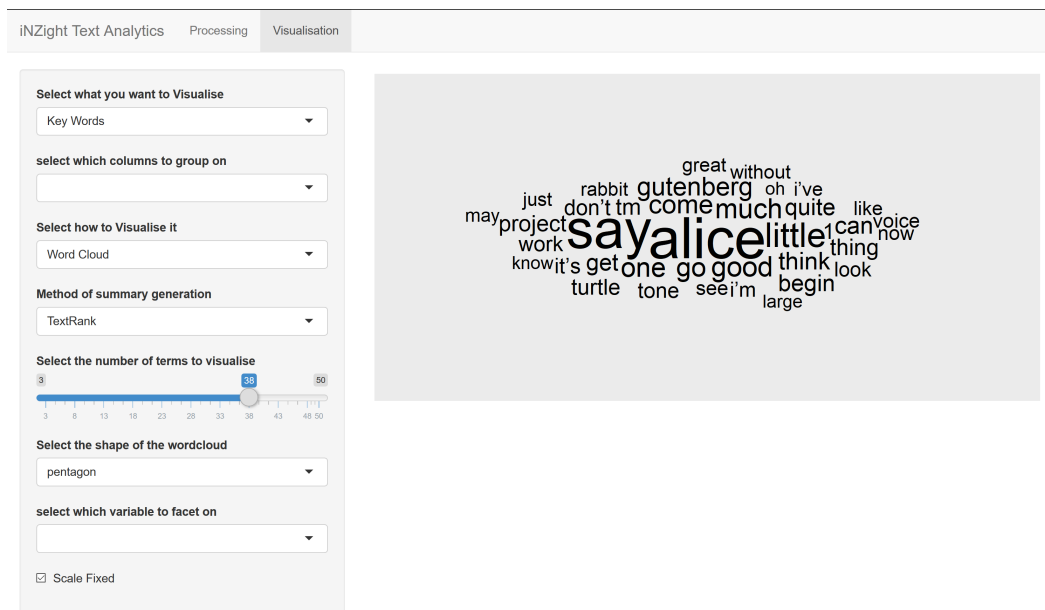


Figure 3.9: Key words visualised with Word Cloud

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. 3.9.

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.

### 3.4.2 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 faceted. Over a large dataset that free-form survey responses may provide, it is useful to calculate the mean sentiment for each response, the assess the resulting distribution — this program provides such capabilities.

### 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 3.3.1 and fig. 3.10). A function for visualisation in the form



Figure 3.10: Term Sentiment visualised with histograms, faceted by chapter

of a histogram was created (listing 23).

## Density

To visualise the density estimation as a continuous distribution for some score, a function (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 3.3.2). In the application, they are treated as any other visualisation, just requiring selection from a drop-down menu, as in fig. 3.11.

### 3.4.3 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.

## 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 func-



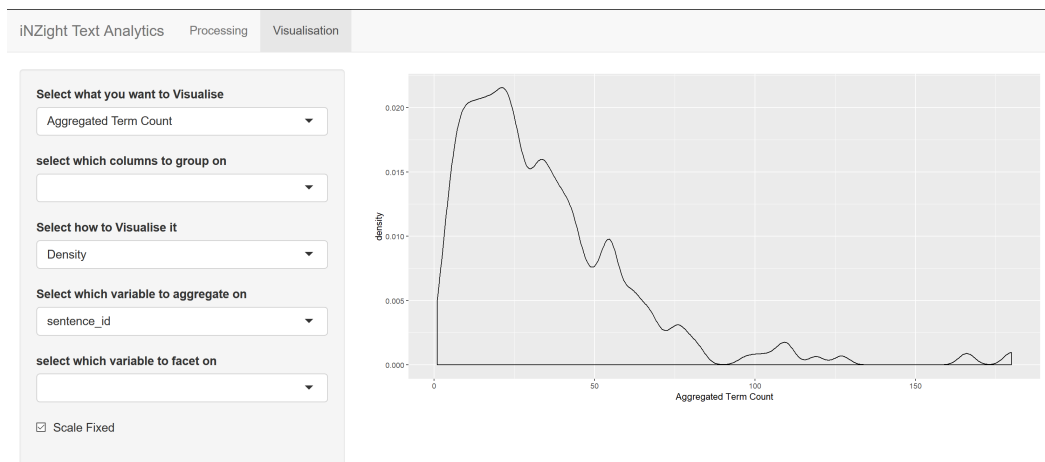


Figure 3.11: Aggregate Term Count over sentences visualised as a density

tion (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 3.3.1 (listing 15).

## Page View

The concept and functional implementation (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. 3.12).

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

The package “`ggpage`” is set up for making immediate plots, but by using the constructor `ggpage_build()` and `ggpage_plot()`, complex functions can be formed in the immediate representation from build before plotting[20]. Facetting is an issue, with standard implementation lacking clear support.

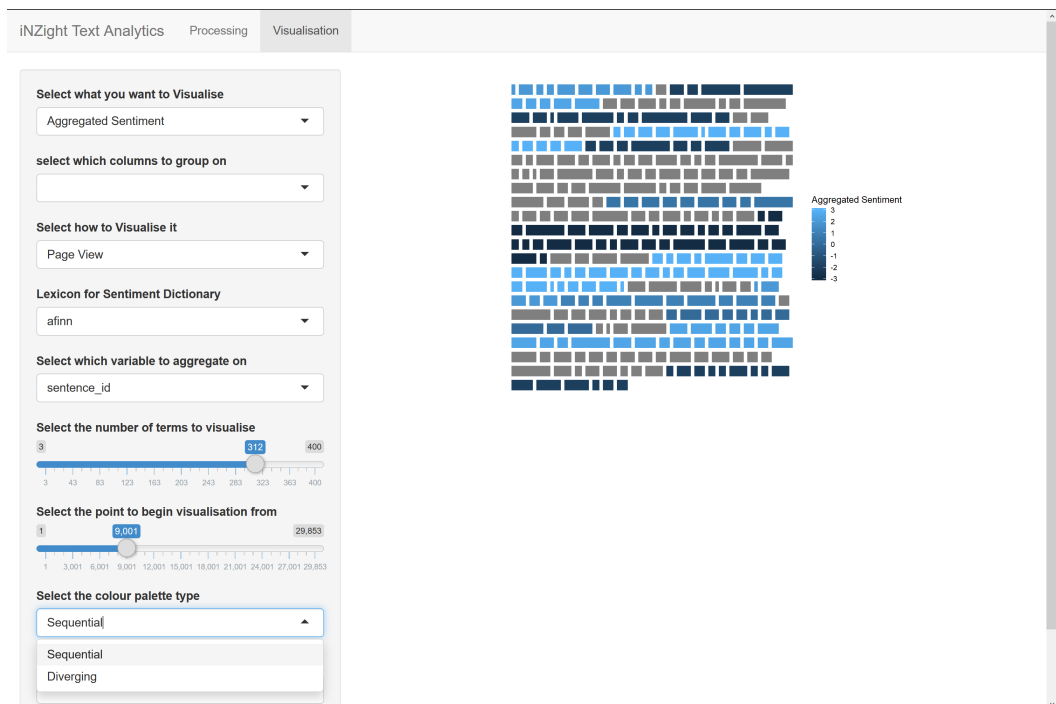


Figure 3.12: Aggregate Sentiment visualised with Page View

Creating this function was the 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.

