Natural Language Processing for sensory and consumer scientists. A tidy introduction in R

Ruben Rama

8 Sep, 2024

Housekeeping

Why Are We Here?

The Sensometric Society

EUROSENSE 2024

A Sense of Global Culture

11th Conference on Sensory and Consumer Research

8-11 September 2024

Dublin, Ireland

symrise

Intro

Hello! 

Intro

Hello! 👋

Ruben Rama

Global Sensory and Consumer Insights Data and Knowledge Manager

symrise

Where to find the workshop material?

https://github.com/RubenRama/EuroSense2024_NLP

Preface

What Are We Going to Do?

- Step-by-step guide to help sensory and consumer scientists start their journey into Natural Language Processing (NLP) analysis.
- Natural Language Processing (NLP) is a field of Artificial Intelligence that makes human language intelligible to machines.
- NLP studies the rules and structure of language, and create *intelligent* systems capable of:
  - understanding,
  - analyzing, and
  - extracting meaning from text.

What Are We Going to Do?

- *Part One*: Text Mining and Exploratory Analysis
- *Part Two*: Tidy Sentiment Analysis in R
- *Part Three*: Topic Modelling

Tools 🔧 - R & RStudio

Tools - R & RStudio

R is the underlying statistical computing environment, but using R alone is no fun.

RStudio is a graphical integrated development environment (IDE) that makes using R much easier and more interactive.

Tools - R & RStudio

R and RStudio are separate downloads and installations.

You need to install R before you install RStudio.

Once installed, because RStudio is an IDE, RStudio will run R in the background. You do not need to run it separately.

Tools 🔍 - `tidyverse`

We will be using `tidy` principles.

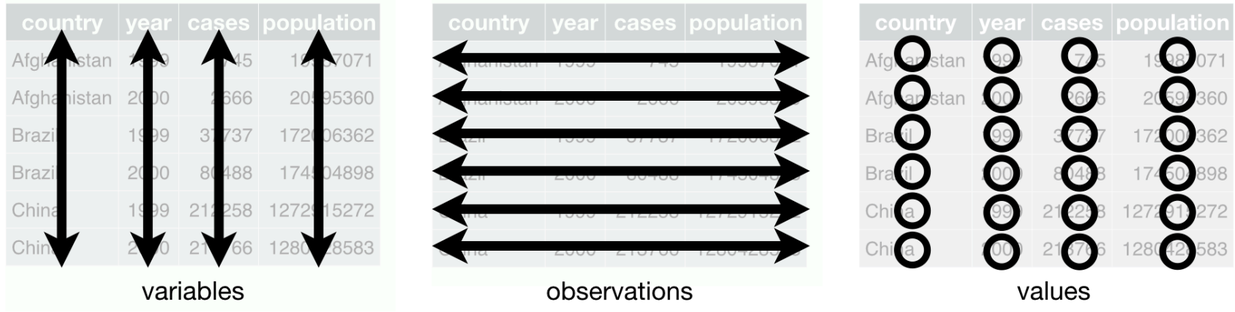
The `tidyverse` is an opinionated collection of R packages designed for data science.

All packages share an underlying design philosophy, grammar, and data structures.

Tools 🔍 - `tidyverse`

`tidy` data has a specific structure:

- Each variable is a column.
- Each observation is a row.
- Each type of observational unit is a table



1

symrise

Tools 🔍 - `tidyverse`

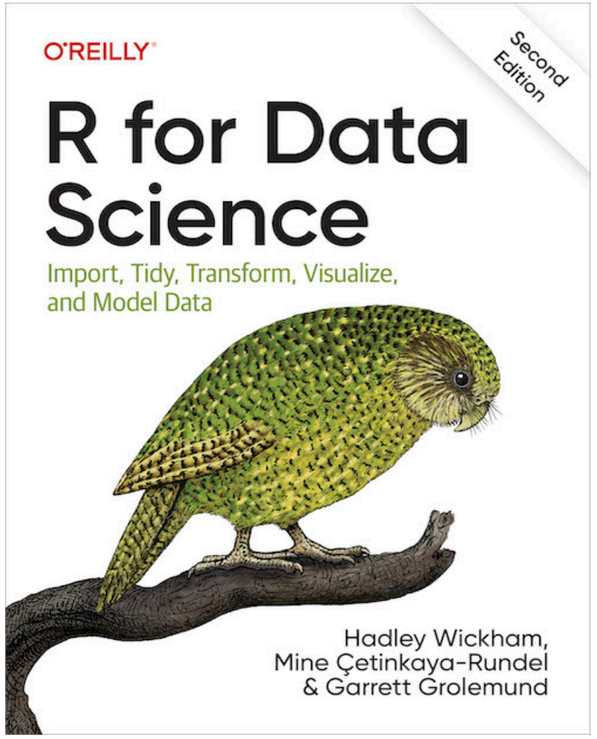
We can install the complete `tidyverse` with:

```
1 install.packages("tidyverse")
```

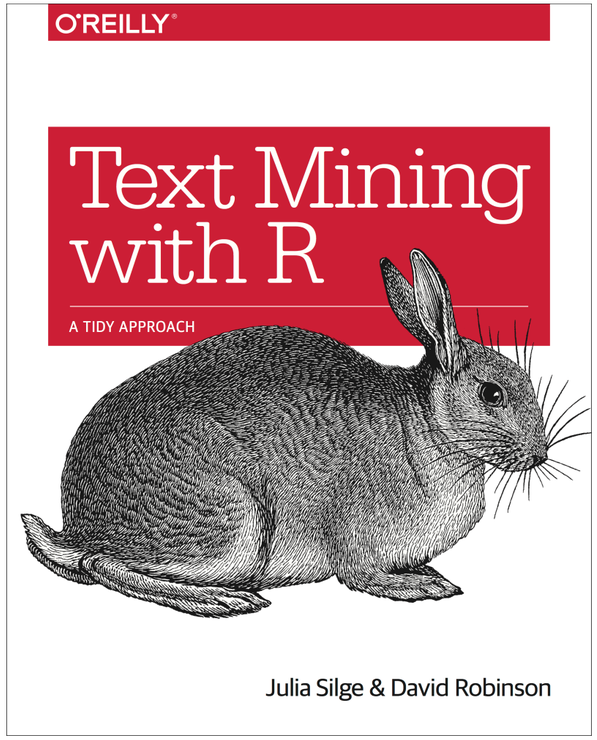
Once installed, we can load it with:

```
1 library(tidyverse)
```

Tools 📖 - `tidyverse`



R for Data Science (Wickham 2023)



Text Mining with R (Silge 2017)

Tools - Basic Piping

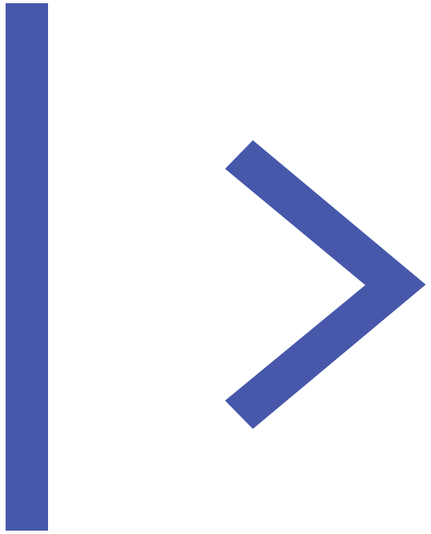
%>%

`magrittr` package

|>

R Native (> 4.1)

Tools - Basic Piping



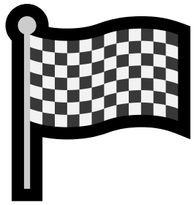
It allows us to link a sequence of analysis steps.

```
function(data, argument(s))

# is equivalent to

data |>
  funtion(argument(s))
```

The pipe operator

- takes the thing that is on the left, and
- places it on the first argument of the function that is on the right!

Prepare your Questions!



Part One: Text Mining and Exploratory Analysis

A Friendly Place

Text Mining and Data Analysis

Data can be organized into three categories:

- **structured** data: predefined and formatted to a tabular format (e.g., an Excel spreadsheet).
- **semi-structured** data: blend between structured and unstructured (e.g., JSON files).
- **unstructured** data: data with no predefined format (e.g., an email).

Text Mining and Data Analysis

Text mining or text analysis is the process of *exploring* and *analyzing* **unstructured** or **semi-structured** text data to identify:

- key concepts,
- patterns,
- relationships or,
- any other attributes of the text.

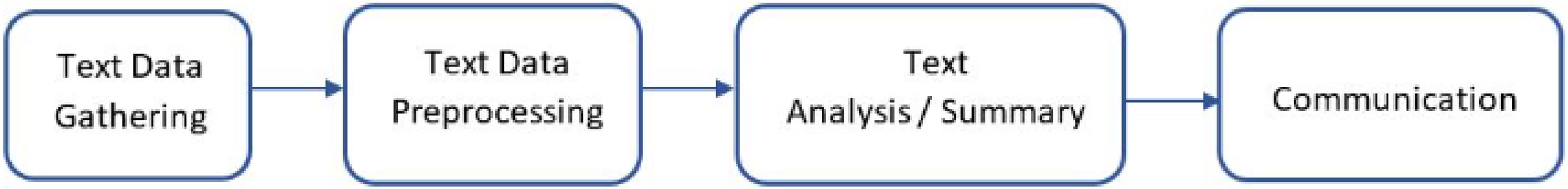
Text Mining and Data Analysis

From a sensory and consumer perspective, text data can come from lots of different sources:

- panel/consumer comments,
- product review,
- interview transcripts,
- MROCS or online discussions,
- digitized text,
- tweets, blogs, social media,
- etc.

Process of Text Mining

A simplistic explanation of a typical text mining can include the following steps:



1. We gather the data, either by creating it or selecting existing datasets.
2. We preprocess or clean the text to get it ready for analysis.
3. We perform text mining or analysis, such as sentiment analysis, topic modelling, etc.
4. We communicate the findings from the text mining.

Importing Review Data

Original data was sourced from Kaggle, from the Amazon Alexa Reviews dataset.

A copy of the data file is available in my github repository.

https://github.com/RubenRama/EuroSense2024_NLP

Importing Review Data

We can use the `read_csv()` function from `readr` (part of the `tidyverse`) to load the data:

```
1 review_data <- read_csv("data/amazon_alexa.csv")
```

Importing Review Data

We can also use the `file.choose()` function.

That will bring up a file explorer window that will allow us to interactively choose the required file:

```
1 review_data <- read_csv(file.choose())
```

Let’s Explore the Data

```
1 review_data
```

```
# A tibble: 3,150 × 5
   stars date      product             review                              feedback
   <dbl> <chr>     <chr>               <chr>                                  <dbl>
 1     5 31-Jul-18 Charcoal Fabric     Love my Echo!                              1
 2     5 31-Jul-18 Charcoal Fabric     Loved it!                                  1
 3     4 31-Jul-18 Walnut Finish       Sometimes while playing a game,…           1
 4     5 31-Jul-18 Charcoal Fabric     I have had a lot of fun with th…           1
 5     5 31-Jul-18 Charcoal Fabric     Music                                      1
 6     5 31-Jul-18 Heather Gray Fabric I received the echo as a gift. …           1
 7     3 31-Jul-18 Sandstone Fabric    Without having a cellphone, I c…           1
 8     5 31-Jul-18 Charcoal Fabric     I think this is the 5th one I'v…           1
 9     5 30-Jul-18 Heather Gray Fabric looks great                                1
10     5 30-Jul-18 Heather Gray Fabric Love it! I’ve listened to songs…           1
# ℹ 3,140 more rows
```

Let’s Explore the Data

We have **3150** reviews.

We may want to remove any duplicated reviews by using the `distinct()` function from `dplyr`:

Traditional approach:

```
distinct(review_data)
```

With `|>`

```
review_data |>
  distinct()
```

Let’s Explore the Data

We have **3150** reviews.

We may want to remove any duplicated reviews by using the `distinct()` function from `dplyr`:

```
review_data <- review_data |>
  distinct()
```

Let’s Explore the Data

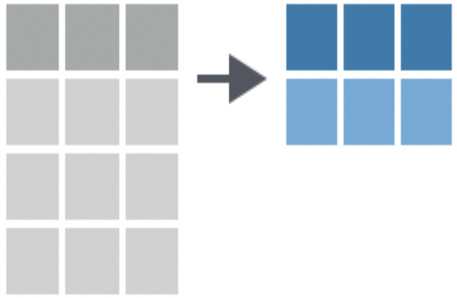
We have **3150** reviews.

We may want to remove any duplicated reviews by using the `distinct()` function from `dplyr`:

```
1  review_data <- review_data |>
2    distinct()
3
4  review_data
```

```
# A tibble: 2,435 × 5
   stars date      product             review                                feedback
   <dbl> <chr>     <chr>               <chr>                                    <dbl>
 1     5 31-Jul-18 Charcoal Fabric     Love my Echo!                                1
 2     5 31-Jul-18 Charcoal Fabric     Loved it!                                    1
 3     4 31-Jul-18 Walnut Finish       Sometimes while playing a game,…             1
 4     5 31-Jul-18 Charcoal Fabric     I have had a lot of fun with th…             1
 5     5 31-Jul-18 Charcoal Fabric     Music                                        1
 6     5 31-Jul-18 Heather Gray Fabric I received the echo as a gift. …             1
 7     3 31-Jul-18 Sandstone Fabric    Without having a cellphone, I c…             1
 8     5 31-Jul-18 Charcoal Fabric     I think this is the 5th one I'v…             1
 9     5 30-Jul-18 Heather Gray Fabric looks great                                  1
10     5 30-Jul-18 Heather Gray Fabric Love it! I’ve listened to songs…             1
# ℹ 2,425 more rows
```

Let’s Explore the Data



Briefly, let’s just focus on one product.

We use `filter()` and `summarise()` (or `summarize()`) functions (from `dplyr`):

Traditional approach:

```
1  df <- filter(review_data,
2               product == "Charcoal Fabric")
3
4  df <- aggregate(df$stars,
5                  by = list(df$product),
6                  FUN = mean)
7
8  df
```

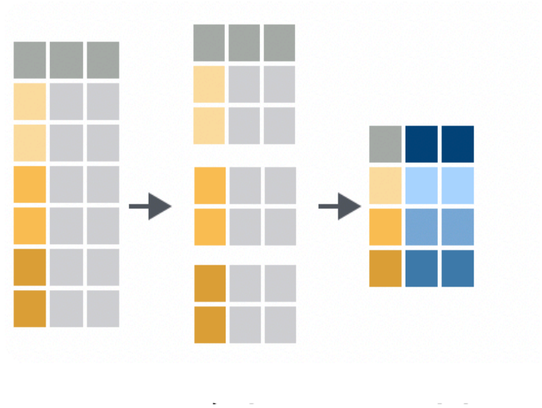
```
          Group.1       x
1 Charcoal Fabric 4.73516
```

with `|>`:

```
1  review_data |>
2    filter(product == "Charcoal Fabric") |>
3    summarise(stars_mean = mean(stars))
```

```
# A tibble: 1 × 1
  stars_mean
       <dbl>
1       4.74
```

Let’s Explore the Data



We may want to group by product and then obtain a summary of the star rating.

We can use `group_by()` and `summarise()` (also from `dplyr`):

```
1  review_data |>
2    group_by(product) |>
3    summarise(stars_mean = mean(stars))
```

```
# A tibble: 16 × 2
   product                          stars_mean
   <chr>                                 <dbl>
 1 Black                                  4.23
 2 Black  Dot                            4.45
 3 Black  Plus                           4.37
 4 Black  Show                           4.48
 5 Black  Spot                           4.31
 6 Charcoal Fabric                       4.74
 7 Configuration: Fire TV Stick          4.59
 8 Heather Gray Fabric                   4.70
 9 Oak Finish                            4.86
10 Sandstone Fabric                      4.36
11 Walnut Finish                         4.8
12 White                                 4.14
```

Let’s Explore the Data

We can `arrange()` the results to show in descending order (yes! also `dplyr`):

```
1  review_data |>
2    group_by(product) |>
3    summarize(stars_mean = mean(stars)) |>
4    arrange(desc(stars_mean))
```

```
# A tibble: 16 × 2
   product                          stars_mean
   <chr>                                 <dbl>
 1 Oak Finish                             4.86
 2 Walnut Finish                          4.8
 3 Charcoal Fabric                        4.74
 4 Heather Gray Fabric                    4.70
 5 Configuration: Fire TV Stick           4.59
 6 Black  Show                            4.48
 7 Black  Dot                             4.45
 8 White  Dot                             4.42
 9 Black  Plus                            4.37
10 White  Plus                            4.36
11 Sandstone Fabric                       4.36
12 White  Spot                            4.34
```

Let’s Explore the Data

But we cannot summarise unstructured or categorical data!

```
review_data |>
  group_by(product) |>
  summarize(review_mean = mean(review))
```

```
# A tibble: 16 × 2
   product                          review_mean
   <chr>                                  <dbl>
 1 Black                                     NA
 2 Black  Dot                                NA
 3 Black  Plus                               NA
 4 Black  Show                               NA
 5 Black  Spot                               NA
 6 Charcoal Fabric                           NA
 7 Configuration: Fire TV Stick              NA
 8 Heather Gray Fabric                       NA
 9 Oak Finish                                NA
10 Sandstone Fabric                          NA
11 Walnut Finish                             NA
12 White                                     NA
```

Counting Categorical Data

If what we want is to understand the number of reviews per product, we can summarize with `n()` after grouping by product.

```
review_data |>
  group_by(product) |>
  summarize(number_rows = n())
```

```
# A tibble: 16 × 2
   product                          number_rows
   <chr>                                  <int>
 1 Black                                    261
 2 Black  Dot                              252
 3 Black  Plus                             270
 4 Black  Show                             260
 5 Black  Spot                             241
 6 Charcoal Fabric                         219
 7 Configuration: Fire TV Stick            342
 8 Heather Gray Fabric                      79
 9 Oak Finish                                7
10 Sandstone Fabric                         45
11 Walnut Finish                             5
12 White                                    91
```

Counting Categorical Data

Alternatively, there is a tidy way to achieve the same by using `count()` (thanks `dplyr`!):

```
1 review_data |>
2   count(product)
```

```
# A tibble: 16 × 2
   product                              n
   <chr>                            <int>
 1 Black                              261
 2 Black  Dot                        252
 3 Black  Plus                       270
 4 Black  Show                       260
 5 Black  Spot                       241
 6 Charcoal Fabric                   219
 7 Configuration: Fire TV Stick      342
 8 Heather Gray Fabric                79
 9 Oak Finish                          7
10 Sandstone Fabric                   45
11 Walnut Finish                       5
12 White                              91
13 White  Dot                         92
```

Counting Categorical Data

Including `sort = TRUE` arranges the results in descending order:

```
1 review_data |>
2   count(product, sort = TRUE)
```

```
# A tibble: 16 × 2
   product                          n
   <chr>                        <int>
 1 Configuration: Fire TV Stick   342
 2 Black  Plus                    270
 3 Black                          261
 4 Black  Show                    260
 5 Black  Dot                     252
 6 Black  Spot                    241
 7 Charcoal Fabric                219
 8 White  Spot                    108
 9 White  Dot                      92
10 White                           91
11 White  Show                     85
12 Heather Gray Fabric             79
```

Cleaning the Text

There are different methods you can use to condition the text data:

- An advanced option would be to convert the data frame to a Corpus and Document Term Matrix using the `tm` text mining package and then use the `tm_map()` function to do the cleaning.
- But for this tutorial, we will be using the `tidyverse` tidy principles, and the `textclean` package.

Cleaning the Text

textclean is a package containing several functions that automate the

- checking,
- cleaning, and
- normalization of text.

