UNIVERSITY OF SUSSEX | BUSINESS SCHOOL | SCIENCE POLICY RESEARCH UNIT

Textual data and text mining (2)

Daniele Rotolo

Introductory Data Science for Innovation (995N1) – Week 9, 22 November 2021

Outline

Outline

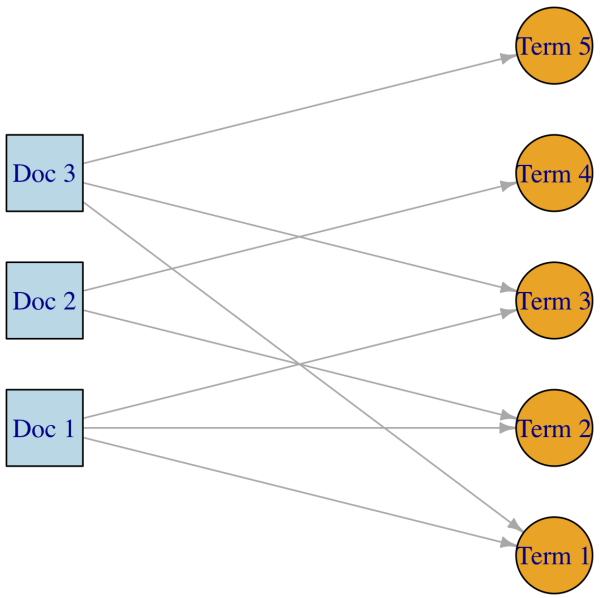
- Visualising text data
- Sentiment analysis
- Regular expressions
- Introduction to topic modelling

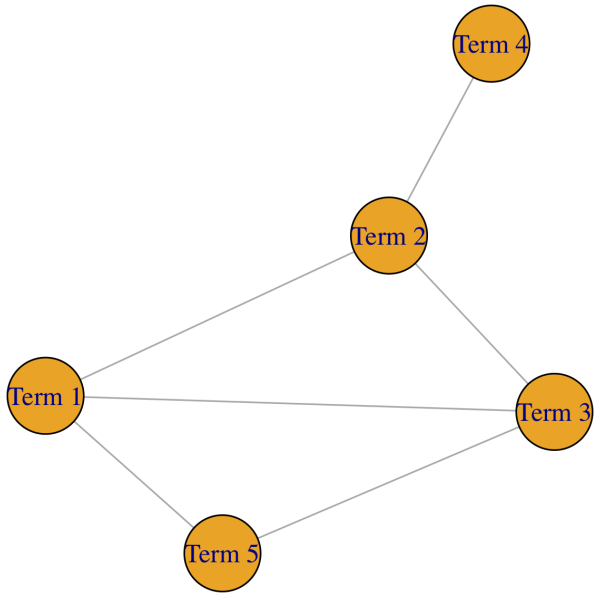
Visualising text data

Visualising text data

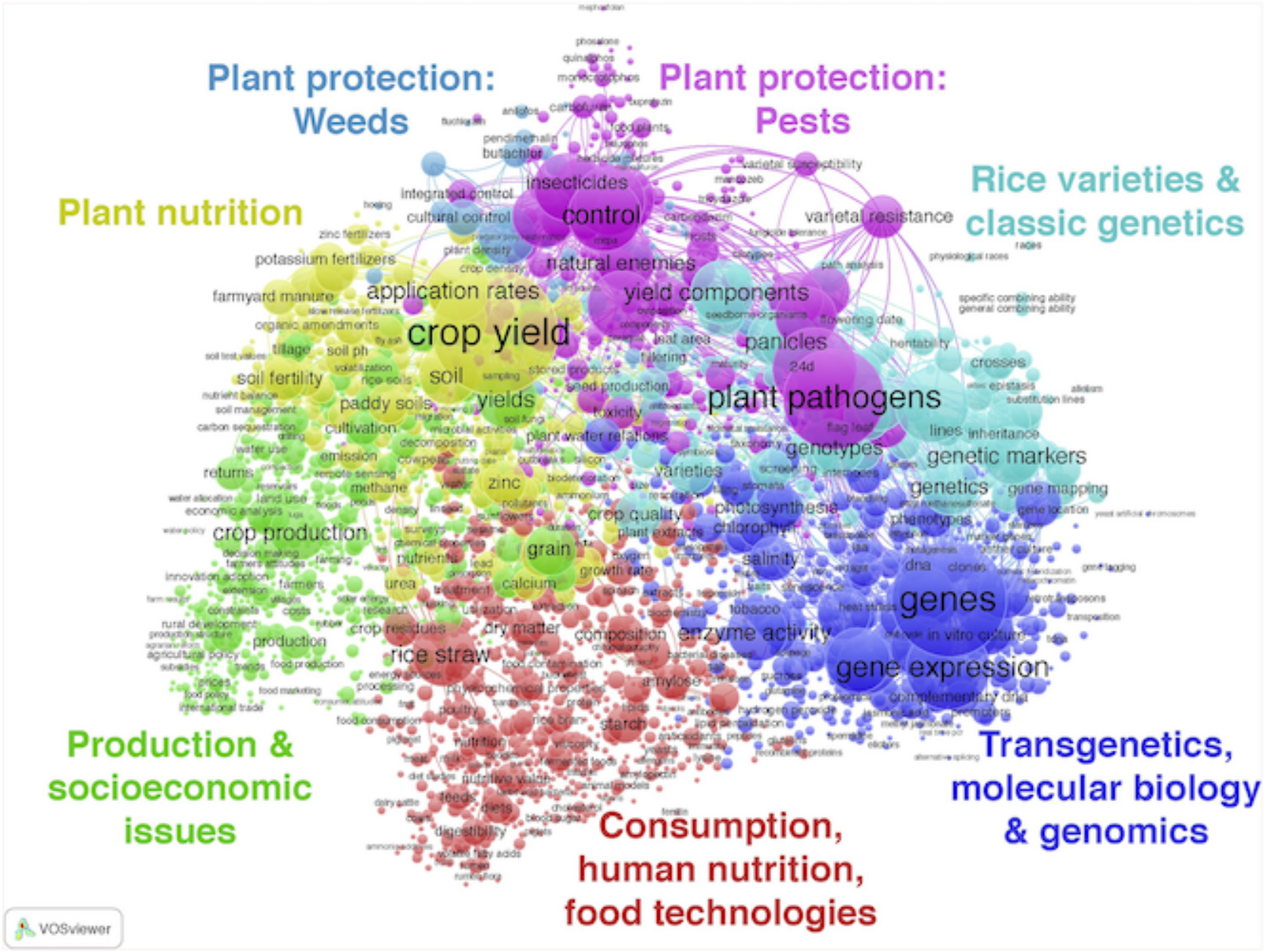
- So far, we explored text mining counting words and visualising their frequency using bar charts or similar
- **Bag of words** approach,
  - We do not consider the order of words
  - Yet, we can account for words occurring in the same sentence, paragraph, article, or, more generally, document
- This analysis is called **co-word analysis** (Callon, Courtial, and Laville 1991)
- A network is generated
  - Nodes = words
  - Links = co-occurrence of words (**cosine similarity normalisation**)

Co-word analysis





Co-word analysis



Source: Research priorities and societal demand(Ciarli and Ràfols 2019)

Example: (1) Co-word analysis

- We load all packages we need for co-word analysis and subsequent examples
- In particular, the `igraph` (**network analysis**) and `ggraph` (**network visualisation**) packages

```
library(tidyverse)
library(tidytext)
library(ggplot2)
library(plotly)
library(igraph)
library(ggraph)
```

Example: (2) Co-word analysis

- We extract unigrams from from 692 news English articles on COP26 published by newspapers on 12 November 2021
- We use text in titles only
- We remove stopwords and numbers

```
my_text_uni <- read_csv("news_articles_example.csv") %>%
  filter(Publication == "The Guardian (London)" |
         Publication == "The New York Times" |
          Publication == "The Independent (United Kingdom)") %>%
  select(id, Title) %>%
  unnest_tokens(output = word, input = Title) %>%
  anti_join(stop_words) %>%
  mutate(word_numeric = as.numeric(word)) %>%
  filter(is.na(word_numeric)) %>%
  select(-word_numeric)
```

```
## Warning in mask$eval_all_mutate(quo): NAs introduced by coercion
```

Example: (3) Co-word analysis

```
my_text_uni
```

```
## # A tibble: 483 × 2
##       id word
##    <dbl> <chr>
##  1    36 island
##  2    36 nations
##  3    36 press
##  4    36 cop26
##  5    36 winds
##  6   101 cop26
##  7   101 compromise
##  8   101 calamity
##  9   102 cop26
## 10   102 hundreds
## # … with 473 more rows
```

Example: (4) Co-word analysis

- We need to build a bipartite network (in this case a doc-term network)

```
my_text_uni <- my_text_uni %>%
  mutate(id = paste0("doc", id))

node1 <- my_text_uni %>%
  distinct(id) %>%
  rename(node = id) %>%
  mutate(type = T)

node2 <- my_text_uni %>%
  distinct(word) %>%
  rename(node = word) %>%
  mutate(type = F)

nodes <- bind_rows(node1, node2)
```

Example: (5) Co-word analysis

```
nodes
```

```
## # A tibble: 389 × 2
##    node   type
##    <chr>  <lgl>
##  1 doc36  TRUE
##  2 doc101 TRUE
##  3 doc102 TRUE
##  4 doc104 TRUE
##  5 doc105 TRUE
##  6 doc106 TRUE
##  7 doc107 TRUE
##  8 doc108 TRUE
##  9 doc109 TRUE
## 10 doc110 TRUE
## # … with 379 more rows
```

Example: (6) Co-word analysis

- The list of links between documents (articles) and words is in the `my_text_uni` object, while nodes are in the `nodes` object
- We can create a network relying on the function `graph_from_data_frame` from the package `igraph`

```
g_bip <- graph_from_data_frame(my_text_uni, vertices = nodes, directed = F)
```

