Main Dissertation Notes

Jason Cairns

September 2, 2019

Contents

1 Initial Survey

1.1 Prescribed Reading

- Text Mining with R: notes
- Text Analysis with R

1.2 Initial Consideration of Application Features

UX/UI must be intuitive, modelling the inherently non-linear workflow, while fitting in with the iNZight / lite environment. I wrote some notes on UX/UI here.

Text Classification seems to be a hot topic in sentiment analysis, but the question remains of whether it is within our scope (I suspect not). If it were, openNLP, along with some pre-made models, would likely serve this topic well. An interesting example of text classification in the Dewey Decimal System is given here.

Zipf's Law is referred to regularly in text analysis. Should we demonstrate this directly, as part of some other analysis, or not at all?

The **form of analysis** will vary enormously with different forms of text. There are some things constant for all forms of text, but a good deal is very specific. For example, the information of interest will differ between novels, discourse (interviews, plays, scripts), twitter posts, survey response data, or others. It may be worthwhile to either focus solely on one, or give the option to specify the type of text.

Data structures will change based on the text type, and the packages used. With a tidy dataframe as our base structure, it is easy enough to convert to specific objects required by various packages, harder to convert back.

There is a natural link between **text analysis and Linguistics**, and a significant amount of the terminology in the field reflects that. Our application requires far more market share than one that a mention in a third year linguistics paper provides, and so linguistics is not going to be our primary focus. Regardless, many forms of analysis require some linguistic theory, such as those dependent on Part of Speech tagging, so it is still worthwhile to keep in mind

1.3 Twitter

Twitter data looks particularly interesting, as it is a constantly updating, rich source of information. I wrote up some notes on text mining twitter here. It would be particularly interesting to view twitter data in the context of discourse analysis.

1.4 Subtitles

Subtitles are a unique form of text that would be very interesting to analyse. Subtitles for films and TV Series can be obtained easily from the site opensubtitles, though obtaining subtitles programatically may be more difficult. It clearly is possible, as VLC has an inbuilt feature, as does subsync, which is written in C#, so would require a port to R (probably not worth it for us at this point). Subtitles usually come as .srt files, and once the file is obtained, it's easy enough to import and work with it in R with the package subtools.

1.5 R Packages

Here is a useful comparison between the major text mining packages. CRAN also has a task view specifically for Natural Language Processing, offering many packages relevant to this project. Interestingly, they are split by linguistic category; Syntax, Semantics, and Pragmatics. The further from syntax the package is, the far more interesting it intuitively appears (eg. word count vs sentiment analysis). Some packages of interest include:

tidytext is a text-mining package using tidy principles, providing excellent interactivity with the tidyverse, as documented in the book Text Mining with R

tm is a text-mining framework that was the go-to for text mining in R, but appears to have been made redundant by tidytext and quanteda of late

quanteda 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 is a "quantitative discourse analysis package", an extremely rich set of tools for the analysis of discourse in text, such as may arise from plays, scripts, interviews etc. Includes output on length of discourse for agents, turn-taking, and sentiment within passages of speech. This looks to me like the most insight that could be gained from a text.

sentimentr is a rich sentiment analysis and tokenising package, with features including dealing with 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 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 produces animations on top of the ggplot package, offering powerful insights. Here is an example demonstrating Zipf's Law

textrank has the unique idea of extracting keywords automatically from a text using the pagerank algorithm (pagerank studied in depth in STATS 320) - my exploration of the package is documented here

Packages for obtaining text:

- **gutenbergr** from Project Gutenberg
- **rtweet** from Twitter
- **wikipediar** from Wikipedia

ggpage produces impressive page-view charts with features such as word highlighting, allowing for a clear overview of a text and it's structure, with probable use in our search feature function

gganimate produces animated charts, which can be useful if additional, regular, and low n dimensions exist in the data

---

Additionally, there are some packages that may not necessarily be useful for the end user, but may help for our development needs. These include:

- udpipe performs

tokenisation, parts of speech tagging (which serves as the foundation for textrank), and more, based on the well-recognised C++ udpipe library, using the Universal Treebank

- BTM 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 provides a modelling

framework, which is currently outside our current scope, but could be useful later

- In the analysis / removal of names, an important component of a text,

humaniformat is likely to be useful

- CRAN Task View: Web Technologies and Services for importing texts from the

internet

1.6 Other Text Analytics Applications

The field of text analytics applications is rather diverse, with most being general analytics applications with text analytics as a feature of the application. Some of the applications (general and specific) are given:

- txtminer is a web app for analysing text at a deep level (with something of a linguistic focus) over multiple languages, for an "educated citizen researcher"

1.7 Scope Determination

The scope of the project is naturally limited by the amount of time available to do it. As such, exploration of topics such as discourse analysis, while interesting, is beyond the scope of the project. Analysis of text must be limited to regular texts, and comparisons between them. The application must give the greatest amount of insight to a regular user, in the shortest amount of time, into what the text is actually about.

Cassidy's project was intended to create this, and I have written notes on it here.

Ultimately, I am not completely sold on the idea that term frequencies and other base-level statistics really give that clear a picture of what a text is about. It can give some direction, and it can allow for broad classification of works (eg. a novel will usually have character names at the highest frequency ranks, scientific works usually have domain specific terms), but I think word frequencies are less useful to the analyst than to the algorithms they feed into, such as tf-idf, that may be more useful. As such, I don't think valuable screen space should be taken up by low-level statistics such as term frequencies. To me, the situation is somewhat akin to Anscombe's Quartet, where the base statistics leave a good deal of information out, term frequencies being analogous to the modal values.

Additionally, sentiment is really just one part of determining the semantics of a text. I think too much focus is put on sentiment, which in practice is something of a "happiness meter". I would like to include other measurement schemes, such as readability, formality, etc.

Some kind of context in relation to the universal set of texts would be ideal as well, I think a lot of this analysis occurs in a vacuum, and insights are hard to come by - something like Google n-grams would be ideal.

I'm picturing a single page, where the analyst can take one look and have a fair idea of what a text is about. In reality it will have to be more complex than that, but that is my lead at the moment. With this in mind, I want to see keywords, more on *structure* of a text, context, and clear, punchy graphics showing not *just* sentiment, but several other key measurements.

2 Initial Feature Considerations

2.1 Introduction

The application essentially consists of a feature-space, with the area being divided in three; Processing, Within-Text Analytics, and Between-Text Analytics. This follows the general format of much of what is capable in text analysis, and what is of interest to us and our end users. The UI will likely reflect this, dividing into seperate windows/panes/tabs to accomodate. Let's look at them in turn:

2.2 Processing

In order for text to be analysed, it must be imported and processed. A lot of this is an iterative process, coming back for further processing after analysis etc. Importing will have a "type" selection ability for the user, where they can choose from a small curated list of easy-access types, such as gutenberg search, twitter, etc. The option for a custom text-type is essential, allowing .txt, and for the particularly advanced end-user, .csv.

Once the file is imported/type is downloaded, the option should exist to allow the specification of divisions in the text. In a literary work, these include "chapter", "part", "canto", etc. A twitter type would allow division by author, by tweet, etc. An important aspect of this processing is to have a clear picture of what the data should look like. Division of a text should be associated with some visualisation of the resulting structure of the text, such as a horizontal bar graph showing the raw count of text (word count) for each division - this would allow immediate insight into the correctness of the division, by sighting obvious errors immediately, and allowing fine tuning so that, for example, the known number of chapters match up with the number of divisions. We could implement a few basic division operators in regex, while following the philosophy of allowing custom input if wanted. Example regex for "Chapter" could be `/[Cc]hapter[.:]?[ ]{0,10}-?[ ]{0,10}([0-9]|[ivxIVX]*))/g`, something the end user is likely not wanting to input themselves.

Removal and transformation is another important processing step for text, with stopwords and lemmatisation being invaluable. The option should exist to remove specific types of words, which can again come from prespecified lists. An aspect worth considering is if this should be done in a table manipulation, or a model - or both, with the length of the text deciding automatically based on sensible defaults. Again, the need for a clear picture of the data is essential, with some visual indication of the data during transformation and removal essential; this could take the form of some basic statistics, such as a ranking of terms by frequencies, and some random passage chosen.

Processing multiple documents is also essential. The importation is something that has to be got right, otherwise it'll be more complex than it already is, and the end-user will lose interest before the show even begins. My initial thoughts are of a tabbed import process, with each tab holding the processing tasks for each individual document, however this won't scale well to large corpus imports.

2.3 Within-Text Analytics

Within-text analytics should have options to look at the whole text as it is, whether to look by division, or whether to look at the entire imported corpus as a whole.

A killer feature here is the production of a summary; a few key sentences that summarise the text. It's a case of using text to describe text, but done effectively, it has the potential to compress a large amount of information into a small, human-understandable object.

Related to the summary, keywords in the text will give a good indication of topics and tone of the text, as well as perhaps more grammatical notions, such as authorial word choices. There is the possibility of using keywords as a basis for other features, such as the ability to use a search engine to find related texts from the keywords.