Can be installed by:

```
install.packages("textclean")
```

Once installed, it can be loaded as usual:

```
library(textclean)
```

Cleaning the Text

The `check_text()` function performs a thorough analysis of the text, suggesting any pre-processing ought to be done (be ready for a long output!):

```
1  check_text(review_data$review)
```

Cleaning the Text

```
===========
CONTRACTION
===========

The following observations contain contractions:

8, 12, 20, 22, 27, 34, 40, 41, 47, 52...[truncated]...

This issue affected the following text:

8: I think this is the 5th one I've purchased. I'm working on getting one in every room of my house. I really like what
features they offer specifily playing music on all Echos and controlling the lights throughout my house.
...[truncated]...
```

Cleaning the Text

We can see that `textclean` has identified several pre-processing suggestions and solutions:

- **Contractions**: `replace_contraction()` function to replace any contractions with their multi-word forms (i.e., *wasn't* to *was not*, *i'd* to *i would*, etc.)
- **Date**: `replace_date()` with `replacement = ""` to replace any date with a blank character.
- **Time**: `replace_time()` with `replacement = ""` to replace any time with a blank character.
- **Emojis**: `replace_emoji()` to replace any emoji (i.e., 👌) with word equivalents.
- **Emoticons**: `replace_emoticon()` to replace any emoticon (i.e., *;)*) with word equivalents.
- **Hashtags**: `replace_hash()` to replace any *#hashtag* with a blank character.

Cleaning the Text

- **Numbers**: `replace_number()` to replace any number (including coma separated numbers) with a blank character.
- **HTML**: `replace_html()` with `symbol = FALSE` to remove any HTML markup.
- **Incomplete Sentences**: `replace_incomplete()` with `replacement = ""` to replace incomplete sentence end marks (i.e. …).
- **URL**: `replace_url()` with `replacement = ""` to replace any URL with a blank character.
- **Kern**: `replace_kern()` to remove any added manual space (i.e., *The B O M B !* to *The BOMB!*).
- **Internet Slang**: `replace_internet_slang()` to replace the slang with longer word equivalents (i.e., *ASAP* to *as soon as possible*).

Cleaning the Text

Traditionally, we would need to call every function one at a time:

```
review_data$review <- replace_contraction(review_data$review)

review_data$review <- replace_date(review_data$review, replacement = "")

review_data$review <- replace_time(review_data$review, replacement = "")

...
```

Cleaning the Text

The benefit of using the |> is very apparent in situations like this:

```
review_data$review <- review_data$review |>
  replace_date(replacement = "") |>
  replace_time(replacement = "") |>
  replace_email() |>
  replace_emoticon() |>
  replace_number() |>
  replace_html(symbol = FALSE) |>
  replace_incomplete(replacement = "") |>
  replace_url(replacement = "") |>
  replace_kern() |>
  replace_internet_slang() |>
  replace_contraction()
```

Cleaning the Text

In addition, we can use `str_remove_all()` from the `stringr` package (part of the `tidyverse`) to remove all matched patterns from a string:

```
review_data$review <- review_data$review |>
  str_remove_all("&#34;")
```

Using `tidytext`

Once our text has been cleaned, we will be using `tidytext` to preprocess text data.

As before, we can install this package by:

```
1  install.package("tidytext")
```

And we will load it with:

```
1  library(tidytext)
```

Tokenizing Text

- Text mining or text analysis methods are based on counting:
  - words,
  - phrases,
  - sentences, or
  - any other meaningful segment.
- These segments are called **tokens**.

Tokenizing Text

- Therefore, we need to
  - break out the reviews into individual words (or tokens) and
  - begin mining for insights.
- This process is called **tokenization**.

Tokenization with `tidytext`

- From a `tidy text` framework, we need to both
  - break the text into individual tokens (tokenization) and
  - transform it to a `tidy` data structure.
- `tidy text` is defined as a **one-token-per-row** dataframe, where a token can be
  - a character,
  - a word,
  - an n-gram,
  - a sentence,
  - a paragraph,
  - a tweet,
  - etc.

Tokenization with `tidytext`

- We can do this by using `unnest_tokens()` from `tidytext`.
- `unnest_tokens()` requires at least two arguments:
  - the *output column name* that will be created as the text is unnested into it (`word` in our case, for simplicity), and
  - the *input column* that hold the current text (i.e., `review` in our case)

Tokenization with `tidytext`

```
tidy_review <- review_data |>
  unnest_tokens(word, review)
```

Tokenization with `tidytext`

```
1 tidy_review <- review_data |>
2   unnest_tokens(word, review)
3
4 tidy_review
```

```
# A tibble: 65,749 × 5
   stars date      product          feedback word
   <dbl> <chr>     <chr>               <dbl> <chr>
 1     5 31-Jul-18 Charcoal Fabric         1 love
 2     5 31-Jul-18 Charcoal Fabric         1 my
 3     5 31-Jul-18 Charcoal Fabric         1 echo
 4     5 31-Jul-18 Charcoal Fabric         1 loved
 5     5 31-Jul-18 Charcoal Fabric         1 it
 6     4 31-Jul-18 Walnut Finish           1 sometimes
 7     4 31-Jul-18 Walnut Finish           1 while
 8     4 31-Jul-18 Walnut Finish           1 playing
 9     4 31-Jul-18 Walnut Finish           1 a
10     4 31-Jul-18 Walnut Finish           1 game
# ℹ 65,739 more rows
```

Counting words (or tokens)

```
1 tidy_review |>
2   count(word, sort = TRUE)
```

```
# A tibble: 4,263 × 2
   word      n
   <chr> <int>
 1 the    2675
 2 i      2563
 3 to     2241
 4 it     2205
 5 and    1810
 6 a      1224
 7 is     1202
 8 my     1118
 9 for     849
10 love    743
# ℹ 4,253 more rows
```

Removal of stop words

- Stop words are overly common words that may not add any meaning to our results (e.g., *“the”*, *“have”*, *“is”*, *“are”*).
- We want to exclude them from our textual data and our analysis completely.
- There is no single universal list of stop words.
- Nor any agreed upon rules for identifying stop words!
- Luckily, there are several different lists to choose from…

Removal of stop words

We can get a specific stop word lexicon via the `stopwords()` function from the `stopwords` package, in a `tidy` format with one word per row.

We first need to install the package:

```
1 install.package("stopwords")
```

Followed by loading it:

```
1 library(stopwords)
```

Then we can obtain the different sources for stop words:

```
1 stopwords_getsources()
```

```
[1] "snowball"      "stopwords-iso" "misc"          "smart"
[5] "marimo"        "ancient"       "nltk"          "perseus"
```

Removal of stop words

Stop words lists are available in multiple languages too!

```
1 stopwords_getlanguages("snowball")
```

```
 [1] "da" "de" "en" "es" "fi" "fr" "hu" "ir" "it" "nl" "no" "pt" "ro" "ru" "sv"
```

```
1 stopwords_getlanguages("marimo")
```

```
[1] "en"    "de"    "ru"    "ar"    "he"    "zh_tw" "zh_cn" "ko"    "ja"
```

```
1 stopwords_getlanguages("stopwords-iso")
```

```
 [1] "af" "ar" "hy" "eu" "bn" "br" "bg" "ca" "zh" "hr" "cs" "da" "nl" "en" "eo"
[16] "et" "fi" "fr" "gl" "de" "el" "ha" "he" "hi" "hu" "id" "ga" "it" "ja" "ko"
[31] "ku" "la" "lt" "lv" "ms" "mr" "no" "fa" "pl" "pt" "ro" "ru" "sk" "sl" "so"
[46] "st" "es" "sw" "sv" "th" "tl" "tr" "uk" "ur" "vi" "yo" "zu"
```

Removal of stop words

Different word lists contain different words!

```
get_stopwords(language = "en", source = "snowball") |>
  count()
```

```
# A tibble: 1 × 1
      n
  <int>
1   175
```

```
get_stopwords(language = "en", source = "stopwords-iso") |>
  count()
```

```
# A tibble: 1 × 1
      n
  <int>
1  1298
```

```
get_stopwords(language = "en", source = "smart") |>
  count()
```

```
# A tibble: 1 × 1
      n
  <int>
1   571
```

Removal of stop words

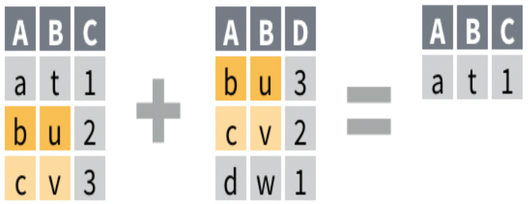
We can sample a random list of these stop words.

By default, from `smart` in English (en).

```
1 head(sample(stop_words$word, 15), 15)
```

```
 [1] "about"      "appear"     "why"        "everything" "did"
 [6] "both"       "everybody"  "ain't"      "her"        "most"
[11] "throughout" "did"        "regarding"  "any"        "shouldn't"
```

Removal of stop words



To remove stops words from our tidy tibble using `tidytext`, we will use a join.

After we tokenize the reviews into words, we can use `anti_join()` to remove stop words.

```
tidy_review <- review_data |>
  unnest_tokens(word, review) |>
  anti_join(stop_words)
```

Removal of stop words

If we want to select another source or another language, we can join using the `get_stopwords()` function directly:

```
tidy_review_clean <- review_data |>
  unnest_tokens(word, review) |>
  anti_join(tidytext::get_stopwords(language = 'es', source = 'stopwords-iso'))
```

Removal of stop words

Notice that `stop_words` already has a `word` column.

```
stop_words
```

```
# A tibble: 1,149 × 2
   word        lexicon
   <chr>       <chr>
 1 a           SMART
 2 a's         SMART
 3 able        SMART
 4 about       SMART
 5 above       SMART
 6 according   SMART
 7 accordingly SMART
 8 across      SMART
 9 actually    SMART
10 after       SMART
# ℹ 1,139 more rows
```

and a new column called `word` was created by the `unest_tokens()` function,

```
review_data |>
  unnest_tokens(word, review)
```

```
# A tibble: 65,749 × 5
   stars date      product          feedback word
   <dbl> <chr>     <chr>               <dbl> <chr>
 1     5 31-Jul-18 Charcoal Fabric         1 love
 2     5 31-Jul-18 Charcoal Fabric         1 my
 3     5 31-Jul-18 Charcoal Fabric         1 echo
 4     5 31-Jul-18 Charcoal Fabric         1 loved
 5     5 31-Jul-18 Charcoal Fabric         1 it
 6     4 31-Jul-18 Walnut Finish           1 sometimes
 7     4 31-Jul-18 Walnut Finish           1 while
 8     4 31-Jul-18 Walnut Finish           1 playing
 9     4 31-Jul-18 Walnut Finish           1 a
10     4 31-Jul-18 Walnut Finish           1 game
# ℹ 65,739 more rows
```

Removal of stop words

so `anti_join()` automatically joins on the column `word`.

```
# A tibble: 65,749 × 5
   stars date      product          feedback word
   <dbl> <chr>     <chr>               <dbl> <chr>
 1     5 31-Jul-18 Charcoal Fabric         1 love
 2     5 31-Jul-18 Charcoal Fabric         1 my
 3     5 31-Jul-18 Charcoal Fabric         1 echo
 4     5 31-Jul-18 Charcoal Fabric         1 loved
 5     5 31-Jul-18 Charcoal Fabric         1 it
 6     4 31-Jul-18 Walnut Finish           1 sometimes
 7     4 31-Jul-18 Walnut Finish           1 while
 8     4 31-Jul-18 Walnut Finish           1 playing
 9     4 31-Jul-18 Walnut Finish           1 a
10     4 31-Jul-18 Walnut Finish           1 game
# ℹ 65,739 more rows
```

```
# A tibble: 22,144 × 5
   stars date      product          feedback word
   <dbl> <chr>     <chr>               <dbl> <chr>
 1     5 31-Jul-18 Charcoal Fabric         1 love
 2     5 31-Jul-18 Charcoal Fabric         1 echo
 3     5 31-Jul-18 Charcoal Fabric         1 loved
 4     4 31-Jul-18 Walnut Finish           1 playing
 5     4 31-Jul-18 Walnut Finish           1 game
 6     4 31-Jul-18 Walnut Finish           1 answer
 7     4 31-Jul-18 Walnut Finish           1 question
 8     4 31-Jul-18 Walnut Finish           1 correctly
 9     4 31-Jul-18 Walnut Finish           1 alexa
10     4 31-Jul-18 Walnut Finish           1 wrong
# ℹ 22,134 more rows
```

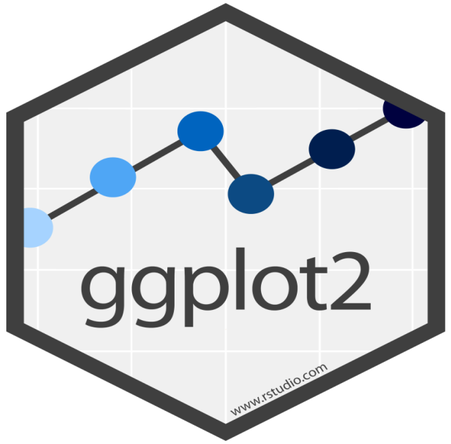
Removal of stop words

Let’s check the result.

```
 1 tidy_review |>
 2   count(word, sort = TRUE)
```

```
# A tibble: 3,728 × 2
   word        n
   <chr>   <int>
 1 love      743
 2 echo      658
 3 alexa     473
 4 music     363
 5 easy      268
 6 sound     237
 7 set       231
 8 amazon    218
 9 dot       211
10 product   205
# ℹ 3,718 more rows
```

Plotting Word Counts



We will use `ggplot2` for the data visualization.

The library is automatically loaded as part of `tidyverse`, but can be called separately:

```
1  library(ggplot2)
```

Plotting Word Counts

Starting with our `tidy text`, we want to create an extra column called `id` to be able to identify the review.

```
tidy_review <- review_data |>
  mutate(id = row_number()) |>
  unnest_tokens(word, review) |>
  anti_join(stop_words)
```

Plotting Word Counts

Starting with our `tidy text`, we want to create an extra column called `id` to be able to identify the review.

```
1 tidy_review <- review_data |>
2   mutate(id = row_number()) |>
3   unnest_tokens(word, review) |>
4   anti_join(stop_words)
5
6 tidy_review
```

```
# A tibble: 22,144 × 6
   stars date      product          feedback    id word
   <dbl> <chr>     <chr>               <dbl> <int> <chr>
 1     5 31-Jul-18 Charcoal Fabric         1     1 love
 2     5 31-Jul-18 Charcoal Fabric         1     1 echo
 3     5 31-Jul-18 Charcoal Fabric         1     2 loved
 4     4 31-Jul-18 Walnut Finish           1     3 playing
 5     4 31-Jul-18 Walnut Finish           1     3 game
 6     4 31-Jul-18 Walnut Finish           1     3 answer
 7     4 31-Jul-18 Walnut Finish           1     3 question
 8     4 31-Jul-18 Walnut Finish           1     3 correctly
 9     4 31-Jul-18 Walnut Finish           1     3 alexa
10     4 31-Jul-18 Walnut Finish           1     3 wrong
# ℹ 22,134 more rows
```

Plotting Word Counts

Visualizing counts with `geom_col()`:

```
word_counts <- tidy_review |>
  count(word, sort = TRUE)
```

Plotting Word Counts

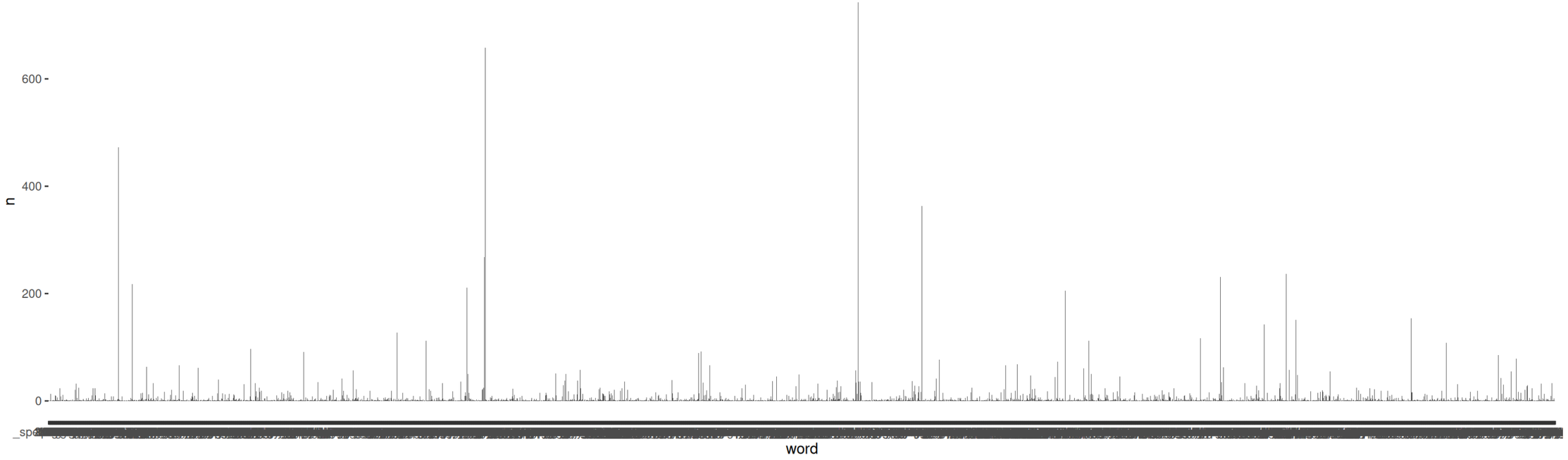
Visualizing counts with `geom_col()`:

```
word_counts <- tidy_review |>
  count(word, sort = TRUE)

word_counts |>
  ggplot(
    aes(
      x = word,
      y = n
    )
  ) +
  geom_col()
```

Plotting Word Counts

We can combine using the pipe |> to make it easier to read and more concise!

```
tidy_review |>
  count(word, sort = TRUE) |>
  ggplot(
    aes(
      x = word,
      y = n)
    ) +
  geom_col()
```



Plotting Word Counts

Too many words? We can `filter()` before visualizing:

```
word_counts_filter <- tidy_review |>
  count(word) |>
  filter(n > 100) |>
  arrange(desc(n))
```

Plotting Word Counts

Too many words? We can `filter()` before visualizing:

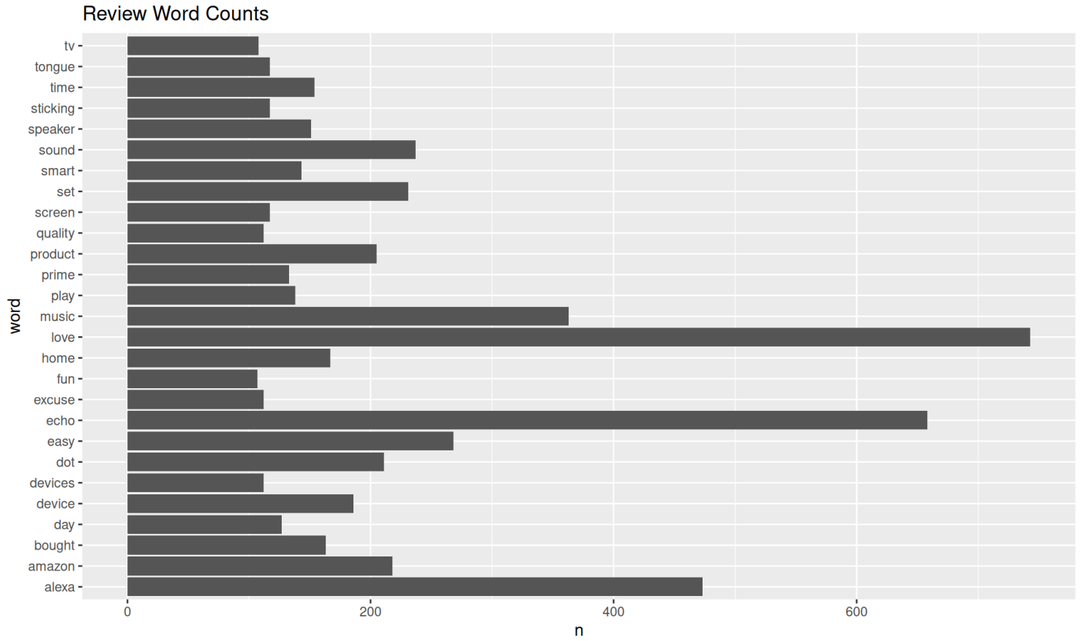
```
word_counts_filter <- tidy_review |>
  count(word) |>
  filter(n > 100) |>
  arrange(desc(n))

word_counts_filter
```

```
# A tibble: 27 × 2
   word        n
   <chr>   <int>
 1 love      743
 2 echo      658
 3 alexa     473
 4 music     363
 5 easy      268
 6 sound     237
 7 set       231
 8 amazon    218
 9 dot       211
10 product   205
# ℹ 17 more rows
```

Plotting Word Counts

We can do a few tweaks to improve the count visualization.