Example: (7) Co-word analysis

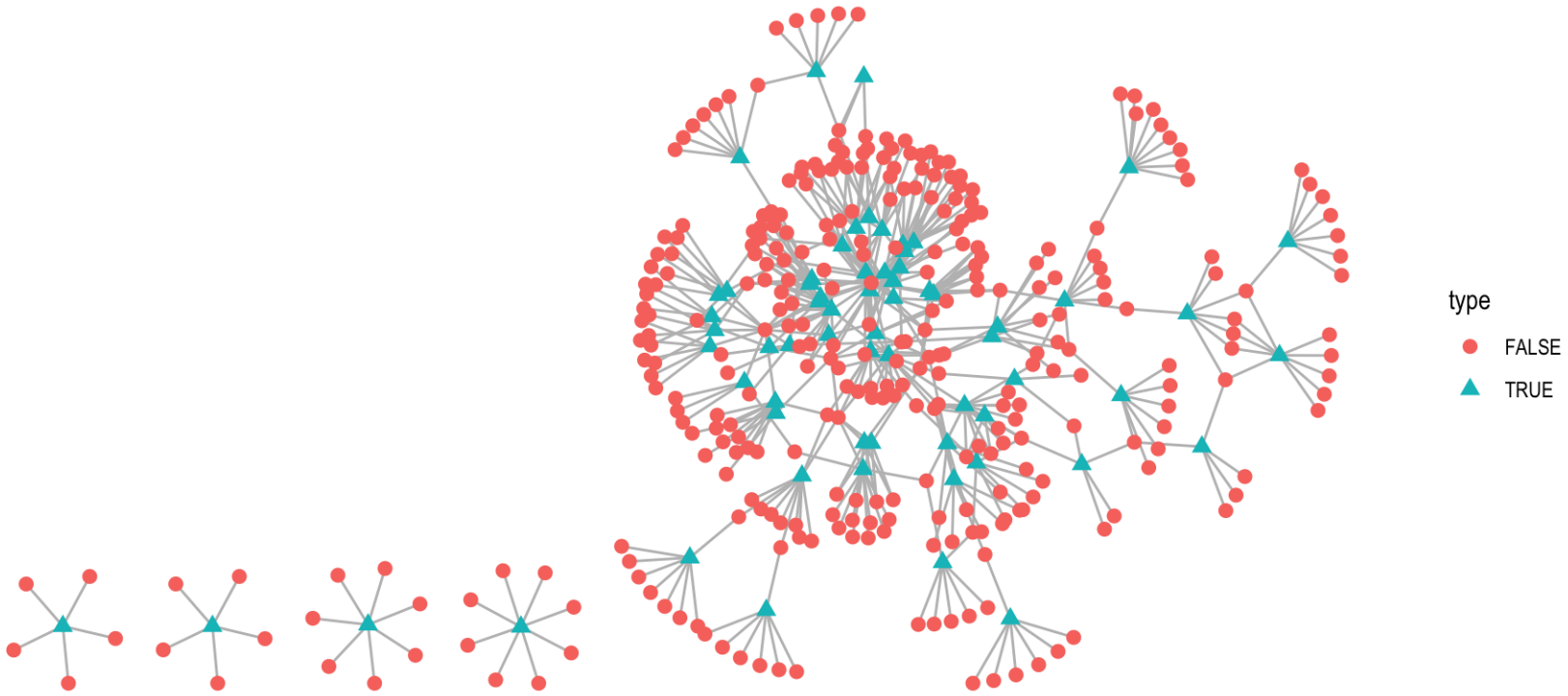
```
g_bip
```

```
## IGRAPH feee55f UN-B 389 483 --
## + attr: name (v/c), type (v/l)
## + edges from feee55f (vertex names):
##  [1] doc36 --island      doc36 --nations     doc36 --press
##  [4] doc36 --cop26       doc36 --winds       doc101--cop26
##  [7] doc101--compromise  doc101--calamity    doc102--cop26
## [10] doc102--hundreds    doc102--academics   doc102--denounce
## [13] doc102--glasgow     doc102--summit      doc102--failure
## [16] doc102--call        doc102--real        doc102--green
## [19] doc102--revolution  doc104--alternative doc104--worth
## [22] doc104--thinking    doc104--nations     doc104--redouble
## + ... omitted several edges
```

Example: (8) Co-word analysis

```
g_bip_vis <- ggraph(g_bip, layout = "stress") +
  geom_edge_link0(edge_colour = "grey") +
  geom_node_point(aes(shape = type, color =  type), size = 2.5) +
  theme_graph()
```

Example: (9) Co-word analysis



Example: (10) Co-word analysis

- We can transform this network so to have a word-word network (**bipartite projection**)
- We focus on the **largest component**
- We increase the size of nodes/words on the basis of degree a word has with other words

```
g <- bipartite_projection(g_bip, multiplicity = T, which = F)
g <- decompose.graph(g)[[1]]
V(g)$size <- degree(g)
```

Example: (11) Co-word analysis

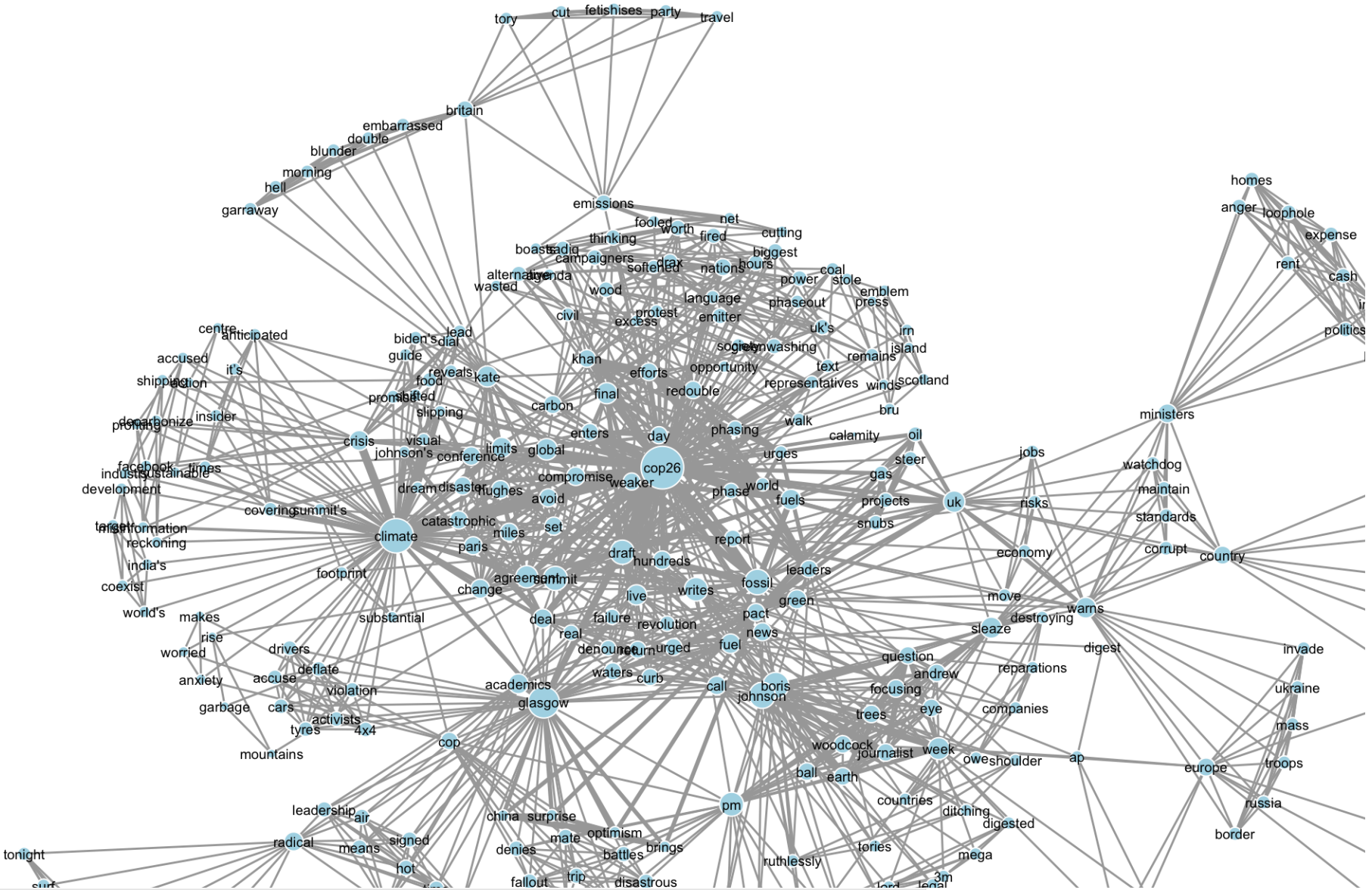
```
g
```

```
## IGRAPH 097bb81 UNW- 299 1540 --
## + attr: name (v/c), size (v/n), weight (e/n)
## + edges from 097bb81 (vertex names):
##  [1] island --nations     island --press       island --cop26
##  [4] island --winds       nations--press       nations--cop26
##  [7] nations--winds       nations--alternative nations--worth
## [10] nations--thinking    nations--redouble    nations--efforts
## [13] nations--final       nations--hours       nations--sadiq
## [16] nations--khan        press  --cop26       press  --winds
## [19] cop26  --winds       cop26  --compromise  cop26  --calamity
## [22] cop26  --hundreds    cop26  --academics   cop26  --denounce
## + ... omitted several edges
```

Example: (12) Co-word analysis

```
g_vis <- ggraph(g, layout = "stress") +
  geom_edge_link0(aes(edge_width = weight), edge_colour = "grey66") +
  geom_node_point(aes(size = size), fill = "lightblue", colour = "white", shape = 21) +
  geom_node_text(aes(label = name), size = 2.5) +
  scale_edge_width(range = c(0.5, 3)) +
  scale_size(range = c(1, 10)) +
  theme_graph()
```

Example: (13) Co-word analysis



Example: (14) Co-word analysis (cosine similarity)

- Co-occurrence only may have serious limitations (e.g. length of a document and normalisation challenges)
- **Cosine similarity** enable us to assess how “similar” are two words since they occur in same set of documents

$$\text{cosine\_sim} = \frac{w_1 * w_2}{||w_1|| * ||w_2||} = \frac{\sum_{i=1}^{n} w_{1,i} * w_{2,i}}{\sqrt{\sum_{i=1}^{n} w_{1,i}^2} * \sqrt{\sum_{i=1}^{n} w_{2,i}^2}}$$

Example: (15) Co-word analysis (cosine similarity)

```
w1 <- c(1, 0, 0, 0, 2)
w2 <- c(1, 0, 1, 0, 1)
w3 <- c(0, 0, 3, 0, 1)
lsa::cosine(w1, w2)
```

```
##           [,1]
## [1,] 0.7745967
```

```
lsa::cosine(w1, w3)
```

```
##           [,1]
## [1,] 0.2828427
```

Example: (16) Co-word analysis (cosine similarity)

- Let's calculate cosine similarity for our unigrams

```
my_text_uni <-  my_text_uni %>%
  count(id, word) %>%
  cast_dtm(id, word, n)

my_text_uni
```

```
## <<DocumentTermMatrix (documents: 65, terms: 324)>>
## Non-/sparse entries: 483/20577
## Sparsity           : 98%
## Maximal term length: 15
## Weighting          : term frequency (tf)
```

Example: (17) Co-word analysis (cosine similarity)