Bigrams and associated terms are also excellent indicators of a text. Something I particularly liked in Cassidy's project was the ability to search for a term, and see what was related to it. In that case, the text was "Peter Pan", and searching for a character's name yielded a wealth of information of the emotions and events attached to the character.

Sentiment is a feature that has been heavily developed by the field of text analytics, seeing a broad variety of uses. here, it would be worth examining sentiment, by word and over the length of the text overall.

2.4 Between-Text Analytics

As in within-text analytics, between-text analytics should have options for specifying the component of the text that is of interest; here, the two major categories would be comparisons between divisions within an individual text, and comparisons between full texts.

Topic modelling gives an idea of what some topics are between texts - something odd to me is that there isn't a huge amount of information on topic modelling purely within a text, it always seems to be between texts (LDA etc.)

tf-idf for a general overview of terms more or less unique to different texts.

Summarisation between all texts would also be enormously useful.

2.5 Stopwords

After noting that stopword removal impacted important n-grams when a stopword made up some component of the n-gram, it becomes very worthwhile to not only include an active capacity to view what current stopwords exist, but also to have alternative lists of stopwords. The following summarises some research into stopwords and common practices around them;

- StackOverflow removes the top 10,000 most common english words in "related queries" for their SQL search engine (`https://stackoverflow.blog/2008/12/04/podcast-32/`)
- The stopwords `R` package includes several lists of stopwords. Among these, of note are:
  - SMART: The stopword lists based on the SMART (System for the Mechanical Analysis and Retrieval of Text)Information Retrieval System, an information retrieval system developed at Cornell University inthe 1960s.
  - snowball: It is a small string processing language designed for creating stemming algorithms for use in Information Retrieval.
  - iso: The most comprehensive stopwords for any language

The package we are using extensively, tidytext, has both SMART and snowball lists, as well as onix, which bills itself as " probably the most widely used stopword list. It covers a wide number of stopwords without getting too aggressive and including too many words which a user might search upon." Of note is that all of the lists are included in one dataframe, so it should be filtered before being used, unlike how we have been using it. snowball is clearly the shortest, and I think may be worth having as the default, with SMART (the most extensive) and onix as secondary options. We are not in the role of providing a computationally efficient search engine, only removing words that contribute little but noise.

In terms of implementation within our program, we ought to have the ability to add custom stopwords. In keeping with the philosophy of having our data clearly visible, this will necessitate a "temporary stopwords" list. In the process of implementation, we will have to make assesments of whether it will run too slowly if allowed to influence charts and output in real timme, so manual refreshes would be required. Additionally, it will be good to have a running set of statistics keeping available what has been done to the data (including more than just stopword removal)

2.6 Visualisation

With so much of the conceptual space of text analytic visualisation being taken up with far from optimal charts, there is a need to experiment with alternative visualisations; We explore some here

2.7 Text Summarisation

Wikipedia: Automatic Summarisation

Text summarisation creates enormous insight, especially from a long text. There are a variety of different techniques, of varying effectiveness and efficiency. A famous example of automatic text summarisation comes from autotoldr, a bot on reddit that automatically generates summaries of news articles in 4-5 sentences. Autotldr is powered by SMMRY, which explains it's algorithm as working through the following steps:

1. Associate words with their grammatical counterparts. (e.g "city" and "cities")
2. Calculate the occurrence of each word in the text.
3. Assign each word with points depending on their popularity.
4. Detect which periods represent the end of a sentence. (e.g "Mr." does not).
5. Split up the text into individual sentences.
6. Rank sentences by the sum of their words' points.
7. Return X of the most highly ranked sentences in chronological order.

The two main approaches to automatic summarisation are extractive and abstractive; **Extractive** uses some subset of the original text to form a summary, while **abstractive** techniques form semantic representations of the text. Here, we will stick to the clearer, simpler, extractive techniques for now.

textrank has the unique idea of extracting keywords automatically from a text using the pagerank algorithm (pagerank studied in depth in STATS 320) - my exploration of the package is documented here. At present, the R implementation of it creates errors for large text files, but it is worth exploring more into it - whether it is the implementation, or if it is the algorithm itself.

Hvidfeldt is a prolific blogger focussing on text analysis - he put up this tutorial on incorporating textrank with tidy methods: tidy textRank

Further summarisation experimentation is continued here

After further testing, I have found LexRank to work significantly faster, while generating similar results, thus being favourable for summarisation. It appears that Textrank wins in the ability to generate keywords, and does so extremely quickly. Despite the speed gain in using LexRank for summarisation, it still takes several seconds on my i5 dual-core, to run, however this is offset by the verbosity of the function assuring me that it isn't hanging.

LexRank and textRank appear to exist complimentarily to one another. Below is a brief summary of how they work

2.7.1 TextRank

TextRank essentially finds the most representative sentence of a text based on some similarity measure to other sentence.

By dividing a text into sentences, measures of similarity between every sentence is calculated (by any number of possible similarity measures), producing an adjacency matrix of a graph with nodes being sentences, edge weights being similarity. The PageRank algorithm is then run on this graph, deriving the best connected sentences, and thereby the most representative sentences. A list is produced giving sentences with their corresponding PageRank. The top n sentences can be chosen, then output in chronological order, to produce a summary.

In the generation of keywords, the same process described is typically run on unigrams, with the similarity measure being co-occurance.

2.7.2 LexRank

LexRank is essentially the same as textRank, however 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)"

2.8 Search Function

The analyst is not expected to be entirely familiar with the texts under analysis; this is partly the purpose of this program. Hence, there are likely to be terms, keywords, and relationships that the program reveals, and are a surprise to the analyst, and context is necessary to understand them. A search function has been identified as useful in meeting this problem, where a word is entered in search, and contextual passages are returned. Useful in the results would be indications of location of each passage in the greater text, as well as if multiple texts are present, the name of the text it belongs to.

2.9 Topic Modelling

Topic Modelling appears to serve a useful purpose in text analytics, with LDA being the primary implementation, requiring multiple texts, and a Document-Term Matrix. My exploration with topic modelling is located here. It could be worth investigating other forms of topic modelling, especially within-text.

[2019-05-17 Fri] I checked other forms - their complexity requires a great deal of time to understand if I want to implement them intelligently; better to stick with LDA, which, while also complex, is well used enough to be considered standard.

2.10 Sentiment Distribution

Over a large n dataset such as free-response surveys, it may be useful to calculate the sentiment for each response, and consider the statistical properties of the distribution of sentiments. Here is an exploration of free-response data forming a sentiment distribution.

2.11 Conditional Analytics

The idea of conditional analytics is of interest to me, especially for high n datasets such as large free-response surveys. Particularly, I want to know, given some condition, how does the subset behave? For example, given a negative sentiment, what is the most representative response? Or, given that some common word, what is the distribution of sentiment

2.12 Wrapper Functions

In order to begin implementation, I have defined wrapper functions for the primary features. The intention is to create a higher layer of abstraction for the features as well as ease of use. I begin with the text summarisation feature; the details are below

2.12.1 Text Summarisation

Link to src Link to test Arguments:

- x = input dataframe with column titled "word"
- n = n-many sentences
- style = style of output (chart, dataframe, text)
- dim = dimension of chart
- engine = textrank/lexrank
- type = sentences/keywords etc.

Working through, I have come to come realisation that a complete wrapper function may not necessarily be ideal; rather, a pipeline may be better

- this is because a wrapper function, with, e.g., a plotting function at the outermost layer, would require a full recalculation of the inner functions for every parameter change in the plot - what may be better is the creation of a pipeline that leaves most functions as they are but just creates more suitable objects to pass as arguments to the functions. This is something of a "memory cheap; processing expensive" principle. The display wrapper functions would then be taking complete objects only

<2019-05-22 Wed> Chris clarified the role of wrappers here being more of a "layer" level, layers being:

- word/n-gram;
  - Word frequency
  - Bigram frequency
  - pairwise word correlations
  - textrank keywords
- sentence;
  - textrank
  - lexrank
- topic level
- sentiment level

2.13 Visualisation

Visualisation of text is proving to be a more complex area than I first assumed. Prior to this project, the only visualisation I knew of was word clouds, which I have come to understand to be about as useless as an unlabelled pie chart.

Text visualisation is essentially the attempt to efficiently relay insights gained from text analytics. In the preparation-insight-visualisation layers, it is the final layer. Visualisation is not limited to just charts; for our purposes, a well crafted and formatted table may be just as good at conveying information.

The form of the insight determines the form of the visualisation. So far, insights all give a "score". Thus, the visualisation, showing a mapping between a text (categorical) and a numerical insight (numberical) varaiable, can only take a few forms, ideally showing the relative scores and ranking of specific text items, or a distribution of the entire set.

At base, nearly everything is neatly categorical-numeric, able to be represented by bars/lollipops.