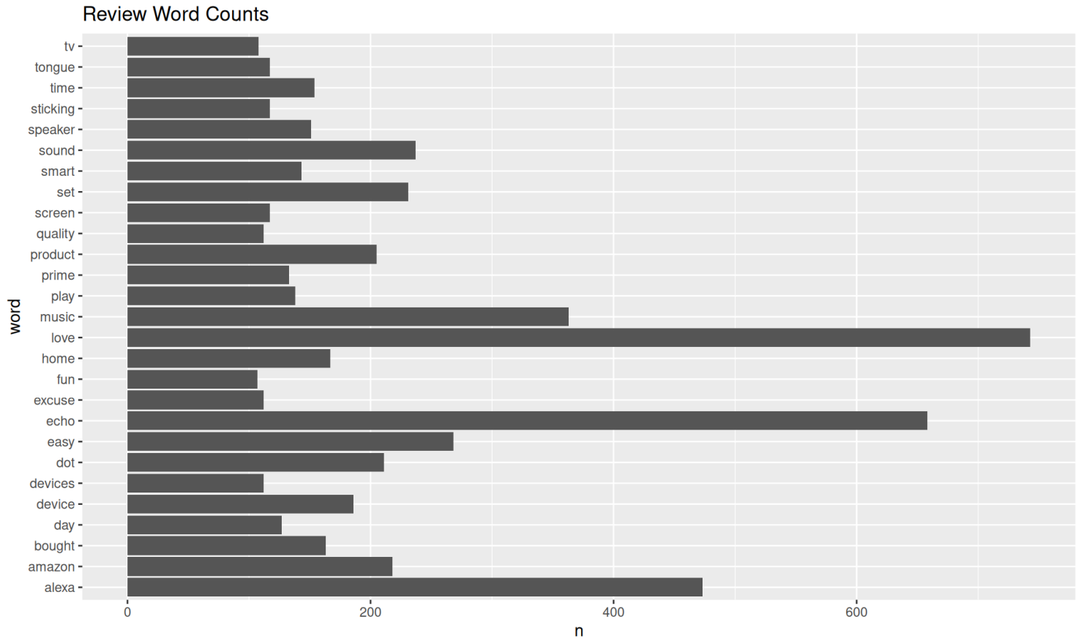
```
word_counts_filter |>
  ggplot(
    aes(
      x = word,
      y = n
    )
  ) +
  geom_col() +
  coord_flip() +
  labs(
    title = "Review Word Counts"
  )
```



Plotting Word Counts

Again, we can pipe everything together using |> to make it more concise:

```
tidy_review |>
  count(word) |>
  filter(n > 100) |>
  arrange(desc(n)) |>
  ggplot(
    aes(
      x = word,
      y = n)
    ) +
  geom_col() +
  coord_flip() +
  labs(
    title = "Review Word Counts"
  )
```



Adding Custom Stop Words

Sometimes, we discover a number of words in the data that aren't informative and should be removed from our final list of words:

```
1 tidy_review |>
2   filter(word == "yr")
```

```
# A tibble: 2 × 6
  stars date      product          feedback    id word
  <dbl> <chr>     <chr>               <dbl> <int> <chr>
1     5 31-Jul-18 Charcoal Fabric         1     4 yr
2     5 30-Jul-18 White  Dot              1  2182 yr
```

We will add a few words to our `custom_stop_words` data frame.

Adding Custom Stop Words

Firstly, let's look at the structure of the `stop_words` data frame:

```
1 stop_words
```

```
# A tibble: 1,149 × 2
   word        lexicon
   <chr>       <chr>
 1 a           SMART
 2 a's         SMART
 3 able        SMART
 4 about       SMART
 5 above       SMART
 6 according   SMART
 7 accordingly SMART
 8 across      SMART
 9 actually    SMART
10 after       SMART
# ℹ 1,139 more rows
```

Adding Custom Stop Words

For that, we can create a custom tibble/data frame called `custom_stop_words`.

The column names of the new data frame of custom stop words should match `stop_words` (i.e., `~word` and `~lexicon`).

```
custom_stop_words <- tribble(
  ~word, ~lexicon,
  "madlibs", "CUSTOM",
  "cd's", "CUSTOM",
  "yr", "CUSTOM"
)
```

Adding Custom Stop Words

For that, we can create a custom tibble/data frame called `custom_stop_words`.

The column names of the new data frame of custom stop words should match `stop_words` (i.e., `~word` and `~lexicon`).

```
1 custom_stop_words <- tribble(
2   ~word, ~lexicon,
3   "madlibs", "CUSTOM",
4   "cd's", "CUSTOM",
5   "yr", "CUSTOM"
6 )
7
8 custom_stop_words
```

```
# A tibble: 3 × 2
  word    lexicon
  <chr>   <chr>
1 madlibs CUSTOM
2 cd's    CUSTOM
3 yr      CUSTOM
```

Adding Custom Stop Words

We can now merge both lists into one that we can use for the analysis by using `bind_rows()`:

```
stop_words_new <- stop_words |>
  bind_rows(custom_stop_words)
```

Adding Custom Stop Words

After that, we can use it with `anti_join()` to remove all the stop words at once!

```
tidy_review <- review_data |>
  unnest_tokens(word, review) |>
  anti_join(stop_words_new)
```

Adding Custom Stop Words

We can combine all the steps together now:

```
tidy_review <- review_data |>
  mutate(id = row_number()) |>
  select(id, date, product, stars, review) |>
  unnest_tokens(word, review) |>
  anti_join(stop_words_new)
```

Adding Custom Stop Words

Let’s check if that word is still there…

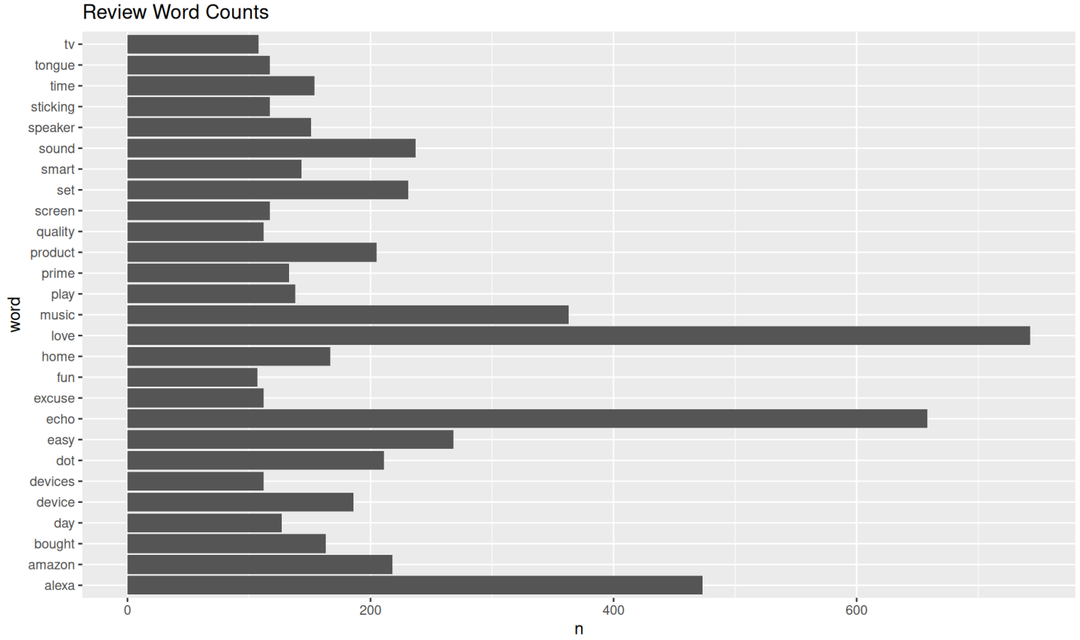
```
1 tidy_review |>
2   filter(word == "yr")
```

```
# A tibble: 0 × 5
# ℹ 5 variables: id <int>, date <chr>, product <chr>, stars <dbl>, word <chr>
```

Back to Plotting Word Counts

We are still able to pipe it all together with |> 

```
review_data |>
  mutate(id = row_number()) |>
  select(id, date, product, stars, review) |>
  unnest_tokens(word, review) |>
  anti_join(stop_words_new) |>
  count(word) |>
  filter(n > 100) |>
  ggplot(
    aes(
      x = word,
      y = n)
    ) +
  geom_col() +
  coord_flip() +
  labs(
    title = "Review Word Counts"
  )
```



(Still) Improving the Count Visualization

To order the different words (i.e., tokens), we can use the `fct_reorder()` function from `forcats`, also part of `tidyverse`:

```
word_counts <- tidy_review |>
  count(word) |>
  filter(n > 100) |>
  mutate(word2 = fct_reorder(word, n))
```

(Still) Improving the Count Visualization

To order the different words (i.e., tokens), we can use the `fct_reorder()` function (or `reorder()`) from `forcats`, also part of `tidyverse`:

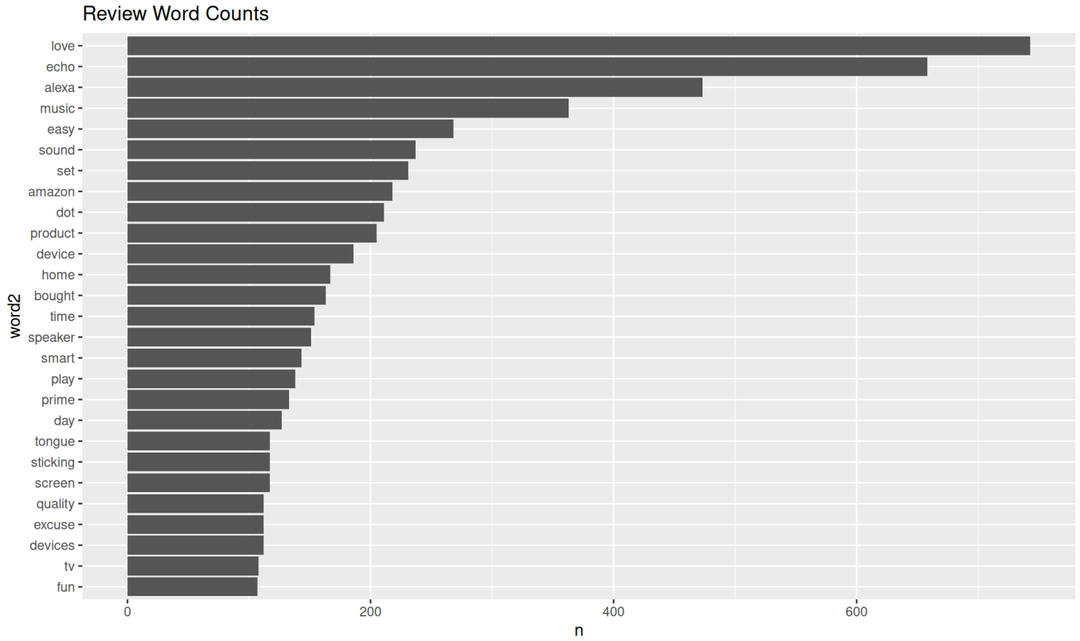
```
word_counts <- tidy_review |>
  count(word) |>
  filter(n > 100) |>
  mutate(word2 = fct_reorder(word, n))

word_counts
```

```
# A tibble: 27 × 3
   word        n word2
   <chr>   <int> <fct>
 1 alexa     473 alexa
 2 amazon    218 amazon
 3 bought    163 bought
 4 day       127 day
 5 device    186 device
 6 devices   112 devices
 7 dot       211 dot
 8 easy      268 easy
 9 echo      658 echo
10 excuse    112 excuse
# ℹ 17 more rows
```

(Still) Improving the Count Visualization

That way, we have a better looking bar plot.

```
word_counts |>
  ggplot(
    aes(
      x = word2,
      y = n
    )
  ) +
  geom_col() +
  coord_flip() +
  labs(
    title = "Review Word Counts"
  )
```



(Still) Improving the Count Visualization

Now, by product!

```
1 tidy_review |>
2   count(word, product, sort = TRUE)
```

```
# A tibble: 9,441 × 3
   word  product                          n
   <chr> <chr>                        <int>
 1 echo  Black  Plus                    128
 2 love  Black  Show                     93
 3 love  Black  Spot                     93
 4 echo  Black  Show                     92
 5 alexa Black  Plus                     89
 6 easy  Configuration: Fire TV Stick    87
 7 love  Configuration: Fire TV Stick    84
 8 love  Black  Dot                      78
 9 love  Black  Plus                     78
10 echo  Black  Spot                     69
# ℹ 9,431 more rows
```

(Still) Improving the Count Visualization

Better to `group_by()`:

```
tidy_review |>
  count(word, product) |>
  group_by(product)
```

```
# A tibble: 9,441 × 3
# Groups:   product [16]
   word  product                 n
   <chr> <chr>               <int>
 1 07    Black  Spot             1
 2 1     Black                   1
 3 1     Black  Plus             2
 4 1     Charcoal Fabric         1
 5 1     White  Show             1
 6 1     White  Spot             1
 7 10    Black                   1
 8 10    Black  Spot             1
 9 10    Heather Gray Fabric     1
10 10    White                   1
# ℹ 9,431 more rows
```

(Still) Improving the Count Visualization

Using `slice_max()` allows us to select the largest values of a variable:

```
1 tidy_review |>
2   count(word, product) |>
3   group_by(product) |>
4   slice_max(n, n = 10)
```

```
# A tibble: 232 × 3
# Groups:   product [16]
   word        product     n
   <chr>       <chr>   <int>
 1 love        Black      62
 2 echo        Black      58
 3 refurbished Black      47
 4 dot         Black      46
 5 alexa       Black      37
 6 bought      Black      31
 7 music       Black      27
 8 amazon      Black      20
 9 product     Black      20
10 time        Black      20
# ℹ 222 more rows
```

(Still) Improving the Count Visualization

We will use `ungroup()` to remove the groups.

Followed by `fct_reorder()`.

```
word_counts <- tidy_review |>
  count(word, product) |>
  group_by(product) |>
  slice_max(n, n = 10) |>
  ungroup() |>
  mutate(word2 = fct_reorder(word, n))
```

(Still) Improving the Count Visualization

We will use `ungroup()` to remove the groups.

Followed by `fct_reorder()`.

```
word_counts <- tidy_review |>
  count(word, product) |>
  group_by(product) |>
  slice_max(n, n = 10) |>
  ungroup() |>
  mutate(word2 = fct_reorder(word, n))

word_counts
```

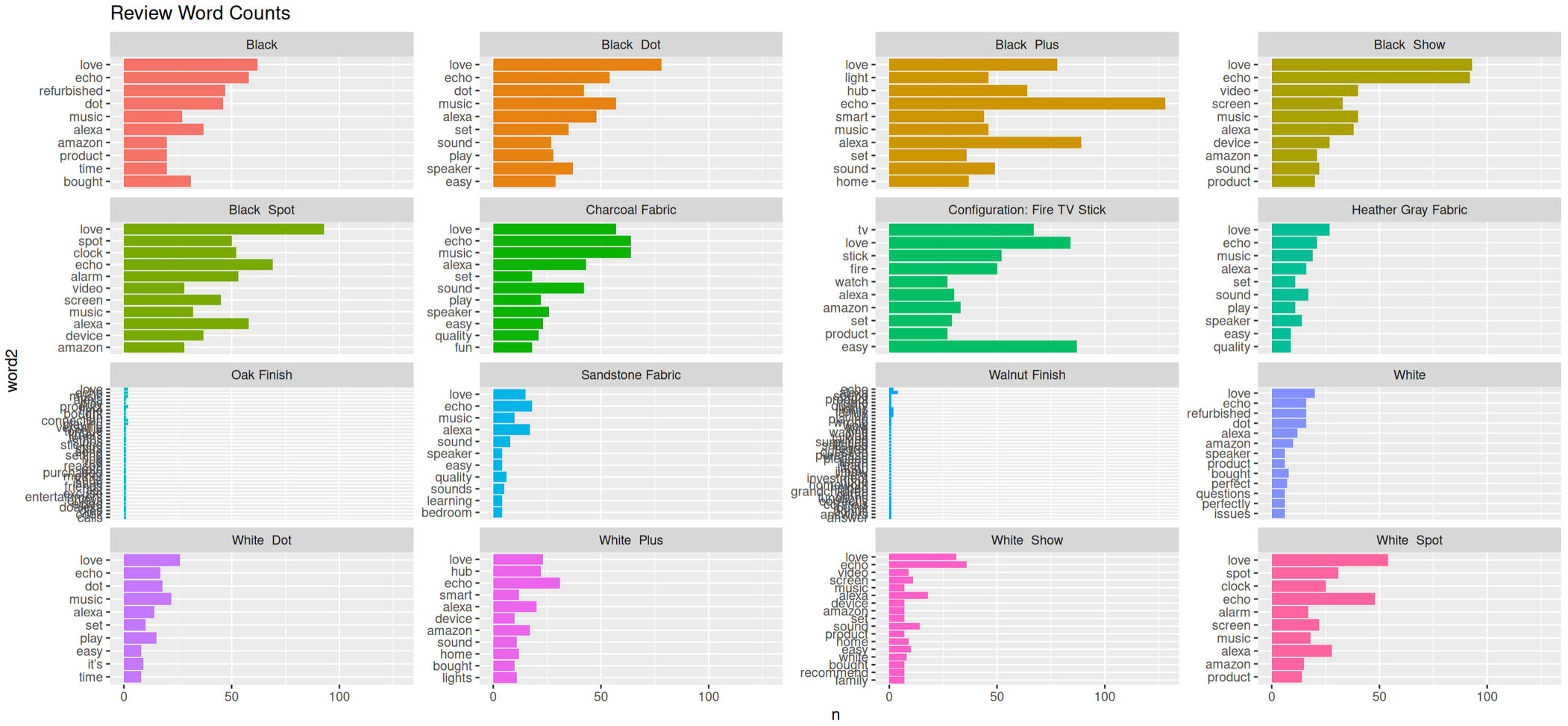
```
# A tibble: 232 × 4
   word        product     n word2
   <chr>       <chr>   <int> <fct>
 1 love        Black      62 love
 2 echo        Black      58 echo
 3 refurbished Black      47 refurbished
 4 dot         Black      46 dot
 5 alexa       Black      37 alexa
 6 bought      Black      31 bought
 7 music       Black      27 music
 8 amazon      Black      20 amazon
 9 product     Black      20 product
10 time        Black      20 time
# ℹ 222 more rows
```

(Still) Improving the Count Visualization

To visualize, we need to use `facet_wrap()`, which allows us to “split” the graph by a determined `facet`:

```
word_counts |>
  ggplot(
    aes(
      x = word2,
      y = n,
      fill = product
    )
  ) +
  geom_col(show.legend = FALSE) +
  facet_wrap(~product, scales = "free_y") +
  coord_flip() +
  labs(
    title = "Review Word Counts"
  )
```

(Still) Improving the Count Visualization



symrise

(Still) Improving the Count Visualization

As explained before, we can |> our way across the code:

```
tidy_review |>
  count(word, product) |>
  group_by(product) |>
  slice_max(n, n = 10) |>
  ungroup() |>
  mutate(word2 = fct_reorder(word, n)) |>
  ggplot(
    aes(
      x = word2,
      y = n,
      fill = product
    )
  ) +
  geom_col(show.legend = FALSE) +
  facet_wrap(~product, scales = "free_y") +
  coord_flip() +
  labs(
    title = "Review Word Counts"
  )
```

Creating Word Clouds

- A `word cloud` is a visual representation of text data.
- It is often used to visualize free form text.
- They are usually composed of single words.
- The importance of each tag is shown with font size or color.

