# 4

# *Stemming*

When we deal with text, often documents contain different versions of one base word, often called a *stem*. "The Fir-Tree," for example, contains more than one version (i.e., inflected form) of the word `"tree"`.

```
library(hcandersenr)
library(tidyverse)
library(tidytext)

fir_tree <- hca_fairytales() %>%
  filter(book == "The fir tree",
         language == "English")

tidy_fir_tree <- fir_tree %>%
  unnest_tokens(word, text) %>%
  anti_join(get_stopwords())

tidy_fir_tree %>%
  count(word, sort = TRUE) %>%
  filter(str_detect(word, "^tree"))
```

```
#> # A tibble: 3 x 2
#>   word       n
#>   <chr>  <int>
#> 1 tree      76
#> 2 trees     12
#> 3 tree's     1
```

Trees, we see once again, are important in this story; the singular form appears 76 times and the plural form appears 12 times. (We'll come back to how we might handle the apostrophe in `"tree's"` later in this chapter.)

What if we aren't interested in the difference between `"trees"` and `"tree"` and we want to treat both together? That idea is at the heart of *stemming*, the process of identifying the base word (or stem) for a data set of words. Stemming is

DOI: 10.1201/9781003093459-4

concerned with the linguistics subfield of morphology, how words are formed. In this example, `"trees"` would lose its letter `"s"` while `"tree"` stays the same. If we counted word frequencies again after stemming, we would find that there are 88 occurrences of the stem `"tree"` (89, if we also find the stem for `"tree's"`).

## 4.1 How to stem text in R

There have been many algorithms built for stemming words over the past half century or so; we'll focus on two approaches. The first is the stemming algorithm of Porter (1980), probably the most widely used stemmer for English. Porter himself released the algorithm implemented in the framework Snowball[1] with an open-source license; you can use it from R via the **SnowballC** package (Bouchet-Valat 2020). (It has been extended to languages other than English as well.)

```
library(SnowballC)

tidy_fir_tree %>%
  mutate(stem = wordStem(word)) %>%
  count(stem, sort = TRUE)
```

```
#> # A tibble: 570 x 2
#>    stem        n
#>    <chr>   <int>
#>  1 tree       88
#>  2 fir        34
#>  3 littl      23
#>  4 said       22
#>  5 stori      16
#>  6 thought    16
#>  7 branch     15
#>  8 on         15
#>  9 came       14
#> 10 know       14
#> # ... with 560 more rows
```

[1] https://snowballstem.org/

Take a look at those stems. Notice that we do now have 88 incidences of "tree". Also notice that some words don't look like they are spelled as real words; this is normal and expected with this stemming algorithm. The Porter algorithm identifies the stem of both "story" and "stories" as "stori", not a regular English word but instead a special stem object.

If you want to tokenize *and* stem your text data, you can try out the function `tokenize_word_stems()` from the **tokenizers** package, which implements Porter stemming just as we demonstrated here. For more on tokenization, see Chapter 2.

Does Porter stemming only work for English? Far from it! We can use the `language` argument to implement Porter stemming in multiple languages. First we can tokenize the text and `nest()` into list-columns.

```
stopword_df <- tribble(~language, ~two_letter,
                       "danish",  "da",
                       "english", "en",
                       "french",  "fr",
                       "german",  "de",
                       "spanish", "es")

tidy_by_lang <- hca_fairytales() %>%
  filter(book == "The fir tree") %>%
  select(text, language) %>%
  mutate(language = str_to_lower(language)) %>%
  unnest_tokens(word, text) %>%
  nest(data = word)
```

Then we can remove stop words (using `get_stopwords(language = "da")` and similar for each language) and stem with the language-specific Porter algorithm. What are the top-20 stems for "The Fir-Tree" in each of these five languages, after removing the Snowball stop words for that language?

```
tidy_by_lang %>%
  inner_join(stopword_df) %>%
  mutate(data = map2(
    data, two_letter, ~ anti_join(.x, get_stopwords(language = .y)))
  ) %>%
```

```
  unnest(data) %>%
  mutate(stem = wordStem(word, language = language)) %>%
  group_by(language) %>%
  count(stem) %>%
  top_n(20, n) %>%
  ungroup %>%
  ggplot(aes(n, fct_reorder(stem, n), fill = language)) +
  geom_col(show.legend = FALSE) +
  facet_wrap(~language, scales = "free_y", ncol = 2) +
  labs(x = "Frequency", y = NULL)
```

Figure 4.1 demonstrates some of the challenges in working with languages other than English; the stop word lists may not be even from language to language, and tokenization strategies that work for a language like English may struggle for a language like French with more stop word contractions. Given that, we see here words about little fir trees at the top for all languages, in their stemmed forms.

The Porter stemmer is an algorithm that starts with a word and ends up with a single stem, but that's not the only kind of stemmer out there. Another class of stemmer are dictionary-based stemmers. One such stemmer is the stemming algorithm of the Hunspell[2] library. The "Hun" in Hunspell stands for Hungarian; this set of NLP algorithms was originally written to handle Hungarian but has since been extended to handle many languages with compound words and complicated morphology. The Hunspell library is used mostly as a spell checker, but as part of identifying correct spellings, this library identifies word stems as well. You can use the Hunspell library from R via the **hunspell** (Ooms 2020b) package.

```
library(hunspell)

tidy_fir_tree %>%
  mutate(stem = hunspell_stem(word)) %>%
  unnest(stem) %>%
  count(stem, sort = TRUE)
```

[2] http://hunspell.github.io/

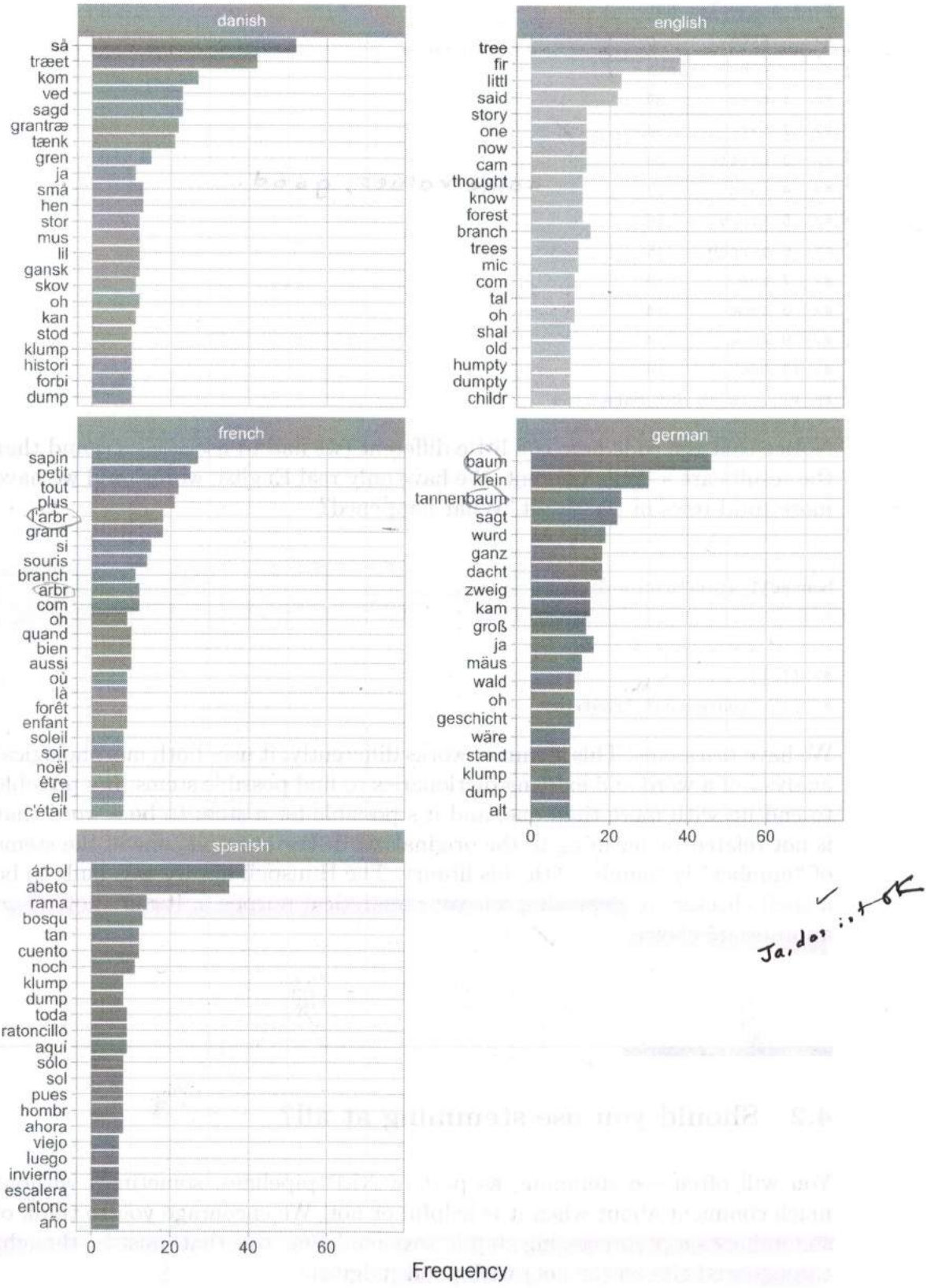



**FIGURE 4.1:** Porter stemming results in five languages

```
#> # A tibble: 595 x 2
#>    stem       n
#>    <chr>  <int>
#>  1 tree      89
#>  2 fir       34
#>  3 little    23
#>  4 said      22
#>  5 story     16
#>  6 branch    15
#>  7 one       15
#>  8 came      14
#>  9 know      14
#> 10 now       14
#> # ... with 585 more rows
```

Notice that the code here is a little different (we had to use `unnest()`) and that the results are a little different. We have only real English words, and we have more total rows in the result. What happened?

```
hunspell_stem("discontented")
```

```
#> [[1]]
#> [1] "contented" "content"
```

We have *two* stems! This stemmer works differently; it uses both morphological analysis of a word and existing dictionaries to find possible stems. It's possible to end up with more than one, and it's possible for a stem to be a word that is not related by meaning to the original word. For example, one of the stems of "number" is "numb" with this library. The Hunspell library was built to be a spell checker, so depending on your analytical purposes, it may not be an appropriate choice.