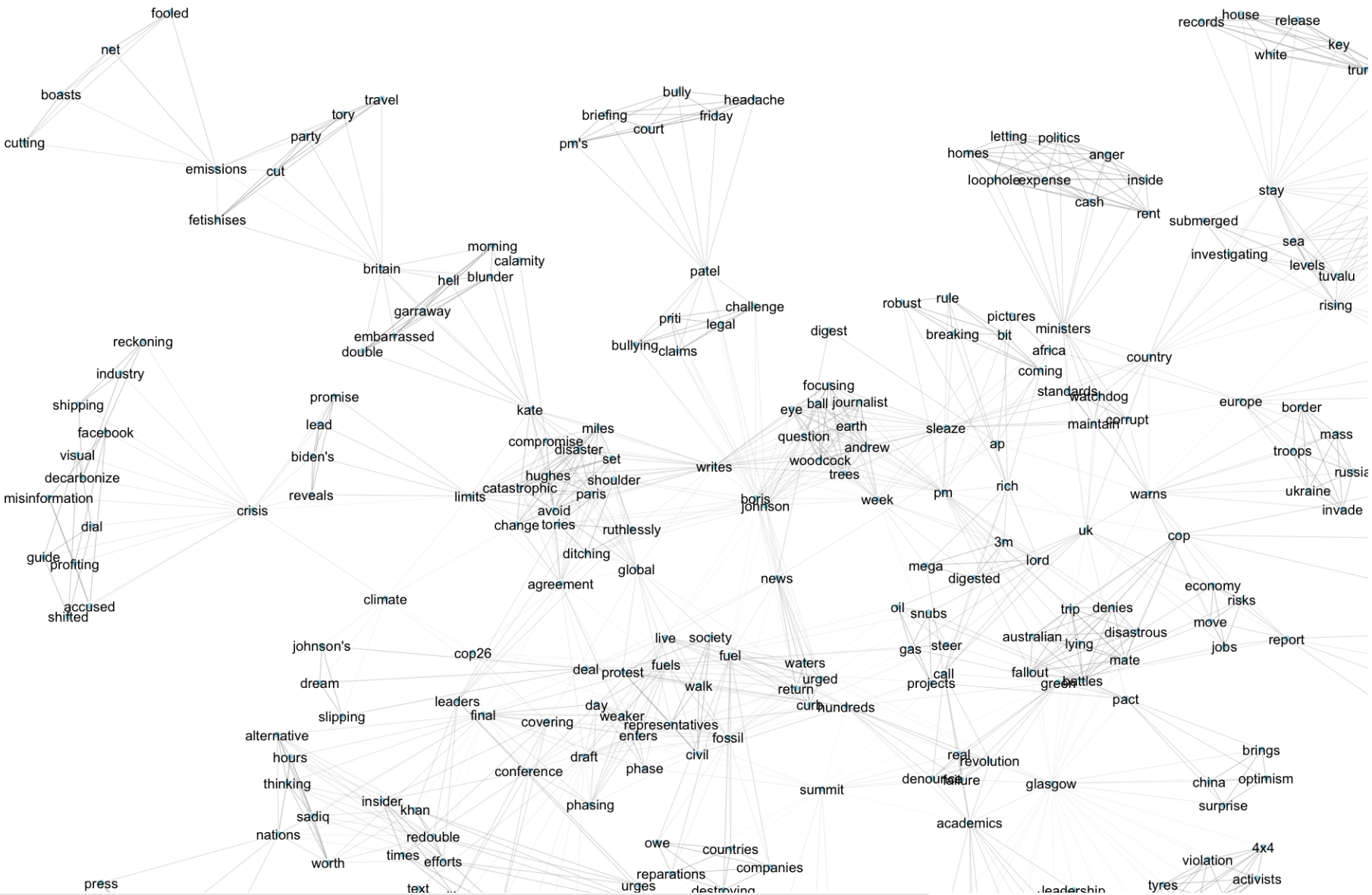
```
my_text_uni <- lsa::cosine(as.matrix(my_text_uni))
g_cos <- graph_from_adjacency_matrix(my_text_uni, mode = "undirected", weighted = T)
g_cos <- delete.edges(g_cos, which(E(g_cos)$weight < 0.3))
g_cos <- decompose.graph(g_cos)[[1]]
g_cos
```

```
## IGRAPH 58eaad5 UNW- 281 1562 --
## + attr: name (v/c), weight (e/n)
## + edges from 58eaad5 (vertex names):
##  [1] calamity  --calamity     calamity  --compromise   compromise--compromise
##  [4] compromise--agreement    compromise--change       compromise--avoid
##  [7] compromise--catastrophic compromise--disaster     compromise--global
## [10] compromise--hughes       compromise--kate         compromise--limits
## [13] compromise--miles        compromise--paris        compromise--set
## [16] compromise--writes       cop26     --cop26        cop26     --final
## [19] cop26     --agreement    cop26     --draft        cop26     --fossil
## [22] cop26     --fuels        cop26     --climate      cop26     --deal
## + ... omitted several edges
```

Example: (18) Co-word analysis (cosine similarity)

```
g_vis <- ggraph(g_cos, layout = "stress") +
  geom_edge_link0(aes(edge_width = weight), edge_colour = "grey66") +
  geom_node_point(fill = "lightblue", colour = "white", shape = 21) +
  geom_node_text(aes(label = name), size = 2.5) +
  scale_edge_width(range = c(0.01, 0.1)) +
  scale_size(range = c(1, 10)) +
  theme_graph()
```

Example: (19) Co-word analysis (cosine similarity)



Additional tools for co-word analysis

- Gephi: Export the network data as a “gml” file (see function `write_gml`) to read it in Gephi
- VOSViewer: Focussed on publication data
- VantagePoint: Focussed on publication and patent data

Sentiment analysis

Sentiment analysis

- Sentiment analysis aims to extract **emotional intentions** from documents (e.g. happiness, surprise, negative/positive feelings)
- It builds on linguistic, psychology, and NLP
- A simple approach to classify documents is to rely on **subjectivity lexicons**
- The `tidtext` package includes several lexicons, which are based on **classification of unigrams**
  - Bing
  - AFIN
  - NRC

Subjectivity lexicons: Bing

- We are often just interested in understanding the **polarity of a document**
- The Bing lexicon classifies unigrams as **positive** or **negative**

```
get_sentiments("bing")
```

```
## # A tibble: 6,786 × 2
##    word        sentiment
##    <chr>       <chr>
##  1 2-faces     negative
##  2 abnormal    negative
##  3 abolish     negative
##  4 abominable  negative
##  5 abominably  negative
##  6 abominate   negative
##  7 abomination negative
##  8 abort       negative
##  9 aborted     negative
## 10 aborts      negative
## # … with 6,776 more rows
```

Subjectivity lexicons: AFINN

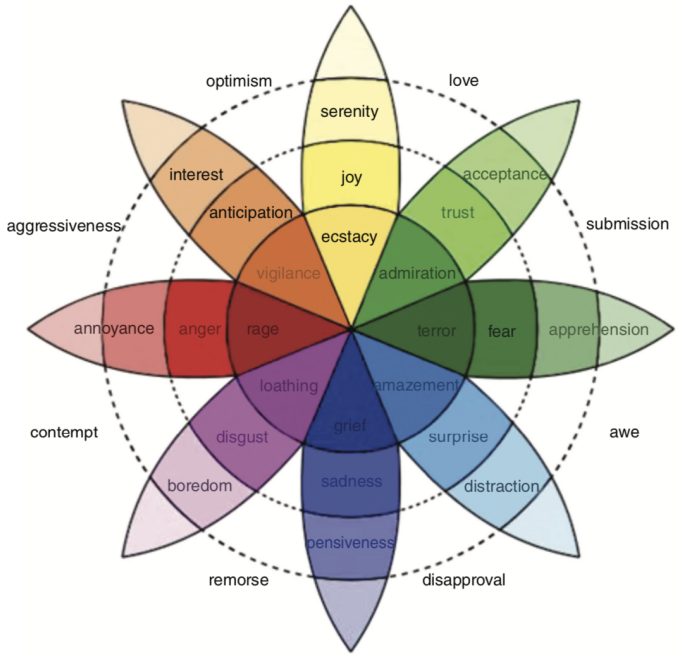
- The AFINN lexicon goes beyond the binary classification of the Bing lexicon
- It assigns a score from -5 (**negative**) to +5 (**positive**) to each unigram

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

```
## # A tibble: 2,477 × 2
##    word       value
##    <chr>      <dbl>
##  1 abandon       -2
##  2 abandoned     -2
##  3 abandons      -2
##  4 abducted      -2
##  5 abduction     -2
##  6 abductions    -2
##  7 abhor         -3
##  8 abhorred      -3
##  9 abhorrent     -3
## 10 abhors        -3
## # … with 2,467 more rows
```

Subjectivity lexicons: NRC

- Plutchik's primary emotions
  - anger
  - anticipation
  - joy
  - trust
  - fear
  - surprise
  - sadness
  - disgust
- All emotions can be derived from the primary ones



Source: Plutchik's wheel of emotion with eight primary emotional states (Kwartler 2017)

Subjectivity lexicons: NRC

```
get_sentiments("nrc")
```

```
## # A tibble: 13,901 × 2
##    word        sentiment
##    <chr>       <chr>
##  1 abacus      trust
##  2 abandon     fear
##  3 abandon     negative
##  4 abandon     sadness
##  5 abandoned   anger
##  6 abandoned   fear
##  7 abandoned   negative
##  8 abandoned   sadness
##  9 abandonment anger
## 10 abandonment fear
## # … with 13,891 more rows
```

Other subjectivity lexicons

- SentiWords: 155,000 English words with a sentiment score from -1 to +1
- WordStat: a negative sentiment is measured on the basis of two rules
  - Negative words that are not preceded by a negation (no, not never) within four words in the same sentence
  - Positive words that are preceded by a negation within four words in the same sentence
  - Similarly in the case of positive sentiments
- Pattern: 2,888 words are scored according to four indicators polarity, subjectivity, intensity and reliability
- SENTIMENT140: unigrams, bigrams, and pairs (unigrams-bigrams; bigrams-unigrams; bigrams-bigramns) classified with negative and postivie sentiments on the basis of 1.6 million tweets
- …

Example: (1) Tokenisation, stopwords, numbers

- Text from 692 news English articles on COP26 published by newspapers on 12 November 2021
- We use the all text field (Title, Headline, Hlead)
- We tokenize data into unigram and remove stopwords as well as numbers

```
my_text_uni <- read_csv("news_articles_example.csv") %>%
  mutate(all_text = paste(Title, Headline, Hlead, sep = " ")) %>%
  select(id, all_text) %>%
  unnest_tokens(output = word, input = all_text) %>%
  anti_join(stop_words) %>%
  mutate(word_numeric = as.numeric(word)) %>%
  filter(is.na(word_numeric)) %>%
  select(-word_numeric)
```

```
## Warning in mask$eval_all_mutate(quo): NAs introduced by coercion
```

Example: (2) Tokenisation, stopwords, numbers

- Our corpus include 191,069 id-unigram pairs

```
print(my_text_uni, n = 6)
```

```
## # A tibble: 191,069 × 2
##       id word
##    <dbl> <chr>
## 1      1 unreal
## 2      1 spectacle
## 3      1 cop26
## 4      1 cop26
## 5      1 prepackaged
## 6      1 feel
## # … with 191,063 more rows
```

Example: (3) Bing lexicon

- 18,108 id-unigram pairs are classified with the Bing lexicon

```
my_text_uni_bing <- my_text_uni %>%
  inner_join(get_sentiments("bing"))

print(my_text_uni_bing, n = 6)
```

```
## # A tibble: 18,108 × 3
##      id word       sentiment
##   <dbl> <chr>      <chr>
## 1     1 unreal     positive
## 2     1 indulgence positive
## 3     1 unreal     positive
## 4     1 indulgence positive
## 5     1 hard       negative
## 6     1 bore       negative
## # … with 18,102 more rows
```