Creating Word Clouds

There are several alternative packages to generate word clouds in R.

For this workshop, we will use the `ggwordcloud` package, as it follows `ggplot2` syntax.

It needs to be installed, following the normal procedure:

```
1 install.packages("ggwordcloud")
```

After that, we need to load it before usage:

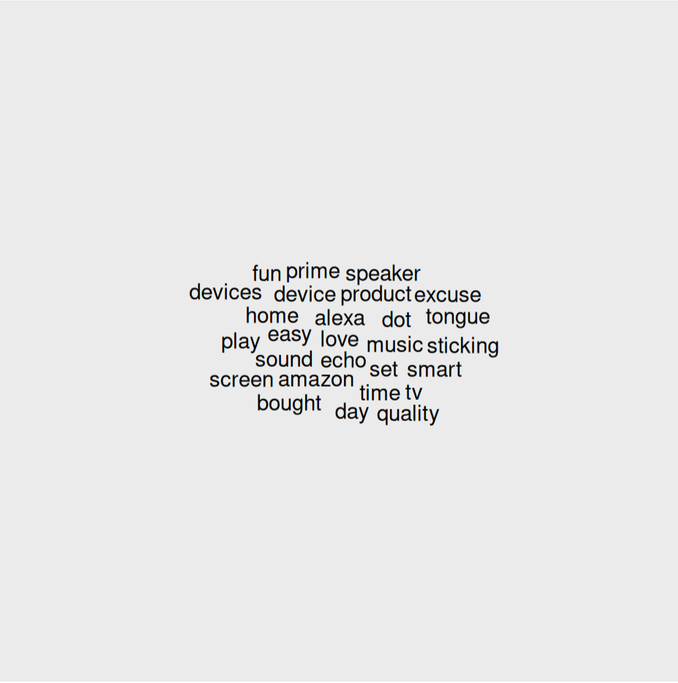
```
1 library(ggwordcloud)
```

Creating Word Clouds

We will use the previously created `word_count_filter` containing the words with more than 100 mentions.

On the most basic level, we only need to use the `geom_text_wordcloud()` function for our `ggplot` plot:

```
set.seed(123)
word_counts_filter |>
  ggplot(
    aes(
      label = word
      )
    ) +
  geom_text_wordcloud()
```



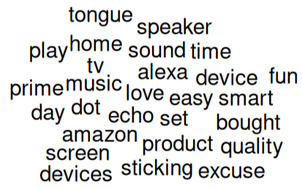
Creating Word Clouds

That seems to create a rather ugly word cloud.

We can improve it by piping a theme (i.e., `theme_minimal()`).

This theme displays the words and nothing else.

```
set.seed(123)
word_counts_filter |>
  ggplot(
    aes(
      label = word
      )
    ) +
  geom_text_wordcloud() +
  theme_minimal()
```



Creating Word Clouds

So far, all the words are the same size.

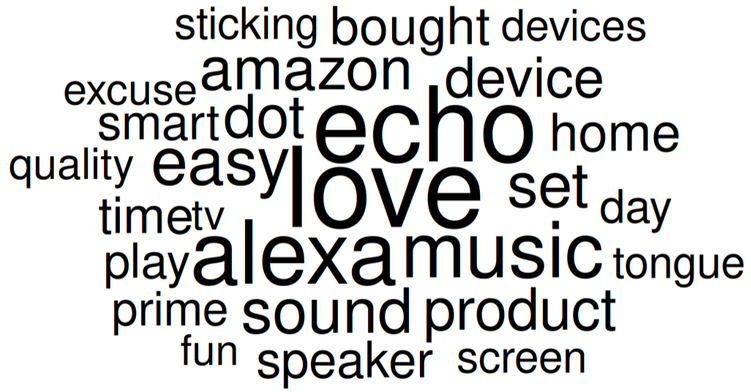
We can introduce the total count of words that we have already calculated as the size.

```
set.seed(123)
word_counts_filter |>
  ggplot(
    aes(
      label = word,
      size = n
      )
    ) +
  geom_text_wordcloud() +
  theme_minimal()
```

Creating Word Clouds

To obtain better proportionality, we need to use `scale_size_area()`:

```
set.seed(123)
word_counts_filter |>
  ggplot(
    aes(
      label = word,
      size = n
      )
     ) +
  geom_text_wordcloud() +
  theme_minimal() +
  scale_size_area(max_size = 20)
```



Creating Word Clouds

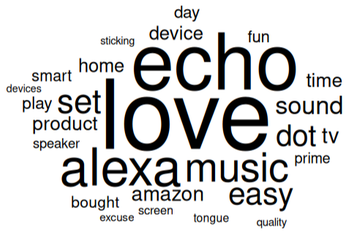
If we want a tighter knitted word cloud with more exagerated sizes, we can use `scale_radius()` instead:

```
set.seed(123)
word_counts_filter |>
  ggplot(
    aes(
      label = word,
      size = n
      )
     ) +
  geom_text_wordcloud() +
  theme_minimal() +
  scale_radius(range = c(0, 30),
               limits = c(0, NA))
```

Creating Word Clouds

`ggwordcloud` also allows us to change the shape of our word cloud, by using `geom_text_wordcloud_area(shape = shape)`:

```
set.seed(123)
word_counts_filter |>
  ggplot(
    aes(
      label = word,
      size = n
      )
     ) +
  geom_text_wordcloud_area(shape = "pentagon") +
  theme_minimal() +
  scale_radius(range = c(0, 30),
               limits = c(0, NA))
```



Creating Word Clouds

Finally, we can apply some colour to our word cloud:

```
set.seed(123)
word_counts_filter |>
  ggplot(
    aes(
      label = word,
      size = n,
      color = n
      )
    ) +
  geom_text_wordcloud() +
  scale_size_area(max_size = 20) +
  theme_minimal() +
  scale_color_gradient2()
```

Creating Word Clouds by Star Review

We can first group by stars using `group_by()` and then `filter()` by the desired rating:

```
set.seed(13)

word_counts_stars <- tidy_review |>
  group_by(stars) |>
  filter(stars == 1) |>
  count(word) |>
  filter(n > 5) |>
  arrange(desc(n))
```

Creating Word Clouds by Star Review

We can first group by stars using `group_by()` and then `filter()` by the desired rating:

```
set.seed(13)

word_counts_stars <- tidy_review |>
  group_by(stars) |>
  filter(stars == 1) |>
  count(word) |>
  filter(n > 5) |>
  arrange(desc(n))

word_counts_stars |>
  ggplot(
    aes(
      label = word,
      size = n,
      color = n
      )
     ) +
  geom_text_wordcloud() +
  scale_size_area(max_size = 20) +
  theme_minimal() +
  scale_color_gradient2()
```

Creating Word Clouds by Product Review

We can first filter by the desired product using `filter()`:

```
set.seed(13)

word_counts_product <- tidy_review |>
  filter(product == "Charcoal Fabric") |>
  count(word) |>
  filter(n > 5) |>
  arrange(desc(n))
```

Creating Word Clouds by Product Review

We can first filter by the desired product using `filter()`:

```
set.seed(13)

word_counts_product <- tidy_review |>
  filter(product == "Charcoal Fabric") |>
  count(word) |>
  filter(n > 5) |>
  arrange(desc(n))

word_counts_product |>
  ggplot(
    aes(
      label = word,
      size = n,
      color = n
      )
     ) +
  geom_text_wordcloud() +
  scale_size_area(max_size = 20) +
  theme_minimal() +
  scale_color_gradient2()
```

Part Two: Tidy Sentiment Analysis in R

Through the looking-glass

Sentiment Analysis Overview

- In the previous chapter, we explored in depth what we mean by the `tidy text` format and showed how this format can be used to approach questions about word frequency.
- This allowed us to analyze which words are used most frequently in documents and to compare documents.
- Let's now address the topic of opinion mining or sentiment analysis.
- When human readers approach a text, we use our understanding of the emotional intent of words to infer whether a section of text is positive 👍 or negative 👎 , or perhaps characterized by some other more nuanced emotion like surprise 😳 or confusion 😕.

Sentiment Analysis - Levels

- As we have previously explored, different levels of analysis based on the text are possible:
  - document,
  - sentence, and
  - word.
- In addition, more complex documents can also have dates, volumes, chapters, etc.

Sentiment Analysis - Levels

- Word level analysis exposes detailed information and can be used as foundational knowledge for more advanced practices in topic modeling.
- Therefore, a way to analyze the sentiment of a text is
  - to consider the text as a combination of its individual words and
  - the sentiment content of the whole text as the sum of the sentiment content of the individual words.
- *This* is an often-used approach, *and* an approach that naturally takes advantage of the `tidy` tool ecosystem.

Sentiment Analysis - Methods

- There are different methods used for sentiment analysis, including:
  - training a known dataset,
  - creating your own classifiers with rules, and
  - using predefined lexical dictionaries (lexicons).
- In this tutorial, you will use the *lexicon-based approach*, but I would encourage you to investigate the other methods as well as their associated trade-offs.

Sentiment Analysis - Dictionaries

- Several distinct dictionaries exist to evaluate the opinion or emotion in text.
- The `tidytext` package provides access to several sentiment lexicons, using the `get_sentiments()` function:
  - **AFINN** from Finn Årup Nielsen,
  - **bing** from Bing Liu and collaborators,
  - **nrc** from Saif Mohammad and Peter Turney, and
  - **loughran** from Loughran-McDonald.

Sentiment Analysis - Dictionaries

- All four of these lexicons are based on *unigrams*, i.e., single words.
- These lexicons contain many English words and the words are assigned scores for positive/negative sentiment, and also possibly emotions like joy, anger, sadness, and so forth.
- Dictionary-based methods find the total sentiment of a piece of text by adding up the individual sentiment scores for each word in the text.

Sentiment Analysis - Dictionaries

- Not every English word is present in the lexicons because many English words are pretty neutral.
- These methods do not take into account qualifiers before a word, such as in “no good” or “not true”; a lexicon-based method like this is based on unigrams only.
- For many kinds of text (like the example in this workshop), there are not sustained sections of sarcasm or negated text, so this is not an important effect.
- Also, we can use a `tidy text` approach to begin to understand what kinds of negation words are important in a given text.

Sentiment Analysis - Dictionaries

- The size of the chunk of text that we use to add up unigram sentiment scores can have an effect on an analysis.
- A text the size of many paragraphs can often have positive and negative sentiment averaged out to about zero, while sentence-sized or paragraph-sized text often works better.
- An example of sentence-based analysis using the `sentimentr` package is included in the Appendix (for those who are impatient!).

AFINN Dictionary

The AFINN lexicon (Nielsen 2011) can be loaded by using the `get_sentiments()` function:

```
1 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
# ℹ 2,467 more rows
```

AFINN Dictionary

The AFINN lexicon (Nielsen 2011) assigns words with a score that runs between -5 and 5, with negative scores indicating negative sentiment and positive scores indicating positive sentiment.

```
get_sentiments("afinn") |>
  summarize(
    min = min(value),
    max = max(value)
  )
```

```
# A tibble: 1 × 2
    min   max
  <dbl> <dbl>
1    -5     5
```

Bing Dictionary

The bing lexicon (Hu and Liu 2004) categorizes words in a binary fashion into “positive” and “negative” categories.

```
1 get_sentiments("bing") |>
2   count(sentiment)
```

```
# A tibble: 2 × 2
  sentiment     n
  <chr>     <int>
1 negative   4781
2 positive   2005
```

nrc Dictionary

The nrc lexicon (Mohammad and Turney 2013) categorizes words in a binary fashion ("yes"/"no") into categories of "positive", "negative", "anger", "anticipation", "disgust", "fear", "joy", "sadness", "surprise", and "trust".

```
get_sentiments("nrc") |>
  count(sentiment, sort = TRUE)
```

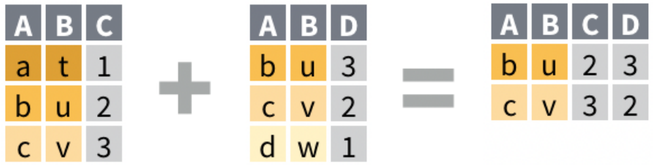
```
# A tibble: 10 × 2
   sentiment        n
   <chr>        <int>
 1 negative      3316
 2 positive      2308
 3 fear          1474
 4 anger         1245
 5 trust         1230
 6 sadness       1187
 7 disgust       1056
 8 anticipation   837
 9 joy            687
10 surprise       532
```

Loughran dictionary

The Loughran lexicon (Loughran and McDonald 2011) was created for use with financial documents, and labels words with six possible sentiments important in financial contexts: “negative”, “positive”, “litigious”, “uncertainty”, “constraining”, or “superfluous”.

```
sentiment_loughran <- get_sentiments("loughran") |>
  count(sentiment) |>
  mutate(sentiment2 = fct_reorder(sentiment, n))

sentiment_loughran |>
  ggplot(
    aes(
      x = sentiment2,
      y = n
    )
  ) +
  geom_col() +
  coord_flip() +
  labs(
    title = "Sentiment Counts in Loughran",
    x = "Counts",
    y = "Sentiment"
  )
```

Using Dictionaries



Dictionaries need to be appended by using `inner_join()`.

The function drops any row in *either* data set that does not have a match in both data sets.

```
1 tidy_review |>
2   inner_join(get_sentiments("nrc"))
```

```
# A tibble: 11,000 × 6
      id date      product         stars word     sentiment
   <int> <chr>     <chr>           <dbl> <chr>    <chr>
 1     1 31-Jul-18 Charcoal Fabric     5 love     joy
 2     1 31-Jul-18 Charcoal Fabric     5 love     positive
 3     3 31-Jul-18 Walnut Finish       4 question positive
 4     3 31-Jul-18 Walnut Finish       4 wrong    negative
 5     4 31-Jul-18 Charcoal Fabric     5 fun      anticipation
 6     4 31-Jul-18 Charcoal Fabric     5 fun      joy
 7     4 31-Jul-18 Charcoal Fabric     5 fun      positive
 8     4 31-Jul-18 Charcoal Fabric     5 music    joy
 9     4 31-Jul-18 Charcoal Fabric     5 music    positive
10     4 31-Jul-18 Charcoal Fabric     5 music    sadness
# ℹ 10,990 more rows
```

Counting Sentiments

After that, we can count the sentiments.

```
tidy_review |>
  inner_join(get_sentiments("nrc")) |>
  count(sentiment)
```

```
# A tibble: 10 × 2
   sentiment        n
   <chr>        <int>
 1 anger          343
 2 anticipation  1275
 3 disgust        209
 4 fear           446
 5 joy           2118
 6 negative       928
 7 positive      3386
 8 sadness        723
 9 surprise       477
10 trust         1095
```

Counting Sentiments

We can also count how many words are linked to which sentiment.

```
tidy_review |>
  inner_join(get_sentiments("nrc")) |>
  count(word, sentiment) |>
  arrange(desc(n))
```

```
# A tibble: 1,500 × 3
   word    sentiment        n
   <chr>   <chr>        <int>
 1 love    joy            743
 2 love    positive       743
 3 music   joy            363
 4 music   positive       363
 5 music   sadness        363
 6 time    anticipation   154
 7 prime   positive       133
 8 excuse  negative       112
 9 fun     anticipation   107
10 fun     joy            107
# ℹ 1,490 more rows
```

Visualizing Sentiments

We will focus only in positive and negative sentiments.

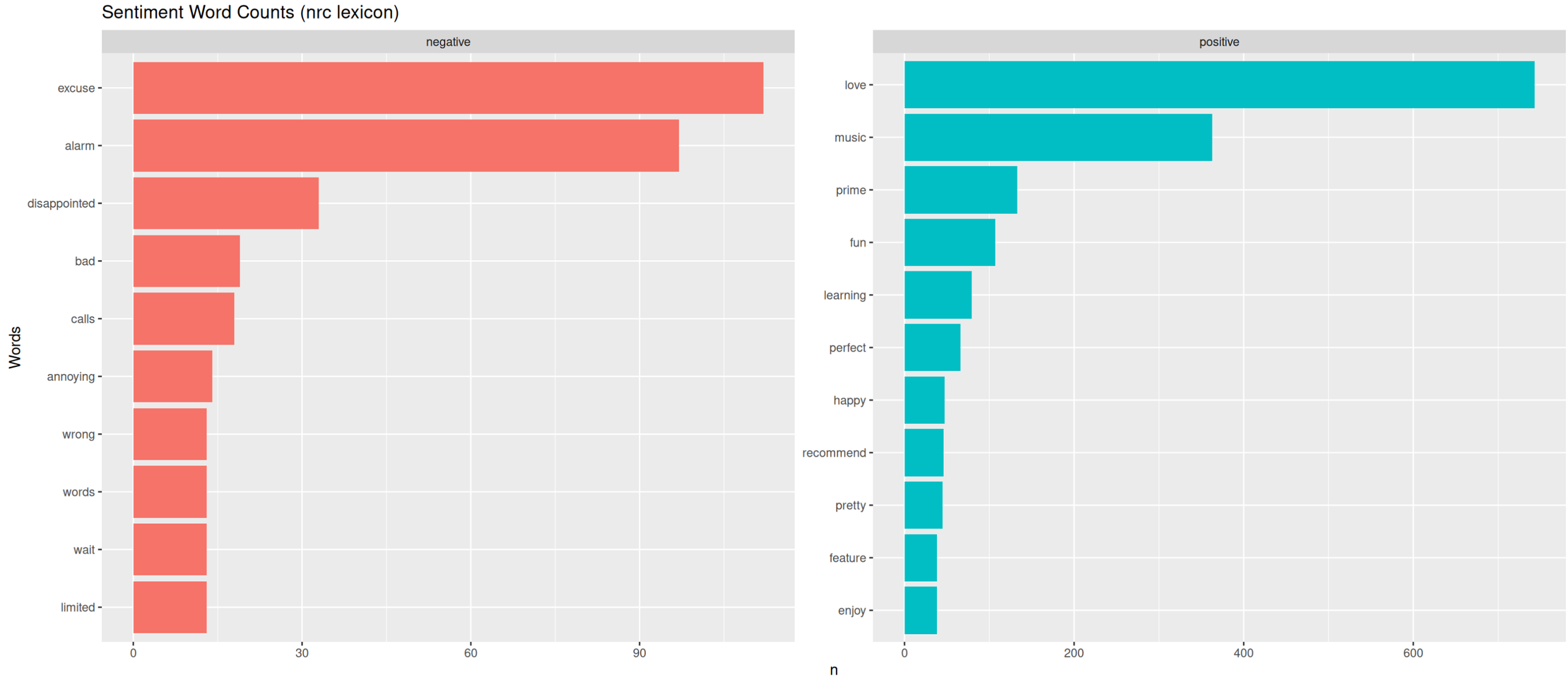
```
sentiment_review_viz <- tidy_review |>
  inner_join(get_sentiments("nrc")) |>
  filter(sentiment %in% c("positive", "negative"))

word_counts <- sentiment_review_viz |>
  count(word, sentiment) |>
  group_by(sentiment) |>
  slice_max(n, n = 10) |>
  ungroup() |>
  mutate(word2 = fct_reorder(word, n))
```

Visualizing Sentiments

We will focus only in positive and negative sentiments.

```
word_counts |>
  ggplot(
    aes(
      x = word2,
      y = n,
      fill = sentiment
      )
    ) +
  geom_col(show.legend = FALSE) +
  facet_wrap(~sentiment, scales = "free") +
  coord_flip() +
  labs(
    title = "Sentiment Word Counts (nrc lexicon)",
    x = "Words"
  )
```

Visualizing Sentiments



Visualizing Sentiments

Of course, we can `tidy` and `|>` all that code 😊

```
tidy_review |>
  inner_join(get_sentiments("nrc")) |>
  filter(sentiment %in% c("positive", "negative")) |>
  count(word, sentiment) |>
  group_by(sentiment) |>
  slice_max(n, n = 10) |>
  ungroup() |>
  mutate(word2 = fct_reorder(word, n)) |>
  ggplot(
    aes(
    x = word2,
    y = n,
    fill = sentiment
    )
   ) +
  geom_col(show.legend = FALSE) +
  facet_wrap(~sentiment, scales = "free") +
  coord_flip() +
  labs(
    title = "Sentiment Word Counts (nrc lexicon)",
    x = "Words"
  )
```

Counting Sentiment by Star Rating

Let’s use the bing lexicon for this experiment.

```
tidy_review |>
  inner_join(get_sentiments("bing")) |>
  count(stars, sentiment)
```

```
# A tibble: 10 × 3
   stars sentiment     n
   <dbl> <chr>     <int>
 1     1 negative    160
 2     1 positive     84
 3     2 negative    114
 4     2 positive     74
 5     3 negative    115
 6     3 positive     91
 7     4 negative    253
 8     4 positive    427
 9     5 negative    461
10     5 positive   2263
```

Counting Sentiment by Star Rating

For a more comfortable exploration, we may want to transpose the results.