## 4.2 Should you use stemming at all?

You will often see stemming as part of NLP pipelines, sometimes without much comment about when it is helpful or not. We encourage you to think of stemming as a preprocessing step in text modeling, one that must be thought through and chosen (or not) with good judgment.

Why does stemming often help, if you are training a machine learning model for text? Stemming *reduces the feature space* of text data. Let's see this in action, with a data set of United States Supreme Court opinions available in the **scotus** package, discussed in more detail in Section B.2. How many words are there, after removing a standard data set of stopwords?

NOTE: tried to install "scotus" package, error said its not available for this version of R, so I typed this code:

> install.packages("remotes")
> remotes::install_github("EmilHvitfeldt/scotus")

```
library(scotus)

tidy_scotus <- scotus_filtered %>%
  unnest_tokens(word, text) %>%
  anti_join(get_stopwords())

tidy_scotus %>%
  count(word, sort = TRUE)
```

✓ ✓

```
#> # A tibble: 167,879 x 2
#>    word        n
#>    <chr>   <int>
#>  1 court  286448
#>  2 v      204176
#>  3 state  148320
#>  4 states 128160
#>  5 case   121439
#>  6 act    111033
#>  7 s.ct   108168
#>  8 u.s    106413
#>  9 upon   105069
#> 10 united 103267
#> # ... with 167,869 more rows
```

✓ exact same output, good

There are 167,879 distinct words in this data set we have created (after removing stopwords) but notice that even in the most common words we see a pair like `"state"` and `"states"`. A common data structure for modeling, and a helpful mental model for thinking about the sparsity of text data, is a matrix. Let's `cast()` this tidy data to a sparse matrix, technically, a document-feature matrix object from the **quanteda** (Benoit et al. 2018) package.

Yay! A non-tidy package :) Also taught in Jockers' textbook.

```
tidy_scotus %>%
  count(case_name, word) %>%
  cast_dfm(case_name, word, n)
```

```
#> Document-feature matrix of: 9,642 documents, 167,879
features (99.49% sparse) and 0 docvars.
```

Look at the sparsity of this matrix. It's high! Think of this sparsity as the sparsity of data that we will want to use to build a supervised machine learning model.

What if instead we use stemming as a preprocessing step here?

```
tidy_scotus %>%
  mutate(stem = wordStem(word)) %>%
  count(case_name, stem) %>%
  cast_dfm(case_name, stem, n)
```

```
#> Document-feature matrix of: 9,642 documents, 135,570
features (99.48% sparse) and 0 docvars.
```

We reduced the number of word features by many thousands, although the sparsity did not change much. Why is it possibly helpful to reduce the number of features? Common sense says that reducing the number of word features in our data set so dramatically will improve the performance of any machine learning model we train with it, *assuming that we haven't lost any important information by stemming*.

There is a growing body of academic research demonstrating that stemming can be counterproductive for text modeling. For example, Schofield and Mimno (2016) and related work explore how choices around stemming and other preprocessing steps don't help and can actually hurt performance when training topic models for text. From Schofield and Mimno (2016) specifically,

> Despite their frequent use in topic modeling, we find that stemmers produce no meaningful improvement in likelihood and coherence and in fact can degrade topic stability.

Topic modeling is an example of unsupervised machine learning for text and is not the same as the predictive modeling approaches we'll be focusing on in this book, but the lesson remains that stemming may or may not be beneficial for any specific context. As we work through the rest of this chapter and learn more about stemming, consider what information we lose when we stem text in exchange for reducing the number of word features. Stemming can be helpful in some contexts, but typical stemming algorithms are somewhat aggressive and have been built to favor sensitivity (or recall, or the true positive rate) at the expense of specificity (or precision, or the true negative rate).

Most common stemming algorithms you are likely to encounter will successfully reduce words to stems (i.e., not leave extraneous word endings on the words) but at the expense of collapsing some words with dramatic differences in meaning, semantics, use, etc. to the same stems. Examples of the latter are numerous, but some include:

- meaning and mean
- likely, like, liking
- university and universe

In a supervised machine learning context, this affects a model's positive predictive value (precision), or ability to not incorrectly label true negatives as positive. In Chapter 7, we will train models to predict whether a complaint to the United States Consumer Financial Protection Bureau was about a mortgage or not. Stemming can increase a model's ability to find the positive examples, i.e., the complaints about mortgages. However, if the complaint text is over-stemmed, the resulting model loses its ability to label the negative examples, the complaints *not* about mortgages, correctly.

## 4.3 Understand a stemming algorithm

If stemming is going to be in our NLP toolbox, it's worth sitting down with one approach in detail to understand how it works under the hood. The Porter stemming algorithm is so approachable that we can walk through its outline in less than a page or so. It involves five steps, and the idea of a word **measure**.

Think of any word as made up alternating groups of vowels $V$ and consonants $C$. One or more vowels together are one instance of $V$, and one or more consonants togther are one instance of $C$. We can write any word as

$$[C](VC)^m[V]$$

where $m$ is called the "measure" of the word. The first $C$ and the last $V$ in brackets are optional. In this framework, we could write out the word `"tree"` as

$$CV$$

with $C$ being "tr" and $V$ being "ee"; it's an `m = 0` word. We would write out the word `"algorithms"` as

*VCVCVC*

and it is an `m = 3` word.

- The first step of the Porter stemmer is (perhaps this seems like cheating) actually made of three substeps working with plural and past participle word endings. In the first substep (1a), "sses" is replaced with "ss," "ies" is replaced with "i," and final single "s" letters are removed. The second substep (1b) depends on the measure of the word `m` but works with endings like "eed," "ed," "ing," adding "e" back on to make endings like "ate," "ble," and "ize" when appropriate. The third substep (1c) replaces "y" with "i" for words of a certain `m`.

- The second step of the Porter stemmer takes the output of the first step and regularizes a set of 20 endings. In this step, "ization" goes to "ize," "alism" goes to "al," "aliti" goes to "al" (notice that the ending "i" there came from the first step), and so on for the other 17 endings.

- The third step again processes the output, using a list of seven endings. Here, "ical" and "iciti" both go to "ic," "ful" and "ness" are both removed, and so forth for the three other endings in this step.

- The fourth step involves a longer list of endings to deal with again (19), and they are all removed. Endings like "ent," "ism," "ment," and more are removed in this step.

- The fifth and final step has two substeps, both which depend on the measure `m` of the word. In this step, depending on `m`, final "e" letters are sometimes removed and final double letters are sometimes removed.

How would this work for a few example words? The word "supervised" loses its "ed" in step 1b and is not touched by the rest of the algorithm, ending at "supervis". The word "relational" changes "ational" to "ate" in step 2 and loses its final "e" in step 5, ending at "relat". Notice that neither of these results are regular English words, but instead special stem objects. This is expected.

This algorithm was first published in Porter (1980) and is still broadly used; read Willett (2006) for background on how and why it has become a stemming standard. We can reach even *further* back and examine what is considered the

first ever published stemming algorithm in Lovins (1968). The domain Lovins worked in was engineering, so her approach was particularly suited to technical terms. This algorithm uses much larger lists of word endings, conditions, and rules than the Porter algorithm and, although considered old-fashioned, is actually faster!

Check out the steps of a Snowball stemming algorithm for German[3].

## 4.4 Handling punctuation when stemming

Punctuation contains information that can be used in text analysis. Punctuation *is* typically less information-dense than the words themselves, and thus it is often removed early in a text mining analysis project, but it's worth thinking through the impact of punctuation specifically on stemming. Think about words like `"they're"` and `"child's"`.

We've already seen how punctuation and stemming can interact with our small example of "The Fir-Tree"; none of the stemming strategies we've discussed so far have recognized `"tree's"` as belonging to the same stem as `"trees"` and `"tree"`.

```
tidy_fir_tree %>%
  count(word, sort = TRUE) %>%
  filter(str_detect(word, "^tree"))
```

```
#> # A tibble: 3 x 2
#>   word       n
#>   <chr>  <int>
#> 1 tree      76
#> 2 trees     12
#> 3 tree's     1
```

It is possible to split tokens not only on white space but **also** on punctuation, using a regular expression (see Appendix A).

[3] https://snowballstem.org/algorithms/german/stemmer.html

```
fir_tree_counts <- fir_tree %>%
  unnest_tokens(word, text, token = "regex", pattern = "\\s+|[[:punct:]]+") %>%
  anti_join(get_stopwords()) %>%
  mutate(stem = wordStem(word)) %>%
  count(stem, sort = TRUE)

fir_tree_counts
```

```
#> # A tibble: 572 x 2
#>    stem        n
#>    <chr>   <int>
#>  1 tree       89
#>  2 fir        34
#>  3 littl      23
#>  4 said       22
#>  5 stori      16
#>  6 thought    16
#>  7 branch     15
#>  8 on         15
#>  9 came       14
#> 10 know       14
#> # ... with 562 more rows
```

Now we are able to put all these related words together, having identified them with the same stem.

```
fir_tree_counts %>%
  filter(str_detect(stem, "^tree"))
```

```
#> # A tibble: 1 x 2
#>   stem      n
#>   <chr> <int>
#> 1 tree     89
```

Handling punctuation in this way further reduces sparsity in word features. Whether this kind of tokenization and stemming strategy is a good choice in any particular data analysis situation depends on the particulars of the text characteristics.

## 4.5 Compare some stemming options

Let's compare a few simple stemming algorithms and see what results we end with. Let's look at "The Fir-Tree," specifically the tidied data set from which we have removed stop words. Let's compare three very straightforward stemming approaches.

- **Only remove final instances of the letter "s."** This probably strikes you as not a great idea after our discussion in this chapter, but it is something that people try in real life, so let's see what the impact is.
- **Handle plural endings with slightly more complex rules in the "S" stemmer.** The S-removal stemmer or "S" stemmer of Harman (1991) is a simple algorithm with only three rules.[4]
- **Implement actual Porter stemming.** We can now compare to the most commonly-used stemming algorithm in English.

```
stemming <- tidy_fir_tree %>%
  select(-book, -language) %>%
  mutate(`Remove S` = str_remove(word, "s$"),
         `Plural endings` = case_when(str_detect(word, "[^e|aies$]ies$") ~
                                        str_replace(word, "ies$", "y"),
                                      str_detect(word, "[^e|a|oes$]es$") ~
                                        str_replace(word, "es$", "e"),
                                      str_detect(word, "[^ss$|us$]s$") ~
                                        str_remove(word, "s$"),
                                      TRUE ~ word),
         `Porter stemming` = wordStem(word)) %>%
  rename(`Original word` = word)
```

Figure 4.2 shows the results of these stemming strategies. All successfully handled the transition from `"trees"` to `"tree"` in the same way, but we have different results for `"stories"` to `"story"` or `"stori"`, different handling of `"branches"`, and more. There are subtle differences in the output of even these straightforward stemming approaches that can effect the transformation of text features for modeling.