Pairwise correlation is slightly different, being a numerical function of two categorical arguments; best represented in either a searchable table, or a correlation matrix

Getting more advanced, for small data, ggpage type visualisations will be excellent for sentiment and word/bigram frequency, as well as ranking keywords.

Finally, when grouping is implemented, colouring or facetting by group will be what makes this analysis package better than any competitors.

[2019-07-01 Mon] After implementing grouping, the issue with arranging bars in a barplot by rank within each group is that ggplot arranges bars through the ordering of factor levels. The problem is that each instance of a word in every group shares the same level ordering, so while a word may rank highly overall, but less than others in a particular group, it will retain the high ordering overall in the facet for that group, leading to inaccuracies.

2.14 ggpage

ggpage is an extension to ggplot to allow the rendering of text in a page-like representation as a manipulable image. Example

```r
library(tidyverse)
library(ggpage)
head(tinderbox)
```

#+RESULTS

text book <chr> <chr> 1 "A soldier came marching along the high road: \"Left, right - le... The tinder-... 2 had his knapsack on his back, and a sword at his side; he had be... The tinder-... 3 and was now returning home. As he walked on, he met a very frigh... The tinder-... 4 witch in the road. Her under-lip hung quite down on her breast, ... The tinder-... 5 "and said, \"Good evening, soldier; you have a very fine sword, ... The tinder-... 6 knapsack, and you are a real soldier; so you shall have as much ... The tinder-...

ggpage can make immediate plots, but using `ggpage_build` and `ggpage_plot`, complex functions can be formed in the immediate representation from build before plotting. The representation takes the following form:

```r
tinderbox %>%
  ggpage_build()
```

word book page line xmin xmax ymin ymax index$_{\text{line}}$ <chr> <chr> <int> <int> <dbl> <dbl> <dbl> <dbl> <chr> 1 a The tinder-box 1 1 91 90 -114 -117 1-1 2 soldier The tinder-box 1 1 99 92 -114 -117 1-1 3 came The tinder-box 1 1 104 100 -114 -117 1-1 4 marching The tinder-box 1 1 113 105 -114 -117 1-1 5 along The tinder-box 1 1 119 114 -114 -117 1-1 6 the

The tinder-box 1 1 123 120 -114 -117 1-1 7 high The tinder-box 1 1 128 124 -114 -117 1-1 8 road The tinder-box 1 1 133 129 -114 -117 1-1 9 left The tinder-box 1 1 138 134 -114 -117 1-1 10 right The tinder-box 1 1 144 139 -114 -117 1-1

This is set up solely for novels, and there is no way yet to implement grouping (as at ggpage v0.2.2.9000), but this may be useful. ggpage requires the scoring to be defined within the $\text{ggpage}_{\text{build}}$ dataframe form - we can make use of this if we apply the insight functions to it. Entirely coincidentally, we have used precisely the same naming conventions for the input dataframe to $\text{ggpage}_{\text{build}}$ (column named 'text'), and the insight functions inside $\text{ggpage}_{\text{build}}$ (working on column named "word"). Some tests are given in the test file. The primary issue with using ggpage is that the insight is applied as a *part* of the visualisation, rather than being seperate to it, as with all the others.

3 Initial Data Types Survey

The application requires the capacity to smoothly work with diverse data types. For this to occur, a test corpus must be developed, and some important data types picked out.

3.1 Test Corpus

It is essential to test on a broad variety of texts in order to create the most general base application, so a "test set" will have to be developed. All data is stored in the folder data

Must have

- Literature (eg. Dante's Divine Comedy)
- Survey response data (eg. nzqhs, Cancer Society)
- Transcript; lack of punctuation may cause difficulties in processing sentences.
- Twitter

Would be nice

- article
  - journal (scientific, social)
  - news
  - blog
  - wikipedia

- discourse
  - interview
  - subtitles
- documentation
  - product manual
  - technical user guide

3.2 Free-Response Data

Free Response Data (as in survey forms etc.) has been identified as an area of high potential for the application. Two datasets have been used to run typical text analyses upon, with the exploration here. Upon close inspection, there are subtleties worth exploring further especially in bigrams and keywords.

3.3 Data types for implementation

In the production of wrapper functions, we require data types that work well with all functions that are required. For the purpose of word-level summarisation, the following features require functions with the associated data types as arguments:

- Word frequency: `tidytext::unnest_tokens`
  - @param tbl: A data frame
- Bigram frequency: `tidytext::unnest_tokens`
  - @param tbl: A data frame
- pairwise word correlations: `widyr::pairwise_cor`
  - @param tbl: Table
  - @param: item: Item to compare; will end up in 'item1' and 'item2' columns
  - @param feature: Column describing the feature that links one item to others
- textrank keywords: `textrank::textrank_keywords`
  - @param x: a character vector of words.

Thinking even earlier in the pipeline, the processing section requires functions to remove stopwords- this requires `tidytext::unnest_tokens` again, meaning a dataframe. The issue is that if we operate on groups, then we require a function that takes a vector as argument. Perhaps more thought is needed in understanding what grouped operations should look like in text analytics. Alternatively, we could create a function that takes a dataframe as input, with the option to name groups to perform group operations upon.

Another issue that arises is the elimination of sentences and structure upon the unnesting of tokens. What may be worthwhile is to create a dataframe such as the following:

grouping vars	...	doc$_{\text{id}}$	paragraph$_{\text{id}}$	sentence$_{\text{id}}$	word$_{\text{id}}$	word

In which case, we should start at the very beginning, looking at text import wrapper functions, enabling them to output a dataframe of this type such that the remaining process is entirely predictable.

Current files for wrappers: prep-for-insight.R

Note: *[2019-06-10 Mon]*: determined that line number is more general than paragraph: paragraph can be inferred from line number.

As @ *[2019-06-13 Thu]*, I have found the dataframe form as described prior to be extremely valuable. The implementation of all wrappers should have as the aim to preserve the structure as much as possible, only adding additional columns to the dataframe resulting from the function.

3.3.1 Text Analytics wrappers

[2019-05-29 Wed]: Chris approved the datatype. Work will begin on the wrappers, using this datatype. He raised the very valid point on how pairwise corelations between words should possibly use groups as their similarity component, rather than sentences. e.g., correlation of words between survey responses. **note: groups are always nested, and conditioning is actually filtering**

Important to note: Different punctuation marks exist, and despite some visual similarities, are not recognised as equivalent on the computer: for example, "’" and "’" are different. Selecting "alice’s" as a stopword will not filter out "alice’s". While on the topic, it may be worthwhile to incorporate regex ability for the application. CLI integration would be a dream, but not so useful for school and undergraduate students.

4 Initial Considerations of Program Structure

4.1 *[2019-06-13 Thu]* Notes

- Read r-pkgs.org. Notes: A working prototype will be built before formally packaging it; this is to allow for greater flexibility and experimentation without worry about breaking the package structure. All the source code for functions are located in the src folder, grouped according to their functional category.

- Further intentions: a rigorous, clean implementation of grouping and conditioning (generalised as filtering) is something I believe to be important to make this package stand out from the crowd. Upon the function set all working, I think this would be worth pursuing. The structure of the internal datatype has been kept specifically so that grouping and filtering are efficient, lossless, and simple operations.

Dataframe form:

grouping vars	…	doc_{id}	$\text{paragraph}_{\text{id}}$	$\text{sentence}_{\text{id}}$	word_{id}	word

4.2 *[2019-07-10 Wed]* Notes

I have done some further thinking today, especially following the meeting yesterday; destructive edits to the text are a serious problem to the integrity of the text, where all insight actions require starting from scratch as soon as any different types of input are needed. An example stems from experimenting with ggpage and realising that when stopwords are removed, the structure of the text is heavily hollowed out. After some thought, my solution is the following;

Processing: Start with the importation and formatting of text, keeping every single word and it's identification, down to the capitalisation. Further options include (for example) lemmatisation, and stopwords. In keeping with the spirit of non-destructive edits, each add a column: lemmatisation adds a lemmatised form of row's word, and stopwords adds a boolean value regarding the status of the lemma. A final processing function creates a new row for the insight to be performed on, based on the processing options (to use lemmas, stopwords etc.). Groups are then declared.

Insights looks for the insight column, and adds some output column based upon it. The only changes I will have to make to the existing functions will be to look for the insight column. A potential difficulty is that they will have to be capable of dealing with missing values (now that stopwords are just removed with NA in place)

Visualisations will be exactly the same. A new, neat bit will be that ggpage is simply a case of `ggpage_build` of the original import and a `cbind`, then `ggpage_plot(aes(fill = insight))`.

(End of Solution) In addition, I have been thinking about UI. Shiny apps often have a paged, scrolling structure like a webpage, but I think text analytics may require a different format, due to the continual return to the processing stage, as well as the large amount of processing required for many operations, thus leading to slow, laggy pages. I think the "SAS format" may be a winning formula, where tickboxes, radios, and inputs on one high level page are tweaked, then a button is pressed to produce the output. This would lend itself really well to going back and tweaking, as well as the feature of code generation. It obeys the KISS principle, which wins it points in my book.