Example: (4) Bing lexicon

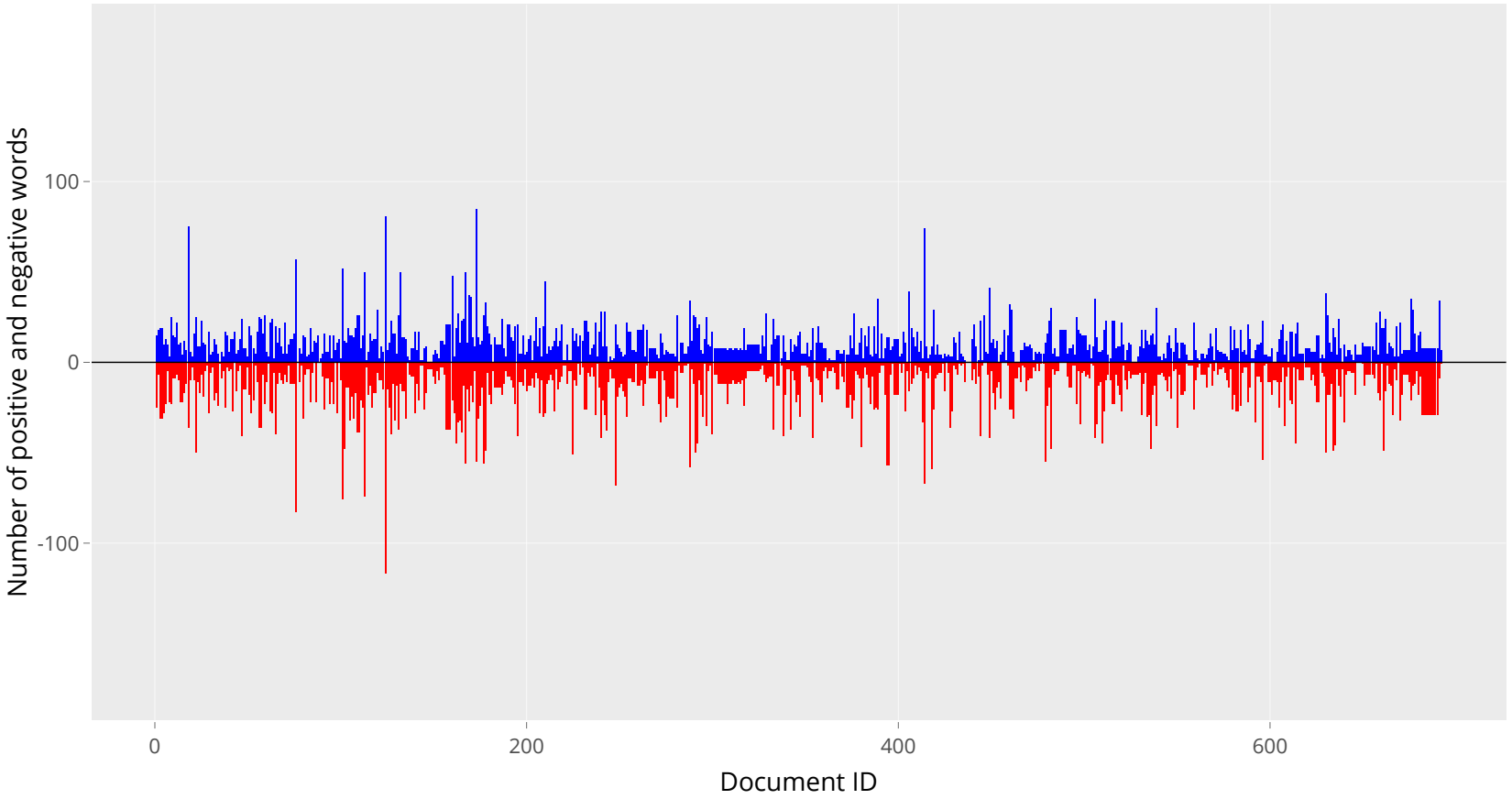
```
my_text_uni_bing_by_id <- my_text_uni_bing %>%
  count(id, sentiment) %>%
  pivot_wider(names_from = sentiment, values_from = n, values_fill = list(n = 0))

print(my_text_uni_bing_by_id, n = 6)
```

```
## # A tibble: 688 × 3
##      id negative positive
##   <dbl>    <int>    <int>
## 1     1       25       15
## 2     2        7       18
## 3     3       31       19
## 4     4       31       19
## 5     5       28       10
## 6     6       23       13
## # … with 682 more rows
```

Example: (5) Bing lexicon

```
g <- my_text_uni_bing_by_id %>%
 ggplot(aes(id)) +
 geom_bar(aes(y =  positive), stat = "identity", show.legend = FALSE, fill = "blue") +
 geom_bar(aes(y = -negative), stat = "identity", show.legend = FALSE, fill = "red") +
 scale_y_continuous(limits = c(-180, +180)) +
 xlab(label = "Document ID") +
 ylab(label =  "Number of positive and negative words") +
 geom_hline(yintercept = 0, color = "black", size = 0.2)
```

Example: (6) Bing lexicon



Example: (7) Bing lexicon

- We can identify the top-15 positive and negative words

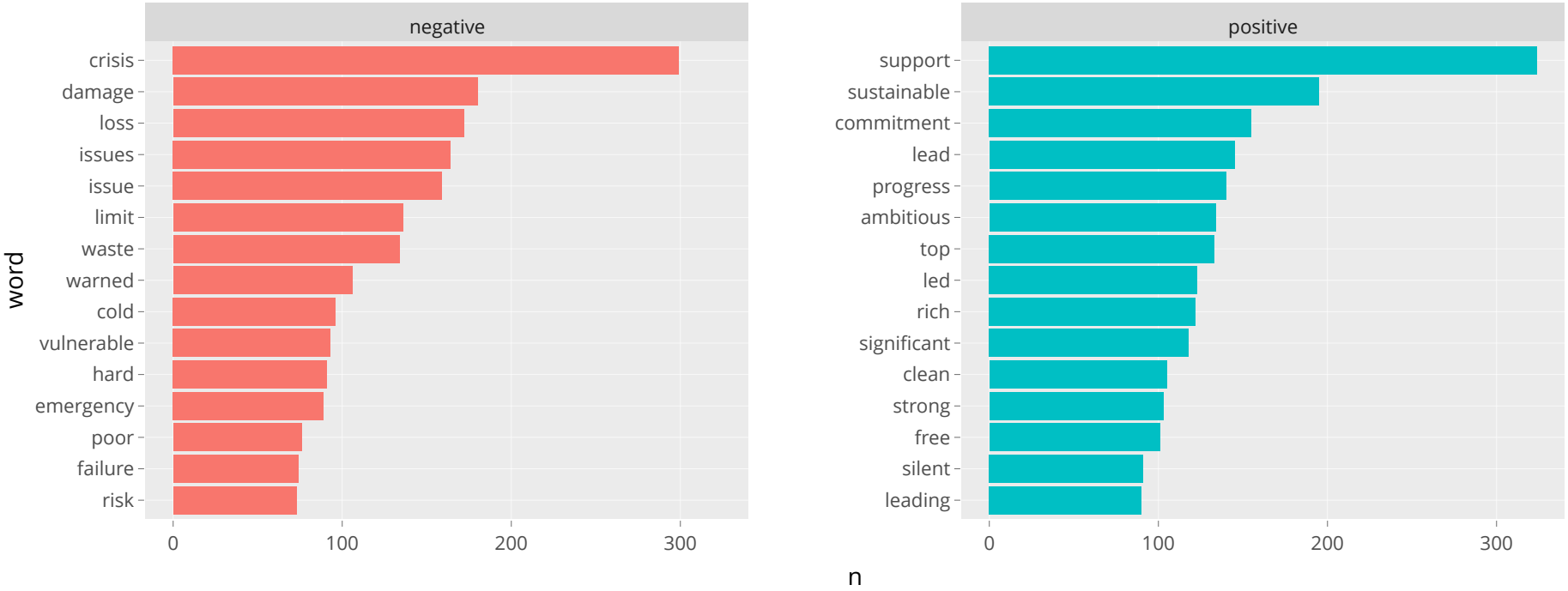
```
my_text_uni_bing_by_count <- my_text_uni_bing %>%
  count(word, sentiment) %>%
  ungroup()

print(my_text_uni_bing_by_count, n = 6)
```

```
## # A tibble: 2,232 × 3
##   word      sentiment     n
##   <chr>     <chr>     <int>
## 1 abolish   negative      2
## 2 abruptly  negative      4
## 3 absence   negative     15
## 4 absurd    negative      3
## 5 abundance positive      7
## 6 abundant  positive      4
## # … with 2,226 more rows
```

Example: (8) Bing lexicon

```
g <- my_text_uni_bing_by_count %>%
  group_by(sentiment) %>%
  top_n(15, n) %>%
  ungroup %>%
  mutate(sentiment = as.factor(sentiment),
         word = reorder_within(word, n, sentiment)) %>%
  ggplot(aes(word, n, fill = sentiment)) +
  geom_col() +
  facet_wrap(~sentiment,  nrow = 1, scales = "free_y") +
  coord_flip() +
  theme(legend.position = "none") +
  scale_x_reordered()
```

Example: (9) Bing lexicon



Example: (10) AFINN lexicon

- 18,823 id-unigram pairs are classified with the AFINN lexicon

```
my_text_uni_afinn <- my_text_uni %>%
  inner_join(get_sentiments("afinn"))

print(my_text_uni_afinn, n = 6)
```

```
## # A tibble: 18,823 × 3
##      id word   value
##   <dbl> <chr>  <dbl>
## 1     1 united     1
## 2     1 hard      -1
## 3     1 chance     2
## 4     1 save       2
## 5     1 bore      -2
## 6     1 agreed     1
## # … with 18,817 more rows
```

Example: (11) AFINN lexicon

- We can calculate a “total sentiment” for a given article/document

```
my_text_uni_afinn_by_id <- my_text_uni_afinn %>%
  group_by(id) %>%
  summarise(sentiment = sum(value)) %>%
  mutate(overall_sentiment = case_when(
    sentiment >  0 ~ "positive",
    sentiment <  0 ~ "negative",
    sentiment == 0 ~ "neutral"))
```

Example: (12) AFINN lexicon

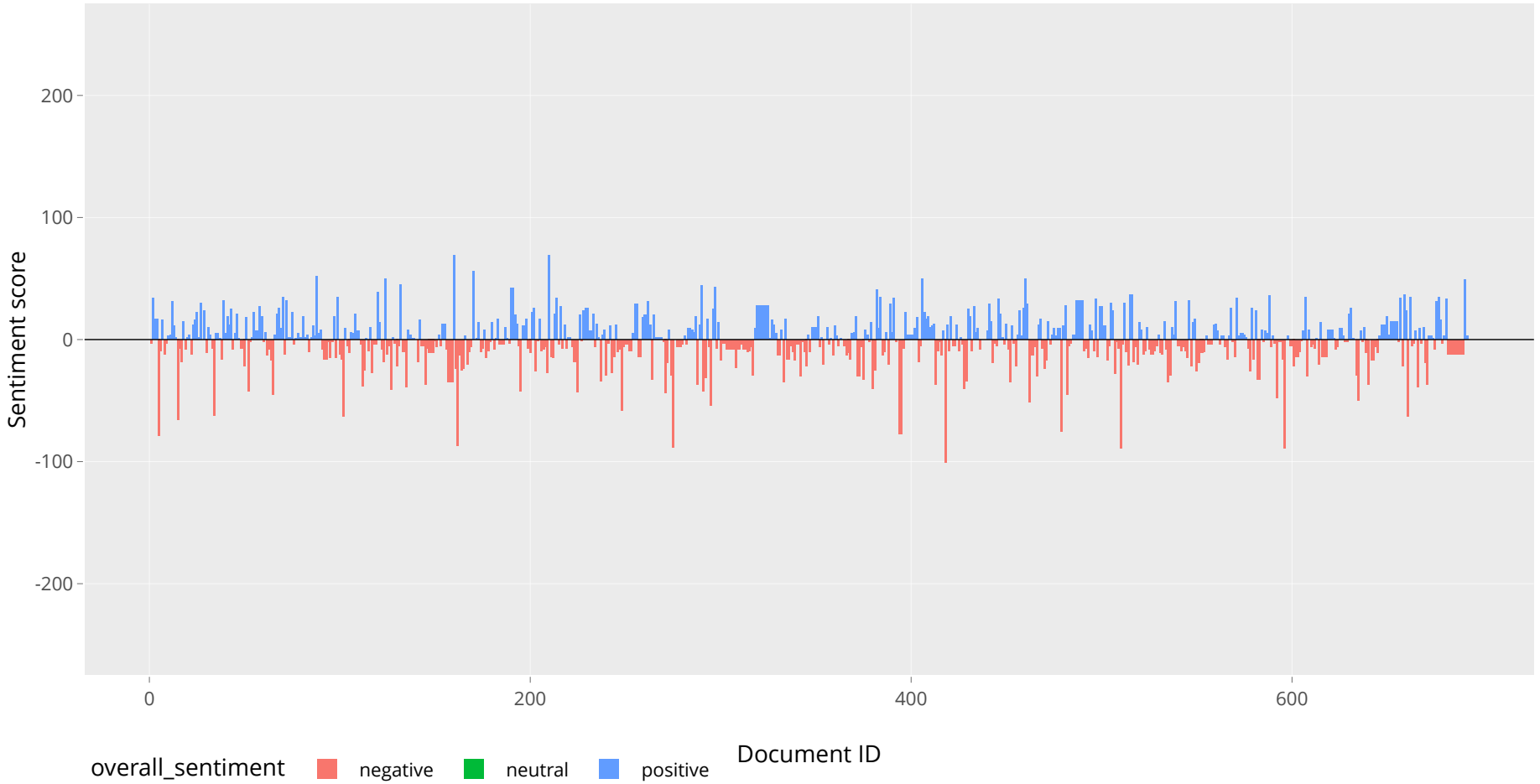
```
print(my_text_uni_afinn_by_id, n = 6)
```

```
## # A tibble: 689 × 3
##       id sentiment overall_sentiment
##    <dbl>     <dbl> <chr>
## 1      1        -3 negative
## 2      2        34 positive
## 3      3        17 positive
## 4      4        17 positive
## 5      5       -79 negative
## 6      6        -9 negative
## # … with 683 more rows
```