[4]This simple, "weak" stemmer is handy to have in your toolkit for many applications. Notice how we implement it here using `dplyr::case_when()`.

```
stemming %>%
  gather(Type, Result, `Remove S`:`Porter stemming`) %>%
  mutate(Type = fct_inorder(Type)) %>%
  count(Type, Result) %>%
  group_by(Type) %>%
  top_n(20, n) %>%
  ungroup %>%
  ggplot(aes(fct_reorder(Result, n),
             n, fill = Type)) +
  geom_col(show.legend = FALSE) +
  facet_wrap(~Type, scales = "free_y") +
  coord_flip() +
  labs(x = NULL, y = "Frequency")
```

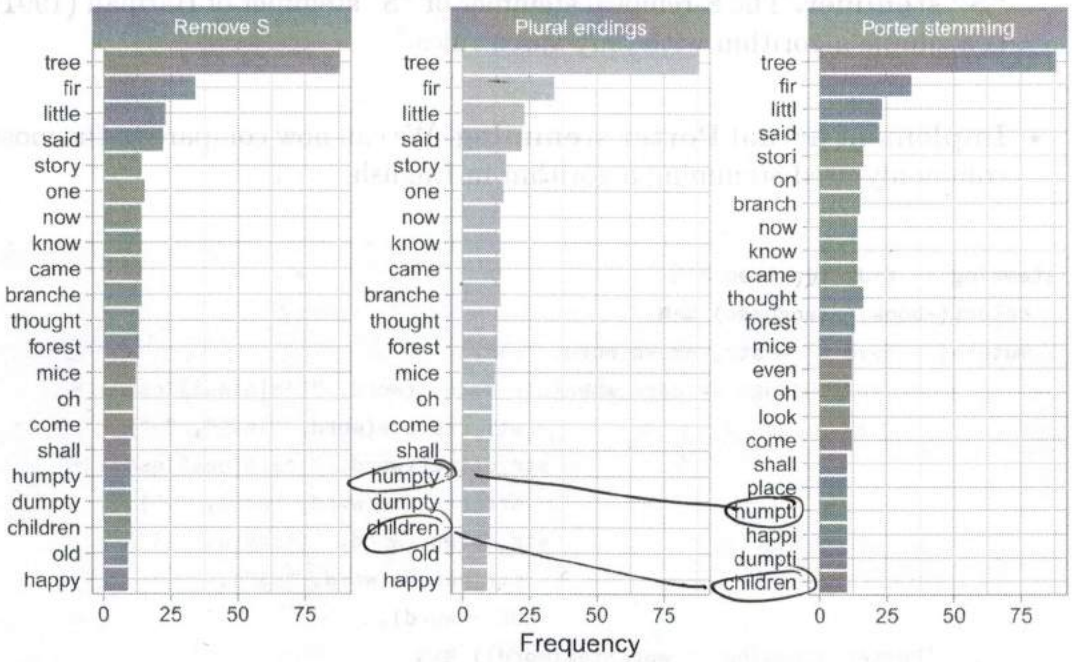



**FIGURE 4.2:** Results for three different stemming strategies

Porter stemming is the most different from the other two approaches. In the top-20 words here, we don't see a difference between removing only the letter "s" and taking the slightly more sophisticated "S" stemmer approach to plural endings. In what situations *do* we see a difference?

```
stemming %>%
  filter(`Remove S` != `Plural endings`) %>%
  distinct(`Remove S`, `Plural endings`, .keep_all = TRUE)
```

```
#> # A tibble: 13 x 4
#>    `Original word` `Remove S`  `Plural endings` `Porter stemming`
#>    <chr>           <chr>       <chr>            <chr>
#>  1 raspberries     raspberrie  raspberry        raspberri
#>  2 strawberries    strawberrie strawberry       strawberri
#>  3 less            les         less             less
#>  4 us              u           us               u
#>  5 brightness      brightnes   brightness       bright
#>  6 conscious       consciou    conscious        consciou
#>  7 faintness       faintnes    faintness        faint
#>  8 happiness       happines    happiness        happi
#>  9 ladies          ladie       lady             ladi
#> 10 babies          babie       baby             babi
#> 11 anxious         anxiou      anxious          anxiou
#> 12 princess        princes     princess         princess
#> 13 stories         storie      story            stori
```

data here is same as graph, except shown as a chart.

We also see situations where the same sets of original words are bucketed differently (not just with different stem labels) under different stemming strategies. In the following very small example, two of the strategies bucket these words into two stems while one strategy buckets them into one stem.

← annoying ... but useful sometimes - good for designing future NLP projects for TOI. studies as you can demonstrate a good grasp of multiple ways

```
stemming %>%
  gather(Type, Result, `Remove S`:`Porter stemming`) %>%
  filter(Result %in% c("come", "coming")) %>%
  distinct(`Original word`, Type, Result)
```

shows all steps of process

```
#> # A tibble: 9 x 3
#>   `Original word` Type            Result
#>   <chr>           <chr>           <chr>
#> 1 come            Remove S        come
#> 2 comes           Remove S        come
#> 3 coming          Remove S        coming
#> 4 come            Plural endings  come
#> 5 comes           Plural endings  come
#> 6 coming          Plural endings  coming
#> 7 come            Porter stemming come
#> 8 comes           Porter stemming come
#> 9 coming          Porter stemming come
```

These different characteristics can either be positive or negative, depending on the nature of the text being modeled and the analytical question being pursued.

Language use is connected to culture and identity. How might the results of stemming strategies be different for text created with the same language (like English) but in different social or cultural contexts, or by people with different identities? With what kind of text do you think stemming algorithms behave most consistently, or most as expected? What impact might that have on text modeling? unconjugated text? eg textbooks/manuals? idk.

## 4.6 Lemmatization vs. stemming

When people use the word "stemming" in natural language processing, they typically mean a system like the one we've been describing in this chapter, with rules, conditions, heuristics, and lists of word endings. Think of stemming as typically implemented in NLP as **rule-based**, operating on the word by itself. There is another option for normalizing words to a root that takes a different approach. Instead of using rules to cut words down to their stems, lemmatization uses knowledge about a language's structure to reduce words down to their lemmas, the canonical or dictionary forms of words. Think of lemmatization as typically implemented in NLP as **linguistics-based**, operating on the word in its context. "exceptions to the rules of conjugation"

Lemmatization requires more information than the rule-based stemmers we've discussed so far. We need to know what part of speech a word is to correctly identify its lemma,[5] and we also need more information about what words mean in their contexts. Often lemmatizers use a rich lexical database like WordNet[6] as a way to look up word meanings for a given part-of-speech use (Miller 1995). Notice that lemmatization involves more linguistic knowledge of a language than stemming.

How does lemmatization work in languages other than English? Lookup dictionaries connecting words, lemmas, and parts of speech for languages other than English have been developed as well.

[5] Part-of-speech information is also sometimes used directly in machine learning

[6] https://wordnet.princeton.edu/

can we use xml tagging to tag parts-of-speech in a .txt file, then process that in R?

A modern, efficient implementation for lemmatization is available in the excellent spaCy[7] library (Honnibal et al. 2020), which is written in Python.

NLP practitioners who work with R can use this library via the **spacyr** package (Benoit and Matsuo 2020), the **cleanNLP**[8] package (Arnold 2017), or as an "engine" in the **textrecipes**[9] package (Hvitfeldt 2020a).

Section 6.6 demonstrates how to use textrecipes with spaCy as an engine and include lemmas as features for modeling. You might also consider using spaCy directly in R Markdown via its Python engine[10].

Let's briefly walk through how to use spacyr.

```
library(spacyr)
spacy_initialize(entity = FALSE)

fir_tree %>%
  mutate(doc_id = paste0("doc", row_number())) %>%
  select(doc_id, everything()) %>%
  spacy_parse() %>%
  anti_join(get_stopwords(), by = c("lemma" = "word")) %>%
  count(lemma, sort = TRUE) %>%
  top_n(20, n) %>%
  ggplot(aes(n, fct_reorder(lemma, n))) +
  geom_col() +
  labs(x = "Frequency", y = NULL)
```

Figure 4.3 demonstrates how different lemmatization is from stemming, especially if we compare to Figure 4.2. Punctuation characters are treated as tokens (these punctuation tokens can have predictive power for some modeling questions!), and all pronouns are lemmatized to `-PRON-`. We see our familiar friends "tree" and "fir," but notice that we see the normalized version "say" instead of "said," "come" instead of "came," and similar. This transformation to the canonical or dictionary form of words is the goal of lemmatization.