### 3.4.4 Wrapper

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

## 3.5 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, resting on the package.

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 a text file has been imported, lemmatised, sectioned, etc. in the processing

tab, the user moves to visualisation, confronted 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 plot associated with it, so that with a few mouse clicks, meaningful output can be created. Different plots can be used in an analysis, also selected with a drop down menu. Additional options, for the control of both the analysis and visualisation, dynamically appear depending on the type of analysis and visualisation.

The application has undergone a rebuild over the course of the project, starting with an app mirroring the underlying text analytics package closely, before moving to a more user-centric style of 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 the intermediate means of achieving this request. This is effectively automatic visualisation based on the insight selection, which is closer to the functionality 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 Summary

4.1.1 summarise successes

4.1.2 summarise failures

4.1.3 general thoughts on the topic

### 4.2 Recommendations

4.2.1 educational potential of text analytics

4.2.2 what else remains

## Appendix A

# Appendix

The following pages include all of the source code needed to build the package and application. This includes a copy of the documentation for the R package, which was automatically generated through the Roxygen2 system from in-source comments.

---

```
1  #' Import text file
2  #'
3  #' @param filepath a string indicating the relative or absolute
4  #'   filepath of the file to import
5  #'
6  #' @return a [tibble][tibble::tibble-package] of each row
7  #'   corresponding to a line of the text file, with the column named
8  #'   "text"
9  import_txt <- function(filepath){
10   paste(readLines(filepath), collapse = "\n") %>%
11     tibble::tibble(text=.)
12 }
```

---

Listing 2: Import txt

---

```
1  #' Import csv file
2  #'
3  #' @param filepath a string indicating the relative or absolute
4  #'   filepath of the file to import
5  #'
6  #' @return a [tibble][tibble::tibble-package] of each row
7  #'   corresponding to a line of the text file, with the column named
8  #'   "text"
9  import_csv <- function(filepath){
10   readr::read_csv(filepath) ## %>%
11     ## table_textcol()
12 }
```

---

Listing 3: Import csv

---

```
1  #' Import excel file
2  #'
3  #' @param filepath a string indicating the relative or absolute
```

```

4 #'      filepath of the file to import
5 #'
6 #' @return a [tibble][tibble::tibble-package] of each row
7 #'      corresponding to a line of the text file, with the column
8 #'      named "text"
9 import_excel <- function(filepath){
10   readxl::read_excel(filepath) ## %>%
11   ## table_textcol()
12 }

```

---

Listing 4: Import excel

```

1 #' Base case for file import
2 #'
3 #' @param filepath string filepath of file for import
4 #'
5 #' @return imported file with document id
6 import_base_file <- function(filepath){
7   filetype <- get_filetype(filepath)
8   filename <- basename(filepath)
9   if (filetype == "csv"){
10     imported <- import_csv(filepath)
11   } else if (filetype == "xlsx" | filetype == "xls") {
12     imported <- import_excel(filepath)
13   } else {
14     imported <- import_txt(filepath)
15   }
16   imported %>%
17     dplyr::mutate(doc_id = filename)
18 }
19
20 #' Import any number of files
21 #'
22 #' @param filepaths char vector of filepaths
23 #'
24 #' @return a [tibble][tibble::tibble-package] imported files with
25 #'      document id
26 #'
27 #' @export
28 import_files <- function(filepaths){
29   filepaths %>%
30     purrr::map(import_base_file) %>%
31     dplyr::bind_rows()
32 }

```

---

Listing 5: Import files

```

1 text_prep <- function(data){
2   data %>%
3     tidytext::unnest_tokens(output = sentence, input = text,
4                             token = "sentences", to_lower = FALSE) %>%
5     dplyr::mutate(sentence_id = dplyr::row_number()) %>%
6     dplyr::group_by(sentence_id, add=TRUE) %>%
7     dplyr::group_modify(~ {
8       .x %>%
9         tidytext::unnest_tokens(output = word, input = sentence,
10                                 token = "words", to_lower=FALSE) %>%
11         dplyr::mutate(word_id = dplyr::row_number())
12     }) %>%
13     ungroup_by("sentence_id")
14 }

```

---

## Listing 6: Prepare text

---

```
1  #' Gets stopwords from a default list and user-provided list
2  #'
3  #' @param lexicon a string name of a stopwords list, one of "smart",
4  #'       "snowball", or "onix"
5  #'
6  #' @param addl user defined character vector of additional stopwords,
7  #'       each element being a stopwords
8  #'
9  #' @return a [tibble][tibble::tibble-package] with one column named "word"
10 get_sw <- function(lexicon = "snowball", addl = NA){
11   addl_char <- as.character(addl)
12   tidytext::get_stopwords(source = lexicon) %>%
13     dplyr::select(word) %>%
14     dplyr::bind_rows(., tibble::tibble(word = addl_char)) %>%
15     stats::na.omit() %>%
16     purrr::as_vector() %>%
17     tolower() %>%
18     as.character()
19 }
20
21 #' determine stopwords status
22 #'
23 #' @param .data vector of words
24 #'
25 #' @param ... arguments of get_sw
26 #'
27 #' @return a [tibble][tibble::tibble-package] equivalent to the input
28 #'   dataframe, with an additional stopwords column
29 #'
30 #' @export
31 determine_stopwords <- function(.data, ...){
32   sw_list <- get_sw(...)
33   .data %in% sw_list
34 }
```

---

## Listing 7: Manage stopwords

---

```
1  #' takes imported one-line-per-row data and prepares it for later analysis
2  #'
3  #' @param .data tibble with one line of text per row
4  #'
5  #' @param lemmatize boolean, whether to lemmatize or not
6  #'
7  #' @param stopwords boolean, whether to remove stopwords or not
8  #'
9  #' @param sw_lexicon string, lexicon with which to remove stopwords
10 #'
11 #' @param addl_stopwords char vector of user-supplied stopwords
12 #'
13 #' @return a [tibble][tibble::tibble-package] with one token per line,
14 #'   stopwords removed leaving NA values, column for analysis named
15 #'   "text"
16 #'
17 #' @export
18 format_data <- function(.data, lemmatize=TRUE, stopwords=TRUE,
19   sw_lexicon="snowball", addl_stopwords=NA){
20   formatted <- .data %>%
21     text_prep()
22
23   text <- ifexp(lemmatize,
```

---

```

24         ifexp(stopwords,
25               dplyr::mutate(formatted,
26                             lemma = tolower(textstem::lemmatize_words(word)),
27                             stopword = determine_stopwords(lemma,
28                                                             sw_lexicon,
29                                                             addl_stopwords),
30                             text = dplyr::if_else(stopword,
31                                                    as.character(NA),
32                                                    lemma)),
33               dplyr::mutate(formatted,
34                             lemma = tolower(textstem::lemmatize_words(word)),
35                             text = lemma)),
36         ifexp(stopwords,
37               dplyr::mutate(formatted,
38                             stopword = determine_stopwords(word,
39                                                             sw_lexicon,
40                                                             addl_stopwords),
41                             text = dplyr::if_else(stopword,
42                                                    as.character(NA),
43                                                    word)),
44               dplyr::mutate(formatted, text = word)))
45     return(text)
46 }