Preparation is now divided into importation, grouping, formatting, then processing. In detail:

Import bringing in text from various formats, convert to simple table

Group section text by groups, for which later operations will be uniquely performed on

Format format the text into a standard object that can be operated on

Process remove stopwords, lemmatise, filter, other lossy transformations

4.3 Note on Non-Destructive Editing

Destructive editing is the practice where the original input can't be attained 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 a programmer with full control over every possible variable assignment, but not 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 many degrees of freedom as possible. The concept of nondestructive edits is not new; graphic design relies upon it, with an example given for photoshop at the Adobe Website

4.4 *[2019-07-25 Thu]* Notes

Present thoughts on visualisation: It should be a manual process, with intentionality behind it, rather than scrolling through pre-made visualisations. This would require (i.e. make clear) a function that takes specifications of x, y, facets etc.

5 Program Dependencies

The following code is to enable tibbles internally in the package

```
## usethis namespace: start
#' @importFrom tibble tibble
## usethis namespace: end
NULL
```

I am considering using furrr for parallel or distributed processing performance enhancements, though I want to get all functionality implemented first before performing that kind of optimisation.

5.1 Helper Functions

5.1.1 Unrestricted if-expression

Having conditionals as expressions rather than statements grants the ability for direct assignment of the evaluation. Base `ifelse` and tidyverse `dplyr::if_else` impose the restriction that the output is the same shape as the test predicate. This helper removes that restriction

```r
#' scheme-like if expression, without restriction of returning same-size table of .test, as ifelse() does
#'
#' @param .test predicate to test
#'
#' @param true expression to return if .test evals to TRUE
#'
#' @param false expression to return if .test evals to TRUE
#'
#' @return either true or false
ifexp <- function(.test, true, false){
  if (.test) {
    return(true)
  } else {
    return(false)
  }
}
```

5.1.2 Filetype from extension

A helper function to attain the document filetype from the file name.

```r
#' Get filetype
#'
#' @param filepath string filepath of document
#'
#' @return filetype (string) - NA if no extension
get_filetype <- function(filepath){
  filepath %>%
    basename %>%
    stringr::str_extract('[a-zA-Z0-9]+\\.[a-zA-Z0-9]+$') %>% #ensure filename.extension form
    stringr::str_extract('[a-zA-Z0-9]+$')                   #extract extension
}
```

5.1.3 Mark the text column of a table

A helper function to determine and mark the text column of a table

```r
#' Interactively determine and automatically mark the text column of a table
#'
#' @param data dataframe with column requiring marking
```

```r
#'
#' @return same dataframe with text column renamed to "text"
table_textcol <- function(data){
cols <- colnames(data)
print("Please enter the number of the column you want selected for text analytics")
print(cols)
textcol_index <- get_valid_input(as.character(1:ncol(data))) %>%
  as.integer
textcol <- cols[textcol_index]
data %>%
    dplyr::rename(text = !! dplyr::sym(textcol))
}
```

5.1.4 Validate User Input

A helper function to get valid user input

```r
#' helper function to get valid input (recursively)
#'
#' @param options vector of options that valid input should be drawn from
#'
#' @param init whether this is the initial attempt, used only as
#'   recursive information
#'
#' @return readline output that exists in the vector of options
get_valid_input <- function(options, init=TRUE){
  input <- ifelse(init,
                  readline(),
                  readline(prompt = "Invalid option. Please try again: "))
  ifelse(input %in% options,
         input,
         get_valid_input(options, init=FALSE))
}
```

5.1.5 TODO Ungroup by

Also needed, but surprisingly missing from dplyr, is an "ungroup$_{\text{by}}$" function, that allows specifice groups to be removed. Currently standard evaluation only, will switch to NSE when time allows

TODO:

- ☐ Make `ungroup_by` NSE

```r
#' helper function to ungroup for dplyr. functions equivalently to
#' group_by() but with standard (string) evaluation
#'
#' @param x tibble to perform function on
#'
#' @param ... string of groups to ungroup on
#'
#' @return x with ... no longer grouped upon
ungroup_by <- function(x,...){
dplyr::group_by_at(x, dplyr::group_vars(x)[!dplyr::group_vars(x) %in% ...])
}
```

6 Program Layer: Preparation

Here I lay out the preparation layer in detail. The culmination of all preparation functions is one wrapper, requesting the possible preparation features, and outputting a final tibble that is worked on by the next insight layer.

Multiple documents are input the same as singular, though with an additional "document" column that can be grouped upon.

The following sections detail the components of text preparation.

6.1 TODO Importing

A variety of filetypes are able to be imported, with one wrapper function intelligently determining the appropriate import function from the file extension. Files with unrecognised extensions are treated as plaintext. Importantly, as we are working in a tidy paradigm, everything is imported as a tibble, with plaintext being one line per row, and tabular data maintaining the original form. Tabular data requires the specification of which column is the text column for analytics. All imports have a document ID, which is an identifier column.

6.1.1 Import .txt

Plaintext is the most important and simplest to work with of all text representations; entire operating systems are built around the concept.

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

6.1.2 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.

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

6.1.3 Import Excel

Unfortunately, much data exists in the Microsoft Excel format, but this must be catered for. As tabular data, it is treated equivalently to csv.

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

6.1.4 TODO Import Gutenberg

Project Gutenberg is an online library containing, at the time of writing, over 57,000 items, primarily plaintext ebooks. This is a goldmine of text ripe for analysis, and once the basic frontend is complete, I will dedicate some thought to the in-app importation of Gutenberg texts

6.1.5 Import

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

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

The base file import is generalised to multiple files with a multiple import function: this will be our sole import function (until we get direct Gutenburg import)

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

6.2 Formatting

To work in a tidy paradigm, following the lead of tidytext, we separate and ID by token. To do this, we take the line ID, the sentence ID, then the word ID, producing a dataframe that takes the following form:

line_{id}	$\text{sentence}_{\text{id}}$	word_{id}	word
1	1	1	the
1	1	2	quick
2	1	3	brown

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 `unnest_tokens` function from tidytext doesn't play as expected with groups at present, so much of grouping is (not ideally) taking place internally in the first `group_modify`. When I have the luxury of time, I will try to optimise this.

[2019-07-17 Wed]: removed first group modify; unnecessary now that grouping has been shifted to take place afterwards

```r
#' formats imported data into an analysis-ready format '
#' @param data a tibble formatted with a text and (optional) group
#'     column
```

```r
#'
#' @return a [tibble][tibble::tibble-package] formatted such that
#'     columns correspond to identifiers of group, line, sentence,
#'     word (groups ignored)
#'
#' @export
format_data <- function(data){
  data %>%
    dplyr::mutate(line_id = dplyr::row_number()) %>%
      tidytext::unnest_tokens(output = sentence, input = text,
                              token = "sentences", to_lower = FALSE) %>%
    dplyr::mutate(sentence_id = dplyr::row_number()) %>%
    dplyr::group_by(sentence_id, add=TRUE) %>%
    dplyr::group_modify(~ {
      .x %>%
        tidytext::unnest_tokens(output = word, input = sentence,
                                token = "words", to_lower=FALSE) %>%
        dplyr::mutate(word_id = dplyr::row_number())
    }) %>%
    ungroup_by("sentence_id")
}
```

6.3 TODO Filtering

Filtering has to be done with code at present, but the intention is that once I have a frontend up, it's design will inform an interactive filter. After some initial analytics have been done in the insight layer, then preparation can be returned to and the text can be filtered on based on the analytics.

6.4 TODO Lemmatisation

Lemmatisation is effectively the process of getting words into dictionary form. It is actually a very complex, stochastic procedure, as natural languages don't follow consistent and clear rules all the time. Hence, models have to be used. Despite the burden, it is generally worthwhile to lemmatise words for analytics, as there are many cases of words not being considered significant, purely due to taking so many different forms relative to others. Additionally, stopwords work better when considering just the lemmatised form, rather than attempting to exhaustively cover every possible form of a word. textstem is an R package allowing for easy lemmatisation, with it's function `lemmatize_words` transforming a vector of words into their lemmatised forms (thus being compatible with `mutate` straight out of the box). 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 we don't need at this point. Worth noting is that, like stopwords, there are different dictionaries available for the lemmatisation process, but we will use the default, as testing has shown it to be the simplest to set up and just as reliable as the rest.

6.5 TODO Stemming

Stemming is far simpler than lemmatisation, being the removal of word endings. This doesn't require as complex a model, as it is deterministic. It is not quite as effective, as the base word ending is not tacked back on at the end, so we are left with word stumps and morphemes. However, it may sometimes be useful when the lemmatisation model isn't working effectively, and textstem provides the capability with `stem_words`

6.6 Stopwords

Stopwords are syntactical features of text that are superfluous and get in the way of text analytics. Typical examples include articles and pronouns, like "the", "to", "I", etc. They would clutter the output of insights such as word frequency. We need a way of generating a list of stopwords, from both a default source, as well as allowing the user to add their own stopwords. `get_sw` performs that, detailed below.