Example: (13) AFINN lexicon

```
g <- my_text_uni_afinn_by_id %>%
 ggplot(aes(x = id, y = sentiment, fill = overall_sentiment)) +
 geom_bar(stat = "identity") +
 scale_y_continuous(limits = c(-250, +250)) +
 xlab(label = "Document ID") +
 ylab(label =  "Sentiment score") +
 geom_hline(yintercept = 0, color = "black", size = 0.2) +
 theme(legend.position = "bottom")
```

Example: (14) AFINN lexicon



Example: (15) NRC lexicon

- 63,590 id-unigram pairs are classified with the NRC lexicon

```
my_text_uni_nrc <- my_text_uni %>%
  inner_join(get_sentiments("nrc"))

print(my_text_uni_nrc, n = 6)
```

```
## # A tibble: 83,912 × 3
##      id word      sentiment
##   <dbl> <chr>     <chr>
## 1     1 spectacle negative
## 2     1 spectacle positive
## 3     1 spectacle negative
## 4     1 spectacle positive
## 5     1 united    positive
## 6     1 united    trust
## # … with 83,906 more rows
```

Example: (16) NRC lexicon

- We can count words by sentiment

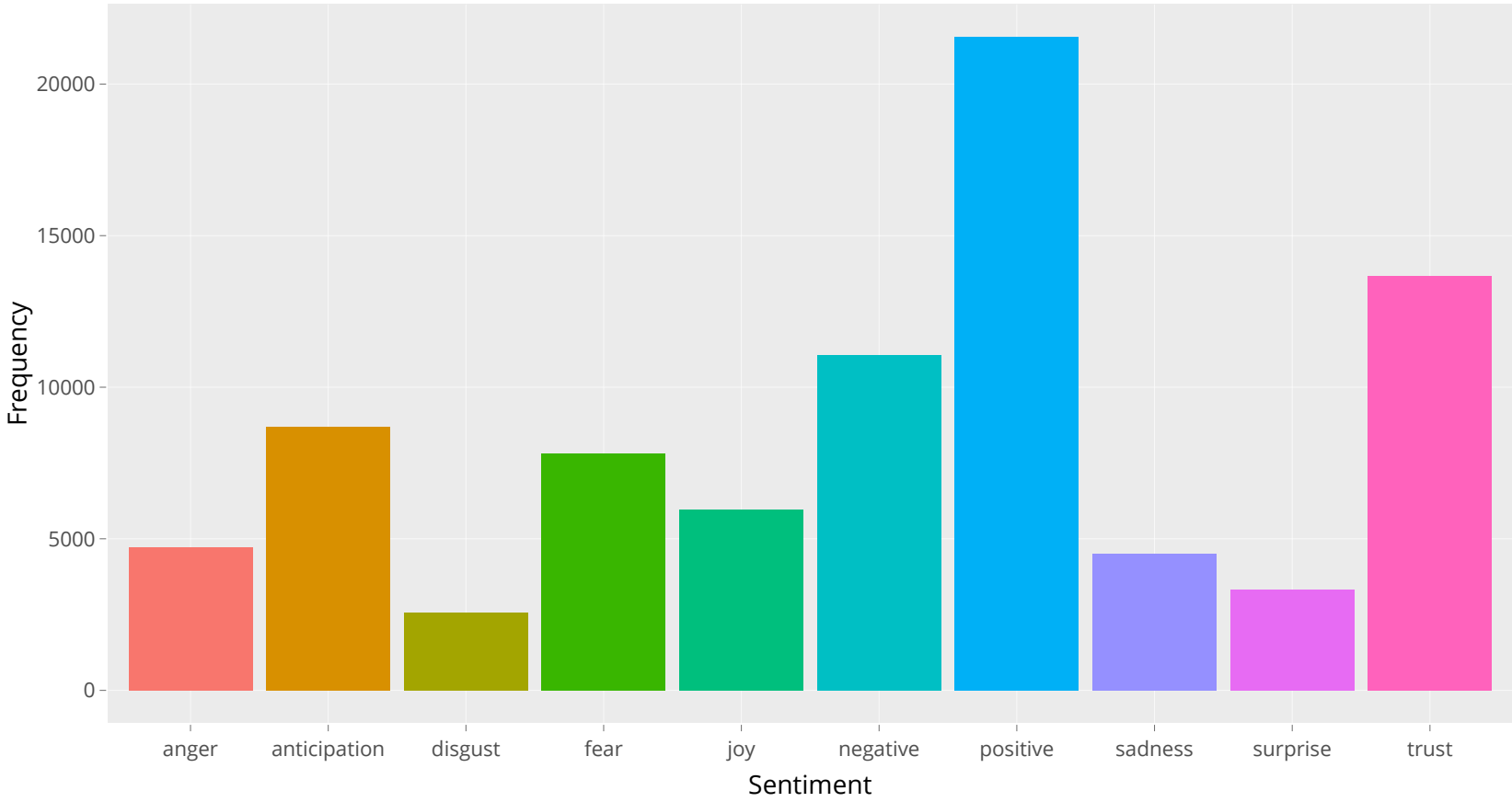
```
my_text_uni_nrc_by_id <- my_text_uni_nrc %>%
  count(sentiment) %>%
  ungroup()

print(my_text_uni_nrc_by_id, n = 6)
```

```
## # A tibble: 10 × 2
##   sentiment        n
##   <chr>        <int>
## 1 anger         4722
## 2 anticipation  8691
## 3 disgust       2577
## 4 fear          7824
## 5 joy           5970
## 6 negative     11069
## # … with 4 more rows
```

Example: (17) NRC lexicon

```
g <- my_text_uni_nrc_by_id %>%
 ggplot(aes(x = sentiment, y = n, fill = sentiment)) +
 geom_bar(stat = "identity") +
 xlab(label = "Sentiment") +
 ylab(label = "Frequency") +
 theme(legend.position = "none")
```

Example: (18) NRC lexicon



Regular expressions

Identifying words in the corpus

- In some cases, it is helpful to identify whether a specif word (and/or its variations) occurs in a corpus
- **Regular expressions** are sequences of characters that enable us to identify detect, extract, or replace patterns of text
- Regular expressions have their own **syntactic rules**
- The `stringr` package provides a number of functions to work with patterns of texts and regular expression

stringr cheat sheat

String manipulation with stringr : : **CHEAT SHEET**

The **stringr** package provides a set of internally consistent tools for working with character strings, i.e. sequences of characters surrounded by quotation marks.

Detect Matches

str_detect(string, **pattern**) Detect the presence of a pattern match in a string.
str_detect(fruit, "a")

str_which(string, **pattern**) Find the indexes of strings that contain a pattern match.
str_which(fruit, "a")

str_count(string, **pattern**) Count the number of matches in a string.
str_count(fruit, "a")

str_locate(string, **pattern**) Locate the positions of pattern matches in a string. Also **str_locate_all**. *str_locate(fruit, "a")*

Subset Strings

str_sub(string, start = 1L, end = -1L) Extract substrings from a character vector.
str_sub(fruit, 1, 3); str_sub(fruit, -2)

str_subset(string, **pattern**) Return only the strings that contain a pattern match.
str_subset(fruit, "b")

str_extract(string, **pattern**) Return the first pattern match found in each string, as a vector. Also **str_extract_all** to return every pattern match. *str_extract(fruit, "[aeiou]")*

str_match(string, **pattern**) Return the first pattern match found in each string, as a matrix with a column for each ( ) group in pattern. Also **str_match_all**.
str_match(sentences, "(a|the) ([^]+)")

Manage Lengths

str_length(string) The width of strings (i.e. number of code points, which generally equals the number of characters). *str_length(fruit)*

str_pad(string, width, side = c("left", "right", "both"), pad = " ") Pad strings to constant width. *str_pad(fruit, 17)*

str_trunc(string, width, side = c("right", "left", "center"), ellipsis = "...") Truncate the width of strings, replacing content with ellipsis.
str_trunc(fruit, 3)

str_trim(string, side = c("both", "left", "right")) Trim whitespace from the start and/or end of a string. *str_trim(fruit)*

Mutate Strings

str_sub() **<-** value. Replace substrings by identifying the substrings with str_sub() and assigning into the results.
str_sub(fruit, 1, 3) <- "str"

str_replace(string, **pattern**, replacement) Replace the first matched pattern in each string. *str_replace(fruit, "a", "-")*

str_replace_all(string, **pattern**, replacement) Replace all matched patterns in each string. *str_replace_all(fruit, "a", "-")*

str_to_lower(string, locale = "en")[1] Convert strings to lower case.
str_to_lower(sentences)

str_to_upper(string, locale = "en")[1] Convert strings to upper case.
str_to_upper(sentences)

str_to_title(string, locale = "en")[1] Convert strings to title case. *str_to_title(sentences)*

Join and Split

str_c(..., sep = "", collapse = NULL) Join multiple strings into a single string.
str_c(letters, LETTERS)

str_c(..., sep = "", **collapse = ""**) Collapse a vector of strings into a single string.
str_c(letters, collapse = "")

str_dup(string, times) Repeat strings times times. *str_dup(fruit, times = 2)*

str_split_fixed(string, **pattern**, n) Split a vector of strings into a matrix of substrings (splitting at occurrences of a pattern match). Also **str_split** to return a list of substrings.
str_split_fixed(fruit, " ", n=2)

str_glue(..., .sep = "", .envir = parent.frame()) Create a string from strings and {expressions} to evaluate. *str_glue("Pi is {pi}")*

str_glue_data(.x, ..., .sep = "", .envir = parent.frame(), .na = "NA") Use a data frame, list, or environment to create a string from strings and {expressions} to evaluate.
str_glue_data(mtcars, "{rownames(mtcars)} has {hp} hp")

Order Strings

str_order(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...)[1] Return the vector of indexes that sorts a character vector. *x[str_order(x)]*

str_sort(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...)[1] Sort a character vector.
str_sort(x)

Helpers

str_conv(string, encoding) Override the encoding of a string. *str_conv(fruit,"ISO-8859-1")*

str_view(string, **pattern**, match = NA) View HTML rendering of first regex match in each string. *str_view(fruit, "[aeiou]")*

str_view_all(string, **pattern**, match = NA) View HTML rendering of all regex matches.
str_view_all(fruit, "[aeiou]")