```

---

## Listing 8: Format data

---

```

1  #' creates a search closure to section text
2  #'
3  #' @param search a string regexp for the term to seperate on, e.g. "Chapter"
4  #'
5  #' @return closure over search expression
6  get_search <- function(search){
7    function(.data){
8      .data %>%
9        stringr::str_detect(search) %>%
10       purrr::accumulate(sum, na.rm=TRUE)
11    }
12  }
13
14  #' sections text based on chapters
15  #'
16  #' @param .data vector to section
17  #'
18  #' @return vector of same length as .data with chapter numbers
19  #'
20  #' @export
21  get_chapters <- get_search("^([\\s]*[Cc][Hh][Aa]?[Pp][Tt]([Ee][Rr])?")
22
23  #' sections text based on parts
24  #'
25  #' @param .data vector to section
26  #'
27  #' @return vector of same length as .data with part numbers
28  #'
29  #' @export
30  get_parts <- get_search("^([\\s]*[Pp]([Aa][Rr])?[Tt]")
31
32  #' sections text based on sections
33  #'
34  #' @param .data vector to section
35  #'
36  #' @return vector of same length as .data with section numbers

```



```

37 #'
38 #' @export
39 get_sections <- get_search("^\\\\s*([Ss][Ss])|([Ss][Ee][Cc][Tt][Ii][Oo][Nn])")
40
41 #' sections text based on cantos
42 #'
43 #' @param .data vector to section
44 #'
45 #' @return vector of same length as .data with canto numbers
46 #'
47 #' @export
48 get_cantos <- get_search("(?i)canto (XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$")
49
50 #' sections text based on book
51 #'
52 #' @param .data vector to section
53 #'
54 #' @return vector of same length as .data with book numbers
55 #'
56 #' @export
57 get_books <- get_search("(?i)book$")
58
59 #' Adds section column to dataframe
60 #'
61 #' @param .data dataframe formatted as per output of prep process
62 #'
63 #' @param section_by character name of what to section over
64 #'
65 #' @return input dataframe with additional section column
66 #'
67 #' @export
68 section <- function(.data, section_by){
69   sec_table <- list("chapter" = get_chapters,
70                     "part" = get_parts,
71                     "section" = get_sections,
72                     "canto" = get_cantos,
73                     "book" = get_books)
74   .data %>%
75     dplyr::mutate(!! section_by := sec_table[[section_by]](word))
76 }

```

---

Listing 9: Detect and add sections

```

1 #' Determine term frequency
2 #'
3 #' @param .data character vector of terms
4 #'
5 #' @return numeric vector of term frequencies
6 #'
7 #' @export
8 term_freq <- function(.data){
9   .data %>%
10     tibble::enframe() %>%
11     dplyr::add_count(value) %>%
12     dplyr::mutate(n = dplyr::if_else(is.na(value),
13                                     as.integer(NA),
14                                     n)) %>%
15     dplyr::pull(n)
16 }

```

---

Listing 10: Determine Term Frequencies

---

```

1  #' Returns the n-grams, skipping NA values
2  #'
3  #' @param .data vector to get n-grams from
4  #'
5  #' @param n number of n-grams to attain
6  #'
7  #' @return n-gram vector without NA values
8  #'
9  #' @export
10 get_ngram <- function(.data, n){
11   main_n <- n
12   ngrams <- rep(NA_character_, length(.data))
13   for (i in seq_along(.data)){
14     if (i - 1 > length(.data) - n) break
15     if (is.na(.data[i])){
16       next
17     } else {
18       ngram <- .data[i]
19     }
20     j <- i + 1
21     while(n > 1){
22       if (j > length(.data)){
23         ngram <- NA_character_
24         break
25       }
26       if (is.na(.data[j])){
27         j <- j + 1
28       } else {
29         ngram <- paste(ngram, .data[j])
30         n <- n - 1
31         j <- j + 1
32       }
33     }
34     ngrams[i] <- ngram
35     n <- main_n
36   }
37   return(ngrams)
38 }

```

---

Listing 11: Get n-grams

---

```

1  #' NOT FOR PRODUCTION - STILL IN TESING. Returns the count of n-grams, skipping NA values
2  #'
3  #' @param .data vector to get n-grams from
4  #'
5  #' @param n number of n-grams to attain
6  #'
7  #' @return count of each associated n-gram
8  #'
9  #' @export
10 ngram_freq <- function(.data, n){
11   term_freq(get_ngram(.data, n))
12 }

```

---

Listing 12: Get n-grams frequencies

---

```

1  #' Determine textrank score for vector of words
2  #'
3  #' @param .data character vector of words
4  #'

```

---

```

5  #' @param summ_method method to use for summarisation: textrank or
6  #'      lewrank. Doesn't do anything yet
7  #'
8  #' @return vector of scores for each word
9  #'
10 #' @export
11 keywords_tr <- function(.data, summ_method){
12   relevent <- !is.na(.data)
13   tr <- textrank::textrank_keywords(.data, relevent, p=+Inf)
14   score <- tr$pagerank$vector %>% tibble::enframe()
15   data <- .data %>% tibble::enframe("number", "name")
16   dplyr::full_join(data, score, by="name") %>%
17     dplyr::pull(value)
18 }

```

---

Listing 13: Determine Key Words with Textrank

---

```

1  #' Determine sentiment of terms
2  #'
3  #' @param .data vector of terms
4  #'
5  #' @param lexicon sentiment lexicon to use, based on the corpus
6  #'      provided by tidytext
7  #'
8  #' @return vector with sentiment score of each word in the vector
9  #'
10 #' @export
11 term_sentiment <- function(.data, lexicon="afinn"){
12   data <- tibble::enframe(.data, "number", "word")
13   tidytext::get_sentiments(lexicon) %>%
14     dplyr::select(word, value) %>%
15     dplyr::right_join(data, by="word") %>%
16     dplyr::pull(value)
17 }

```

---

Listing 14: Determine Term Sentiments

---

```

1  #' Determine the lagged sentiment of terms
2  #'
3  #' @param .data vector of terms
4  #'
5  #' @param lexicon sentiment lexicon to use, based on the corpus
6  #'      provided by tidytext
7  #'
8  #' @param lag how many (inclusive) terms to compute statistic over
9  #'
10 #' @param statistic base statistic used to summarise the data, capable
11 #'      of taking an na.rm argument
12 #'
13 #' @return vector with lagged sentiment score of each term in the input vector
14 #'
15 #' @export
16 ma_term_sentiment <- function(.data, lexicon="afinn", lag = 10, statistic = mean){
17   sents <- term_sentiment(.data, lexicon)
18   ## lagged_sents <- rep(NA, length(sents))
19   ## for (i in seq(lag, length(sents))){
20   ##   lagged_sents[i] <- statistic(sents[(seq(i - lag + 1, i))], na.rm = TRUE)
21   ## }
22   ## lagged_sents
23   c(rep(NA, lag - 1),
24     sapply(seq(lag, length(sents)),