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

The status of the stopwords are then added to the data with `determine_stopwords`

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

6.7 Object Preparation

The `preparation` wrapper takes all combinations of stopwords and lemmatisation options and intelligently connects them for the "insight column", which the insight is performed upon. For the purpose of standard interoperability with, e.g., ggpage, we name this column "text"

The gnarly `ifexp` taking up the heart of the function encodes the logic involving the interaction of stopwords and lemmatisation:

	Stopwords True
Lemmatise True	Lemmatise, determine stopwords on lemmatisation, perform insight on lemmas s
Lemmatise False	Determine stopwords on original words (no lemmatisation), perform insight on v

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

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

6.8 Sectioning

Plaintext, as might exist as a Gutenberg Download, differs from more complex representations in many ways, including a lack of sectioning - Chapters require a specific search in order to jump to them. Here, I compose a closure that searches and sections text based on a Regular Expression intended to capture a particular section. Several functions are created from that. In time, advanced users could be given the option to compose their own regular expressions for sectioning.

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

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

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

#' sections text based on sections
#'
#' @param .data vector to section
#'
#' @return vector of same length as .data with section numbers
#'
#' @export
get_sections <- get_search("^[\\s]*([Ss][Ss])|([Ss][Ee][Cc][Tt][Ii][Oo][Nn])")
```

How to implement sectioning in a way that fits in a shiny UI is still very much TBC. Presumably, after object preparation, the option to section would appear, followed by a group selection option. I will implement these only after implementing the shiny app.

6.9 TODO Grouping

Grouping is a killer feature of our app. The intention is to run a `group_by` dplyr command in the wrapper over user-specified groups, and all further insights and visualisations are performed groupwise. This allows for immediate and clear comparisons.

Like filtering, after some initial analytics have been done in the insight layer, then preparation can be returned to and the text can be grouped on based on the analytics.

7 Program Layer: Insight

Insight is the meat of this package. After some initial resistance, I have decided to jump all-in with tidyverse-style transformations, especially for the non-destructive editing, as an immutable functional programming paradigm suits such functions. Insight may be divided into word insight, and higher-level (aggregated) insights. The higher level insights include sentence and document level insights, such as sentence sentiment, tf-idf, etc. Importantly for the document level insights is that our program doesn't necessarily have to work purely on documents - any identifying column could potentially stand in.

At present, all insight functions haven't yet been tested with the new output of the Preparation layer. I want to make the following changes to all of them for this to be effective:

TODO:

- ☒ Have all insight functions work on vector input and output, so as to work with `mutate`
- ☒ Ensure correctness of output under grouping

7.1 Term Insight

7.1.1 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

```r
#' Determine term frequency
#'
#' @param .data character vector of terms
```

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

7.1.2 Bigrams [0/1]

Bigrams are two words that occur in sequence. For example, in the phrase, "The quick brown dog.", the following bigrams exist: "The quick", "quick brown", "brown dog". This can be generalised to any number of sequential words as *n-grams*. They are useful in text analytics to determine word sequences, as well as common adverb-verb and adjective-noun pairs. This exists partly between word and aggregate insight, but by measure is closer to the word-level.

When we attain the bigrams, we can use the word frequency function defined previously to attain a bigram frequency.

TODO:

- ☐ generalise to n-grams (make closure, have bigrams as special case)

I determine bigrams by matching the vector of words with itself, sans the first element in the second list.

```r
#' Determine bigrams
#'
#' @param .data character vector of words
#'
#' @return character vector of bigrams
#'
#' @export
get_bigram <- function(.data){
  1:length(.data) %>%
    purrr::map_chr(index_bigram, .data, .data[-1])
}
```

However, it is more complex than that; we need a way to deal with NA values. This beautiful recursive function (designed by myself) does just that:

```r
#' get bigram at index i of list1 & 2
#'
#' @param i numeric index to attain bigram at
#'
#' @param list1 list or vector for first bigram token
#'
```

```r
#' @param list2 list or vector for second bigram token
#'
#' @return bigram of list1 and list2 at index i, skipping NA's
index_bigram <- function(i, list1, list2){
  ifelse(length(list2) < i | is.na(list1[i]),
          as.character(NA),
  ifelse(!(is.na(list1[i]) | is.na(list2[i])),
         paste(list1[i], list2[i]),
         index_bigram(i,list1, list2[-1])))
}
```

It works as follows; if our list appears as (1 2 X 4 5 X 7 X), we expect the following bigrams: ((1 2) (2 4) X (4 5) (5 7) X X X), due to bigrams taking the lead of list1 as list2, as per the following table:

list1	list2	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

```r
x <- c(1, 2, NA, 4, 5, NA, 7, NA)
get_bigram(x)
```

```
1 2
2 4
nil
4 5
5 7
nil
nil
nil
```

7.1.3 Key Words (TextRank)

Key words are another killer feature of this app. The algorithm is explained previously. The `textrank` package is used to perform textrank. Of note is that all words other than stopwords (indicated by NA) are relevent, but the standard algorithm works on data that has had POS tagging, typically assessing only nouns and adjectives. We don't do that here as the processing burden for POS tagging is enormous, though it may be implemented in the future.

```r
#' Determine textrank score for vector of words
#'
#' @param .data character vector of words
#'
#' @return vector of scores for each word
#'
#' @export
keywords_tr <- function(.data){
  relevent <- !is.na(.data)
  tr <- textrank::textrank_keywords(.data, relevent, p=+Inf)
  score <- tr$pagerank$vector %>% tibble::enframe()
  data <- .data %>% tibble::enframe("number", "name")
  dplyr::full_join(data, score, by="name") %>%
    dplyr::pull(value)
}
```

7.1.4 Term Sentiment [1/1]

Sentiment has been discussed earlier. Effectively, for any text analytics it is essential. There are numerous sentiment dictionaries, but we will use AFINN for the nice numeric properties it has, allowing for statistics on them. Categorical dictionaries will be implemented later.

I **really** want to move away from dependence on tidytext; the maintainers are clearly incompetent. As an example, the `sentiments` data provided by the package previously contained several different lexicons, and had a column indicating the lexicon. After an update, this column was removed, leaving only the bing lexicon, breaking my functions, with no indication of where the other lexicons were in the documentation. The inconsistincies don't stop there; `sentiments` has two columns, word, and sentiment, whereas `get_sentiments` is

> A tbl_{df} with a word column, and either a 'sentiment' column (if 'lexicon' is not "afinn") or a numeric 'score' column (if 'lexicon' is "afinn").

So bing has sentiment if `sentiment` is invoked, but score if `get_sentiment` is invoked. All the while I have to completely unnecessarily program different selection names based on the different lexicons, instead of being able to rely on a simple static name. If I want different names, that should be my decision, not the packages'. The greatest irony is that in practice, as at *[2019-07-20 Sat]* on CRAN, the documentation isn't even correct; `get_sentiment` returns columns "word" and "value"

```r
get_sentiments("afinn")
```

```
get_sentiments("afinn")
# A tibble: 2,477 x 2
   word        value
   <chr>       <dbl>
```

Additionally, the `reorder_within` function is very poorly coupled with the rest of the package. It should have been a pull request to forcats, or an alternative, but it is far more general than the scope of the `tidytext` package. Moreover, the style of the package is very inconsistent and generally awful. Literally the only purpose it has served has been getting me up to speed through it's associated ebook, but even then if that book didn't exist, someone else (maybe me) would have written a better one with a better package. The ebook itself encourages very poor data practices, mandating lossy forms of working, and it has taken me half a year to work through the issues encouraged by them. I can see the lack of seriousness in the development where recent commit messages include "Do not check these in either see$_{noevil}$" with a monkey face emoji.

Without any options, they also forced the requirement to (interactively) download sentiment files (through the textdata package) instead of including them in the package. **These changes break backwards compatibility**. I have no idea how they will affect my shiny application. Likely, I'll just get the data myself and distribute that with my package, if I don't want specific licenses, I just won't include them.

TODO:

☒ Include option for additional dictionaries

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

7.1.5 TODO Word Correlation

This is the word-level insight that will be the most difficult to perform, due to my requirements that the dataframe remains tidy and lossless. The only way I can conceive of doing this is by adding columns for each distinct word, 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 like the one Cassidy created.

7.2 TODO Aggregate Insight

This should work effectively the same as the word-level insight, however the wrapper may have to be different. This is TBC. I think an "aggregate on ..." user option would be useful

7.2.1 Term Count

Word count on some aggregate group is following the pattern I have been noticing where the simpler a function is, the more analytical power it seems to give. This may be generalised in the future to give a nested aggregate count (e.g. sentences/paragraph, lines/document etc.)

Note in the following function the near canonical example of split-apply-combine, or MapReduce style. This allows for performance gains in parallel processing, ideal for the large datasets we typically work with in text analytics.

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

7.2.2 Key Sentence (LexRank)

Often keywords aren't very explanatory on their own; patterns only really develop in aggregate. We use lexrank as the algorithm for key-sentences, as textrank takes too long, though lexrank seems to take just as long at high n - it may be worth exploring the option of textrank again.

Testing shows a performance of around 3-4 mins for ~30,000 words of text aggregated of ~3000 sentences. Not bad for something graph based, but a warning will be required at the user end.