That can be achieved with the `pivot_wider()` function (from `tidyr`), which will transform data from long to wide format.

```
tidy_review |>
  inner_join(get_sentiments("bing")) |>
  count(stars, sentiment) |>
  pivot_wider(names_from = sentiment, values_from = n)
```

```
# A tibble: 5 × 3
  stars negative positive
  <dbl>    <int>    <int>
1     1      160       84
2     2      114       74
3     3      115       91
4     4      253      427
5     5      461     2263
```

Computing Overall Sentiment by Star Rating

After that, we can use `mutate()` to create a new column with the overall sentiment rating:

```
tidy_review |>
  inner_join(get_sentiments("bing")) |>
  count(stars, sentiment) |>
  pivot_wider(names_from = sentiment, values_from = n) |>
  mutate(overall_sentiment = positive - negative)
```

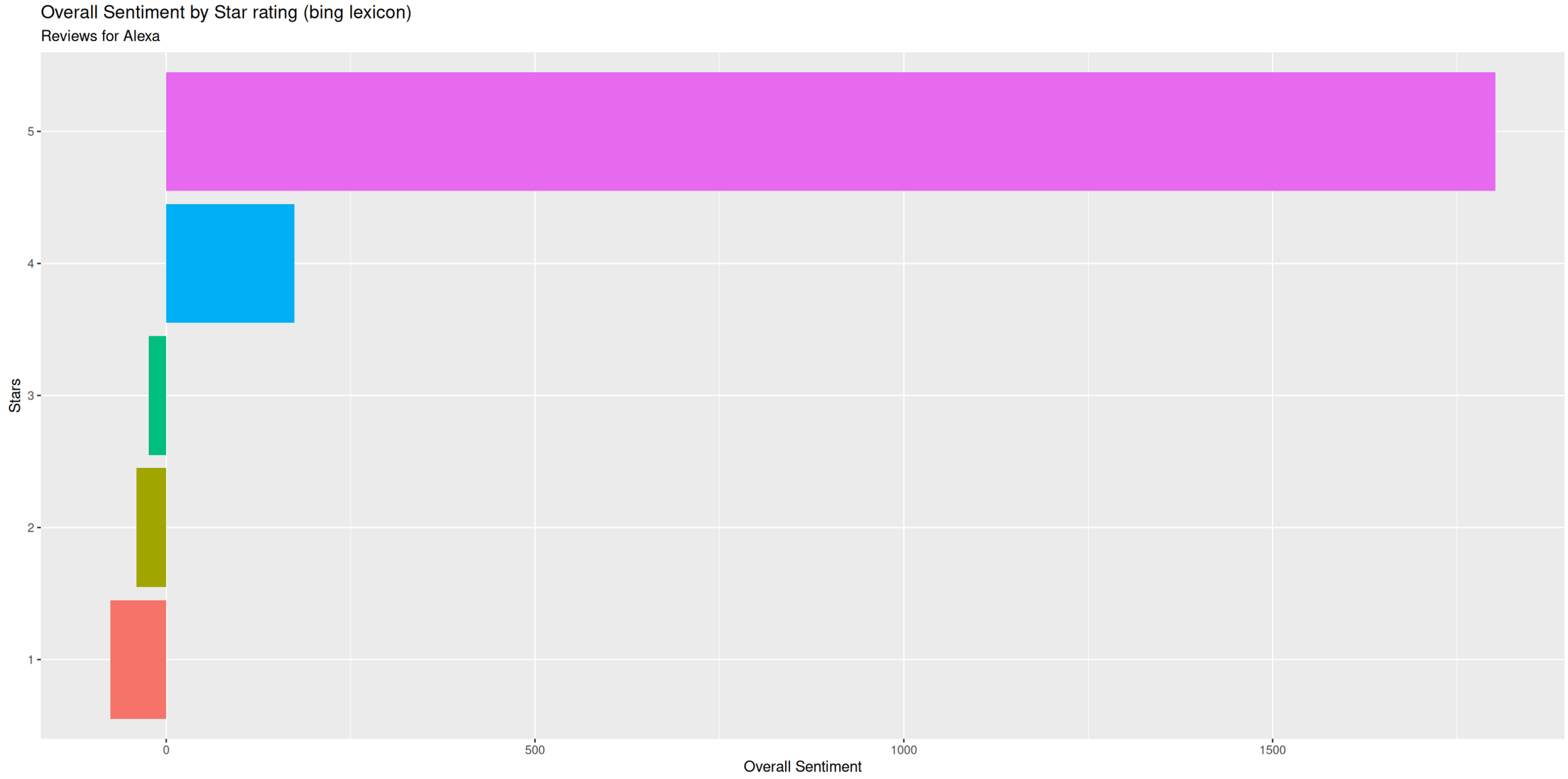
```
# A tibble: 5 × 4
  stars negative positive overall_sentiment
  <dbl>    <int>    <int>             <int>
1     1      160       84               -76
2     2      114       74               -40
3     3      115       91               -24
4     4      253      427               174
5     5      461     2263              1802
```

Visualizing Sentiment by Star Rating

We can put it all together to obtain a visualization 🎉

```
tidy_review |>
  inner_join(get_sentiments("bing")) |>
  count(stars, sentiment) |>
  pivot_wider(names_from = sentiment, values_from = n) |>
  mutate(
    overall_sentiment = positive - negative,
    stars2 = reorder(stars, overall_sentiment)
        ) |>
  ggplot(
      aes(
        x = stars2,
        y = overall_sentiment,
        fill = as.factor(stars)
        )
      ) +
  geom_col(show.legend = FALSE) +
  coord_flip() +
  labs(
    title = "Overall Sentiment by Star rating (bing lexicon)",
    subtitle = "Reviews for Alexa",
    x = "Stars",
    y = "Overall Sentiment"
  )
```

Visualizing Sentiment by Star Rating



Most Common Positive and Negative Words

- One advantage of having the data frame with both sentiment and word is that we can analyze word counts that contribute to each sentiment.
- By implementing `count()` here with arguments of both `word` and `sentiment`, we find out how much each word contributed to each sentiment.

Most Common Positive and Negative Words

```
bing_word_counts <- tidy_review |>
  inner_join(get_sentiments("bing")) |>
  count(word, sentiment, sort = TRUE)
```

Most Common Positive and Negative Words

```r
bing_word_counts <- tidy_review |>
  inner_join(get_sentiments("bing")) |>
  count(word, sentiment, sort = TRUE)

bing_word_counts
```

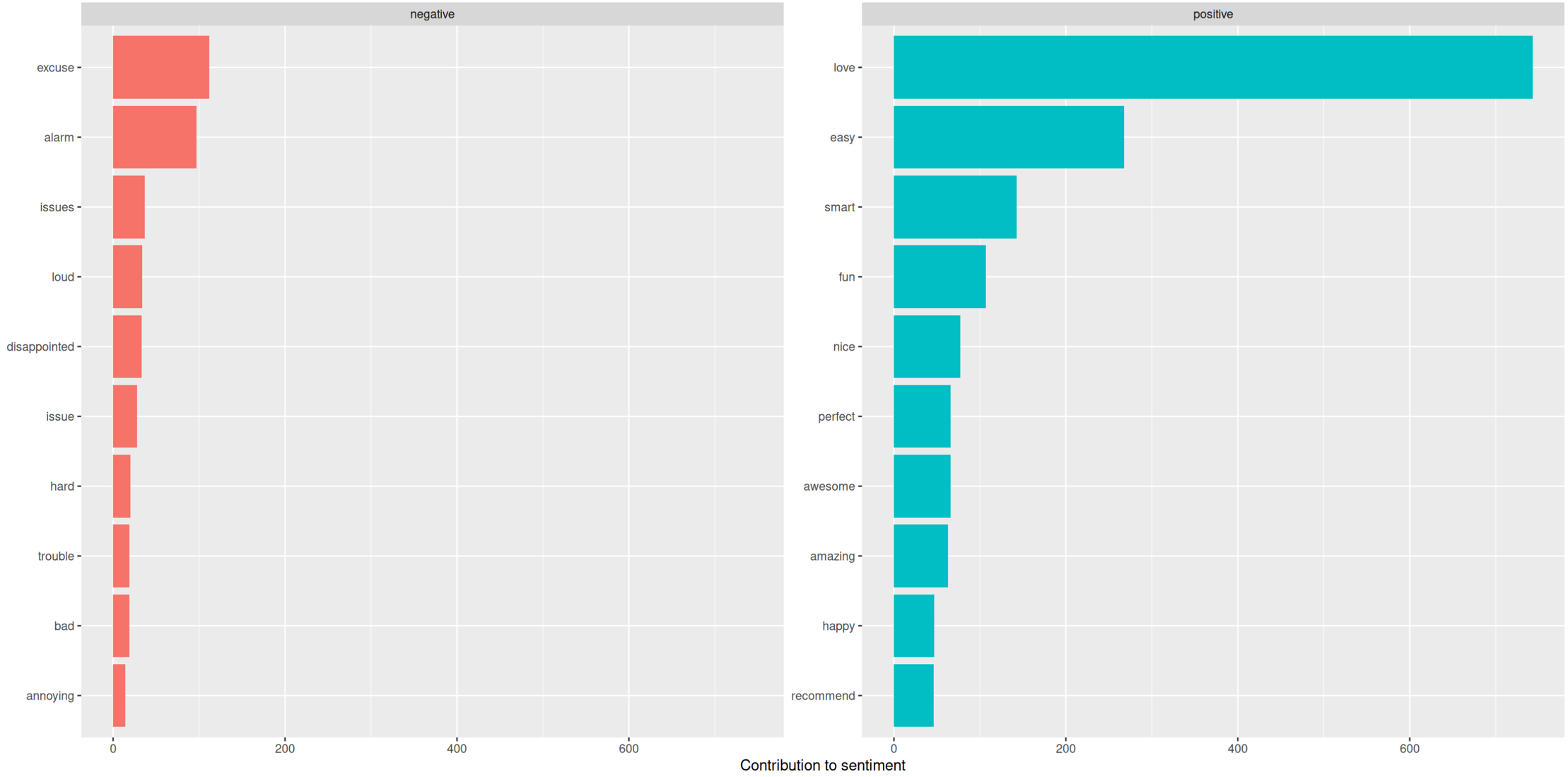
```
# A tibble: 585 × 3
   word    sentiment     n
   <chr>   <chr>     <int>
 1 love    positive    743
 2 easy    positive    268
 3 smart   positive    143
 4 excuse  negative    112
 5 fun     positive    107
 6 alarm   negative     97
 7 nice    positive     77
 8 awesome positive     66
 9 perfect positive     66
10 amazing positive     63
# ℹ 575 more rows
```

Most Common Positive and Negative Words

This can be shown visually, and we can pipe straight into `ggplot2`, if we like, because of the way we are consistently using tools built for handling `tidy` data frames:

```
bing_word_counts |>
  group_by(sentiment) |>
  slice_max(n, n = 10) |>
  ungroup() |>
  mutate(word = reorder(word, n)) |>
  ggplot(
    aes(
      x = n,
      y = word,
      fill = sentiment
      )
    ) +
  geom_col(show.legend = FALSE) +
  facet_wrap(~sentiment, scales = "free_y") +
  labs(
    x = "Contribution to sentiment",
    y = NULL
    )
```

Most Common Positive and Negative Words



Most Common Positive and Negative Words (by Star Rating)

We can do the same, but slicing the data by the star rating gave by the consumers using `group_by()`:

```
bing_word_counts_by_stars <- tidy_review |>
  group_by(stars) |>
  inner_join(get_sentiments("bing")) |>
  count(word, sentiment, sort = TRUE) |>
  ungroup()
```

Most Common Positive and Negative Words (by Star Rating)

We can do the same, but slicing the data by the star rating gave by the consumers using `group_by()`:

```
bing_word_counts_by_stars <- tidy_review |>
  group_by(stars) |>
  inner_join(get_sentiments("bing")) |>
  count(word, sentiment, sort = TRUE) |>
  ungroup()

bing_word_counts_by_stars
```

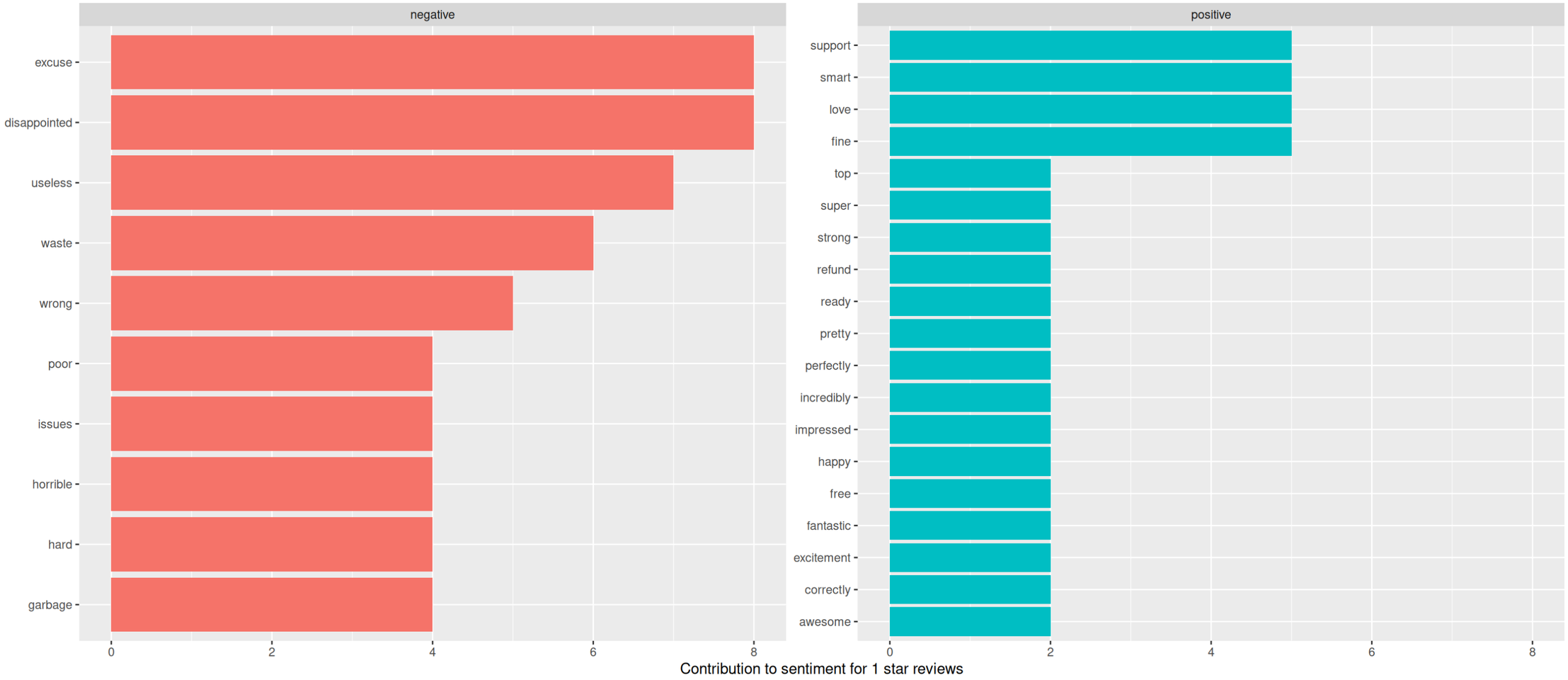
```
# A tibble: 1,001 × 4
   stars word    sentiment     n
   <dbl> <chr>   <chr>     <int>
 1     5 love    positive    658
 2     5 easy    positive    232
 3     5 smart   positive    109
 4     5 fun     positive     86
 5     4 love    positive     72
 6     5 excuse  negative     69
 7     5 amazing positive     57
 8     5 alarm   negative     56
 9     5 awesome positive     56
10     5 perfect positive     56
# ℹ 991 more rows
```

Most Common Positive and Negative Words (by Star Rating)

We can focus on 1 star rating using `filter()`:

```
bing_word_counts_by_stars |>
  filter(stars == 1) |>
  group_by(sentiment) |>
  slice_max(n, n = 10) |>
  ungroup() |>
  mutate(word = reorder(word, n)) |>
  ggplot(
    aes(
      x = n,
      y = word,
      fill = sentiment
      )
    ) +
  geom_col(show.legend = FALSE) +
  facet_wrap(~sentiment, scales = "free_y") +
  labs(
    x = "Contribution to sentiment for 1 star reviews",
    y = NULL
    )
```

Most Common Positive and Negative Words (by Star Rating)