[7] https://spacy.io/

[8] https://statsmaths.github.io/cleanNLP/

[9] https://textrecipes.tidymodels.org/

[10] https://rstudio.github.io/reticulate/articles/r_markdown.html

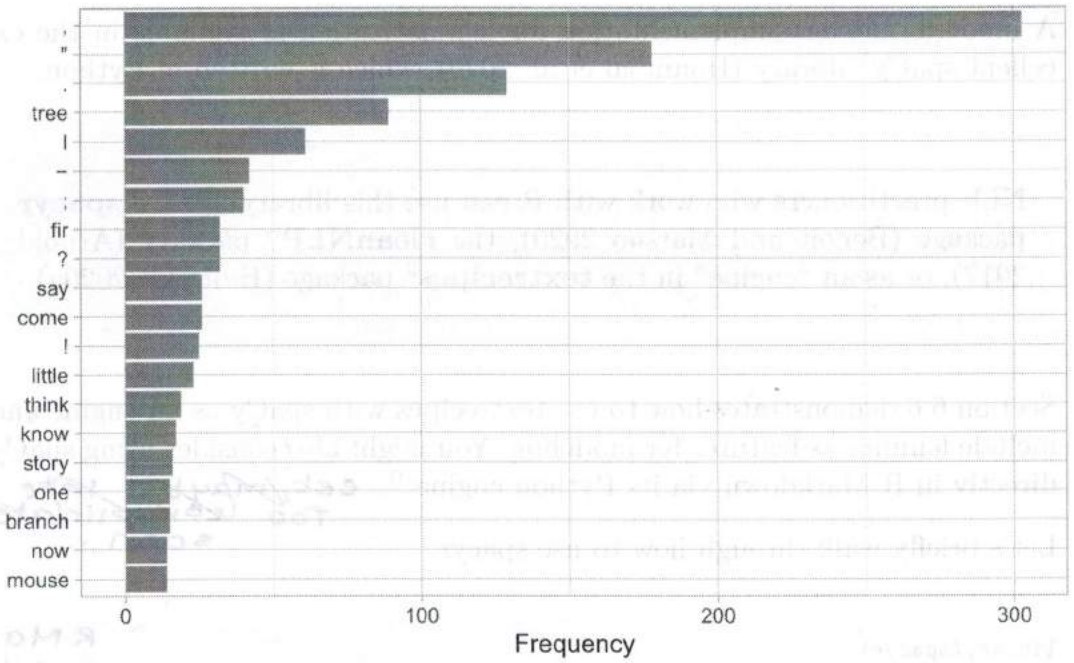



**FIGURE 4.3:** Results for lemmatization, rather than stemming

Why did we need to initialize the spaCy library? You may not need to, but spaCy is a full-featured NLP pipeline that not only tokenizes and identifies lemmas but also performs entity recognition. We will not use entity recognition in modeling or analysis in this book, and it takes a lot of computational power. Initializing with `entity = FALSE` will allow lemmatization to run much faster.

Implementing lemmatization is slower and more complex than stemming. Just like with stemming, lemmatization often improves the true positive rate (or recall) but at the expense of the true negative rate (or precision) compared to not using lemmatization, but typically less so than stemming.

## 4.7 Stemming and stop words

Our deep dive into stemming came *after* our chapters on tokenization (Chapter 2) and stop words (Chapter 3) because this is typically when you will want to implement stemming, if appropriate to your analytical question. Stop

word lists are usually unstemmed, so you need to remove stop words before stemming text data. For example, the Porter stemming algorithm transforms words like "themselves" to "themselv", so stemming first would leave you without the ability to match up to the commonly-used stop word lexicons.

A handy trick is to use the following function on your stop word list to return the words that don't have a stemmed version in the list. If the function returns a length 0 vector then you can stem and remove stop words in any order.

```
library(stopwords)
not_stemmed_in <- function(x) {
  x[!SnowballC::wordStem(x) %in% x]
}

not_stemmed_in(stopwords(source = "snowball"))
```

```
#>  [1] "ourselves"  "yourselves" "his"        "they"       "themselves"
#>  [6] "this"       "are"        "was"        "has"        "does"
#> [11] "you're"     "he's"       "she's"      "it's"       "we're"
#> [16] "they're"    "i've"       "you've"     "we've"      "they've"
#> [21] "let's"      "that's"     "who's"      "what's"     "here's"
#> [26] "there's"    "when's"     "where's"    "why's"      "how's"
#> [31] "because"    "during"     "before"     "above"      "once"
#> [36] "any"        "only"       "very"
```

Here we see that many of the words that are lost are the contractions.

In Section 3.2, we explored whether to include "tree" as a stop word for "The Fir-Tree." Now we can understand that this is more complicated than we first discussed, because there are different versions of the base word ("trees," "tree's") in our data set. Interactions between preprocessing steps can have a major impact on your analysis.

## 4.8 Summary

In this chapter, we explored stemming, the practice of identifying and extracting the base or stem for a word using rules and heuristics. Stemming reduces

the sparsity of text data, which can be helpful when training models, but at the cost of throwing information away. Lemmatization is another way to normalize words to a root, based on language structure and how words are used in their context.

### 4.8.1 In this chapter, you learned:

- about the most broadly-used stemming algorithms
- how to implement stemming
- that stemming changes the sparsity or feature space of text data
- the differences between stemming and lemmatization

Great, though technically annoying, chapter. Good for thinking through implications of pre-processing steps we take.

# 5

# Word Embeddings

When developing a new project, ask → les vrais questions sont:
① comment savons-nous combien nettoyer?
② Comment savons-nous que nous l'avons nettoyé sans perdre de sens?
③ comment pouvons-nous justifier nos étapes dans la section document de recherche / méthodologie? et
④ Devrait-il y avoir une norme dans tout le domaine des études sur Tolkien?

> You shall know a word by the company it keeps.
> — John Rupert Firth[1]

So far in our discussion of natural language features, we have discussed preprocessing steps such as tokenization, removing stop words, and stemming in detail. We implement these types of preprocessing steps to be able to represent our text data in some data structure that is a good fit for modeling.

"cleaning"

## 5.1 Motivating embeddings for sparse, high-dimensional data

→ types of data structures:
→ dfm
→ dtm
→ csv
...

What kind of data structure might work well for typical text data? Perhaps, if we wanted to analyze or build a model for consumer complaints to the United States Consumer Financial Protection Bureau (CFPB)[2], described in Section B.3, we would start with straightforward word counts. Let's create a sparse matrix, where the matrix elements are the counts of words in each document.

First, download the .csv file !

```
library(tidyverse)
library(tidytext)
library(SnowballC)

complaints <- read_csv("data/complaints.csv.gz")
```

[1] https://en.wikiquote.org/wiki/John_Rupert_Firth

[2] https://www.consumerfinance.gov/data-research/consumer-complaints/

DOI: 10.1201/9781003093459-5

```
complaints %>%
  unnest_tokens(word, consumer_complaint_narrative) %>%
  anti_join(get_stopwords(), by = "word") %>%
  mutate(stem = wordStem(word)) %>%
  count(complaint_id, stem) %>%
  cast_dfm(complaint_id, stem, n)
```

```
#> Document-feature matrix of: 117,214 documents, 46,099
features (99.88% sparse) and 0 docvars.
```

A *sparse matrix* is a matrix where most of the elements are zero. When working with text data, we say our data is "sparse" because most documents do not contain most words, resulting in a representation of mostly zeroes. There are special data structures and algorithms for dealing with sparse data that can take advantage of their structure. For example, an array can more efficiently store the locations and values of only the non-zero elements instead of all elements.

The data set of consumer complaints used in this book has been filtered to those submitted to the CFPB since January 1, 2019 that include a consumer complaint narrative (i.e., some submitted text).

Another way to represent our text data is to create a sparse matrix where the elements are weighted, rather than straightforward counts only. The *term frequency* of a word is how frequently a word occurs in a document, and the *inverse document frequency* of a word decreases the weight for commonly-used words and increases the weight for words that are not used often in a collection of documents. It is typically defined as:

$$idf(\text{term}) = \ln\left(\frac{n_{\text{documents}}}{n_{\text{documents containing term}}}\right)$$

These two quantities can be combined to calculate a term's tf-idf (the two quantities multiplied together). This statistic measures the frequency of a term adjusted for how rarely it is used, and it is an example of a weighting scheme that can often work better than counts for predictive modeling with text features.

```
complaints %>%
  unnest_tokens(word, consumer_complaint_narrative) %>%
  anti_join(get_stopwords(), by = "word") %>%
  mutate(stem = wordStem(word)) %>%
  count(complaint_id, stem) %>%
  bind_tf_idf(stem, complaint_id, n) %>%
  cast_dfm(complaint_id, stem, tf_idf)
```

```
#> Document-feature matrix of: 117,214 documents, 46,099
features (99.88% sparse) and 0 docvars.
```

Notice that, in either case, our final data structure is incredibly sparse and of high dimensionality with a huge number of features. Some modeling algorithms and the libraries that implement them can take advantage of the memory characteristics of sparse matrices for better performance; an example of this is regularized regression implemented in **glmnet** (Friedman, Hastie, and Tibshirani 2010). Some modeling algorithms, including tree-based algorithms, do not perform better with sparse input, and then some libraries are not built to take advantage of sparse data structures, even if it would improve performance for those algorithms. We have some computational tools to take advantage of sparsity, but they don't always solve all the problems that come along with big text data sets.

As the size of a corpus increases in terms of words or other tokens, both the sparsity and RAM required to hold the corpus in memory increase. Figure 5.1 shows how this works out; as the corpus grows, there are more words used just a few times included in the corpus. The sparsity increases and approaches 100%, but even more notably, the memory required to store the corpus increases with the square of the number of tokens.

```
get_dfm <- function(frac) {
  complaints %>%
    sample_frac(frac) %>%
    unnest_tokens(word, consumer_complaint_narrative) %>%
    anti_join(get_stopwords(), by = "word") %>%
    mutate(stem = wordStem(word)) %>%
    count(complaint_id, stem) %>%
    cast_dfm(complaint_id, stem, n)
}

set.seed(123)
tibble(frac = 2 ^ seq(-16, -6, 2)) %>%
```

```
mutate(dfm = map(frac, get_dfm),
       words = map_dbl(dfm, quanteda::nfeat),
       sparsity = map_dbl(dfm, quanteda::sparsity),
       `RAM (in bytes)` = map_dbl(dfm, lobstr::obj_size)) %>%
pivot_longer(sparsity:`RAM (in bytes)`, names_to = "measure") %>%
ggplot(aes(words, value, color = measure)) +
geom_line(size = 1.5, alpha = 0.5) +
geom_point(size = 2) +
facet_wrap(~measure, scales = "free_y") +
scale_x_log10(labels = scales::label_comma()) +
scale_y_continuous(labels = scales::label_comma()) +
theme(legend.position = "none") +
labs(x = "Number of unique words in corpus (log scale)",
     y = NULL)
```