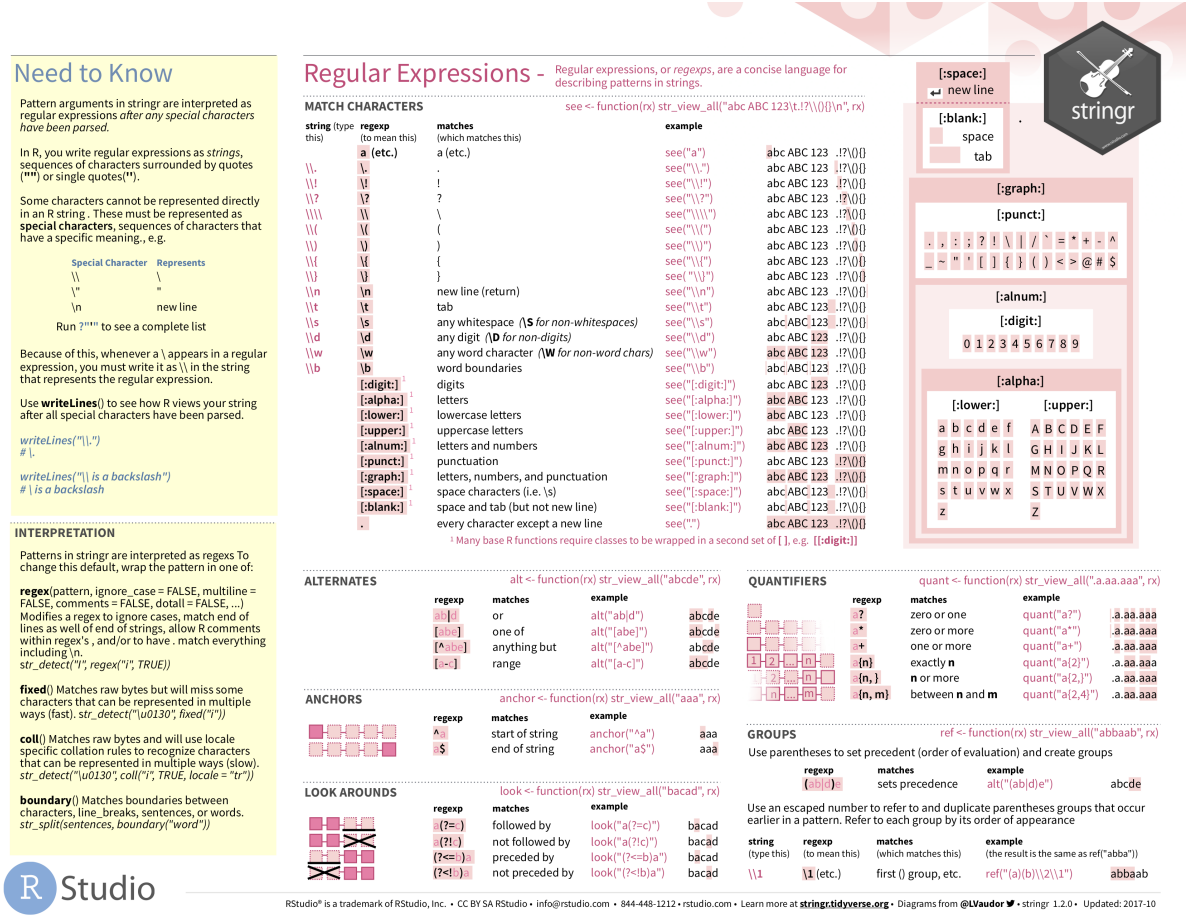
str_wrap(string, width = 80, indent = 0, exdent = 0) Wrap strings into nicely formatted paragraphs. *str_wrap(sentences, 20)*

RStudio

[1] See bit.ly/ISO639-1 for a complete list of locales.

Source: https://rstudio.com/resources/cheatsheets/

stringr cheat sheat



Need to Know

Pattern arguments in stringr are interpreted as regular expressions *after any special characters have been parsed.*

In R, you write regular expressions as *strings*, sequences of characters surrounded by quotes (**""**) or single quotes(**''**).

Some characters cannot be represented directly in an R string . These must be represented as **special characters**, sequences of characters that have a specific meaning., e.g.

Special Character	Represents
\\	\
\"	"
\n	new line

Run **?"'"** to see a complete list

Because of this, whenever a \ appears in a regular expression, you must write it as \\ in the string that represents the regular expression.

Use **writeLines**() to see how R views your string after all special characters have been parsed.

```
writeLines("\\.")
# \.

writeLines("\\ is a backslash")
# \ is a backslash
```

INTERPRETATION

Patterns in stringr are interpreted as regexs To change this default, wrap the pattern in one of:

regex(pattern, ignore_case = FALSE, multiline = FALSE, comments = FALSE, dotall = FALSE, ...) Modifies a regex to ignore cases, match end of lines as well of end of strings, allow R comments within regex's , and/or to have . match everything including \n.
str_detect("i", regex("i", TRUE))

fixed() Matches raw bytes but will miss some characters that can be represented in multiple ways (fast). *str_detect("\u0130", fixed("i"))*

coll() Matches raw bytes and will use locale specific collation rules to recognize characters that can be represented in multiple ways (slow). *str_detect("\u0130", coll("i", TRUE, locale = "tr"))*

boundary() Matches boundaries between characters, line_breaks, sentences, or words. *str_split(sentences, boundary("word"))*

Regular Expressions -

Regular expressions, or *regexps*, are a concise language for describing patterns in strings.

MATCH CHARACTERS

see <- function(rx) str_view_all("abc ABC 123\t.!?\\(){}\n", rx)

string (type this)	regexp (to mean this)	matches (which matches this)	example	
	a (etc.)	a (etc.)	see("a")	abc ABC 123 .!?\(){}
\\.	\.	.	see("\\.")	abc ABC 123 .!?\(){}
\\!	\!	!	see("\\!")	abc ABC 123 .!?\(){}
\\?	\?	?	see("\\?")	abc ABC 123 .!?\(){}
\\\\	\\	\	see("\\\\")	abc ABC 123 .!?\(){}
\\(	\(	(	see("\\(")	abc ABC 123 .!?\(){}
\\)	\)	)	see("\\)")	abc ABC 123 .!?\(){}
\\{	\{	{	see("\\{")	abc ABC 123 .!?\(){}
\\}	\}	}	see("\\}")	abc ABC 123 .!?\(){}
\\n	\n	new line (return)	see("\\n")	abc ABC 123 .!?\(){}
\\t	\t	tab	see("\\t")	abc ABC 123 .!?\(){}
\\s	\s	any whitespace (**\S** *for non-whitespaces*)	see("\\s")	abc ABC 123 .!?\(){}
\\d	\d	any digit (**\D** *for non-digits*)	see("\\d")	abc ABC 123 .!?\(){}
\\w	\w	any word character (**\W** *for non-word chars*)	see("\\w")	abc ABC 123 .!?\(){}
\\b	\b	word boundaries	see("\\b")	abc ABC 123 .!?\(){}
	[:digit:][1]	digits	see("[:digit:]")	abc ABC 123 .!?\(){}
	[:alpha:][1]	letters	see("[:alpha:]")	abc ABC 123 .!?\(){}
	[:lower:][1]	lowercase letters	see("[:lower:]")	abc ABC 123 .!?\(){}
	[:upper:][1]	uppercase letters	see("[:upper:]")	abc ABC 123 .!?\(){}
	[:alnum:][1]	letters and numbers	see("[:alnum:]")	abc ABC 123 .!?\(){}
	[:punct:][1]	punctuation	see("[:punct:]")	abc ABC 123 .!?\(){}
	[:graph:][1]	letters, numbers, and punctuation	see("[:graph:]")	abc ABC 123 .!?\(){}
	[:space:][1]	space characters (i.e. \s)	see("[:space:]")	abc ABC 123 .!?\(){}
	[:blank:][1]	space and tab (but not new line)	see("[:blank:]")	abc ABC 123 .!?\(){}
	.	every character except a new line	see(".")	abc ABC 123 .!?\(){}

[1] Many base R functions require classes to be wrapped in a second set of **[ ]**, e.g. **[[:digit:]]**

[:space:] ↵ new line

[:blank:] space, tab

.

[:graph:]

[:punct:]

. , : ; ? ! \ | / ` = * + - ^
_ ~ " ' [ ] { } ( ) < > @ # $

[:alnum:]

[:digit:]

0 1 2 3 4 5 6 7 8 9

[:alpha:]

[:lower:]

a b c d e f
g h i j k l
m n o p q r
s t u v w x
z

[:upper:]

A B C D E F
G H I J K L
M N O P Q R
S T U V W X
Z

ALTERNATES

alt <- function(rx) str_view_all("abcde", rx)

regexp	matches	example	
ab|d	or	alt("ab|d")	abcde
[abe]	one of	alt("[abe]")	abcde
[^abe]	anything but	alt("[^abe]")	abcde
[a-c]	range	alt("[a-c]")	abcde

ANCHORS

anchor <- function(rx) str_view_all("aaa", rx)

regexp	matches	example	
^a	start of string	anchor("^a")	aaa
a$	end of string	anchor("a$")	aaa

LOOK AROUNDS

look <- function(rx) str_view_all("bacad", rx)

regexp	matches	example	
a(?=c)	followed by	look("a(?=c)")	bacad
a(?!c)	not followed by	look("a(?!c)")	bacad
(?<=b)a	preceded by	look("(?<=b)a")	bacad
(?<!b)a	not preceded by	look("(?<!b)a")	bacad

QUANTIFIERS

quant <- function(rx) str_view_all(".a.aa.aaa", rx)

regexp	matches	example	
a?	zero or one	quant("a?")	.a.aa.aaa
a*	zero or more	quant("a*")	.a.aa.aaa
a+	one or more	quant("a+")	.a.aa.aaa
a{n}	exactly **n**	quant("a{2}")	.a.aa.aaa
a{n,}	**n** or more	quant("a{2,}")	.a.aa.aaa
a{n, m}	between **n** and **m**	quant("a{2,4}")	.a.aa.aaa

GROUPS

ref <- function(rx) str_view_all("abbaab", rx)

Use parentheses to set precedent (order of evaluation) and create groups

regexp	matches	example	
(ab|d)e	sets precedence	alt("(ab|d)e")	abcde

Use an escaped number to refer to and duplicate parentheses groups that occur earlier in a pattern. Refer to each group by its order of appearance

string (type this)	regexp (to mean this)	matches (which matches this)	example (the result is the same as ref("abba"))	
\\1	\1 (etc.)	first () group, etc.	ref("(a)(b)\\2\\1")	abbaab

RStudio® is a trademark of RStudio, Inc. • CC BY SA RStudio • info@rstudio.com • 844-448-1212 • rstudio.com • Learn more at **stringr.tidyverse.org** • Diagrams from **@LVaudor** • stringr 1.2.0 • Updated: 2017-10

Source: https://rstudio.com/resources/cheatsheets/

Basic matches

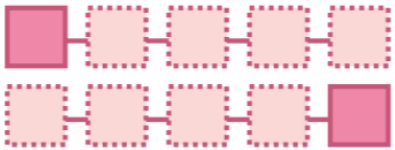
- We have a vector of strings (e.g. single words or groups of words)
- We want to identify all elements with the “sussex” pattern

```
my_text <- c("brighton and hove sussex",
             "sussex university",
             "sussex science policy research unit",
             "research impact sussex")

str_view(my_text, "sussex")
```

- brighton and hove sussex
- sussex university
- sussex science policy research unit
- research impact sussex

Anchors

ANCHORS



anchor <- function(rx) str_view_all("aaa", rx)

regexp	matches	example	
^a	start of string	anchor("^a")	aaa
a$	end of string	anchor("a$")	aaa

Source: https://rstudio.com/resources/cheatsheets/

Anchors

```
my_text <- c("brighton and hove sussex",
             "sussex university",
             "sussex science policy research unit",
             "research impact sussex")

str_view(my_text, "^sussex")
```

- brighton and hove sussex
- sussex university
- sussex science policy research unit
- research impact sussex

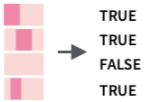
Anchors

```
my_text <- c("brighton and hove sussex",
             "sussex university",
             "sussex science policy research unit",
             "research impact sussex")

str_view(my_text, "sussex$")
```

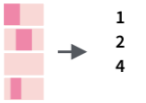
- brighton and hove `sussex`
- sussex university
- sussex science policy research unit
- research impact `sussex`

Beyond `str_view`

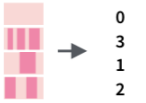
Detect Matches



str_detect(string, **pattern**) Detect the presence of a pattern match in a string.
str_detect(fruit, "a")



str_which(string, **pattern**) Find the indexes of strings that contain a pattern match.
str_which(fruit, "a")

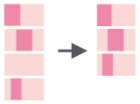


str_count(string, **pattern**) Count the number of matches in a string.
str_count(fruit, "a")

str_locate(string, **pattern**) Locate the positions of pattern matches in a string. Also **str_locate_all**. *str_locate(fruit, "a")*

Subset Strings

str_sub(string, start = 1L, end = -1L) Extract substrings from a character vector.
str_sub(fruit, 1, 3); str_sub(fruit, -2)



str_subset(string, **pattern**) Return only the strings that contain a pattern match.
str_subset(fruit, "b")

str_extract(string, **pattern**) Return the first pattern match found in each string, as a vector. Also **str_extract_all** to return every pattern match. *str_extract(fruit, "[aeiou]")*

str_match(string, **pattern**) Return the first pattern match found in each string, as a matrix with a column for each ( ) group in pattern. Also **str_match_all**.
str_match(sentences, "(a|the) ([^]+)")

Source: https://rstudio.com/resources/cheatsheets/

Matching characters and quantifiers

Regular Expressions -

Regular expressions, or *regexps*, are a concise language for describing patterns in strings.