```

```

25         function(i){x <- statistic(sents[[seq(i - lag + 1, i)]],
26                                na.rm = TRUE)
27                                ifelse(is.nan(x),
28                                       NA,
29                                       x)
30        })
31 }

```

---

### Listing 15: Determine the Moving Average Term Sentiment

---

```

1  #' Determine the number of terms at each aggregate level
2  #'
3  #' @param .data character vector of terms
4  #'
5  #' @param aggregate_on vector to split .data on for insight
6  #'
7  #' @return vector of number of terms for each aggregate level, same
8  #'         length as .data
9  #'
10 #' @export
11 term_count <- function(.data, aggregate_on){
12   split(.data, aggregate_on) %>%
13     purrr::map(function(x){rep(length(x), length(x))}) %>%
14     dplyr::combine()
15 }

```

---

### Listing 16: Determine the term count over some aggregate

---

```

1  #' get score for key sentences as per Lexrank
2  #'
3  #' @param .data character vector of words
4  #'
5  #' @param summ_method method to use for summarisation: textrank or
6  #'         lexrank. Doesn't do anything yet
7  #'
8  #' @param aggregate_on vector to aggregate .data over; ideally, sentence_id
9  #'
10 #' @return lexrank scores of aggregates
11 #'
12 #' @export
13 key_aggregates <- function(.data, aggregate_on, summ_method){
14   ## prepare .data for lexrank
15   base <- tibble::tibble(word = !! .data, aggregate = aggregate_on)
16   aggregated <- base %>%
17     dplyr::group_by(aggregate) %>%
18     stats::na.omit() %>%
19     dplyr::summarise(sentence = paste(word, collapse = " ")) %>%
20     dplyr::mutate(sentence = paste0(sentence, "."))
21   ## lexrank
22   lr <- aggregated %>%
23     dplyr::pull(sentence) %>%
24     lexRank::lexRank(., n=length(.), removePunc = FALSE, returnTies = FALSE,
25                      removeNum = FALSE, toLower = FALSE, stemWords = FALSE,
26                      rmStopWords = FALSE, Verbose = TRUE)
27   ## match lexrank output to .data
28   lr %>%
29     dplyr::distinct(sentence, .keep_all = TRUE) %>%
30     dplyr::full_join(aggregated, by="sentence") %>%
31     dplyr::full_join(base, by="aggregate") %>%
32     dplyr::arrange(aggregate) %>%
33     dplyr::pull(value)

```

34 }

---

### Listing 17: Determine the Key Sections

---

```
1  #' Get statistics for sentiment over some group, such as sentence.
2  #'
3  #' @param .data character vector of words
4  #'
5  #' @param aggregate_on vector to aggregate .data over; ideally,
6  #'   sentence_id, but could be chapter, document, etc.
7  #'
8  #' @param lexicon as per term sentiment
9  #'
10 #' @param statistic function that accepts na.rm argument; e.g. mean,
11 #'   median, sd.
12 #'
13 #' @return sentiment of same length as input vector aggregated over the aggregate_on vector
14 #'
15 #' @export
16 aggregate_sentiment <- function(.data, aggregate_on, lexicon = "afinn", statistic = mean){
17   tibble::enframe(.data, "nil1", "word") %>%
18     dplyr::bind_cols(tibble::enframe(aggregate_on, "nil2", "aggregate")) %>%
19     dplyr::select(word, aggregate) %>%
20     dplyr::mutate(sentiment = term_sentiment(word, lexicon)) %>%
21     dplyr::group_by(aggregate) %>%
22     dplyr::mutate(aggregate_sentiment =
23       (function(.x){
24         rep(statistic(.x, na.rm = TRUE), length(.x))
25       })(sentiment)) %>%
26     dplyr::pull(aggregate_sentiment)
27 }
```

---

### Listing 18: Determine the Aggregate Sentiments

---

```
1  #' perform group-aware term operations on the data
2  #'
3  #' @param .data dataframe of terms as per output of format_data
4  #'
5  #' @param operations character vector of term operations to perform
6  #'
7  #' @param ... additional arguments to the operation - only sensible for singular operations
8  #'
9  #' @return .data with operation columns added
10 #'
11 #' @export
12 get_term_insight <- function(.data, operations, ...){
13   opstable <- list("Term Frequency" = term_freq,
14     "n-gram Frequency" = ngram_freq,
15     "n-grams" = get_ngram,
16     "Key Words" = keywords_tr,
17     "Term Sentiment" = term_sentiment,
18     "Moving Average Term Sentiment" = ma_term_sentiment)
19   ops <- opstable[operations]
20   lapply(seq(length(ops)),
21     function(x){
22       name <- dplyr::sym(names(ops[x]))
23       operation <- ops[x][[1]]
24       df <- dplyr::mutate(.data,
25         !!name := operation(text, ...))
26       df[names(ops[x])]
27     }) %>%
```

```

28     dplyr::bind_cols(.data, .)
29   }
30
31   #' perform group-aware aggregate operations on the data
32   #'
33   #' @param .data dataframe of terms as per output of format_data
34   #'
35   #' @param operations character vector of operations to perform
36   #'
37   #' @param aggregate_on character name of the column to perform aggregate operations on
38   #'
39   #' @param ... additional arguments to the operation - only sensible for singular operations
40   #'
41   #' @return .data with operation columns added
42   #'
43   #' @export
44   get_aggregate_insight <- function(.data, operations, aggregate_on, ...){
45     opstable <- list("Aggregated Term Count" = term_count,
46                     "Key Sections" = key_aggregates,
47                     "Aggregated Sentiment" = aggregate_sentiment,
48                     "Bound Aggregates" = bind_aggregation)
49     ops <- opstable[operations]
50     lapply(seq(length(ops)),
51            function(x){
52              name <- dplyr::sym(names(ops[x]))
53              operation <- ops[x][[1]]
54              agg_on <- dplyr::sym(aggregate_on)
55              df <- if (names(ops[x]) == "Bound Aggregates"){
56                dplyr::mutate(.data,
57                              !!name := operation(word, !! agg_on))
58              } else {
59                dplyr::mutate(.data,
60                              !!name := operation(text, !! agg_on, ...))
61              }
62              df[names(ops[x])]
63            }) %>%
64     dplyr::bind_cols(.data, .)
65   }

```

---

Listing 19: Insight functions wrapper

---

```

1   #' bind aggregate terms together
2   #'
3   #' @param data vector of terms
4   #'
5   #' @param aggregate_on vector of aggregations
6   #'
7   #' @return data with every aggregation bound, as in a sentence
8   #'
9   #'
10  bind_aggregation <- function(data, aggregate_on){
11    tibble::tibble(data, agg = aggregate_on) %>%
12      dplyr::group_by(agg) %>%
13      dplyr::mutate(bound = paste(data, collapse = " ")) %>%
14      dplyr::pull(bound)
15  }

```

---

Listing 20: Bind aggregate terms

---

```

1   #' output a ggplot column graph of the top texts from some insight function
2   #'