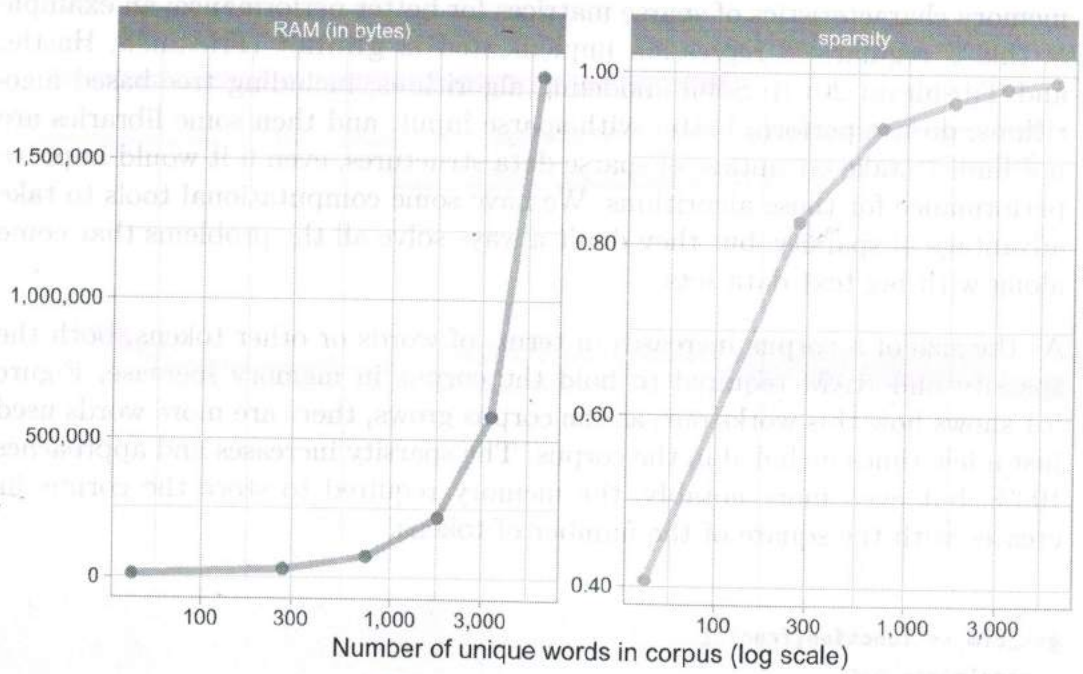



**FIGURE 5.1:** Memory requirements and sparsity increase with corpus size

Linguists have long worked on vector models for language that can reduce the number of dimensions representing text data based on how people use language; the quote that opened this chapter dates to 1957. These kinds of dense word vectors are often called *word embeddings.*

## 5.2 Understand word embeddings by finding them yourself

Word embeddings are a way to represent text data as vectors of numbers based on a huge corpus of text, capturing semantic meaning from words' context.

Modern word embeddings are based on a statistical approach to modeling language, rather than a linguistics or rules-based approach.

gah, how annoying

We can determine these vectors for a corpus of text using word counts and matrix factorization, as outlined by Moody (2017). This approach is valuable because it allows practitioners to find word vectors for their own collections of text (with no need to rely on pre-trained vectors) using familiar techniques that are not difficult to understand. Let's walk through how to do this using tidy data principles and sparse matrices, on the data set of CFPB complaints.

? "Matrix factorization is a filtering algorithm that sorts matrices

(1) First, let's filter out words that are used only rarely in this data set and create a nested dataframe, with one row per complaint.

```
tidy_complaints <- complaints %>%
  select(complaint_id, consumer_complaint_narrative) %>%
  unnest_tokens(word, consumer_complaint_narrative) %>%
  add_count(word) %>%
  filter(n >= 50) %>%
  select(-n)

nested_words <- tidy_complaints %>%
  nest(words = c(word))

nested_words
```

couldn't read in

```
#> # A tibble: 117,170 x 2
#>    complaint_id words
#>           <dbl> <list>
#>  1      3384392 <tibble [18 x 1]>
#>  2      3417821 <tibble [71 x 1]>
#>  3      3433198 <tibble [77 x 1]>
#>  4      3366475 <tibble [69 x 1]>
```

```
#>  5      3385399 <tibble [213 x 1]>
#>  6      3444592 <tibble [19 x 1]>
#>  7      3379924 <tibble [121 x 1]>
#>  8      3446975 <tibble [22 x 1]>
#>  9      3214857 <tibble [64 x 1]>
#> 10      3417374 <tibble [44 x 1]>
#> # ... with 117,160 more rows
```

Next, let's create a `slide_windows()` function, using the `slide()` function from the **slider** package (Vaughan 2021a) that implements fast sliding window computations written in C. Our new function identifies skipgram windows in order to calculate the skipgram probabilities, how often we find each word near each other word. We do this by defining a fixed-size moving window that centers around each word. Do we see `word1` and `word2` together within this window? We can calculate probabilities based on when we do or do not.

One of the arguments to this function is the `window_size`, which determines the size of the sliding window that moves through the text, counting up words that we find within the window. The best choice for this window size depends on your analytical question because it determines what kind of semantic meaning the embeddings capture. A smaller window size, like three or four, focuses on how the word is used and learns what other words are functionally similar. A larger window size, like 10, captures more information about the domain or topic of each word, not constrained by how functionally similar the words are (Levy and Goldberg 2014). A smaller window size is also faster to compute.

```
slide_windows <- function(tbl, window_size) {
  skipgrams <- slider::slide(
    tbl,
    ~.x,
    .after = window_size - 1,
    .step = 1,
    .complete = TRUE
  )

  safe_mutate <- safely(mutate)

  out <- map2(skipgrams,
              1:length(skipgrams),
              ~ safe_mutate(.x, window_id = .y))

  out %>%
    transpose() %>%
    pluck("result") %>%
```

```
    compact() %>%
    bind_rows()
}
```

Now that we can find all the skipgram windows, we can calculate how often words occur on their own, and how often words occur together with other words. We do this using the point-wise mutual information (PMI), a measure of association that measures exactly what we described in the previous sentence: it's the logarithm of the probability of finding two words together, normalized for the probability of finding each of the words alone. We use PMI to measure which words occur together more often than expected based on how often they occurred on their own.

For this example, let's use a window size of *four*.

This next step is the computationally expensive part of finding word embeddings with this method, and can take a while to run. Fortunately, we can use the **furrr** package (Vaughan and Dancho 2021) to take advantage of parallel processing because identifying skipgram windows in one document is independent from all the other documents.

```
library(widyr)
library(furrr)

plan(multisession)  ## for parallel processing

tidy_pmi <- nested_words %>%
  mutate(words = future_map(words, slide_windows, 4L)) %>%
  unnest(words) %>%
  unite(window_id, complaint_id, window_id) %>%
  pairwise_pmi(word, window_id)

tidy_pmi
```

```
#> # A tibble: 4,818,402 x 3
#>    item1    item2           pmi
#>    <chr>    <chr>         <dbl>
#>  1 systems transworld  7.09
```

printer failed to print
pg 80.

Notes:

- when PMI is high, two words are more probable to occur beside each other
- Next, ~~determine~~ turn PMI into those vectors (word embeddings) by doing SVD: singular value decomposition.

```
    nv = 100, maxit = 1000
  )

tidy_word_vectors
```

```
#> # A tibble: 747,500 x 3
#>    item1   dimension   value
#>    <chr>       <int>   <dbl>
#>  1 systems         1 0.0165
#>  2 inc             1 0.0191
#>  3 is              1 0.0202
#>  4 trying          1 0.0423
#>  5 to              1 0.00904
#>  6 collect         1 0.0370
#>  7 a               1 0.0126
#>  8 debt            1 0.0430
#>  9 that            1 0.0136
#> 10 not             1 0.0213
#> # ... with 747,490 more rows
```

`tidy_word_vectors` is not drastically smaller than `tidy_pmi` since the vocabulary is not enormous and `tidy_pmi` is represented in a sparse format.

We have now successfully found word embeddings, with clear and understandable code. This is a real benefit of this approach; this approach is based on counting, dividing, and matrix decomposition and is thus easier to understand and implement than options based on deep learning. Training word vectors or embeddings, even with this straightforward method, still requires a large data set (ideally, hundreds of thousands of documents or more) and a not insignificant investment of time and computational power.

## 5.3 Exploring CFPB word embeddings

Now that we have determined word embeddings for the data set of CFPB complaints, let's explore them and talk about how they are used in modeling. We have projected the sparse, high-dimensional set of word features into a more dense, 100-dimensional set of features.

Each word can be represented as a numeric vector in this new feature space. A single word is mapped to only one vector, so be aware that all senses of a word are conflated in word embeddings. Because of this, word embeddings are limited for understanding lexical semantics.

Which words are close to each other in this new feature space of word embeddings? Let's create a simple function that will find the nearest words to any given example in using our newly created word embeddings.

```
nearest_neighbors <- function(df, token) {
  df %>%
    widely(
      ~ {
        y <- .[rep(token, nrow(.)), ]
        res <- rowSums(. * y) /
          (sqrt(rowSums(. ^ 2)) * sqrt(sum(.[token, ] ^ 2)))

        matrix(res, ncol = 1, dimnames = list(x = names(res)))
        },
      sort = TRUE
    )(item1, dimension, value) %>%
    select(-item2)
}
```

This function takes the tidy word embeddings as input, along with a word (or token, more strictly) as a string. It uses matrix multiplication and sums to calculate the cosine similarity between the word and all the words in the embedding to find which words are closer or farther to the input word, and returns a dataframe sorted by similarity.

What words are closest to `"error"` in the data set of CFPB complaints, as determined by our word embeddings?

```
tidy_word_vectors %>%
    nearest_neighbors("error")
```

```
#> # A tibble: 7,475 x 2
#>    item1            value
```

```
#>    <chr>            <dbl>
#>  1 error            1
#>  2 mistake          0.683
#>  3 clerical         0.627
#>  4 problem          0.582
#>  5 glitch           0.580
#>  6 errors           0.571
#>  7 miscommunication 0.512
#>  8 misunderstanding 0.486
#>  9 issue            0.478
#> 10 discrepancy      0.474
#> # ... with 7,465 more rows
```

Mistakes, problems, glitches – sounds bad!

What is closest to the word `"month"`?

```
tidy_word_vectors %>%
    nearest_neighbors("month")
```

```
#> # A tibble: 7,475 x 2
#>    item1        value
#>    <chr>        <dbl>
#>  1 month        1
#>  2 year         0.607
#>  3 months       0.593
#>  4 monthly      0.454
#>  5 installments 0.446
#>  6 payment      0.429
#>  7 week         0.406
#>  8 weeks        0.400
#>  9 85.00        0.399
#> 10 bill         0.396
#> # ... with 7,465 more rows
```

We see words about installments and payments, along with other time periods such as years and weeks. Notice that we did not stem this text data (see Chapter 4), but the word embeddings learned that "month," "months," and "monthly" belong together.