MATCH CHARACTERS

see <- function(rx) str_view_all("abc ABC 123\t.!?\\(){}\n", rx)

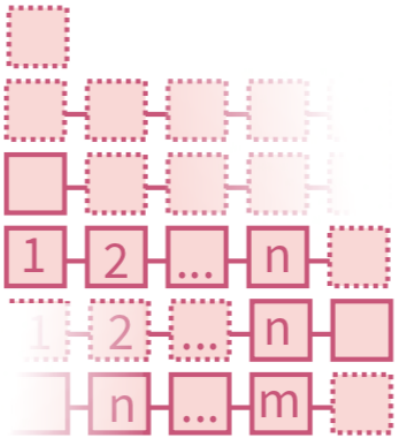
string (type this)	**regexp** (to mean this)	**matches** (which matches this)	**example**	
	a (etc.)	a (etc.)	see("a")	abc ABC 123 .!?\(){}
\\.	**\.**	.	see("\\.")	abc ABC 123 .!?\(){}
\\!	**\!**	!	see("\\!")	abc ABC 123 .!?\(){}
\\?	**\?**	?	see("\\?")	abc ABC 123 .!?\(){}
\\\\	****	\	see("\\\\")	abc ABC 123 .!?\(){}
\\(	**\(**	(	see("\\(")	abc ABC 123 .!?\(){}
\\)	**\)**	)	see("\\)")	abc ABC 123 .!?\(){}
\\{	**\{**	{	see("\\{")	abc ABC 123 .!?\(){}
\\}	**\}**	}	see("\\}")	abc ABC 123 .!?\(){}
\\n	**\n**	new line (return)	see("\\n")	abc ABC 123 .!?\(){}
\\t	**\t**	tab	see("\\t")	abc ABC 123 .!?\(){}
\\s	**\s**	any whitespace (**\S** *for non-whitespaces*)	see("\\s")	abc ABC 123 .!?\(){}
\\d	**\d**	any digit (**\D** *for non-digits*)	see("\\d")	abc ABC 123 .!?\(){}
\\w	**\w**	any word character (**\W** *for non-word chars*)	see("\\w")	abc ABC 123 .!?\(){}
\\b	**\b**	word boundaries	see("\\b")	abc ABC 123 .!?\(){}
	[:digit:] [1]	digits	see("[:digit:]")	abc ABC 123 .!?\(){}
	[:alpha:] [1]	letters	see("[:alpha:]")	abc ABC 123 .!?\(){}
	[:lower:] [1]	lowercase letters	see("[:lower:]")	abc ABC 123 .!?\(){}
	[:upper:] [1]	uppercase letters	see("[:upper:]")	abc ABC 123 .!?\(){}
	[:alnum:] [1]	letters and numbers	see("[:alnum:]")	abc ABC 123 .!?\(){}
	[:punct:] [1]	punctuation	see("[:punct:]")	abc ABC 123 .!?\(){}
	[:graph:] [1]	letters, numbers, and punctuation	see("[:graph:]")	abc ABC 123 .!?\(){}
	[:space:] [1]	space characters (i.e. \s)	see("[:space:]")	abc ABC 123 .!?\(){}
	[:blank:] [1]	space and tab (but not new line)	see("[:blank:]")	abc ABC 123 .!?\(){}
	.	every character except a new line	see(".")	abc ABC 123 .!?\(){}

[1] Many base R functions require classes to be wrapped in a second set of **[ ]**, e.g. **[[:digit:]]**

Source: https://rstudio.com/resources/cheatsheets/

Matching characters and quantifiers

QUANTIFIERS



quant <- function(rx) str_view_all(".a.aa.aaa", rx)

regexp	matches	example	
a?	zero or one	quant("a?")	.a.aa.aaa
a*	zero or more	quant("a*")	.a.aa.aaa
a+	one or more	quant("a+")	.a.aa.aaa
a{n}	exactly **n**	quant("a{2}")	.a.aa.aaa
a{n,}	**n** or more	quant("a{2,}")	.a.aa.aaa
a{n, m}	between **n** and **m**	quant("a{2,4}")	.a.aa.aaa

Source: https://rstudio.com/resources/cheatsheets/

Matching characters and quantifiers

- Suppose you have the text below and that you want to know whether that text includes the word “interchangeability” or variations of this word (e.g. interchangeable, interchange)
- Suppose the text also includes typos (e.g. OCR from historical sources)

```
my_text  <- c("Parts from the same car manufacturer areinterchangeable",
              "Interchangeability is a principle of mass production of cars")

my_regex <- regex("[:alpha:]*interchang[:alpha:]*", ignore_case = T)

str_extract(my_text, my_regex)


## [1] "areinterchangeable" "Interchangeability"
```

Matching characters and quantifiers

- Detecting singulars and plurals

```
my_text  <- c("Parts from the same car manufacturer areinterchangeable",
              "Interchangeability is a principle of mass production of cars")

my_regex <- regex("[:alpha:]*car[:alpha:]*", ignore_case = T)

str_extract(my_text, my_regex)


## [1] "car"  "cars"
```

Alternates

ALTERNATES

```
alt <- function(rx) str_view_all("abcde", rx)
```

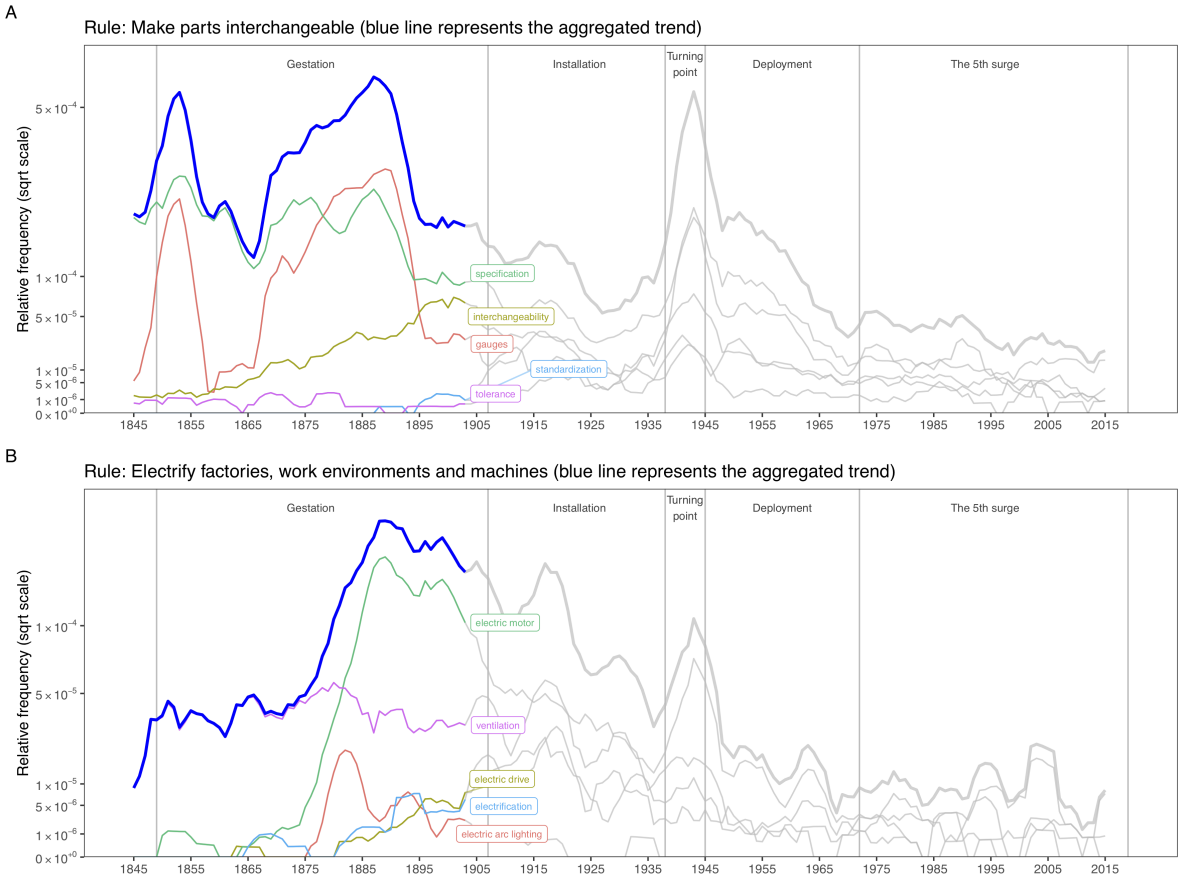
regexp	matches	example	
ab|d	or	alt("ab|d")	abcde
[abe]	one of	alt("[abe]")	abcde
[^abe]	anything but	alt("[^abe]")	abcde
[a-c]	range	alt("[a-c]")	abcde

Source: https://rstudio.com/resources/cheatsheets/

Alternates

```
my_text  <- c("That book has been digitalized",
              "Access to digitalised books has increased")

my_regex <- regex("digitali[z|s]e[:alpha:]*", ignore_case = T)

str_extract(my_text, my_regex)


## [1] "digitalized" "digitalised"
```

Example of application of regex



Source: Bone and Rotolo (2020)

Topic modelling

Topic modelling

- **Topic modelling** allows us to assign documents to topics
- The classification is often **unsupervised** - we ask the computer to make sense of the content of documents
- **Latent Dirichlet Allocation (LDA)** has become a very popular topic modelling approach (David M. Blei et al. 2003; David M. Blei 2012)

Latent Dirichlet Allocation (LDA)