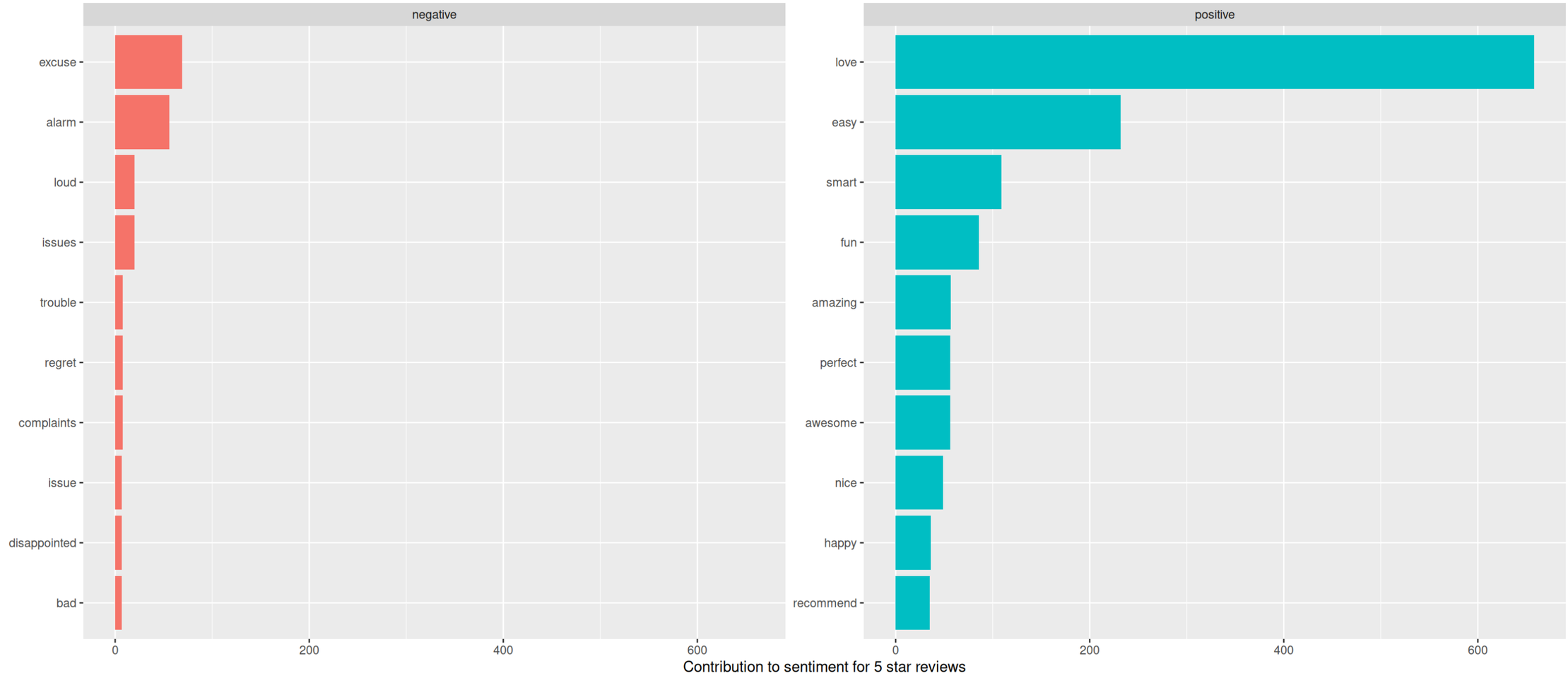


Most Common Positive and Negative Words (by Star Rating)

Let’s see now the 5 star reviews:

```
bing_word_counts_by_stars |>
  filter(stars == 5) |>
  group_by(sentiment) |>
  slice_max(n, n = 10) |>
  ungroup() |>
  mutate(word = reorder(word, n)) |>
  ggplot(
    aes(
      x = n,
      y = word,
      fill = sentiment)) +
  geom_col(show.legend = FALSE) +
  facet_wrap(~sentiment, scales = "free_y") +
  labs(
    x = "Contribution to sentiment for 5 star reviews",
    y = NULL
    )
```

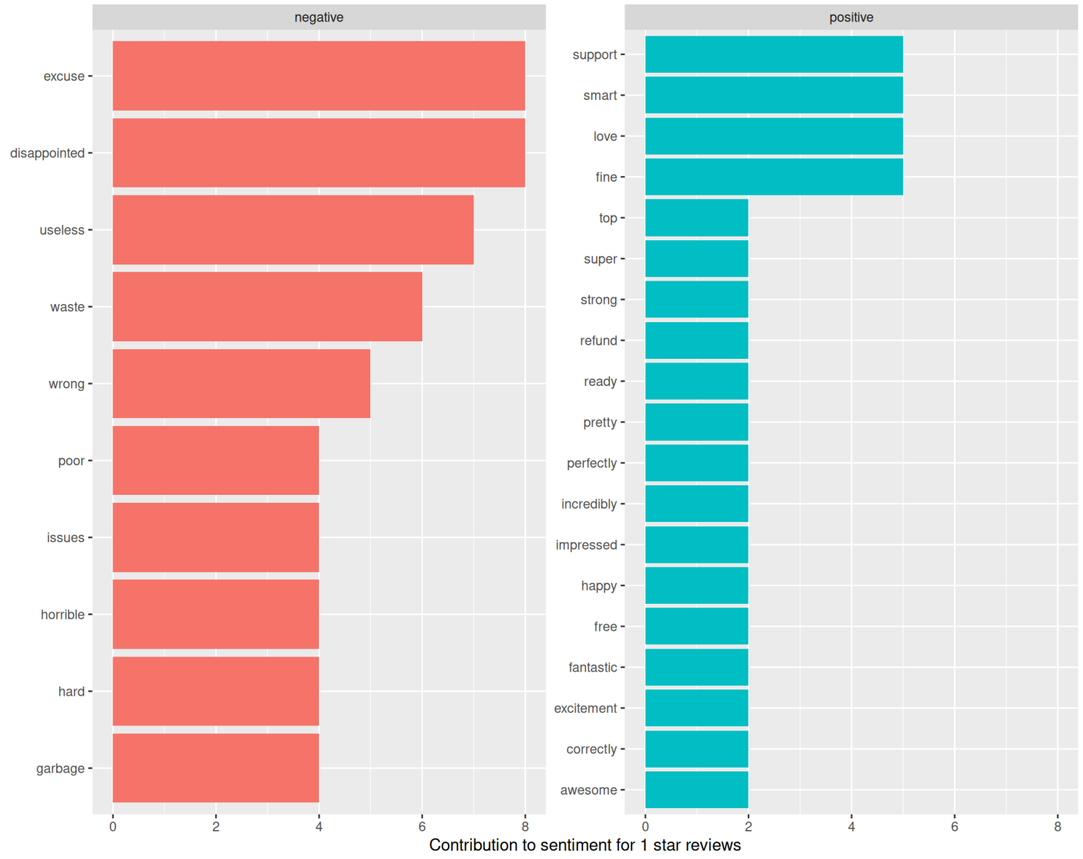
Most Common Positive and Negative Words (by Star Rating)

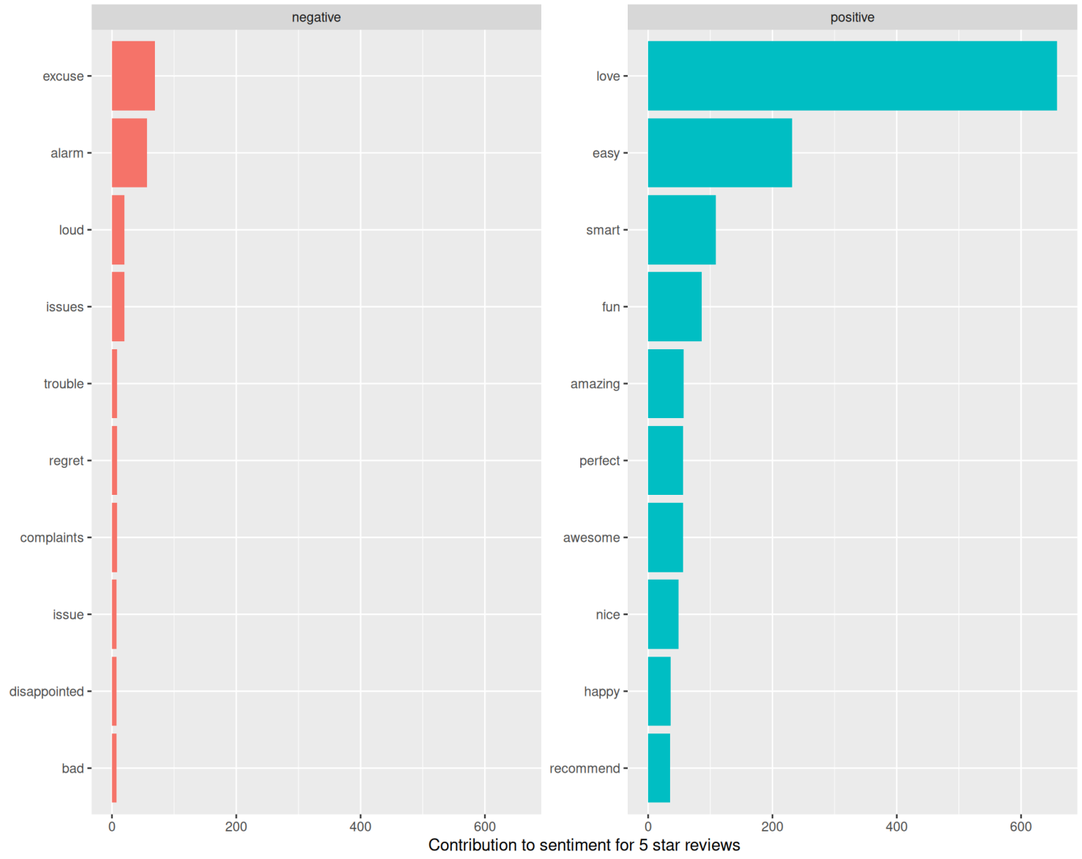


symrise

Most Common Positive and Negative Words (by Star Rating)

Let’s compare side by side





symrise

More Word Clouds!

Sometimes we want to visually present positive and negative words for the same text.

We can use the `comparison.cloud()` function from the `wordcloud` package.

First we need to install the package:

```
1 install.package(wordcloud)
```

After that, we load it as usual:

```
1 library(wordcloud)
```

More Word Clouds!

For `comparison.cloud()`, we may need to turn the data frame into a matrix with the `acast()` function from the `reshape2` package.

The size of a word's text is in proportion to its frequency within its sentiment.

We can see the most important positive and negative words, but the sizes of the words are not comparable across sentiments.

```
library(reshape2)

tidy_review |>
  inner_join(get_sentiments("bing")) |>
  count(word, sentiment, sort = TRUE) |>
  acast(word ~ sentiment, value.var = "n", fill = 0) |>
  comparison.cloud(colors = c("red", "green"),
                   max.words = 100)
```

More Word Clouds!

negative

positive

More Word Clouds! (by Star Rating)

We can do the same as before, focusing on the different star ratings:

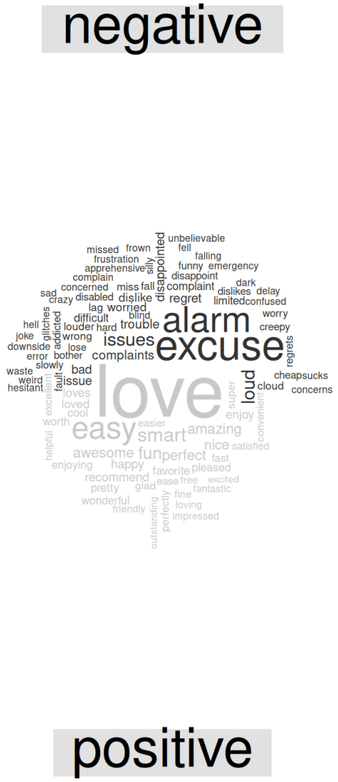
```
library(reshape2)

tidy_review |>
  inner_join(get_sentiments("bing")) |>
  filter(stars == 1) |>
  count(word, sentiment, sort = TRUE) |>
  acast(word ~ sentiment, value.var = "n", fill = 0) |>
  comparison.cloud(colors = c("gray20", "gray80"),
                   max.words = 100)
```

More Word Clouds! (by Star Rating)

More Word Clouds! (by Star Rating)

1 star review

5 star review



symrise

Part Three: Topic Modelling

Into the woods!

Topic Modelling

In this last part, we will build a model using Latent Dirichlet Allocation (LDA), to give a simple example of using LDA to *generate* collections of words that together suggest themes.

Clustering vs. Topic Modelling

- Clustering:
  - Clusters are uncovered based on distance, which is continuous.
  - Every object is assigned to a single cluster.
- Topic Modelling:
  - Topics are uncovered based on word frequency, which is discrete.
  - Every document is a mixture (i.e., partial member) of every topic.

Topic Modelling

- Topic modelling is an unsupervised machine learning approach that can:
  - scan a collection of documents,
  - find word and phrase patterns within them, and
  - automatically *group* word groupings and related expressions into topics.

Topic Modelling with LDA

- Latent Dirichlet Allocation (LDA) is a machine learning algorithm which discovers different topics underlying a collection of documents, where each document is a collection of words.
- LDA makes the following two assumptions:
  1. Every document is a combination of one or more topic(s)
  2. Every topic is a mixture of words

Building Models with LDA

- LDA seeks to find groups of related words.
- It is an iterative, generative algorithm, with two main steps:
  1. During initialization, each word is assigned to a random topic.
  2. The algorithm goes through each word iteratively and reassigns the word to a topic with the following considerations:
     - the probability the word belongs to a topic and,
     - the probability the document can be *generated* by a topic

Creating the Document-Term Matrix

- The LDA algorithm requires the data to be presented as a *document-term matrix (DTM)*.
- Each document is a row, and each column is a term.

	it	is	puppy	cat	pen	a	this
it is a puppy	1	1	1	0	0	1	0
it is a kitten	1	1	0	0	0	1	0
it is a cat	1	1	0	1	0	1	0
that is a dog and this is a pen	0	2	0	0	1	2	1
it is a matrix	1	1	0	0	0	1	0

1

Creating the Document-Term Matrix

We can achieve that by piping (`|>`) our `tidy` data to the `cast_dtm()` function from `tidytext`, where:

- `id` is the name of the field with the document name, and
- `word` is the name of field with the term.

```
1 tidy_review |>
2   count(word, id) |>
3   cast_dtm(id, word, n)
```

```
<<DocumentTermMatrix (documents: 2330, terms: 3725)>>
Non-/sparse entries: 20224/8659026
Sparsity           : 100%
Maximal term length: NA
Weighting          : term frequency (tf)
```

This tells us how many documents and terms we have, and that this is a very sparse matrix.

The word sparse implies that the DTM contains mostly empty fields.

Exploring the Document-Term Matrix

We can look into the contents of a few rows and columns of the DTM by piping it into the `as.matrix()` function.

You will see that each row is a review, and each column is a term.

```
dtm_review <- tidy_review |>
  count(word, id) |>
  cast_dtm(id, word, n) |>
  as.matrix()
```

Exploring the Document-Term Matrix

We can look into the contents of a few rows and columns of the DTM by piping it into the `as.matrix()` function.

You will see that each row is a review, and each column is a term.

```
1 dtm_review <- tidy_review |>
2   count(word, id) |>
3   cast_dtm(id, word, n) |>
4   as.matrix()
5
6 dtm_review[1:4, 2000:2004]
```

```
     Terms
Docs  louis lov love loved lovee
  946     0   0    0     0     0
  90      0   0    0     0     0
  369     0   0    0     0     0
  864     0   0    0     0     0
```

Fitting the Model

We will use the `LDA()` function from the `topicmodels` package.

First we need to install the package:

```
1 install.packages("topicmodels")
```

And load the package as usual:

```
1 library(topicmodels)
```

Fitting the Model

- For our purposes, we will just need to know three parameters for the `LDA()` function:
  - the number of topics `k` (let's start with two, so `k = 2`),
  - the sampling method (`method =`), and
  - the seed (for repeatable results, so `seed = 123`).

Fitting the Model

- The `method` parameter defines the sampling algorithm to use.
- The default is `method = "VEM"`.
- We will use the `method = "Gibbs"` sampling method (it does perform better in my experience).
- An explanation of `VEM` or `Gibbs` methods is beyond this workshop, but I encourage everyone to read a bit more about these two methods.

Fitting the Model

Let’s fit the LDA model and explore the output:

```
lda_tidy <- dtm_review |>
  LDA(
    k = 2,
    method = "Gibbs",
    control = list(seed = 123)
    )
```

Fitting the Model

Let’s fit the LDA model and explore the output:

```
lda_tidy <- dtm_review |>
  LDA(
    k = 2,
    method = "Gibbs",
    control = list(seed = 123)
    )

lda_tidy
```

```
A LDA_Gibbs topic model with 2 topics.
```

Exploring the `LDA()` output

If you REALLY want more details, we can use the `glimpse()` function:

```
1 glimpse(lda_tidy)
```

```
Formal class 'LDA_Gibbs' [package "topicmodels"] with 16 slots
  ..@ seedwords      : NULL
  ..@ z              : int [1:22139] 2 2 1 1 1 1 1 1 2 2 ...
  ..@ alpha          : num 25
  ..@ call           : language LDA(x = dtm_review, k = 2, method = "Gibbs", control = list(seed = 123))
  ..@ Dim            : int [1:2] 2330 3725
  ..@ control        :Formal class 'LDA_Gibbscontrol' [package "topicmodels"] with 14 slots
  ..@ k              : int 2
  ..@ terms          : chr [1:3725] "07" "1" "10" "10.00" ...
  ..@ documents      : chr [1:2330] "946" "90" "369" "864" ...
  ..@ beta           : num [1:2, 1:3725] -11.65 -9.25 -7.54 -11.64 -11.65 ...
  ..@ gamma          : num [1:2330, 1:2] 0.516 0.517 0.475 0.398 0.5 ...
  ..@ wordassignments:List of 5
  .. ..$ i   : int [1:20224] 1 1 1 1 1 1 1 1 1 1 ...
  .. ..$ j   : int [1:20224] 1 12 23 144 774 1669 1794 2119 2469 2630 ...
```

Exploring the `LDA()` output

Most easily, we can use the `tidy()` function with the `matrix = "beta"` argument to put it into a format that is easy to understand.

Passing `beta` provides us with the *per-topic-per-word* probabilities from the model:

```
lda_tidy |>
  tidy(matrix = "beta") |>
  arrange(desc(beta))
```

```
# A tibble: 7,450 × 3
   topic term     beta
   <int> <chr>   <dbl>
 1     2 love   0.0651
 2     1 echo   0.0574
 3     2 alexa  0.0403
 4     2 music  0.0318
 5     2 easy   0.0235
 6     1 sound  0.0207
 7     2 set    0.0203
 8     1 amazon 0.0190
 9     1 dot    0.0184
10     2 device 0.0163
# ℹ 7,440 more rows
```

Interpreting Topics

To understand the model clearly, we need to see what terms are in each topic.