What words are closest in this embedding space to `"fee"`?

```
tidy_word_vectors %>%
    nearest_neighbors("fee")
```

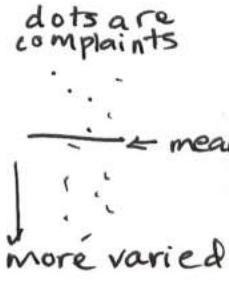
```
#> # A tibble: 7,475 x 2
#>    item1     value
#>    <chr>     <dbl>
#>  1 fee       1
#>  2 fees      0.746
#>  3 overdraft 0.678
#>  4 12.00     0.675
#>  5 14.00     0.645
#>  6 37.00     0.632
#>  7 charge    0.630
#>  8 11.00     0.630
#>  9 36.00     0.627
#> 10 28.00     0.624
#> # ... with 7,465 more rows
```

We find a lot of dollar amounts, which makes sense. Let us filter out the numbers to see what non-dollar words are similar to "fee."

```
tidy_word_vectors %>%
  nearest_neighbors("fee") %>%
  filter(str_detect(item1, "[0-9]*.[0-9]{2}", negate = TRUE))
```

```
#> # A tibble: 7,047 x 2
#>    item1     value
#>    <chr>     <dbl>
#>  1 fee       1
#>  2 fees      0.746
#>  3 overdraft 0.678
#>  4 charge    0.630
#>  5 nsf       0.609
#>  6 charged   0.594
#>  7 od        0.552
#>  8 waived    0.547
#>  9 assessed  0.538
#> 10 charges   0.530
#> # ... with 7,037 more rows
```

We now find words about overdrafts and charges. The top two words are "fee" and "fees"; word embeddings can learn that singular and plural forms of words are related and belong together. In fact, word embeddings can accomplish many of the same goals of tasks like stemming (Chapter 4) but more reliably and less arbitrarily.




Since we have found word embeddings via singular value decomposition, we can use these vectors to understand what principal components explain the most variation in the CFPB complaints. The orthogonal axes that SVD used to represent our data were chosen so that the first axis accounts for the most variance, the second axis accounts for the next most variance, and so on. We can now explore which and how much each *original* dimension (tokens in this case) contributed to each of the resulting principal components produced using SVD.

```
tidy_word_vectors %>%
    filter(dimension <= 24) %>%
    group_by(dimension) %>%
    top_n(12, abs(value)) %>%
    ungroup %>%
    mutate(item1 = reorder_within(item1, value, dimension)) %>%
    ggplot(aes(item1, value, fill = dimension)) +
    geom_col(alpha = 0.8, show.legend = FALSE) +
    facet_wrap(~dimension, scales = "free_y", ncol = 4) +
    scale_x_reordered() +
    coord_flip() +
    labs(
      x = NULL,
      y = "Value",
      title = "First 24 principal components for text of CFPB complaints",
      subtitle = paste("Top words contributing to the components that explain",
                       "the most variation")
    )
```

It becomes very clear in Figure 5.2 that stop words have not been removed, but notice that we can learn meaningful relationships in how very common words are used. Component 12 shows us how common prepositions are often used with words like `"regarding"`, `"contacted"`, and `"called"`, while component 9 highlights the use of *different* common words when submitting a complaint about unethical, predatory, and/or deceptive practices. Stop words do carry information, and methods like determining word embeddings can make that information usable.

We created word embeddings and can explore them to understand our text data set, but how do we use this vector representation in modeling? The

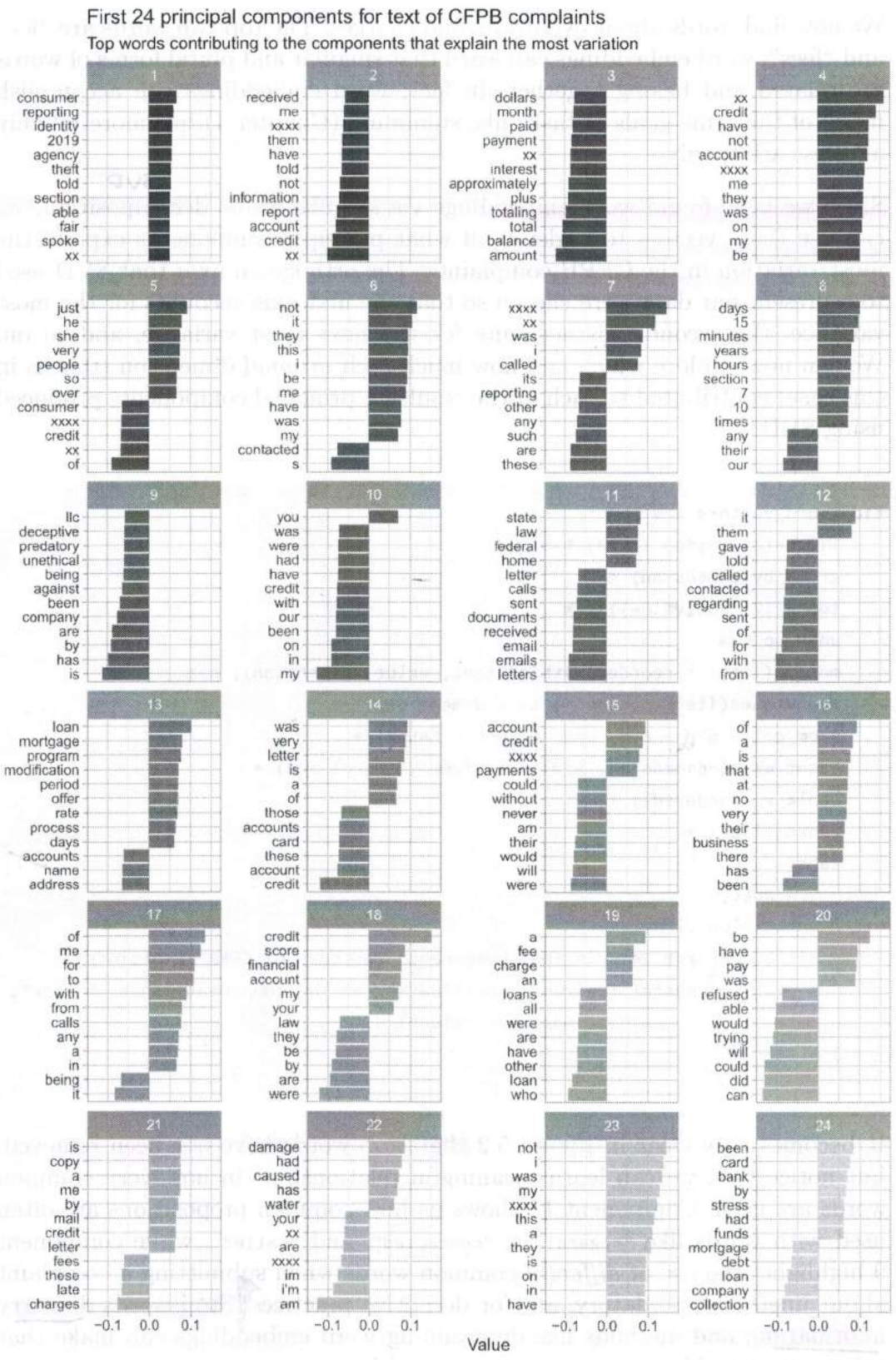



**FIGURE 5.2:** Word embeddings for Consumer Finance Protection Bureau complaints

classic and simplest approach is to treat each document as a collection of words and summarize the word embeddings into *document embeddings*, either using a mean or sum. This approach loses information about word order but is straightforward to implement. Let's `count()` to find the sum here in our example.

```
word_matrix <- tidy_complaints %>%
  count(complaint_id, word) %>%
  cast_sparse(complaint_id, word, n)

embedding_matrix <- tidy_word_vectors %>%
  cast_sparse(item1, dimension, value)

doc_matrix <- word_matrix %*% embedding_matrix

dim(doc_matrix)
```

```
#> [1] 117170    100
```

We have a new matrix here that we can use as the input for modeling. Notice that we still have over 100,000 documents (we did lose a few complaints, compared to our example sparse matrices at the beginning of the chapter, when we filtered out rarely used words) but instead of tens of thousands of features, we have exactly 100 features.

These hundred features are the word embeddings we learned from the text data itself.

If our word embeddings are of high quality, this translation of the high-dimensional space of words to the lower-dimensional space of the word embeddings allows our modeling based on such an input matrix to take advantage of the semantic meaning captured in the embeddings.

This is a straightforward method for finding and using word embeddings, based on counting and linear algebra. It is valuable both for understanding what word embeddings are and how they work, but also in many real-world applications. This is not the method to reach for if you want to publish an academic NLP paper, but is excellent for many applied purposes. Other methods for determining word embeddings include GloVe (Pennington, Socher, and Manning 2014), implemented in R in the **text2vec** package (Selivanov, Bickel, and Wang 2020), word2vec (Mikolov et al. 2013), and FastText (Bojanowski et al. 2017).

## 5.4 Use pre-trained word embeddings

If your data set is too small, you typically cannot train reliable word embeddings.

How small is too small? It is hard to make definitive statements because being able to determine useful word embeddings depends on the semantic and pragmatic details of *how* words are used in any given data set. However, it may be unreasonable to expect good results with data sets smaller than about a million words or tokens. (Here, we do not mean about a million unique tokens, i.e., the vocabulary size, but instead about that many observations in the text data.)