```r
#' get score for key sentences as per Lexrank
#'
#' @param .data character vector of words
#'
#' @param aggregate_on vector to aggregate .data over; ideally, sentence_id
#'
#' @return lexrank scores of aggregates
#'
```

```r
#' @export
key_aggregates <- function(.data, aggregate_on){
  ## prepare .data for lexrank
  base <-  tibble::tibble(word = !! .data, aggregate = aggregate_on)
  aggregated <- base %>%
    dplyr::group_by(aggregate) %>%
    stats::na.omit() %>%
    dplyr::summarise(sentence = paste(word, collapse = " ")) %>%
    dplyr::mutate(sentence = paste0(sentence, "."))
  ## lexrank
  lr <- aggregated %>%
    dplyr::pull(sentence) %>%
    lexRankr::lexRank(., n=length(.),removePunc = FALSE, returnTies = FALSE,
            removeNum = FALSE, toLower = FALSE, stemWords = FALSE,
            rmStopWords = FALSE, Verbose = TRUE)
  ## match lexrank output to .data
  lr %>%
    dplyr::distinct(sentence, .keep_all = TRUE) %>%
    dplyr::full_join(aggregated, by="sentence") %>%
    dplyr::full_join(base, by="aggregate") %>%
    dplyr::arrange(aggregate) %>%
    dplyr::pull(value)
}
```

7.2.3 Aggregate Sentiment

Like the added context that key sentences bring over key words, a similar situation is true of sentiment. I'll make it so that it can deliver any statistic of a sentence; mean, median, variance etc. Importantly, it will only work with numeric sentiment lexicons, in our case, AFINN.

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

7.2.4 TODO Term Frequency - Inverse Document Frequency (tf-idf)

7.2.5 TODO Topic Modelling

7.3 CLOSED Wrapper

The insights of choice can all be combined into a wrapper function, taking the forms and arguments of the insights and applying those chosen. Deprecated now that insight functions work vector-wise.

TODO:

- ☒ Take multiple insights

```r
#' perform group-aware term operations on the data
#'
#' @param .data dataframe of terms as per output of text_prep
#'
#' @param operations character vector of term operations to perform
#'
#' @return .data with operation columns added
#'
#' @export
get_term_insight <- function(.data, operations){
    opstable <- list("Term Frequency" = term_freq,
                     "Bigrams" = get_bigram,
                     "Key Words" = keywords_tr,
                     "Term Sentiment" = term_sentiment)
    ops <- opstable[operations]
    lapply(seq(length(ops)),
           function(x){
               name <- dplyr::sym(names(ops[x]))
               operation <- ops[x][[1]]
               df <- dplyr::mutate(.data,
                                   !!name := operation(text))
               df[names(ops[x])]
           }) %>%
        dplyr::bind_cols(.data, .)
}

#' perform group-aware aggregate operations on the data
#'
#' @param .data dataframe of terms as per output of text_prep
#'
#' @param operations character vector of operations to perform
#'
#' @param aggregate_on character name of the column to perform aggregate operations on
#'
#' @return .data with operation columns added
#'
#' @export
get_aggregate_insight <- function(.data, operations, aggregate_on){
    opstable <- list("Term Count" = term_count,
                     "Key Sections" = key_aggregates,
                     "Aggregated Sentiment" = aggregate_sentiment)
    ops <- opstable[operations]
    lapply(seq(length(ops)),
           function(x){
               name <- dplyr::sym(names(ops[x]))
```

```
                operation <- ops[x][[1]]
                agg_on <- dplyr::sym(aggregate_on)
                df <- dplyr::mutate(.data,
                                    !!name := operation(text, !! agg_on))
                df[names(ops[x])]
            }) %>%
        dplyr::bind_cols(.data, .)
}
```

8 Program Layer: Visualisation

I have grouped visualisations by their output intention, rather than their implementation, as an ends-based focus, with the means being details. The following are the most useful visualisations. A present issue with visualisation is how grouping is performed; If I want to have a set of charts separated by group, performing by group creates as many separate charts as there are groups, as separate graphics. I want to make use of `facet_wrap` from ggplot, which requires some maneuvering with a wrapper function. Ultimately, I will have to create two forms of each chart; one grouped, one ungrouped. Potentially more, for differentiation between singly and multiply grouped charts.

8.1 TODO Rank

8.2 TODO Score

8.2.1 TODO Barplot [0/2]

There are issues with the barplot, as documented by:

> *[2019-07-01 Mon]* After implementing grouping, the issue with arranging bars in a barplot by rank within each group is that ggplot dplyr::arranges bars through the ordering of factor levels. The problem is that each instance of a word in every group shares the same level ordering, so while a word may rank highly overall, but less than others in a particular group, it will retain the high ordering overall in the facet for that group, leading to potential confusion

[2019-07-15 Mon]: `https://juliasilge.com/blog/reorder-within/` may be a solution

Which I do want to fix, though it isn't necessarily *incorrect*. Additionally, this function takes too many arguments.

TODO:

- ☐ Find way to better order score

- ☐ Find way to lower number of arguments

```r
#' output a bar graph of the top words from some insight function
#'
#' @param std_tib the standard dataframe, modified so the last column
#'     is the output of some insight function (eg. output from
#'     term_freq)
#'
#' @param insight_name string name of the column insight
#'     was performed on
#'
#' @param insight_col string name of the column insight was
#'     outputted to
#'
#' @param n number of bars to display
#'
#' @param desc bool: show bars in descending order
#'
word_bar <- function(std_tib, insight_name, insight_col,
                     n = 15, desc = TRUE){
    dist <- std_tib %>%
        dplyr::distinct(word, .keep_all=TRUE)
    if (desc) {
        arr <-  dplyr::arrange(dist, desc(!! sym(insight_col)))
    }else{
        arr <- dplyr::arrange(dist, !! sym(insight_col))
    }
    arr %>%
        group_modify(~{.x %>% head(n)}) %>%
        ungroup() %>%
        dplyr::mutate(!! sym(insight_name) := fct_reorder(!! sym(insight_name),
                                                         !! sym(insight_col),
                                                         .desc = desc)) %>%
        ggplot(aes(x = !! sym(insight_name))) +
        geom_col(aes(y = !! sym(insight_col)))
}
```

8.3 TODO Relation

8.3.1 Correlation Matrix

8.4 TODO Distribution [1/4]

8.4.1 CLOSED Density

```r
#' output a histogram of the distribution of some function of words
#'
#' @param std_tib the standard dataframe, modified so the last column
#'     is the output of some insight function (eg. output from
#'     term_freq)
#'
#' @param insight_col string name of the column insight was
#'     performed on
word_dist <- function(std_tib, insight_col){
std_tib %>%
    ggplot(aes(x = !! sym(insight_col))) +
    geom_density()
}
```

8.4.2 TODO Histogram

8.4.3 TODO Boxplot

8.4.4 TODO Ungrouped Boxplot

8.5 TODO Structure [1/2]

8.5.1 TODO Time Series

8.5.2 CLOSED ggpage

ggpage allows us to show off the importance of our non-destructive editing - the original document can be displayed, with the insights highlighted. There was more discussion on ggpage under an earlier section.

1. Ungrouped

```r
#' Colours a ggpage based on an insight function
#'
#' @param .data a dataframe containing "word" and insight columns as
#'     per the output of the get_(term|aggregate)_insight wrapper
#'     function
#'
#' @param col_name symbol name of the insight column intended to
#'     colour plot
#'
#' @return ggplot object as per ggpage
#'
#' @export
struct_ggpage_ungrouped <- function(.data, col_name){
    q_col_name <- dplyr::enquo(col_name)
    .data %>%
        dplyr::pull(word) %>%
        ggpage::ggpage_build() %>%
        dplyr::bind_cols(.data) %>%
        ggpage::ggpage_plot(ggplot2::aes(fill = !! q_col_name)) ## +
        ## ggplot2::labs(title = "")
}
```

8.6 TODO Wrapper

This is an attempt to create a group-aware visualisation, automatically facetting by group. I feel like it is not ideal, though haven't had any major bugs with it yet

```r
#' create a group-aware visualisation
#'
#' @param .data the standard dataframe, modified so the last column
#'     is the output of some insight function (eg. output from
#'     term_freq)
#'
#' @param vis character name of visualisation function
#'
#' @param col character name of the column to get insight from
#'
```