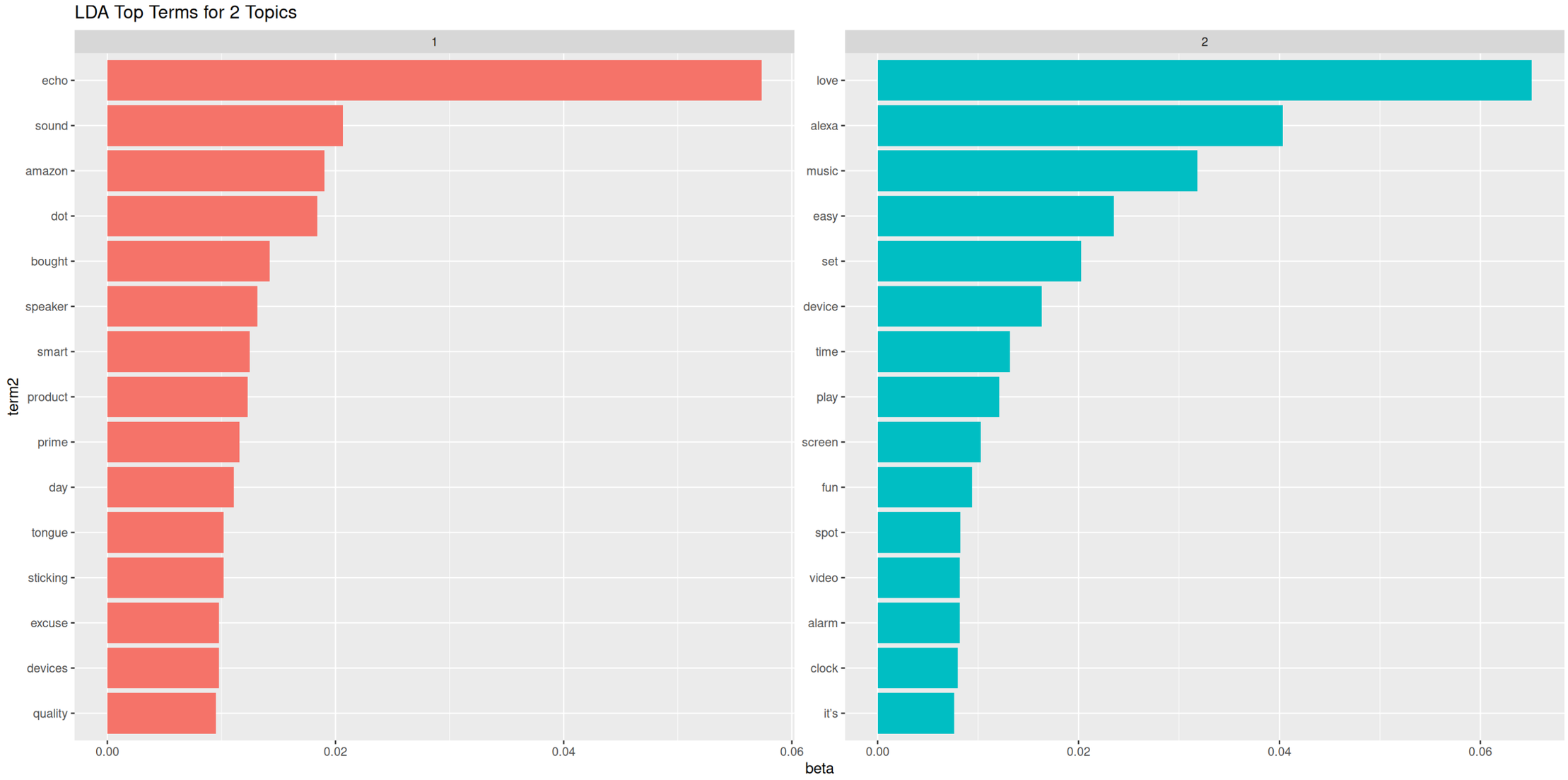
Starting with two topics:

```
lda_2_topics <- dtm_review |>
  LDA(
    k = 2,
    method = "Gibbs",
    control = list(seed = 123)
    ) |>
  tidy(matrix = "beta")

word_2_probs <- lda_2_topics |>
  group_by(topic) |>
  slice_max(beta, n = 15) |>
  ungroup() |>
  mutate(term2 = fct_reorder(term, beta))
```

Interpreting Topics - Two Topics

```
word_2_probs |>
  ggplot(
    aes(
      x = term2,
      y = beta,
      fill = as.factor(topic)
      )
  ) +
  geom_col(show.legend = FALSE) +
  facet_wrap(~topic, scales = "free") +
  coord_flip() +
  labs(
    title = "LDA Top Terms for 2 Topics"
  )
```

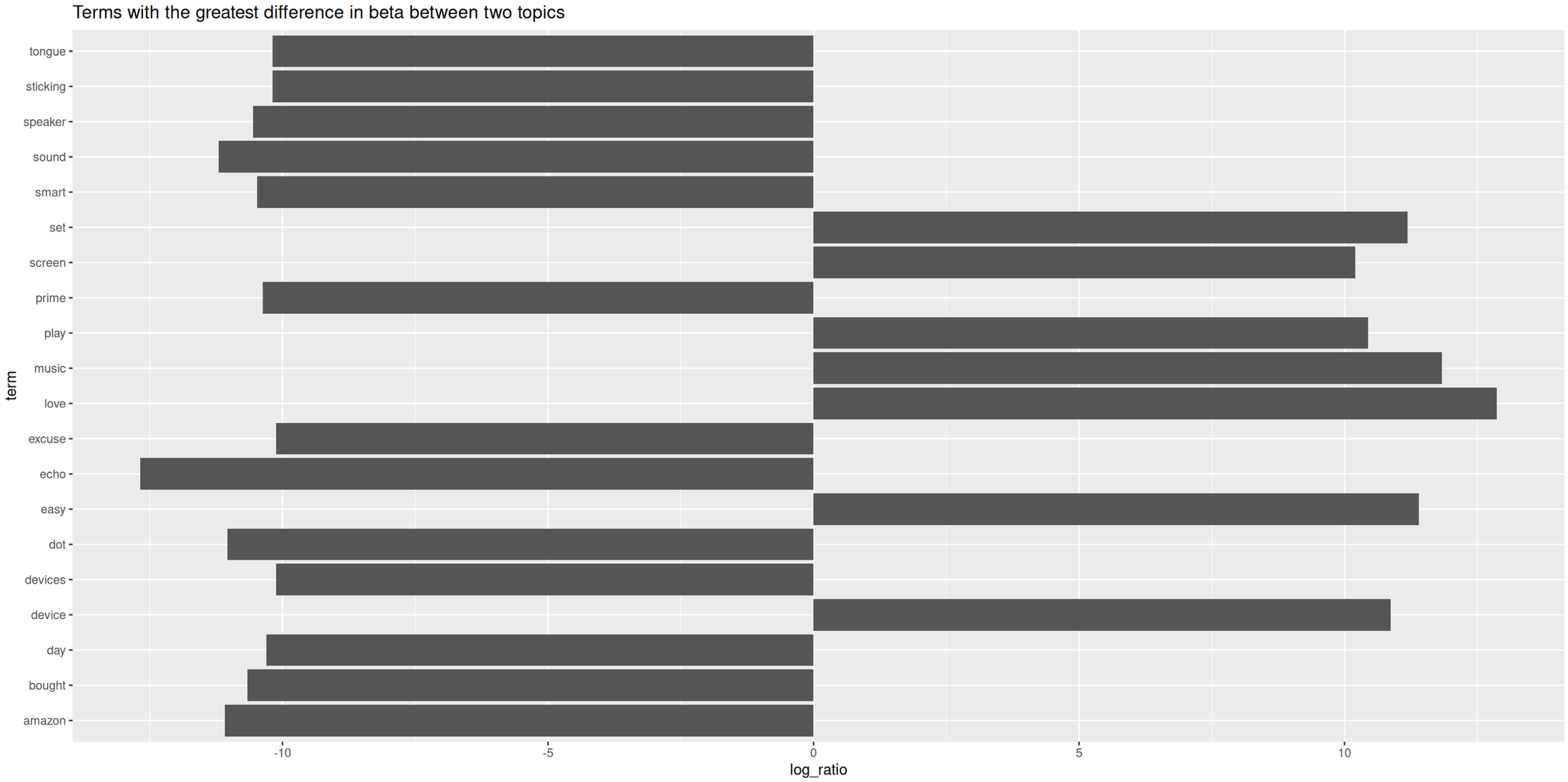
Interpreting Topics - Two Topics



Finding the Terms Generating the Greatest Differences

```r
beta_wide <- lda_2_topics |>
  mutate(topic = paste0("topic", topic)) |>
  pivot_wider(names_from = topic, values_from = beta) |>
  filter(topic1 > .001 | topic2 > .001) |>
  mutate(log_ratio = log2(topic2 / topic1))

beta_wide |>
  arrange(desc(abs(log_ratio))) |>
  head(20) |>
  arrange(desc(log_ratio)) |>
  ggplot(
    aes(
      x = log_ratio,
      y = term
      )
    ) +
  geom_col(show.legend = FALSE) +
  labs(
    title = "Terms with the greatest difference in beta between two topics"
  )
```

Finding the Terms Generating the Greatest Differences



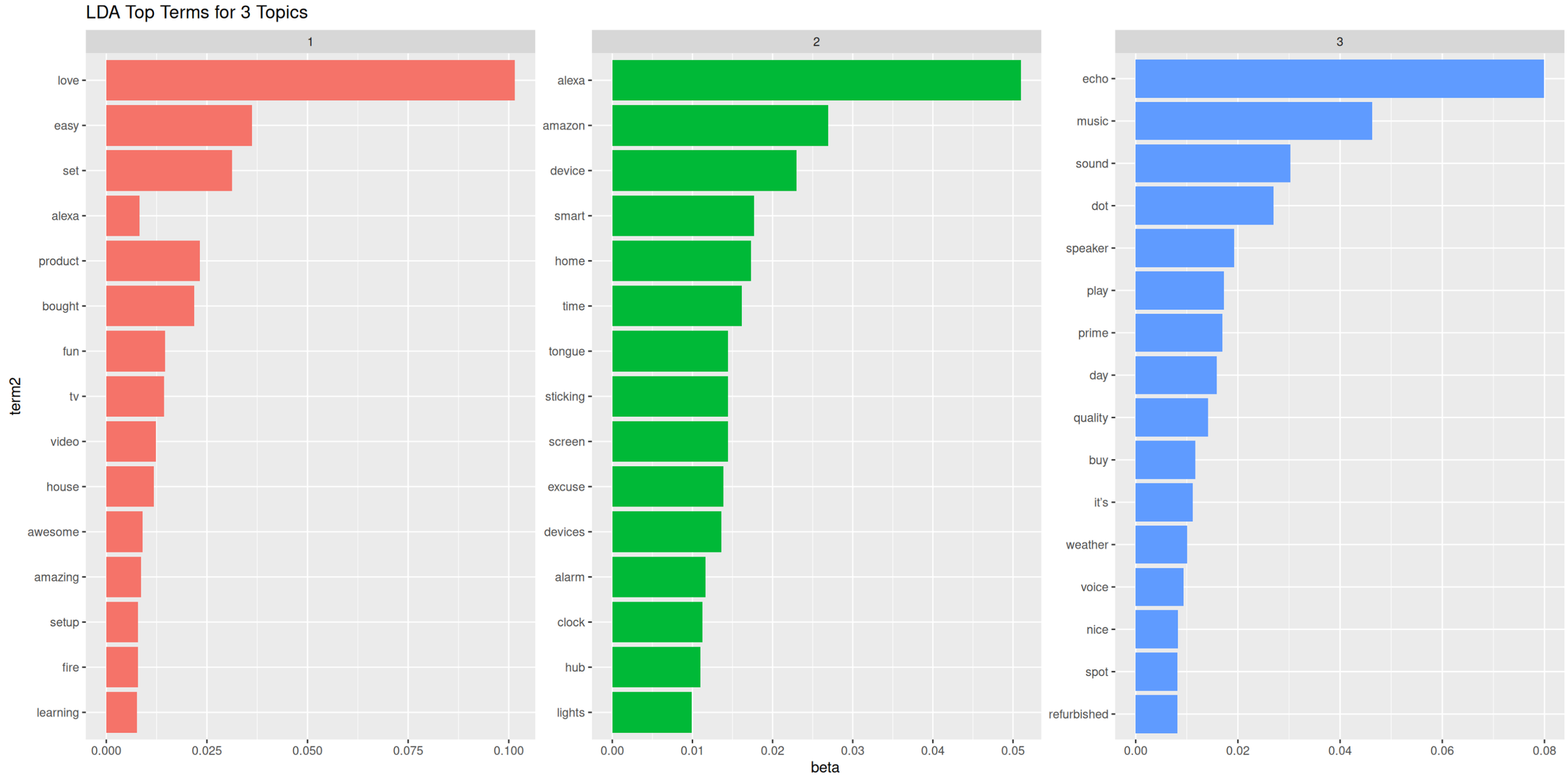
Interpreting Topics - Three Topics

```
lda_3_topics <- dtm_review |>
  LDA(
    k = 3,
    method = "Gibbs",
    control = list(seed = 123)
    ) |>
  tidy(matrix = "beta")

word_3_probs <- lda_3_topics |>
  group_by(topic) |>
  slice_max(beta, n = 15) |>
  ungroup() |>
  mutate(term2 = fct_reorder(term, beta))
```

Interpreting Topics - Three Topics

```
word_3_probs |>
  ggplot(
    aes(
      x = term2,
      y = beta,
      fill = as.factor(topic)
      )
  ) +
  geom_col(show.legend = FALSE) +
  facet_wrap(~topic, scales = "free") +
  coord_flip() +
  labs(
    title = "LDA Top Terms for 3 Topics"
  )
```

Interpreting Topics - Three Topics



symrise

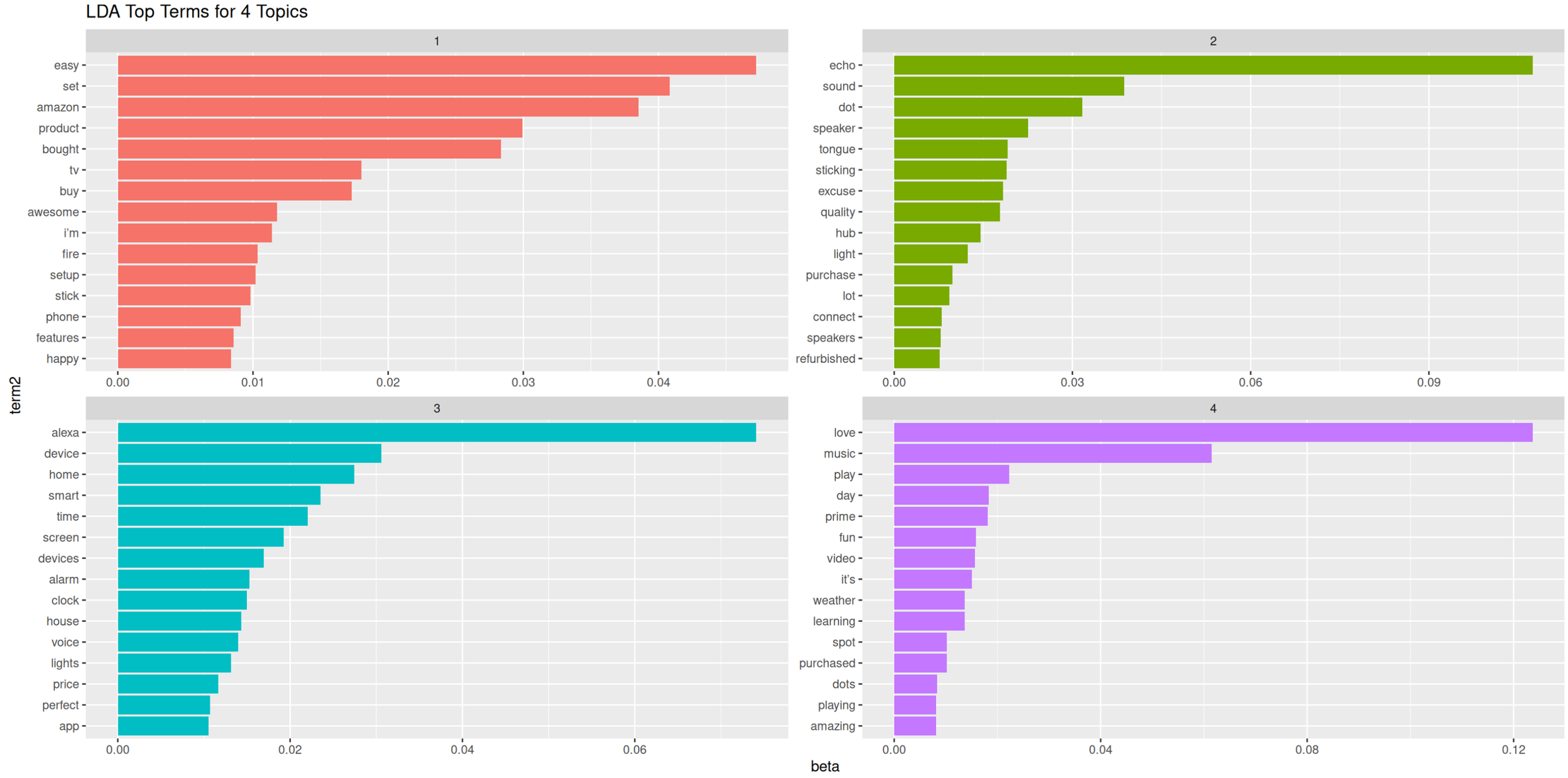
Interpreting Topics - Four Topics

```
lda_4_topics <- LDA(
  dtm_review,
  k = 4,
  method = "Gibbs",
  control = list(seed = 123)
) |>
  tidy(matrix = "beta")

word_4_probs <- lda_4_topics |>
  group_by(topic) |>
  slice_max(beta, n = 15) |>
  ungroup() |>
  mutate(term2 = fct_reorder(term, beta))
```

Interpreting Topics - Four Topics

```
word_4_probs |>
  ggplot(
    aes(
      x = term2,
      y = beta,
      fill = as.factor(topic)
      )
    ) +
  geom_col(show.legend = FALSE) +
  facet_wrap(~ topic, scales = "free") +
  coord_flip() +
  labs(
    title = "LDA Top Terms for 4 Topics"
  )
```

Interpreting Topics - Four Topics



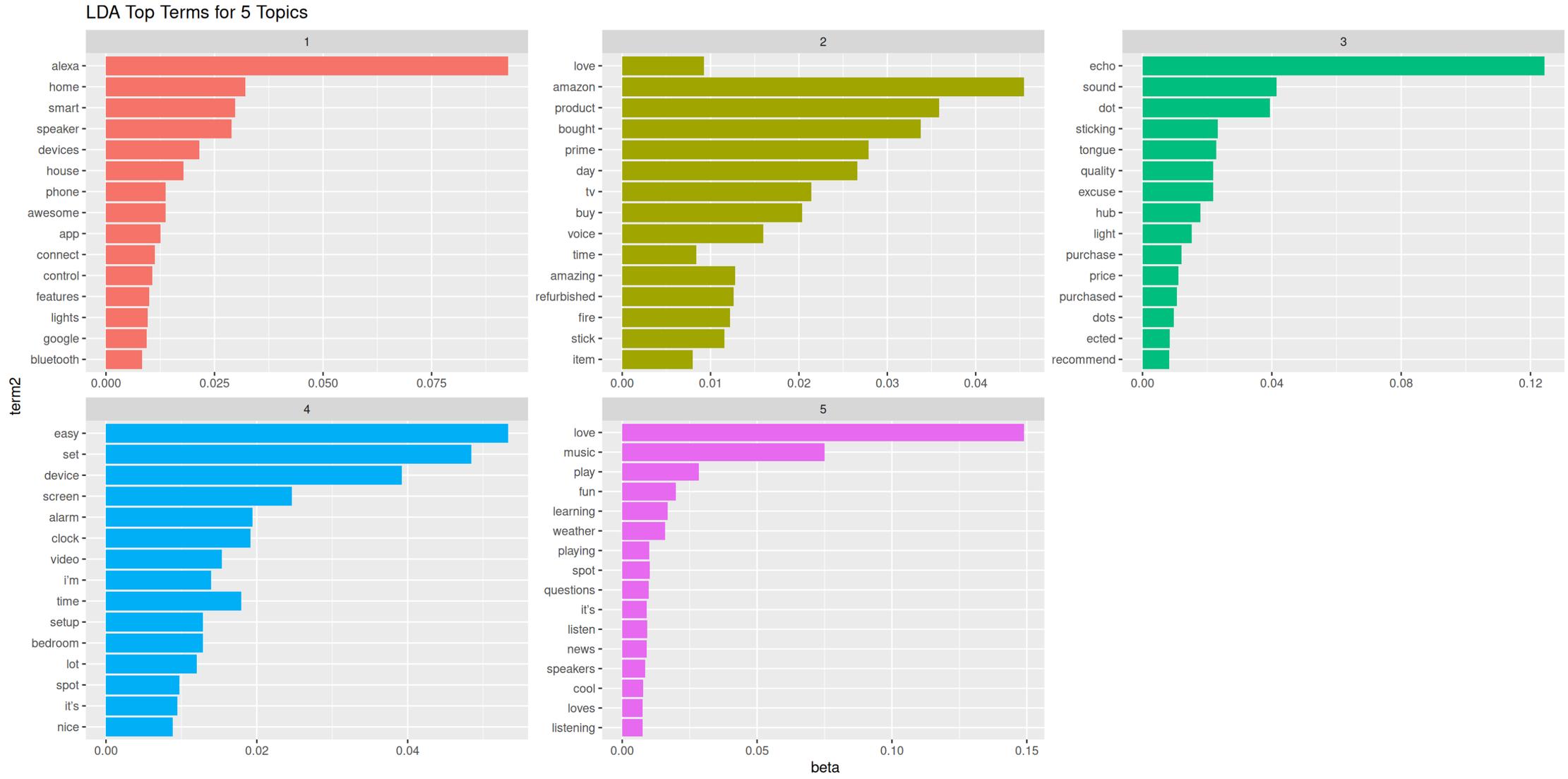
Interpreting Topics - Five Topics

```r
lda_5_topics <- LDA(
  dtm_review,
  k = 5,
  method = "Gibbs",
  control = list(seed = 123)
) |>
  tidy(matrix = "beta")

word_5_probs <- lda_5_topics |>
  group_by(topic) |>
  slice_max(beta, n = 15) |>
  ungroup() |>
  mutate(term2 = fct_reorder(term, beta))
```

Interpreting Topics - Five Topics

```
word_5_probs |>
ggplot(
  aes(
    x = term2,
    y = beta,
    fill = as.factor(topic)
    )
  ) +
  geom_col(show.legend = FALSE) +
  facet_wrap(~ topic, scales = "free") +
  coord_flip() +
  labs(
    title = "LDA Top Terms for 5 Topics"
  )
```

Interpreting Topics - Five Topics



symrise

The Art of Topic Selection

- Unsupervised learning always requires human interpretation…
- Including new topics which are different is always a good thing.
- When topics start repeating, we have selected too many!
- Topics can be named based on the combination of high-probability words.