In such situations, we can still use word embeddings for feature creation in modeling, just not embeddings that we determine ourselves from our own data set. Instead, we can turn to *pre-trained* word embeddings, such as the GloVe word vectors trained on six billion tokens from Wikipedia and news sources. Several pre-trained GloVe vector representations are available in R via the **textdata** package (Hvitfeldt 2020b). Let's use `dimensions = 100`, since we trained 100-dimensional word embeddings in the previous section.

```
library(textdata)

glove6b <- embedding_glove6b(dimensions = 100)
glove6b
```

```
#> # A tibble: 400,000 x 101
#>    token      d1      d2      d3      d4      d5      d6      d7      d8      d9
#>    <chr>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
#>  1 "the" -0.0382 -0.245   0.728  -0.400   0.0832  0.0440 -0.391   0.334  -0.575
#>  2 ","   -0.108   0.111   0.598  -0.544   0.674   0.107   0.0389  0.355   0.0635
#>  3 "."   -0.340   0.209   0.463  -0.648  -0.384   0.0380  0.171   0.160   0.466
#>  4 "of"  -0.153  -0.243   0.898   0.170   0.535   0.488  -0.588  -0.180  -1.36
#>  5 "to"  -0.190   0.0500  0.191  -0.0492 -0.0897  0.210  -0.550   0.0984 -0.201
#>  6 "and" -0.0720  0.231   0.0237 -0.506   0.339   0.196  -0.329   0.184  -0.181
#>  7 "in"   0.0857 -0.222   0.166   0.134   0.382   0.354   0.0129  0.225  -0.438
#>  8 "a"   -0.271   0.0440 -0.0203 -0.174   0.644   0.712   0.355   0.471  -0.296
```

```
#>  9 "\""  -0.305  -0.236   0.176  -0.729  -0.283  -0.256   0.266   0.0253 -0.0748
#> 10 "'s"   0.589  -0.202   0.735  -0.683  -0.197  -0.180  -0.392   0.342  -0.606
#> # ... with 399,990 more rows, and 91 more variables: d10 <dbl>, d11 <dbl>,
#> #   d12 <dbl>, d13 <dbl>, d14 <dbl>, d15 <dbl>, d16 <dbl>, d17 <dbl>,
#> #   d18 <dbl>, ...
```

exact same values, good ✓

We can transform these word embeddings into a more tidy format, using `pivot_longer()` from **tidyr**. Let's also give this tidied version the same column names as `tidy_word_vectors`, for convenience.

```
tidy_glove <- glove6b %>%
  pivot_longer(contains("d"),
               names_to = "dimension") %>%
  rename(item1 = token)

tidy_glove
```

```
#> # A tibble: 40,000,000 x 3
#>    item1 dimension   value
#>    <chr> <chr>       <dbl>
#>  1 the   d1        -0.0382
#>  2 the   d2        -0.245
#>  3 the   d3         0.728
#>  4 the   d4        -0.400
#>  5 the   d5         0.0832
#>  6 the   d6         0.0440
#>  7 the   d7        -0.391
#>  8 the   d8         0.334
#>  9 the   d9        -0.575
#> 10 the   d10        0.0875
#> # ... with 39,999,990 more rows
```

~~newly name~~ ✓

~~each item is a new dimension?~~

'the' has 100 dimensions, next word has 100 d, etc.

got same values

We've already explored some sets of "synonyms" in the embedding space we determined ourselves from the CPFB complaints. What about this embedding space learned via the GloVe algorithm on a much larger data set? We just need to make one change to our `nearest_neighbors()` function and add `maximum_size = NULL`, because the matrices we are multiplying together are much larger this time. ✓

```
nearest_neighbors <- function(df, token) {
  df %>%
    widely(
      ~ {
        y <- .[rep(token, nrow(.)), ]
        res <- rowSums(. * y) /
          (sqrt(rowSums(. ^ 2)) * sqrt(sum(.[token, ] ^ 2)))
        matrix(res, ncol = 1, dimnames = list(x = names(res)))
        },
      sort = TRUE,
      maximum_size = NULL
    )(item1, dimension, value) %>%
    select(-item2)
}
```

Pre-trained word embeddings are trained on very large, general purpose English language data sets. Commonly used word2vec embeddings[3] are based on the Google News data set, and GloVe embeddings[4] (what we are using here) and FastText embeddings[5] are learned from the text of Wikipedia plus other sources. Keeping that in mind, what words are closest to `"error"` in the GloVe embeddings?

```
tidy_glove %>%
    nearest_neighbors("error")
```

```
#> # A tibble: 400,000 x 2
#>    item1       value
#>    <chr>       <dbl>
#>  1 error       1
#>  2 errors      0.792
#>  3 mistake     0.664
#>  4 correct     0.621
#>  5 incorrect   0.613
#>  6 fault       0.607
#>  7 difference  0.594
#>  8 mistakes    0.586
#>  9 calculation 0.584
#> 10 probability 0.583
#> # ... with 399,990 more rows
```

[3] https://code.google.com/archive/p/word2vec/

[4] https://nlp.stanford.edu/projects/glove/

[5] https://fasttext.cc/docs/en/english-vectors.html

Instead of problems and mistakes like in the CFPB embeddings, we now see words related to sports, especially baseball, where an error is a certain kind of act recorded in statistics. This could present a challenge for using the GloVe embeddings with the CFPB text data. Remember that different senses or uses of the same word are conflated in word embeddings; the high-dimensional space of any set of word embeddings cannot distinguish between different uses of a word, such as the word "error."

(2) What is closest to the word `"month"` in these pre-trained GloVe embeddings?

```
tidy_glove %>%
    nearest_neighbors("month")
```

```
#> # A tibble: 400,000 x 2
#>    item1     value
#>    <chr>     <dbl>
#>  1 month     1
#>  2 week      0.939
#>  3 last      0.924
#>  4 months    0.898
#>  5 year      0.893
#>  6 weeks     0.865
#>  7 earlier   0.859
#>  8 tuesday   0.846
#>  9 ago       0.844
#> 10 thursday  0.841
#> # ... with 399,990 more rows
```

Instead of words about payments, the GloVe results here focus on different time periods only.

(3) What words are closest in the GloVe embedding space to `"fee"`?

```
tidy_glove %>%
    nearest_neighbors("fee")
```

```
#> # A tibble: 400,000 x 2
#>    item1       value
#>    <chr>       <dbl>
#>  1 fee         1
#>  2 fees        0.832
#>  3 payment     0.741
```

```
#>  4 pay          0.711
#>  5 salary       0.700
#>  6 paid         0.668
#>  7 payments     0.653
#>  8 subscription 0.647
#>  9 paying       0.623
#> 10 expenses     0.619
#> # ... with 399,990 more rows
```

The most similar words are, like with the CPFB embeddings, generally financial, but they are largely about salary and pay instead of about charges and overdrafts.

These examples highlight how pre-trained word embeddings can be useful because of the incredibly rich semantic relationships they encode, but also how these vector representations are often less than ideal for specific tasks.

If we do choose to use pre-trained word embeddings, how do we go about integrating them into a modeling workflow? Again, we can create simple document embeddings by treating each document as a collection of words and summarizing the word embeddings. The GloVe embeddings do not contain all the tokens in the CPFB complaints, and vice versa, so let's use `inner_join()` to match up our data sets.

```
word_matrix <- tidy_complaints %>%
  inner_join(by = "word",
             tidy_glove %>%
               distinct(item1) %>%
               rename(word = item1)) %>%
  count(complaint_id, word) %>%
  cast_sparse(complaint_id, word, n)

glove_matrix <- tidy_glove %>%
  inner_join(by = "item1",
             tidy_complaints %>%
               distinct(word) %>%
               rename(item1 = word)) %>%
  cast_sparse(item1, dimension, value)

doc_matrix <- word_matrix %*% glove_matrix
```

```
dim(doc_matrix)
```

```
#> [1] 117163    100
```

Since these GloVe embeddings had the same number of dimensions as the word embeddings we found ourselves (100), we end up with the same number of columns as before but with slightly fewer documents in the data set. We have lost documents that contain only words not included in the GloVe embeddings.

The package **wordsalad** (Hvitfeldt 2020c) provides a unified interface for finding different kinds of word vectors from text using pre-trained embeddings. The options include fastText, GloVe, and word2vec.

## 5.5 Fairness and word embeddings

Perhaps more than any of the other preprocessing steps this book has covered so far, using word embeddings opens an analysis or model up to the possibility of being influenced by systemic unfairness and bias.

Embeddings are trained or learned from a large corpus of text data, and whatever human prejudice or bias exists in the corpus becomes imprinted into the vector data of the embeddings.

This is true of all machine learning to some extent (models learn, reproduce, and often amplify whatever biases exist in training data), but this is literally, concretely true of word embeddings. Caliskan, Bryson, and Narayanan (2017) show how the GloVe word embeddings (the same embeddings we used in Section 5.4) replicate human-like semantic biases.

- Typically Black first names are associated with more unpleasant feelings than typically white first names.

- Women's first names are more associated with family and men's first names are more associated with career.
- Terms associated with women are more associated with the arts and terms associated with men are more associated with science.

Results like these have been confirmed over and over again, such as when Bolukbasi et al. (2016) demonstrated gender stereotypes in how word embeddings encode professions or when Google Translate exhibited apparently sexist behavior when translating text from languages with no gendered pronouns[6]. Google has since worked to correct this problem[7], but in 2021 the problem still exists for some languages[8]. Garg et al. (2018) even used the way bias and stereotypes can be found in word embeddings to quantify how social attitudes towards women and minorities have changed over time.

Remember that word embeddings are *learned* or trained from some large data set of text; this training data is the source of the biases we observe when applying word embeddings to NLP tasks. Bender et al. (2021) outline how the very large data sets used in large language models do not mean that such models reflect representative or diverse viewpoints, or even can respond to changing social views. As one concrete example, a common data set used to train large embedding models is the text of Wikipedia, but Wikipedia itself has problems with, for example, gender bias[9]. Some of the gender discrepancies on Wikipedia can be attributed to social and historical factors, but some can be attributed to the site mechanics of Wikipedia itself (Wagner et al. 2016).

It's safe to assume that any large corpus of language will contain latent structure reflecting the biases of the people who generated that language.

When embeddings with these kinds of stereotypes are used as a preprocessing step in training a predictive model, the final model can exhibit racist, sexist, or otherwise biased characteristics. Speer (2017) demonstrated how using pre-trained word embeddings to train a straightforward sentiment analysis model can result in text such as

[6] https://twitter.com/seyyedreza/status/935291317252493312

[7] https://www.blog.google/products/translate/reducing-gender-bias-google-translate/

[8] https://twitter.com/doravargha/status/1373211762108076034

[9] https://en.wikipedia.org/wiki/Gender_bias_on_Wikipedia

Let's go get Italian food

being scored much more positively than text such as

Let's go get Mexican food

because of characteristics of the text the word embeddings were trained on.

## 5.6 Using word embeddings in the real world

Given these profound and fundamental challenges with word embeddings, what options are out there? First, consider not using word embeddings when building a text model. Depending on the particular analytical question you are trying to answer, another numerical representation of text data (such as word frequencies or tf-idf of single words or n-grams) may be more appropriate. Consider this option even more seriously if the model you want to train is already entangled with issues of bias, such as the sentiment analysis example in Section 5.5.

Consider whether finding your own word embeddings, instead of relying on pre-trained embeddings created using an algorithm like GloVe or word2vec, may help you. Building your own vectors is likely to be a good option when the text domain you are working in is *specific* rather than general purpose; some examples of such domains could include customer feedback for a clothing e-commerce site, comments posted on a coding Q&A site, or legal documents.