```r
#' @export
get_vis <- function(.data, vis, col, distribution=FALSE){
    vistable <- list("struct_ggpage_ungrouped" = struct_ggpage_ungrouped)
    if (dplyr::is_grouped_df(.data)){
        grouping <- dplyr::group_vars(.data)
    } else {
      col_name <- dplyr::sym(col)
      vistable[[vis]](.data, !! col_name)
    }
}
```

9 Testing / Demonstration

9.1 Source

```r
library(inzightta)
```

9.2 Ungrouped

9.2.1 Preparation

```r
imported <- import_files(tcltk::tk_choose.files())
lemmatize <- TRUE
stopwords <- TRUE
sw_lexicon <- "snowball"
addl_stopwords <- NA
data <- text_prep(imported, lemmatize, stopwords, sw_lexicon, addl_stopwords)
```

9.2.2 Insights

```r
insighted <- data %>%
  dplyr::mutate(
  term_freq = term_freq(text),
  bigram = get_bigram(text),
  bigram_freq = term_freq(bigram),
  word_sentiment = term_sentiment(text),
  term_count_sentence = term_count(text, sentence_id),
  mean_aggregate_sentiment_sentence = aggregate_sentiment(text, sentence_id, mean),
  sd_aggregate_sentiment_sentence = aggregate_sentiment(text, sentence_id, sd)
  )
```

9.2.3 Visualisation

```r
# ... with 29,944 more rows, and 13 more variables: line_id <int>, word1 <chr>,
#   word_id <int>, lemma <chr>, stopword <lgl>, text <chr>, term_freq <int>,
#   bigram <chr>, bigram_freq <int>, word_sentiment <int>,
#   term_count_sentence <int>, mean_aggregate_sentiment_sentence <dbl>,
#   sd_aggregate_sentiment_sentence <dbl>

## Structure: ggpage --------------------------------

insighted %>%
  dplyr::pull(word) %>%
  ggpage::ggpage_build() %>%
  dplyr::bind_cols(insighted) %>%
  ggpage::ggpage_plot(ggplot2::aes(colour=mean_aggregate_sentiment_sentence)) +
  ggplot2::scale_color_gradient2()

insighted %>%
  dplyr::pull(word) %>%
  ggpage::ggpage_build() %>%
  dplyr::bind_cols(insighted) %>%
  ggpage::ggpage_plot(ggplot2::aes(colour=term_count_sentence)) +
  ggplot2::labs(title = "Word Count of Sentences")

## Distribution: Histogram -------------------------------

insighted %>%
  ggplot2::ggplot(ggplot2::aes(term_freq)) +
  ggplot2::geom_histogram() +
  ggplot2::labs(title = "Histogram of Word Frequency")

## Score: barplot -------------------------------

n <- 10

insighted %>%
  dplyr::distinct(bigram, .keep_all = TRUE) %>%
  dplyr::top_n(n, bigram_freq) %>%
  dplyr::mutate(bigram = forcats::fct_reorder(bigram, dplyr::desc(bigram_freq))) %>%
  ggplot2::ggplot(ggplot2::aes(bigram, bigram_freq)) +
    ggplot2::geom_col() +
    ggplot2::coord_flip() +
  ggplot2::labs(title = "Bigrams by Bigram Frequency")
```

9.3 Grouped

9.3.1 Preparation

We import a file downloaded from Project Gutenberg, and run through some basic preparation, with additional stopwords to be removed

```
imported <- import_files(tcltk::tk_choose.files())
lemmatize <- TRUE
stopwords <- TRUE
sw_lexicon <- "snowball"
addl_stopwords <- NA
prepped <- text_prep(imported, lemmatize, stopwords, sw_lexicon, addl_stopwords)
sectioned <- prepped %>% dplyr::mutate(chapter = get_chapters(text))
data <- sectioned %>%
  dplyr::group_by(doc_id, chapter)

## .data <- data$text
## aggregate_by <- data$chapter
```

9.3.2 Insights

```
insighted <- data %>%
  dplyr::mutate(
```

```r
  term_freq = term_freq(text),
  bigram = get_bigram(text),
  bigram_freq = term_freq(bigram),
  word_sentiment = word_sentiment(text),
  term_count_sentence = term_count(text, sentence_id),
  mean_aggregate_sentiment_sentence = aggregate_sentiment(text, sentence_id, mean),
  sd_aggregate_sentiment_sentence = aggregate_sentiment(text, sentence_id, sd)
  )

## alt_insighted <- data %>%
##   group_modify(~ {
##     .x %>%
##   dplyr::mutate(
##   term_freq = term_freq(text),
##   bigram = get_bigram(text),
##   bigram_freq = term_freq(bigram),
##   word_sentiment = word_sentiment(text),
##   term_count_sentence = term_count(text, sentence_id),
##   mean_aggregate_sentiment_sentence = aggregate_sentiment(text, sentence_id, mean),
##   sd_aggregate_sentiment_sentence = aggregate_sentiment(text, sentence_id, sd)
##   )
##   })

## testthat::test_that("groups work with dplyr::mutate as with group_modify",
## {
##   expect_equal(insighted, alt_insighted)
## })
```

9.3.3 Visualisation

```r
## Structure: ggpage --------------------------------
groups <- dplyr::group_vars(insighted)

insighted %>% #base data
  dplyr::group_modify(~ { #build ggpage
    .x %>%
      dplyr::pull(word) %>%
      ggpage::ggpage_build() %>%
      dplyr::bind_cols(.x)
  }) %>%
  ggpage::ggpage_plot(ggplot2::aes(colour=mean_aggregate_sentiment_sentence)) + #plot ggpage
  ggplot2::scale_color_gradient2() +
  ggplot2::facet_wrap(groups) +
  ggplot2::labs(title = glue::glue("Mean Sentiment of Sentences by {paste(groups, collapse = \", \")}"))

ggplot2::ggsave(filename = "mean-sent-ggpage.png", device = "png", path="~/stats-781/out/")

insighted %>% #base data
  dplyr::group_modify(~ { #build ggpage
    .x %>%
      dplyr::pull(word) %>%
      ggpage::ggpage_build() %>%
      dplyr::bind_cols(.x)
  }) %>%
  ggpage::ggpage_plot(ggplot2::aes(colour=sd_aggregate_sentiment_sentence)) + #plot ggpage
  ggplot2::scale_color_gradient2() +
  ggplot2::facet_wrap(groups) +
    ggplot2::labs(title = glue::glue("Sentiment Standard Deviation of Sentences by {paste(groups, collapse =


ggsave(filename = "sd-sent-ggpage.png", device = "png", path="~/stats-781/out/")


insighted %>% #base data
   dplyr::group_modify(~ { #build ggpage
    .x %>%
      dplyr::pull(word) %>%
      ggpage::ggpage_build() %>%
      dplyr::bind_cols(.x)
  }) %>%
  ggpage::ggpage_plot(ggplot2::aes(colour=term_count_sentence)) + #plot ggpage
  ## scale_color_gradient2() +
  ggplot2::facet_wrap(groups) +
  ggplot2::labs(title = glue::glue("Word Count of Sentences by {paste(groups, collapse = \", \")}"))

insighted %>%
  dplyr::pull(word) %>%
  ggpage::ggpage_build() %>%
  dplyr::bind_cols(insighted) %>%
  ggpage::ggpage_plot(ggplot2::aes(colour=term_count_sentence)) +
  ggplot2::labs(title = "Word Count of Sentences")

## Distribution: Histogram --------------------------------

insighted %>%
  ggplot2::ggplot(ggplot2::aes(term_freq)) +
  ggplot2::geom_histogram() +
  ggplot2::labs(title = "Histogram of Word Frequency") +
  ggplot2::facet_wrap(groups)

ggplot2::ggsave(filename = "word-freq-hist.png", device = "png", path="~/stats-781/out/")

## Score: barplot --------------------------------

n <- 10

insighted %>%
    dplyr::group_modify(~ {.x %>%
                    dplyr::distinct(bigram, .keep_all = TRUE) %>%
                    dplyr::arrange(desc(bigram_freq)) %>%
                    head(n)
  }) %>%
  dplyr::ungroup() %>%
    dplyr::mutate(bigram = tidytext::reorder_within(bigram,
                                                    bigram_freq,
                                                    !! ifexp(length(groups) > 1,
                                                             dplyr::syms(groups),
                                                             dplyr::sym(groups)))) %>%
  ggplot2::ggplot(ggplot2::aes(bigram, bigram_freq)) +
  ggplot2::geom_bar(stat="identity") +
  ggplot2::facet_wrap(groups, scales = "free_y") +
  tidytext::scale_x_reordered() +
  ggplot2::coord_flip() +
    ggplot2::labs(title = "Bigrams by Bigram Frequency")

ggsave(filename = "bigram-freq-bar.png", device = "png", path="~/stats-781/out/")
```

9.4 ggpage

ggpage is a very interesting piece of visualisation, tested here. Once I build up the correct preparation format, I will perform more intensive testing here