```

```

3 #' @param .data a dataframe containing "text" and insight columns as
4 #'   per the output of the get_(term/aggregate)_insight wrapper
5 #'   function
6 #'
7 #' @param y symbol name of the column insight was
8 #'   outputted to
9 #'
10 #' @param x symbol name of column for insight labels
11 #'
12 #' @param n number of bars to display
13 #'
14 #' @param desc bool: show bars in descending order
15 #'
16 #' @export
17 score_barplot <- function(.data, y, n = 15,
18                           x = text, desc = FALSE){
19   wrap <- 50
20   text <- dplyr::enquo(x)
21   insight_col <- dplyr::enquo(y)
22   .data %>%
23     dplyr::distinct(! text, .keep_all=TRUE) %>%
24     dplyr::arrange(dplyr::desc(! insight_col)) %>%
25     dplyr::group_modify(~{.x %>% head(n)}) %>%
26     dplyr::ungroup() %>%
27     dplyr::mutate(text = forcats::fct_reorder(shorten(! text, wrap),
28                                               !! insight_col,
29                                               .desc = desc)) %>%
30     ggplot2::ggplot(ggplot2::aes(x = text)) +
31     ggplot2::geom_col(ggplot2::aes(y = !! insight_col)) +
32     ggplot2::coord_flip()
33 }
34
35 #' Shorten some text up to n characters
36 #'
37 #' @param .data character vector
38 #'
39 #' @param n wrap length of text
40 #'
41 #' @return shortened form of .data
42 #'
43 shorten <- function(.data, n){
44   ifelse(nchar(.data) > n,
45         paste(substr(.data, 1, n), "...", sep = ""),
46         .data)
47 }

```

---

### Listing 21: Create Bar Plot

---

```

1 #' output a ggplot wordcloud graph of the top texts from some insight function
2 #'
3 #' @param .data a dataframe containing "text" and insight columns as
4 #'   per the output of the get_(term/aggregate)_insight wrapper
5 #'   function
6 #'
7 #' @param y symbol name of the column insight was
8 #'   outputted to
9 #'
10 #' @param x symbol name of column for insight labels
11 #'
12 #' @param n number of words to display
13 #'
14 #' @param shape character: shape of the wordcloud

```

```

15 #'
16 #' @export
17 score_wordcloud <- function(.data, y, n = 15,
18                             x = text, shape = "circle"){
19   text <- dplyr::enquo(x)
20   insight_col <- dplyr::enquo(y)
21   .data %>%
22     dplyr::distinct(!! text, .keep_all=TRUE) %>%
23     dplyr::arrange(dplyr::desc(!! insight_col)) %>%
24     dplyr::group_modify(~{.x %>% head(n)}) %>%
25     dplyr::ungroup() %>%
26     ggplot2::ggplot(ggplot2::aes(label = stringr::str_wrap(!! text, 30), size = !! insight_col)) +
27     ggwordcloud::geom_text_wordcloud(shape = shape, rm_outside = TRUE) +
28     ggplot2::scale_size_area(max_size = 24)
29 }

```

---

Listing 22: Create Word Cloud

---

```

1 #' output a histogram of the distribution of some function of words
2 #'
3 #' @param .data the standard dataframe, modified so the last column
4 #'   is the output of some insight function (eg. output from
5 #'   term_freq)
6 #'
7 #' @param col_name symbol name of the column insight was
8 #'   performed on
9 dist_hist <- function(.data, col_name){
10   q_col_name <- dplyr::enquo(col_name)
11   .data %>%
12     ggplot2::ggplot(ggplot2::aes(x = !! q_col_name)) +
13     ggplot2::geom_histogram()
14 }

```

---

Listing 23: Create Histogram

---

```

1 #' output a histogram of the distribution of some function of words
2 #'
3 #' @param .data the standard dataframe, modified so the last column
4 #'   is the output of some insight function (eg. output from
5 #'   term_freq)
6 #'
7 #' @param col_name symbol name of the column insight was
8 #'   performed on
9 dist_density <- function(.data, col_name){
10   q_col_name <- dplyr::enquo(col_name)
11   .data %>%
12     ggplot2::ggplot(ggplot2::aes(x = !! q_col_name)) +
13     ggplot2::geom_density()
14 }

```

---

Listing 24: Create Kernel Density Estimate Plot

---

```

1 #' output a ggplot time series plot of some insight function
2 #'
3 #' @param .data a dataframe containing "text" and insight columns as
4 #'   per the output of the get_(term|aggregate)_insight wrapper
5 #'   function
6 #'
7 #' @param y symbol name of the column insight was
8 #'   outputted to
9 #'

```



```

10 #' @export
11 struct_time_series <- function(.data, y){
12   q_y <- dplyr::enquo(y)
13   .data %>%
14     dplyr::filter(!is.na(!! q_y)) %>%
15     dplyr::mutate(term = seq_along(!! q_y)) %>%
16     ggplot2::ggplot(ggplot2::aes(term, !! q_y)) +
17     ggplot2::geom_line(na.rm = TRUE)
18 }

```

---

### Listing 25: Create Time Series Plot

```

1 #' Colours a ggpage based on an insight function
2 #'
3 #' @param .data a dataframe containing "word" and insight columns as
4 #'   per the output of the get_(term/aggregate)_insight wrapper
5 #'   function
6 #'
7 #' @param col_name symbol name of the insight column intended to
8 #'   colour plot
9 #'
10 #' @param num_terms the number of terms to visualise
11 #'
12 #' @param term_index which term to start the visualisation from
13 #'
14 #' @param palette determine coloration of palette (not yet implemented)
15 #'
16 #' @return ggplot object as per ggpage
17 #'
18 #' @export
19 struct_pageview <- function(.data, col_name, num_terms, term_index, palette){
20   end <- min(nrow(.data), term_index + num_terms)
21   q_col_name <- dplyr::enquo(col_name)
22   .data[seq(term_index, end),] %>%
23     dplyr::pull(word) %>%
24     ggpage::ggpage_build() %>%
25     dplyr::bind_cols(.data[seq(term_index, end),]) %>%
26     ggpage::ggpage_plot(ggplot2::aes(fill = !! q_col_name)) ## +
27 }

```

---

### Listing 26: Create Page View Plot

```

1 #' create a group-aware visualisation
2 #'
3 #' @param .data the standard dataframe, modified so the last column
4 #'   is the output of some insight function (eg. output from
5 #'   term_freq)
6 #'
7 #' @param vis character name of visualisation function
8 #'
9 #' @param col character name of the column to get insight from
10 #'
11 #' @param facet_by character name of the column to facet by
12 #'
13 #' @param scale_fixed force scales to be fixed in a facet
14 #'
15 #' @param ... additional arguments to the visualisation
16 #'
17 #' @export
18 get_vis <- function(.data, vis, col, facet_by="", scale_fixed = TRUE, ...){
19   vistable <- list("Page View" = struct_pageview,

```

```

20         "Time Series" = struct_time_series,
21         "Bar" = score_barplot,
22         "Density" = dist_density,
23         "Histogram" = dist_hist,
24         "Word Cloud" = score_wordcloud)
25     y <- dplyr::sym(col)
26     chart <- vlistable[[vis]](.data, !! y, ...)
27     if (shiny::isTruthy(facet_by)){
28       facet_name <- dplyr::sym(facet_by)
29       q_facet_name <- dplyr::enquo(facet_name)
30       return(chart + ggplot2::facet_wrap(ggplot2::vars(!! q_facet_name),
31                                         scales = ifelse(vis == "struct_pageview" | scale_fixed,
32                                                         "fixed",
33                                                         "free")))
34     } else {
35       return(chart)
36     }
37   }
38 }