The Art of Topic Selection

Don’t forget about the mixed-membership concept and that these topics are not meant to be completely disjoint.

You can fine tune the LDA algorithm using extra parameters to improve the model.

Document-Topic Probabilities

LDA models each document as a mix of topics and words.

With `matrix = "gamma"` we can investigate *per-document-per-topic* probabilities.

Document-Topic Probabilities

Each of these values represents an estimated percentage of the document's words that are from each topic.

Most of these reviews belong to more than one topic:

```
lda_tidy |>
  tidy(matrix = "gamma") |>
  arrange(desc(gamma))
```

```
# A tibble: 4,660 × 3
   document topic gamma
   <chr>    <int> <dbl>
 1 1666         1 0.728
 2 1466         1 0.716
 3 1496         1 0.688
 4 1646         1 0.683
 5 1688         1 0.677
 6 1601         1 0.671
 7 1658         1 0.670
 8 1624         1 0.655
 9 1464         1 0.651
10 1038         2 0.648
# ℹ 4,650 more rows
```

Colophon - Summary

Part One: Text Mining and Exploratory Analysis

- We covered several topics:
  - `Tidy` and `|>`
  - Process of Text Mining
  - Data Exploration
  - Text Cleaning with `textclean`
  - Tokenizing Text with `tidytext`
  - Removal of stop words
  - Plotting Words (Counts and Clouds)

Part Two: Tidy Sentiment Analysis in R

- We explored:
  - Sentiment Analysis Dictionaries
  - Counting and Visualizing Sentiments
  - Comparing Sentiments (Counts and Clouds)

Part Three: Topic Modelling

- We discussed:
  - Clustering vs Topic Modelling
  - Topic Modelling with LDA
  - Create the Document-Term Matrix
  - Interpreting Topics
  - Finding the Terms Generating Differences
  - Document-Topic Probabilities
  - The Art of Topic Selection

Outro

Thank you! 

Feel free to explore the Appendix!

References

Hu, Minqing, and Bing Liu. 2004. *Mining and Summarizing Customer Reviews*. KDD ’04. New York, NY, USA: ACM. https://doi.org/10.1145/1014052.1014073.

Loughran, Tim, and Bill McDonald. 2011. “When Is a Liability Not a Liability? Textual Analysis, Dictionaries, and 10-Ks.” *The Journal of Finance* 66 (1): 35–65. https://doi.org/10.1111/j.1540-6261.2010.01625.x.

Mohammad, Saif M., and Peter D. Turney. 2013. “Crowdsourcing a Word-Emotion Association Lexicon.” *Computational Intelligence* 29 (3): 436–65.

Nielsen, F. Å. 2011. “AFINN.” Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby: Informatics; Mathematical Modelling, Technical University of Denmark. http://www2.compute.dtu.dk/pubdb/pubs/6010-full.html.

Rinker, Tyler W. 2021. *sentimentr: Calculate Text Polarity Sentiment*. Buffalo, New York. https://github.com/trinker/sentimentr.

Silge, & Robinson, J. 2017. *Text Mining with r*. O’Reilly Media.

Wickham, Cetinkaya-Rundel, H. 2023. *R for Data Science (2e)*. O’Reilly Media.

Appendix

Sentiment Analysis with `sentimentr`

- Another package for lexicon-based sentiment analysis is `sentimentr` (Rinker 2021).
- Unlike the `tidytext` package, `sentimentr` takes valence shifters (e.g., negation) into account, which can easily flip the polarity of a sentence with one word.

Sentiment Analysis with `sentimentr`

- For example, the sentence *"I am not unhappy"* is actually positive.
- But if we analyze it word by word, the sentence may seem to have a negative sentiment due to the words *"not"* and *"unhappy"*.
- Similarly, *"I hardly like this book"* is a negative sentence.
- But the analysis of individual words, *"hardly"* and *"like"*, may yield a positive sentiment score.

Sentiment Analysis with `sentimentr`

In contrast to `tidytext`, for `sentimentr` we need the actual sentences rather than the individual tokens.

Therefore, we can:

- use the original cleaned `review_data`,
- get individual sentences for each media briefing using the `get_sentences()` function, and
- calculate sentiment scores per sentence via `sentiment()`.

As usual, we need to install the package:

```
1 install.package(sentimentr)
```

And load it before usage:

```
1 library(sentimentr)
```

Sentiment Analysis with `sentimentr`

```
sentimentr_review <- review_data |>
  mutate(id = row_number()) |>
  get_sentences() |>
  sentiment()
```

Sentiment Analysis with `sentimentr`

```
1 sentimentr_review <- review_data |>
2   mutate(id = row_number()) |>
3   get_sentences() |>
4   sentiment()
5
6 sentimentr_review
```

```
      stars      date         product
   1:     5 31-Jul-18 Charcoal Fabric
   2:     5 31-Jul-18 Charcoal Fabric
   3:     4 31-Jul-18   Walnut Finish
   4:     4 31-Jul-18   Walnut Finish
   5:     5 31-Jul-18 Charcoal Fabric
  ---
5834:     5 30-Jul-18      White  Dot
5835:     4 29-Jul-18      Black  Dot
5836:     5 29-Jul-18      Black  Dot
5837:     5 31-Jul-18      Black  Dot
5838:     5 31-Jul-18      Black  Dot

review
   1:
```

Sentiment Analysis with `sentimentr` - Plotting by Star Rating

```
sentimentr_review |>
  group_by(stars) |>
  ggplot(
    aes(
      x = stars,
      y = sentiment,
      fill = as.factor(stars)
      )
    ) +
  geom_col(show.legend = FALSE) +
  coord_flip() +
  labs(
    title = "Overall Sentiment by Stars using sentimentr",
    subtitle = "Reviews for Alexa",
    x = "Stars",
    y = "Overall Sentiment"
  )
```

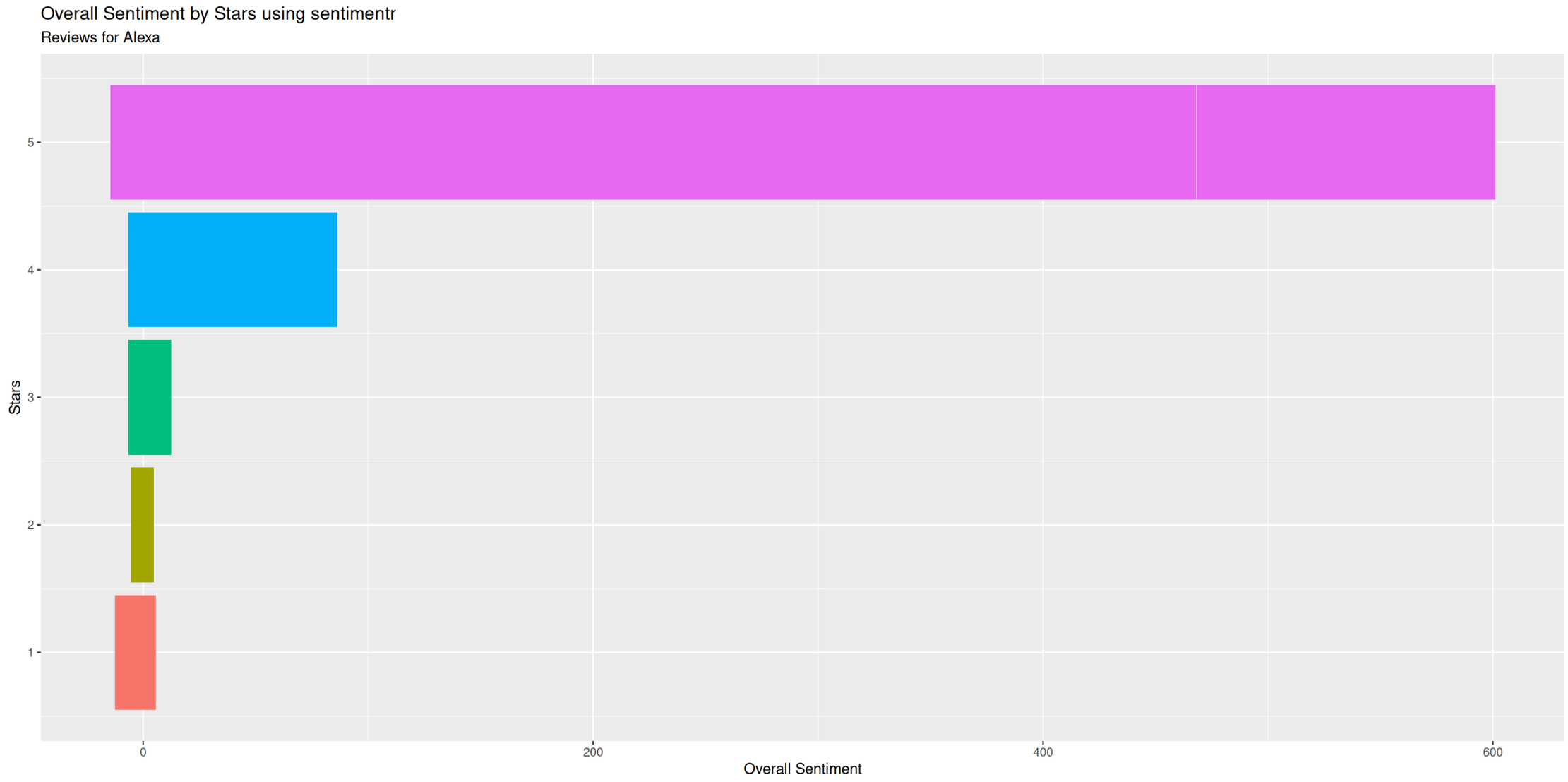
Sentiment Analysis with `sentimentr` - Plotting by Star Rating

Sentiment Analysis with `sentimentr`

We can also look at sentiment analysis by whole reviews, instead of per sentence.

```
sentimentr_sentence <- review_data |>
  get_sentences() |>
  sentiment_by()

review_data_id <- review_data |>
    mutate(id = row_number())

sentimentr_merged <- sentimentr_sentence |>
  inner_join(review_data_id,
             join_by(element_id == id))

sentimentr_merged |>
  group_by(stars) |>
  ggplot(
    aes(
      x = stars,
      y = ave_sentiment,
      fill = as.factor(stars)
      )
    ) +
  geom_col(show.legend = FALSE) +
  coord_flip() +
  labs(
    title = "Overall Sentiment by Stars using sentimentr",
    subtitle = "Reviews for Alexa",
    x = "Stars",
    y = "Overall Sentiment"
  )
```

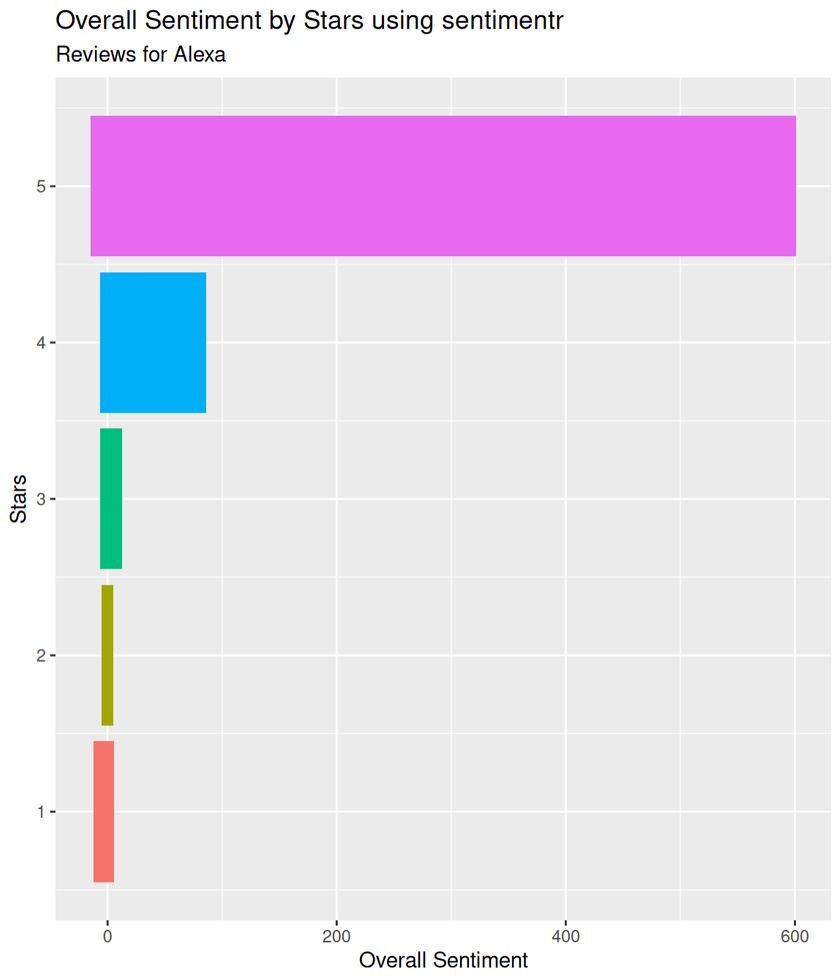
Sentiment Analysis with `sentimentr`



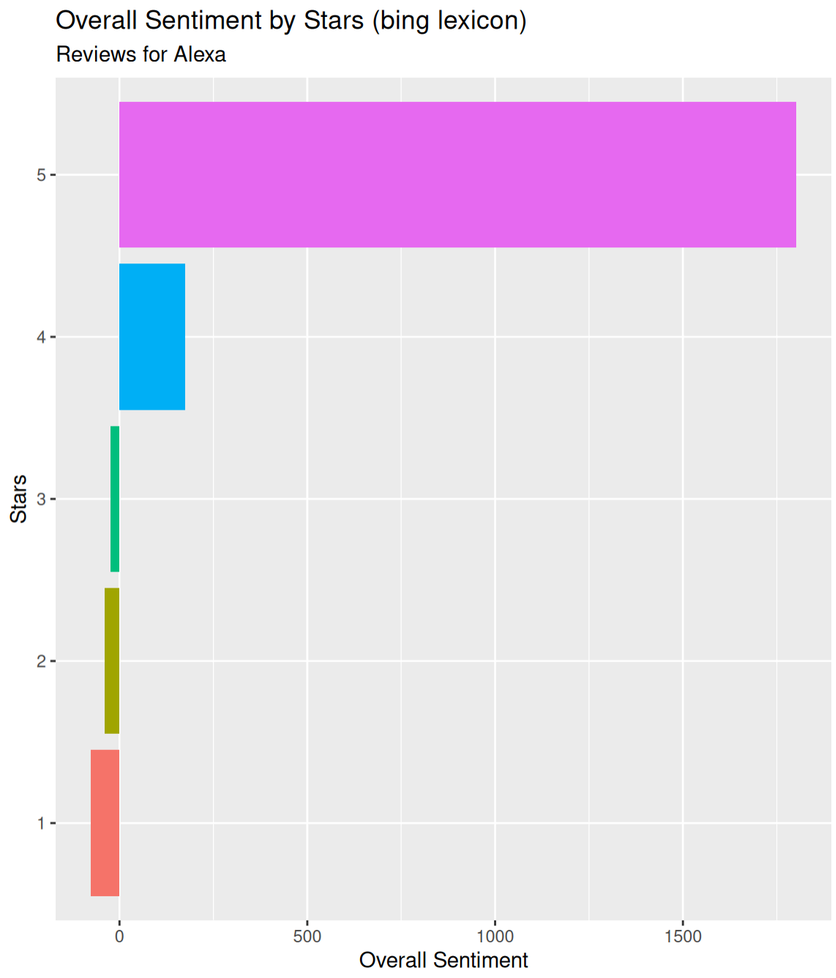
Sentiment Analysis with `sentimentr`

For this specific case, we can see that the sentiment analysis results are very similar between:

`sentimentr` at a sentence level



`bing` lexicon on a word by word basis



(Alternative) Creating Word Clouds

Unfortunately, there is no `tidy` way to create a word cloud with the `wordcloud` package 😒

Regardless, it is time to ask the `wordcloud()` function to read and plot our data:

```
wordcloud(
  word = word_counts$word,
  freq = word_counts$n
)
```

(Alternative) Creating Word Clouds

(Alternative) Creating Word Clouds

There are some useful arguments to experiment with here:

- `min.freq` and `max.words` set boundaries for how populated the wordcloud will be
- `random.order` will put the largest word in the middle if set to FALSE
- `rot.per` is the fraction of words that will be rotated in the graphic

Finally, the words are arranged randomly somehow, and so for a repeatable graphic we need to specify a seed value with `set.seed()`.

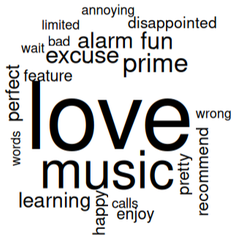
(Alternative) Creating Word Clouds

```
set.seed(13)
wordcloud(
  word = word_counts$word,
  freq = word_counts$n,
  max.words = 30,
  min.freq = 4,
  random.order = FALSE,
  rot.per = 0.25
)
```

(Alternative) Creating Word Clouds

As explained, we can also change the number of words displayed in the cloud:

```
set.seed(13)
wordcloud(
  word = word_counts$word,
  freq = word_counts$n,
  max.words = 70,
  min.freq = 4,
  random.order = FALSE,
  rot.per = 0.25
)
```



(Alternative) Creating Word Clouds

Using pre-defined colours:

```
set.seed(13)
wordcloud(
  word = word_counts$word,
  freq = word_counts$n,
  max.words = 30,
  min.freq = 4,
  random.order = FALSE,
  rot.per = 0.25,
  colors = "blue"
)
```

(Alternative) Creating Word Clouds

Using funky colours, thanks to the `RColorBrewer` package and its large selection of colour palettes:

```
set.seed(13)
wordcloud(
  word = word_counts$word,
  freq = word_counts$n,
  max.words = 30,
  min.freq = 4,
  random.order = FALSE,
  rot.per = 0.25,
  colors = brewer.pal(8, "Paired")
)
```

(Alternative) Creating Word Clouds

If you need more customization (including non-latin characters), you can use the `wordcloud2()` function from the `wordcloud2` package[1]:

```
library(wordcloud2)
set.seed(13)
wordcloud2(word_counts_filter,
           size = 2,
           minRotation = -pi/2,
           maxRotation = -pi/2)
```

(Alternative) Creating Word Clouds