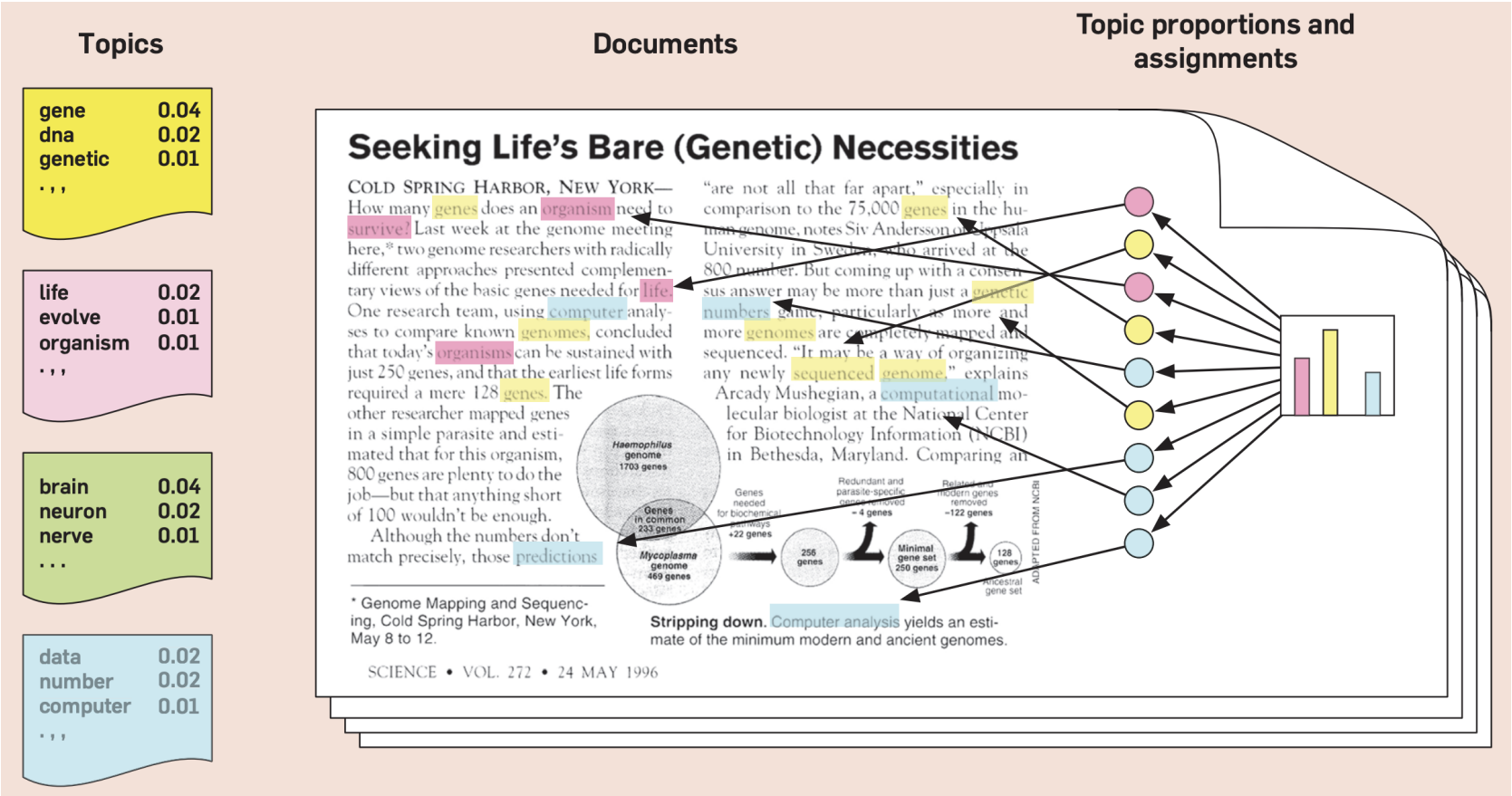
- **Key ideas**:
  - A document is a mix of **latent topics** (e.g. Document 1 is 60% topic A and 40% topic B)
  - A topic is a distribution of words describing it
  - Documents can overlap in terms of content, i.e. topics
  - We know (or we need to guess) a priori how many topics are in the set of documents
- **Probabilistic approach**: a document can belong to multiple topics and a word can have different meaning (not possible with standard cluster analysis)
- Several variations of LDA exist

Latent Dirichlet Allocation (LDA)

$$\begin{bmatrix} 1 & \dots & T \\ \dots & \dots & \dots \\ D & \dots & \dots \end{bmatrix} * \begin{bmatrix} 1 & \dots & W \\ \dots & \dots & \dots \\ T & \dots & \dots \end{bmatrix} = \begin{bmatrix} 1 & \dots & W \\ \dots & \dots & \dots \\ D & \dots & \dots \end{bmatrix}$$

- D documents
- T topics
- W words (vocabulary)

Latent Dirichlet Allocation (LDA)



Source: David M. Blei (2012) (blue = data analysis, pink = evolutionary biology, yellow = genetics)

Latent Dirichlet Allocation (LDA)

- Let's use all text available in our data
- We prepare the text removing stopwords and numbers
- We need to prepare the document-term matrix

```
my_text_uni <- read_csv("news_articles_example.csv") %>%
  mutate(article_text = paste(Title, Headline, Hlead, sep = " ")) %>%
  select(id, article_text) %>%
  unnest_tokens(output = word, input = article_text) %>%
  anti_join(stop_words) %>%
  mutate(word_numeric = as.numeric(word)) %>%
  filter(is.na(word_numeric)) %>%
  select(-word_numeric) %>%
  count(id, word) %>%
  cast_dtm(id, word, n)
```

```
## Warning in mask$eval_all_mutate(quo): NAs introduced by coercion
```

```
my_text_uni
```

```
## <<DocumentTermMatrix (documents: 692, terms: 23633)>>
## Non-/sparse entries: 133282/16220754
## Sparsity           : 99%
```

Latent Dirichlet Allocation (LDA)

- LDA will start with a random assignment of probabilities of documents as belonging to topics (we need to set a `seed` for reproducibility purposes)
- The process is iterative process: it improves the allocation at each step
- We need the package `topicmodels`
- We start assuming we have 4 topics

```
library(topicmodels)
my_topics <- LDA(my_text_uni, k = 4, control = list(seed = 2020))
my_topics <- tidy(my_topics, matrix = "beta")
```

Latent Dirichlet Allocation (LDA)

```
my_topics
```

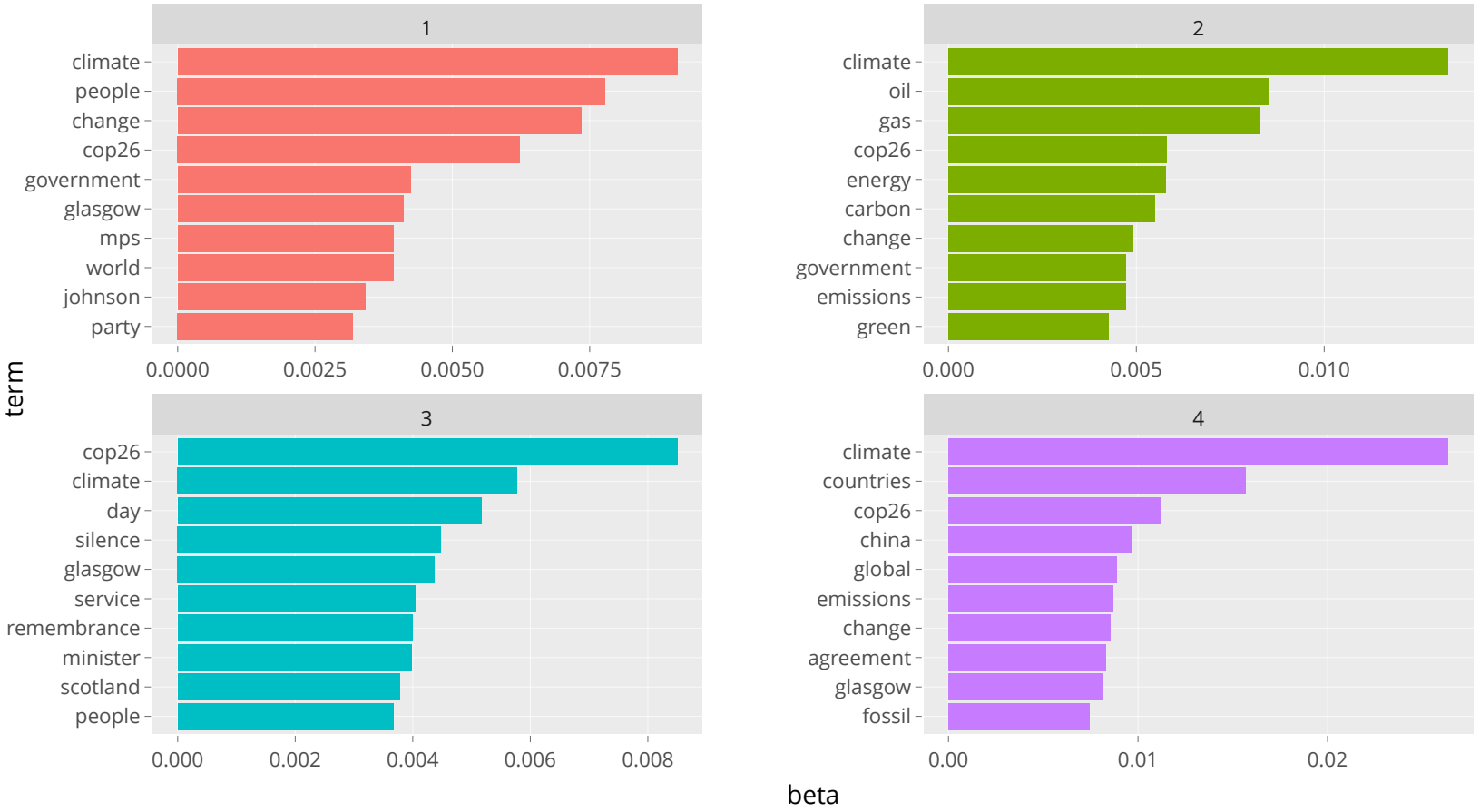
```
## # A tibble: 94,532 × 3
##    topic term         beta
##    <int> <chr>       <dbl>
##  1     1 40,000  1.89e-  5
##  2     2 40,000  5.20e- 11
##  3     3 40,000  1.29e-141
##  4     4 40,000  4.92e-  5
##  5     1 absence 2.76e-  5
##  6     2 absence 9.15e-  5
##  7     3 absence 2.74e-  9
##  8     4 absence 1.44e-  4
##  9     1 absent  1.51e-  4
## 10     2 absent  5.95e-  5
## # … with 94,522 more rows
```

Latent Dirichlet Allocation (LDA)

- We can plot the terms with highest probabilities for each topic

```
g <- my_topics %>%
 group_by(topic) %>%
 top_n(10, beta) %>%
 ungroup() %>%
 arrange(topic, -beta) %>%
 mutate(term = reorder_within(term, beta, topic)) %>%
 ggplot(aes(term, beta, fill = factor(topic))) +
 geom_col(show.legend = FALSE) +
 facet_wrap(~ topic, scales = "free") +
 coord_flip() +
 scale_x_reordered() +
 theme(legend.position = "none")
```

Latent Dirichlet Allocation (LDA)



Latent Dirichlet Allocation (LDA)

- It is good practice to remove very frequent words and words occurring very few times since the algorithm cannot learn much from this for the classification purposes
- Defining the number of topics a priori
- Serious validation and interpretation challenges (you will always get an outcome!)
- Additional tools: `LDAvis` package available here

Questions

Computer session

Plan

1. Students are grouped in randomly generated groups
2. Groups will select at least one **text corpus** and develop a script in R that undertake a text mining analysis (data are available on Canvas)
3. Groups will upload the R script and the main findings in Padlet by the end of Week 9 workshop

Groups

Group 1
Adebisi, Jongho, Maria, Keiho

Group 2
Charunan, Poojani, Abdul, Satoshi

Group 3
Oscar, Tsukumo, Jiyoung, Nicholas

Group 4
Alessandro, Shaunna, Jonathan, Rachel

References

References

Blei, David M. 2012. " Probabilistic topic models." *Communications of the ACM* 55 (4): 77–84. https://doi.org/10.1145/2133806.2133826.

Blei, David M, Blei@cs Berkeley Edu, Andrew Y Ng, Ang@cs Stanford Edu, Michael I Jordan, and Jordan@cs Berkeley Edu. 2003. " Latent Dirichlet Allocation." *Journal of Machine Learning Research* 3: 993–1022. https://doi.org/10.1162/jmlr.2003.3.4-5.993.

Bone, Frederique, and Daniele Rotolo. 2020. " Text mining historical sources to trace technological change. The case of mass production." Working Paper.

Callon, M., J. P. Courtial, and F. Laville. 1991. " Co-word analysis as a tool for describing the network of interactions between basic and technological research: The case of polymer chemsitry." *Scientometrics* 22 (1): 155–205. https://doi.org/10.1007/BF02019280.

Ciarli, Tommaso, and Ismael Ràfols. 2019. " The relation between research priorities and societal demands: The case of rice." *Research Policy* 48 (4): 949–67. https://doi.org/https://doi.org/10.1016/j.respol.2018.10.027.

Kwartler, Ted. 2017. *Text Mining in Practice with R*. Chichester, United Kingdom: John Wiley & Sons Ltd. https://doi.org/10.1002/9781119282105.