```

---

### Listing 27: Create Visualisation

---

```

1  library(shiny)
2  library(inzightta)
3  library(rlang)
4
5  ui <- navbarPage("iNZight Text Analytics",
6                  tabPanel("Processing",
7                            sidebarLayout(
8                              sidebarPanel(
9                                tags$p("Import"),
10                                fileInput("file1", "Choose File(s)",
11                                          multiple = TRUE,
12                                          accept = c("text/csv",
13                                                    "text/comma-separated-values,text/plain",
14                                                    ".csv", ".xlsx", ".xls")),
15                                tags$hr(),
16                                tags$p("Process"),
17                                checkboxInput("lemmatise", "Lemmatise"),
18                                uiOutput("sw_lexicon"),
19                                checkboxInput("stopwords", "Stopwords"),
20                                actionButton("prep_button", "Prepare Text"),
21                                selectInput("section_by", "Section By",
22                                           list("", "chapter", "part", "section", "canto", "book")),
23                                uiOutput("vars_to_filter"),
24                                textInput("filter_pred", "value to match", ""),
25                                mainPanel(
26                                  tableOutput("table")))),
27                            tabPanel("Visualisation",
28                                    sidebarLayout(
29                                      sidebarPanel(selectInput("what_vis",
30                                                                "Select what you want to Visualise",
31                                                                list("Term Frequency",
32                                                                    "n-gram Frequency",
33                                                                    "Key Words",
34                                                                    "Term Sentiment",
35                                                                    "Moving Average Term Sentiment",
36                                                                    "Aggregated Term Count",
37                                                                    "Key Sections",
38                                                                    "Aggregated Sentiment")),
39                                                                uiOutput("group_by"),
40                                                                uiOutput("insight_options"),

```

```

41                                     uiOutput("vis_options"),
42                                     uiOutput("vis_facet_by")),
43                                     mainPanel(
44                                     plotOutput("plot"))))
45                                     )
46
47 server <- function(input, output) {
48   imported <- reactive({
49     inzhightta::import_files(input$file1$datapath)})
50   prepped <- eventReactive(input$prep_button, {
51     imported() %>%
52     format_data(input$lemmatise, input$stopwords, input$sw_lexicon, NA)})
53   sectioned <- reactive({
54     data <- prepped()
55     if (isTruthy(input$section_by)){
56       data <- data %>%
57       section(input$section_by)}
58     data})
59   filtered <- reactive({
60     data <- sectioned()
61     if (isTruthy(input$filter_var) &
62         isTruthy(input$filter_pred)){
63       data <- data %>%
64       dplyr::filter(! dplyr::sym(input$filter_var) == input$filter_pred)}
65     data})
66   grouped <- reactive({
67     data <- filtered()
68     if (isTruthy(input$group_var)){
69       data <- data %>%
70       dplyr::group_by(! dplyr::sym(input$group_var))}
71     data})
72   output$table <- renderTable({
73     filtered() %>% head(300)})
74   output$sw_lexicon <- renderUI(selectInput("sw_lexicon", "Select the Stopword Lexicon",
75     stopwords::stopwords_getsources()))
76   output$vars_to_filter <- renderUI(selectInput("filter_var",
77     "select which column to apply filtering to",
78     c("", names(sectioned())) %||% c("")))
79   output$group_by <- renderUI(selectInput("group_var",
80     "select which columns to group on",
81     c("", names(filtered())) %||% c("")))
82   output$insight_options <- renderUI({
83     switch(input$what_vis,
84       "Term Frequency" = selectInput("vis_type",
85         "Select how to Visualise it",
86         list("Bar",
87             "Word Cloud",
88             "Page View",
89             "Time Series",
90             "Density",
91             "Histogram")),
92       "n-gram Frequency" = tagList(selectInput("vis_type",
93         "Select how to Visualise it",
94         list("Bar",
95             "Word Cloud",
96             "Page View",
97             "Time Series",
98             "Density",
99             "Histogram")),
100         sliderInput("n_gram",
101           "n-gram count",
102           2, 8, 2)),

```

```

103 "Key Words" = tagList(selectInput("vis_type",
104     "Select how to Visualise it",
105     list("Bar",
106         "Word Cloud",
107         "Page View",
108         "Time Series",
109         "Density",
110         "Histogram")),
111     selectInput("summ_method",
112         "Method of summary generation",
113         list("TextRank", "LexRank"))),
114 "Term Sentiment" = tagList(selectInput("vis_type",
115     "Select how to Visualise it",
116     list("Page View",
117         "Word Cloud",
118         "Time Series",
119         "Bar",
120         "Density",
121         "Histogram")),
122     selectInput("sent_lex",
123         "Lexicon for Sentiment Dictionary",
124         list("afinn", "bing",
125             "loughran", "nrc"))),
126 "Moving Average Term Sentiment" = tagList(selectInput("vis_type",
127     "Select how to Visualise it",
128     list("Time Series",
129         "Word Cloud",
130         "Page View",
131         "Bar",
132         "Density",
133         "Histogram")),
134     sliderInput("term_sent_lag",
135         "Lag Length for Calculation of Moving
136         3,500,50),
137     selectInput("sent_lex",
138         "Lexicon for Sentiment Dictionary",
139         list("afinn", "bing",
140             "loughran", "nrc"))),
141 "Aggregated Term Count" = tagList(selectInput("vis_type",
142     "Select how to Visualise it",
143     list("Bar",
144         "Word Cloud",
145         "Page View",
146         "Time Series",
147         "Density",
148         "Histogram")),
149     selectInput("agg_var",
150         "Select which variable to aggregate on",
151         c("", names(grouped())) %||% c("")),
152 "Key Sections" = tagList(selectInput("vis_type",
153     "Select how to Visualise it",
154     list("Bar",
155         "Word Cloud",
156         "Page View",
157         "Time Series",
158         "Density",
159         "Histogram")),
160     selectInput("summ_method",
161         "Method of summary generation",
162         list("TextRank", "LexRank")),
163     selectInput("agg_var",
164         "Select which variable to aggregate on",