```r
filename <- "../data/raw/11-0.txt"

imported <- import_txt(filename)

imported %>%
    ggpage_build() %>%
    filter(page == 1) %>%
    ggpage_plot()

imported %>%
    ggpage_build() %>%
    filter(page == 1) %>%
    get_insight(term_freq) %>%
    ggpage_plot(aes(fill=term_freq))

stopwords <- get_sw()

imported <- import_txt(filename) %>%
    format_data() %>%
    remove_stopwords(stopwords) %>%
    reconstruct()

imported %>%
    ggpage_build() %>%
    get_insight(term_freq) %>%
    ggpage_plot(aes(fill=term_freq))

imported %>%
    ggpage_build() %>%
    get_insight(keywords_tr) %>%
    ggpage_plot(aes(fill=rank))

imported %>%
    ggpage_build() %>%
    get_insight(word_sentiment_AFINN) %>%
    ggpage_plot(aes(fill=score)) +
    scale_fill_gradient2(low = "red", high = "blue", mid = "grey", midpoint = 0)
```

9.5 shiny

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

input <- list(file1 = list(datapath = "~/stats-781/data/raw/conv.txt"),
              lemmatise = TRUE,
              stopwords = TRUE,
              sw_lexicon = "snowball",
              filter_var = NULL,
              filter_pred = NULL,
              group_var = NULL,
              get_term_insight = TRUE,
              term_insight = "Term Frequency",
              get_aggregate_insight = NULL,
                aggregate_insight = NULL,
                aggregate_var = NULL,
                vis = "struct_ggpage_ungrouped",
                vis_col = "Term Frequency",
                distribution = FALSE)

                                    # Import & Process
imported <- inzightta::import_files(input$file1$datapath)
prepped <- {
    data <- imported
        if (isTruthy(input$lemmatise) |
            isTruthy(input$stopwords)){
            data <- data %>%
                text_prep(input$lemmatise, input$stopwords, input$sw_lexicon, NA)
        }
    data}
filtered <- {
    data <- prepped
        if (isTruthy(input$filter_var) &
            isTruthy(input$filter_pred)){
            data <- data %>%
                dplyr::filter(!! dplyr::sym(input$filter_var) == input$filter_pred)
        }
        data
}
grouped <- {
    data <- filtered
    if (isTruthy(input$group_var)){
        data <- data %>%
            dplyr::group_by(!! dplyr::sym(input$group_var))
    }
    data
}
term_insights <- {
    data <- grouped
    if (isTruthy(input$get_term_insight) &
        isTruthy(input$term_insight)){
        data <- data %>%
            get_term_insight(input$term_insight)
    }
    data
}
    aggregate_insights <- {
        data <- term_insights
        if (isTruthy(input$get_aggregate_insight) &
            isTruthy(input$aggregate_insight) &
            isTruthy(input$aggregate_var)){
            data <- data %>%
                get_aggregate_insight(input$aggregate_insight, input$aggregate_var)
        }
        data
    }
get_vis(aggregate_insights, "struct_ggpage_ungrouped", input$vis_col, input$distribution)
```

10 Shiny Frontend

For the user interface, the obvious choice is shiny.

We first tried using an if statement to determine the data, as per

```
 data <- reactive({
    data <- inzightta::import_files(input$file1$datapath)
    if (isTruthy(input$lemmatise) | isTruthy(input$stopwords)){
        data <- data %>%
            text_prep(input$lemmatise, input$stopwords, input$sw_lexicon, NA)
    }
    if (isTruthy(input$filter_var) & isTruthy(input$filter_pred)){
        data <- data %>%
            dplyr::filter(!! dplyr::sym(input$filter_var) == input$filter_pred)
    }
    if (isTruthy(input$group_var)){
        data <- data %>%
            dplyr::group_by(!! dplyr::sym(input$group_var))
    }
    data
})
```

However, this led to variable selectors constantly changing, as they had to recompute the names of data() which ran through several states at every event.

We will use the following as our shiny app:

10.1 Library Calls

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

10.2 UI

```
ui <- navbarPage("iNZight Text Analytics",
                 tabPanel("Processing",
                          sidebarLayout(
                              sidebarPanel(
                                  tags$p("Import"),
                                  fileInput("file1", "Choose File(s)",
                                            multiple = TRUE,
                                            accept = c("text/csv",
                                                       "text/comma-separated-values,text/plain",
                                                       ".csv", ".xlsx", ".xls")),
                                  tags$hr(),
                                  tags$p("Process"),
                                  checkboxInput("lemmatise", "Lemmatise"),
                                  selectInput("sw_lexicon", "Stopword Lexicon", list("snowball")),
                                  checkboxInput("stopwords", "Stopwords"),
                                  uiOutput("vars_to_filter"),
                                  textInput("filter_pred", "value to match", ""),
                                  uiOutput("vars_to_group_on"),
                                  tags$hr(),
                                  tags$p("Term Insight"),
                                  selectInput("term_insight",
                                              "Term Insight",
                                              list("Term Frequency", "Bigrams", "Key Words", "Term Sentiment"
                                              multiple=TRUE),
                                  actionButton("get_term_insight",
                                               "Get Term Insight"),
                                  tags$p("Aggregate Insight"),
                                  uiOutput("var_to_aggregate_insight_on"),
                                  selectInput("aggregate_insight",
                                              "Aggregate Insight",
                                              list("Term Count", "Key Sections", "Aggregated Sentiment"),
                                              multiple=TRUE),
                                  actionButton("get_aggregate_insight",
                                               "Get Aggregate Insight")
                              ),
                              mainPanel(
                                  tableOutput("table")
                              )
                          )),
                 tabPanel("Visualisation",
                          sidebarLayout(
                              sidebarPanel(selectInput("vis",
                                                       "Visualisation Type",
                                                       list("struct_ggpage_ungrouped")),
                                           uiOutput("var_to_visualise"),
                                           checkboxInput("distribution",
                                                         "Distribution")),
                              mainPanel(
                                  plotOutput("plot")
                              )
                          ))
                 )
```

10.3 Server

```
server <- function(input, output) {
    imported <- reactive({
        inzightta::import_files(input$file1$datapath)
    })
    prepped <- reactive({
        data <- imported()
        if (isTruthy(input$lemmatise) |
            isTruthy(input$stopwords)){
            data <- data %>%
                text_prep(input$lemmatise, input$stopwords, input$sw_lexicon, NA)
        }
        data
    })
    filtered <- reactive({
        data <- prepped()
        if (isTruthy(input$filter_var) &
            isTruthy(input$filter_pred)){
            data <- data %>%
                dplyr::filter(!! dplyr::sym(input$filter_var) == input$filter_pred)
        }
        data
    })
    grouped <- reactive({
        data <- filtered()
        if (isTruthy(input$group_var)){
            data <- data %>%
                dplyr::group_by(!! dplyr::sym(input$group_var))
        }
        data
    })
```

```r
    term_insights <- reactive({
        data <- grouped()
        if (isTruthy(input$get_term_insight) &
            isTruthy(input$term_insight)){
            data <- data %>%
                get_term_insight(input$term_insight)
        }
        data
    })
    aggregate_insights <- reactive({
        data <- term_insights()
        if (isTruthy(input$get_aggregate_insight) &
            isTruthy(input$aggregate_insight) &
            isTruthy(input$aggregate_var)){
            data <- data %>%
                get_aggregate_insight(input$aggregate_insight, input$aggregate_var)
        }
        data
    })
    output$table <- renderTable({
        aggregate_insights()
    })
    output$plot <- renderPlot({
        req(input$vis)
        req(input$vis_col)
        get_vis(aggregate_insights(), input$vis, input$vis_col, input$distribution)
    })
    output$vars_to_filter <- renderUI(selectInput("filter_var",
                                                  "select which column to apply filtering to",
                                                  c("", names(prepped()) %||% c(""))))
    output$vars_to_group_on <- renderUI(selectInput("group_var",
                                                    "select which columns to group on",
                                                    c("", names(filtered()) %||% c(""))))
    output$var_to_aggregate_insight_on <- renderUI(selectInput("aggregate_var",
                                                               "select which column to aggregate insight on",
                                                               c("", names(grouped()) %||% c(""))))
    output$var_to_visualise <- renderUI(selectInput("vis_col",
                                                    "select which variable to visualise",
                                                    c("", names(aggregate_insights()) %||% c(""))))
}
```

10.4 Program Call

```r
# Create Shiny app ----
shinyApp(ui, server)
```

10.5 Additional Functionality

todo:

- ☐ Allow additional stopwords
- ☐ Allow choice of stopword lexicon
- ☐ allow multiple groups
- ☐ fix document names

- ☐ allow non-lemmatised or stopworded text preparation
- ☐ add sectioning
- ☐ aggregate sentiment choice of statistic
- ☐ Have insights done automatically when creating visualisations
- ☐ rearrange ui

Interface chapter