```

```

165         c("", names(grouped())) %||% c("")),
166     "Aggregated Sentiment" = tagList(selectInput("vis_type",
167         "Select how to Visualise it",
168         list("Page View",
169             "Word Cloud",
170             "Time Series",
171             "Bar",
172             "Density",
173             "Histogram")),
174         selectInput("sent_lex",
175             "Lexicon for Sentiment Dictionary",
176             list("afinn", "bing",
177                 "loughran", "nrc")),
178         selectInput("agg_var",
179             "Select which variable to aggregate on",
180             c("", names(grouped())) %||% c("")))))})
181 insightful <- reactive({
182     switch(input$what_vis,
183         "Term Frequency" = get_term_insight(grouped(),
184             input$what_vis),
185         "n-gram Frequency" = get_term_insight(grouped(),
186             c("n-grams", "n-gram Frequency"),
187             input$n_gram),
188         "Key Words" = get_term_insight(grouped(),
189             input$what_vis,
190             input$summ_method),
191         "Term Sentiment" = get_term_insight(grouped(),
192             input$what_vis,
193             input$sent_lex),
194         "Moving Average Term Sentiment" = get_term_insight(grouped(),
195             input$what_vis,
196             input$sent_lex,
197             input$term_sent_lag),
198         "Aggregated Term Count" = get_aggregate_insight(grouped(),
199             c("Bound Aggregates", input$what_vis),
200             input$agg_var),
201         "Key Sections" = get_aggregate_insight(grouped(),
202             c("Bound Aggregates", input$what_vis),
203             input$agg_var,
204             input$summ_method),
205         "Aggregated Sentiment" = get_aggregate_insight(grouped(),
206             c("Bound Aggregates", input$what_vis),
207             input$agg_var,
208             input$sent_lex)))})
209 output$vis_options <- renderUI({
210     switch(input$vis_type,
211         "Word Cloud" = tagList(sliderInput("num_terms",
212             "Select the number of terms to visualise",
213             3, 50, 15),
214             selectInput("wordcloud_shape",
215                 "Select the shape of the wordcloud",
216                 list("circle",
217                     "cardioid",
218                     "diamond",
219                     "square",
220                     "triangle-forward",
221                     "triangle-upright",
222                     "pentagon",
223                     "star"))),
224         "Page View" = tagList(sliderInput("num_terms",
225             "Select the number of terms to visualise",
226             3, 400, 100),

```

```

227         sliderInput("term_index",
228                     "Select the point to begin visualisation from",
229                     1, nrow(insighted()), 1),
230         selectInput("palette",
231                     "Select the colour palette type",
232                     list("Sequential", "Diverging")),
233         "Bar" = tagList(sliderInput("num_terms", "Select the number of terms to visualise",
234                                     2,50,15)))})
235 output$vis_facet_by <- renderUI(tagList(selectInput("vis_facet",
236                                                     "select which variable to facet on",
237                                                     c("", names(grouped())) %||% c("")),
238                                         checkboxInput("scale_fixed", "Scale Fixed", value=TRUE)))
239 visualisation <- reactive({
240   switch(input$vis_type,
241     "Word Cloud" = switch(input$what_vis,
242       "n-gram Frequency" = get_vis(
243         insightful(),
244         input$vis_type,
245         input$what_vis,
246         input$vis_facet,
247         input$scale_fixed,
248         input$num_terms,
249         x = `n-grams`,
250         shape = input$wordcloud_shape),
251       "Aggregated Term Count" =,
252       "Key Sections" =,
253       "Aggregated Sentiment" = get_vis(
254         insightful(),
255         input$vis_type,
256         input$what_vis,
257         input$vis_facet,
258         input$scale_fixed,
259         input$num_terms,
260         x = `Bound Aggregates`,
261         shape = input$wordcloud_shape),
262       get_vis(insighted(),
263         input$vis_type,
264         input$what_vis,
265         input$vis_facet,
266         input$scale_fixed,
267         input$num_terms,
268         shape = input$wordcloud_shape)),
269     "Page View" = get_vis(insighted(), input$vis_type,
270       input$what_vis,
271       input$vis_facet,
272       input$scale_fixed,
273       input$num_terms,
274       input$term_index,
275       palette = input$palette),
276     "Time Series" = get_vis(insighted(), input$vis_type,
277       input$what_vis,
278       input$vis_facet,
279       input$scale_fixed),
280     "Bar" = switch(input$what_vis,
281       "n-gram Frequency" = get_vis(insighted(),
282         input$vis_type,
283         input$what_vis,
284         input$vis_facet,
285         input$scale_fixed,
286         input$num_terms,
287         x = `n-grams`),
288       "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)

```

---

Listing 28: UI and Server of Shiny Application

# 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 . . . . .	3
bind_aggregation . . . . .	3
concat_walk . . . . .	4
concat_walk_i . . . . .	4
determine_stopwords . . . . .	5
dist_density . . . . .	5
dist_hist . . . . .	6
format_data . . . . .	6
get_aggregate_insight . . . . .	7
get_bigram . . . . .	7
get_books . . . . .	8
get_cantos . . . . .	8
get_chapters . . . . .	9
get_filetype . . . . .	9
get_ngram . . . . .	10
get_parts . . . . .	10
get_search . . . . .	11
get_sections . . . . .	11
get_sw . . . . .	12
get_term_insight . . . . .	12
get_valid_input . . . . .	13
get_vis . . . . .	13
ifexp . . . . .	14
import_base_file . . . . .	14
import_csv . . . . .	15
import_excel . . . . .	15
import_files . . . . .	16
import_txt . . . . .	16
keywords_tr . . . . .	17
key_aggregates . . . . .	17
ma_term_sentiment . . . . .	18
ngram_freq . . . . .	18
score_barplot . . . . .	19
score_wordcloud . . . . .	19
section . . . . .	20
shorten . . . . .	20
struct_pageview . . . . .	21
struct_time_series . . . . .	21
table_textcol . . . . .	22
term_cooccurrence . . . . .	22
term_corr . . . . .	23
term_count . . . . .	23
term_freq . . . . .	24
term_sentiment . . . . .	24
ungroup_by . . . . .	25

---

aggregate_sentiment	<i>Get statistics for sentiment over some group, such as sentence.</i>
---------------------	--

---

**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	<i>bind aggregate terms together</i>
------------------	--------------------------------------

---

**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	<i>concat list 1 and 2, moving past NA values</i>
-------------	---

---

**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	<i>concat list 1 and 2 at index, skipping NA values</i>
---------------	---

---

**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	<i>determine stopword status</i>
---------------------	----------------------------------

---

**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	<i>output a histogram of the distribution of some function of words</i>
--------------	---

---

**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	<i>output a histogram of the distribution of some function of words</i>
-----------	---

---

**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	<i>takes imported one-line-per-row data and prepares it for later analysis</i>
-------------	--

---

**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	<i>perform group-aware aggregate operations on the data</i>
-----------------------	---

---

**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	<i>Determine bigrams</i>
------------	--------------------------

---

**Description**

Determine bigrams

**Usage**

```
get_bigram(.data)
```

**Arguments**

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

**Value**

character vector of bigrams

---

get_books	<i>sections text based on book</i>
-----------	------------------------------------

---

**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	<i>sections text based on cantos</i>
------------	--------------------------------------

---

**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	<i>sections text based on chapters</i>
--------------	--

---

**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	<i>Get filetype</i>
--------------	---------------------

---

**Description**

Get filetype

**Usage**

```
get_filetype(filepath)
```

**Arguments**

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

**Value**

filetype (string) - NA if no extension



---

get_ngram	<i>Returns the n-grams, skipping NA values</i>
-----------	--

---

**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	<i>sections text based on parts</i>
-----------	-------------------------------------

---

**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	<i>creates a search closure to section text</i>
------------	---

---

**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	<i>sections text based on sections</i>
--------------	--

---

**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	<i>Gets stopwords from a default list and user-provided list</i>
--------	--

---

**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 stopwords list, one of "smart", "snowball", or "onix"
addl	user defined character vector of additional stopwords, each element being a stop-word

**Value**

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

---

get_term_insight	<i>perform group-aware term operations on the data</i>
------------------	--

---

**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	<i>helper function to get valid input (recursively)</i>
-----------------	---

---

**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	<i>create a group-aware visualisation</i>
---------	---

---

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

---

**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	<i>Base case for file import</i>
------------------	----------------------------------

---

**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	<i>Import csv file</i>
------------	------------------------

---

**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 corresponding to a line of the text file, with the column named "text"

---

import_excel	<i>Import excel file</i>
--------------	--------------------------

---

**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 corresponding to a line of the text file, with the column named "text"

---

import_files	<i>Import any number of files</i>
--------------	-----------------------------------

---

**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	<i>Import text file</i>
------------	-------------------------

---

**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 corresponding to a line of the text file, with the column named "text"

---

keywords_tr	<i>Determine textrank score for vector of words</i>
-------------	---

---

**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	<i>get score for key sentences as per Lexrank</i>
----------------	---

---

**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	<i>Determine the lagged sentiment of terms</i>
-------------------	--

---

**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	<i>NOT FOR PRODUCTION - STILL IN TESING. Returns the count of n-grams, skipping NA values</i>
------------	---

---

**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	<i>output a ggplot column graph of the top texts from some insight function</i>
---------------	---

---

**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	<i>output a ggplot wordcloud graph of the top texts from some insight function</i>
-----------------	--

---

**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	<i>Adds section column to dataframe</i>
---------	---

---

**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	<i>Shorten some text up to n characters</i>
---------	---

---

**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	<i>Colours a ggpage based on an insight function</i>
-----------------	--

---

**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_(termlaggregate)_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	<i>output a ggplot time series plot of some insight function</i>
--------------------	--

---

**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_(termlaggregate)_insight wrapper function
y	symbol name of the column insight was outputted to

---

table_textcol	<i>Interactively determine and automatically mark the text column of a table</i>
---------------	--

---

**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_cooccurrence	<i>Determine term cooccurrences - extremely slow</i>
-------------------	--

---

**Description**

Determine term cooccurrences - extremely slow

**Usage**

```
term_cooccurrence(.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	<i>Determine term correlations - extremely slow</i>
-----------	---

---

**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	<i>Determine the number of terms at each aggregate level</i>
------------	--

---

**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	<i>Determine term frequency</i>
-----------	---------------------------------

---

**Description**

Determine term frequency

**Usage**

```
term_freq(.data)
```

**Arguments**

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

**Value**

numeric vector of term frequencies

---

term_sentiment	<i>Determine sentiment of terms</i>
----------------	-------------------------------------

---

**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	<i>helper function to ungroup for dplyr. functions equivalently to group_by() but with standard (string) evaluation</i>
------------	---

---

**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

aggregate\_sentiment, [3](#)

bind\_aggregation, [3](#)

concat\_walk, [4](#)  
concat\_walk\_i, [4](#)

determine\_stopwords, [5](#)  
dist\_density, [5](#)  
dist\_hist, [6](#)

format\_data, [6](#)

get\_aggregate\_insight, [7](#)  
get\_bigram, [7](#)  
get\_books, [8](#)  
get\_cantos, [8](#)  
get\_chapters, [9](#)  
get\_filetype, [9](#)  
get\_ngram, [10](#)  
get\_parts, [10](#)  
get\_search, [11](#)  
get\_sections, [11](#)  
get\_sw, [12](#)  
get\_term\_insight, [12](#)  
get\_valid\_input, [13](#)  
get\_vis, [13](#)

ifexp, [14](#)  
import\_base\_file, [14](#)  
import\_csv, [15](#)  
import\_excel, [15](#)  
import\_files, [16](#)  
import\_txt, [16](#)

key\_aggregates, [17](#)  
keywords\_tr, [17](#)

ma\_term\_sentiment, [18](#)

ngram\_freq, [18](#)

score\_barplot, [19](#)  
score\_wordcloud, [19](#)  
section, [20](#)  
shorten, [20](#)  
struct\_pageview, [21](#)  
struct\_time\_series, [21](#)

table\_textcol, [22](#)  
term\_cooccurrence, [22](#)  
term\_corr, [23](#)  
term\_count, [23](#)  
term\_freq, [24](#)  
term\_sentiment, [24](#)

ungroup\_by, [25](#)

# 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” [28] — 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-

tion. Different languages have different challenges in distinguishing words, where more *synthetic* languages convey meaning through changing word inflection, and *analytic* languages convey meaning through helper words. English straddles the line between these classifications, tending more towards analytic than, e.g. German.. 12

# Bibliography

- [1] SAS Institute Inc. *SAS Text Miner*. Version 15.1. 2019. URL: <https://www.sas.com/software/text-miner.html> (cit. on p. 9).
- [2] IBM Corp. *IBM SPSS Statistics for Windows*. Version 25.0. 2017 (cit. on p. 9).
- [3] 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. 9).
- [4] bnosac. *txtminer - Automated Text Analysis*. 2019. URL: <https://bnosac.be/index.php/products/txtminer> (cit. on p. 9).
- [5] 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. 9).
- [6] Julia Silge and David Robinson. *Text mining with R: A tidy approach*. ” O’Reilly Media, Inc.”, 2017 (cit. on p. 10).
- [7] 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. 10).
- [8] 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. 10).
- [9] 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. 10).
- [10] 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. 10).

- [11] 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. 10).
- [12] Tyler W. Rinker. *qdap: Quantitative Discourse Analysis Package*. 2.3.2. Buffalo, New York, 2019. URL: <http://github.com/trinker/qdap> (cit. on p. 10).
- [13] 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. 10).
- [14] 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. 10).
- [15] 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. 11).
- [16] 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. 11, 31).
- [17] 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. 11).
- [18] 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. 11).
- [19] 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. 11).
- [20] 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. 11, 39).
- [21] 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. 11).
- [22] 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. 11).

- [23] 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. 11).
- [24] 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. 11).
- [25] 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. 13).
- [26] 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. 13).
- [27] 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. 13).
- [28] Merriam-Webster Dictionary, ed. *Term — Definition of Term*. 17th Aug. 2019. URL: <https://www.merriam-webster.com/dictionary/term> (cit. on pp. 14, 88).
- [29] Hadley Wickham and Garrett Grolemund. *R for data science: import, tidy, transform, visualize, and model data.* ” O’Reilly Media, Inc.”, 2016 (cit. on p. 15).
- [30] 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. 18).
- [31] 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. 18).
- [32] 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. 18).
- [33] 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. 19).
- [34] Hadley Wickham et al. ‘Tidy data’. In: *Journal of Statistical Software* 59.10 (2014), pp. 1–23 (cit. on p. 19).

- [35] 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. 21).
- [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. 22).
- [37] Adobe Inc., ed. *Nondestructive Editing in Photoshop*. 14th Nov. 2018. URL: <https://helpx.adobe.com/photoshop/using/nondestructive-editing.html> (cit. on p. 23).
- [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. 23).
- [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. 25).
- [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. 26).
- [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. 33).
- [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. 34).
- [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. 37).