Learning good quality word embeddings is only realistic when you have a large corpus of text data (say, a million tokens), but if you have that much data, it is possible that embeddings learned from scratch based on your own data may not exhibit the same kind of semantic biases that exist in pre-trained word embeddings. Almost certainly there will be some kind of bias latent in any large text corpus, but when you use your own training data for learning word

embeddings, you avoid the problem of *adding* historic, systemic prejudice from general purpose language data sets.

You can use the same approaches discussed in this chapter to check any new embeddings for dangerous biases such as racism or sexism.

NLP researchers have also proposed methods for debiasing embeddings. Bolukbasi et al. (2016) aim to remove stereotypes by postprocessing pre-trained word vectors, choosing specific sets of words that are reprojected in the vector space so that some specific bias, such as gender bias, is mitigated. This is the most established method for reducing bias in embeddings to date, although other methods have been proposed as well, such as augmenting data with counterfactuals (Lu et al. 2020). Recent work (Ethayarajh, Duvenaud, and Hirst 2019) has explored whether the association tests used to measure bias are even useful, and under what conditions debiasing can be effective.

Other researchers, such as Caliskan, Bryson, and Narayanan (2017), suggest that corrections for fairness should happen at the point of *decision* or action rather than earlier in the process of modeling, such as preprocessing steps like building word embeddings. The concern is that methods for debiasing word embeddings may allow the stereotypes to seep back in, and more recent work shows that this is exactly what can happen. Gonen and Goldberg (2019) highlight how pervasive and consistent gender bias is across different word embedding models, *even after* applying current debiasing methods.

## 5.7 Summary

Mapping words (or other tokens) to an embedding in a special vector space is a powerful approach in natural language processing. This chapter started from fundamentals to demonstrate how to determine word embeddings from a text data set, but a whole host of highly sophisticated techniques have been built on this foundation. For example, document embeddings can be learned from text directly (Le and Mikolov 2014) rather than summarized from word embeddings. More recently, embeddings have acted as one part of language models with transformers like ULMFiT (Howard and Ruder 2018) and ELMo (Peters et al. 2018). It's important to keep in mind that even more advanced natural language algorithms, such as these language models with transformers, also exhibit such systemic biases (Sheng et al. 2019).

### 5.7.1 In this chapter, you learned:

- what word embeddings are and why we use them
- how to determine word embeddings from a text data set
- how the vector space of word embeddings encodes word similarity
- about a simple strategy to find document similarity
- how to handle pre-trained word embeddings
- why word embeddings carry historic and systemic bias
- about approaches for debiasing word embeddings

# Part II

# Machine Learning Methods

# *Overview*

It's time to use what we have discussed and learned in the first five chapters of this book in a supervised machine learning context, to make predictions from text data. In the next two chapters, we will focus on putting into practice such machine learning algorithms as:

- naive Bayes,
- support vector machines (SVM) (Boser, Guyon, and Vapnik 1992), and
- regularized linear models such as implemented in glmnet[10] (Friedman, Hastie, and Tibshirani 2010).

We start in Chapter 6 with exploring regression models and continue in Chapter 7 with classification models. These are different types of prediction problems, but in both, we can use the tools of supervised machine learning to connect our *input*, which may exist entirely or partly as text data, with our *outcome* of interest. Most supervised models for text data are built with one of three purposes in mind:

- The main goal of a **predictive model** is to generate the most accurate predictions possible.
- An **inferential model** is created to test a hypothesis or draw conclusions about a population.
- The main purpose of a **descriptive model** is to describe the properties of the observed data.

Many learning algorithms can be used for more than one of these purposes. Concerns about a model's predictive capacity may be as important for an inferential or descriptive model as for a model designed purely for prediction, and model interpretability and explainability may be important for a solely predictive or descriptive model as well as for an inferential model. We will use

[10] https://glmnet.stanford.edu/

the tidymodels[11] framework to address all of these issues, with its consistent approach to resampling, preprocessing, fitting, and evaluation.

The **tidymodels** framework (Kuhn and Wickham 2021a) is a collection of R packages for modeling and machine learning using tidyverse principles (Wickham et al. 2019). These packages facilitate resampling, preprocessing, modeling, and evaluation. There are core packages that you can load all together via `library(tidymodels)` and then extra packages for more specific tasks.

As you read through these next chapters, notice the modeling *process* moving through these stages; we'll discuss the structure of this process in more detail in the overview for the deep learning chapters.

Before we starting fitting these models to real data sets, let's consider how to think about algorithmic bias for predictive modeling. Rachel Thomas proposed a checklist at ODSC West 2019[12] for algorithmic bias in machine learning.

## Should we even be doing this?

This is always the first step. Machine learning algorithms involve math and data, but that does not mean they are neutral. They can be used for purposes that are helpful, harmful, or even unethical.

## What bias is already in the data?

Chapter 6 uses a data set of United States Supreme Court opinions, with an uneven distribution of years. There are many more opinions from more recent decades than from earlier ones. Bias like this is extremely common in data sets and must be considered in modeling. In this case, we show how using regularized linear models results in better predictions across years than other approaches (Section 6.3).

[11] `https://www.tidymodels.org/`

[12] `https://opendatascience.com/odsc-west-2019-keynote-rachel-thomas-on-algorithmic-bias/`

## Can the code and data be audited?

In the case of this book, the code and data are all publicly available. You as a reader can audit our methods and what kinds of bias exist in the data sets. When you take what you have learned in this book and apply it your real-world work, consider how accessible your code and data are to internal and external stakeholders.

## What are the error rates for sub-groups?

In Section 7.6 we demonstrate how to measure model performance for a multiclass classifier, but you can also compute model metrics for sub-groups that are not explicitly in your model as class labels or predictors. Using tidy data principles and the **yardstick** package makes this task well within the reach of data practitioners.

In **tidymodels**, the **yardstick** package (Kuhn and Vaughan 2021a) has functions for model evaluation.

## What is the accuracy of a simple rule-based alternative?

Chapter 7 shows how to train models to predict the category of a user complaint using sophisticated preprocessing steps and machine learning algorithms, but such a complaint could be categorized using simple regular expressions (Appendix A), perhaps combined with other rules. Straightforward heuristics are easy to implement, maintain, and audit, compared to machine learning models; consider comparing the accuracy of models to simpler options.

### What processes are in place to handle appeals or mistakes?

If models such as those built in Chapter 7 were put into production by an organization, what would happen if a complaint was classified incorrectly? We as data practitioners typically (hopefully) have a reasonable estimate of the true positive rate and true negative rate for models we train, so processes to handle misclassifications can be built with a good understanding of how often they will be used. legal gray area? outcomes affect peoples' lives in what ways? how much risk is too much?

### How diverse is the team that built it?

The two-person team that wrote this book includes perspectives from a man and woman, and from someone who has always lived inside the United States and someone who is from a European country. However, we are both white with similar educational backgrounds. We must be aware of how the limited life experiences of individuals training and assessing machine learning models can cause unintentional harm.

"WEIRD"

# 6

# Regression

In this chapter, we will use machine learning to predict *continuous values* that are associated with text data. Like in all predictive modeling tasks, this chapter demonstrates how to use learning algorithms to find and model relationships between an outcome or target variable and other input features. What is unique about the focus of this book is that our features are created from text data following the techniques laid out in Chapters 1 through 5, and what is unique about the focus of this particular chapter is that our outcome is numeric and continuous. For example, let's consider a sample of opinions from the United States Supreme Court, available in the **scotus** (Hvitfeldt 2019b) package.

```
library(tidyverse)
library(scotus)

scotus_filtered %>%
  as_tibble()
```

```
#> # A tibble: 10,000 x 5
#>    year  case_name                  docket_number     id text
#>    <chr> <chr>                      <chr>          <dbl> <chr>
#>  1 1903  Clara Perry, Plff. In Err~ 16             80304 "No. 16.\n State Repor~
#>  2 1987  West v. Conrail            85-1804        96216 "No. 85-1804.\n\n     ~
#>  3 1957  Roth v. United States      582            89930 "Nos. 582, 61.\nNo. 61~
#>  4 1913  McDermott v. Wisconsin     Nos. 112 and ~ 82218 "Nos. 112 and 113.\nMr~
#>  5 1826  Wetzell v. Bussard         <NA>           52899 "Feb. 7th.\nThis cause~
#>  6 1900  Forsyth v. Vehmeyer       180            79609 "No. 180.\nMr. Edward ~
#>  7 1871  Reed v. United States     <NA>           57846 "APPEAL and cross appe~
#>  8 1833  United States v. Mills    <NA>           53394 "CERTIFICATE of Divisi~
#>  9 1940  Puerto Rico v. Rubert Her~ 582           87714 "No. 582.\nMr. Wm. Cat~
#> 10 1910  Williams v. First Nat. Ba~ 130           81588 "No. 130.\nThe defenda~
#> # ... with 9,990 more rows
```

This data set contains the entire text of each opinion in the `text` column, along with the `case_name` and `docket_number`. Notice that we also have the `year` that

DOI: 10.1201/9781003093459-6

each case was decided by the Supreme Court; this is basically a continuous variable (rather than a group membership of discrete label).

If we want to build a model to predict which court opinions were written in which years, we would build a regression model.

- A **classification model** predicts a class label or group membership.
- A **regression model** predicts a numeric or continuous value.

In text modeling, we use text data (such as the text of the court opinions), sometimes combined with other structured, non-text data, to predict the continuous value of interest (such as year of the court opinion). The goal of predictive modeling with text input features and a continuous outcome is to learn and model the relationship between the input features and the numeric target (outcome).

## 6.1 A first regression model

Let's build our first regression model using this sample of Supreme Court opinions. Before we start, let's check out how many opinions we have for each decade in Figure 6.1.

```
scotus_filtered %>%
  mutate(year = as.numeric(year),
         year = 10 * (year %/% 10)) %>%
  count(year) %>%
  ggplot(aes(year, n)) +
  geom_col() +
  labs(x = "Year", y = "Number of opinions per decade")
```

This sample of opinions reflects the distribution over time of available opinions for analysis; there are many more opinions per year in this data set after about 1850 than before. This is an example of bias already in our data, as we discussed in the overview to these chapters, and we will need to account for that in choosing a model and understanding our results.

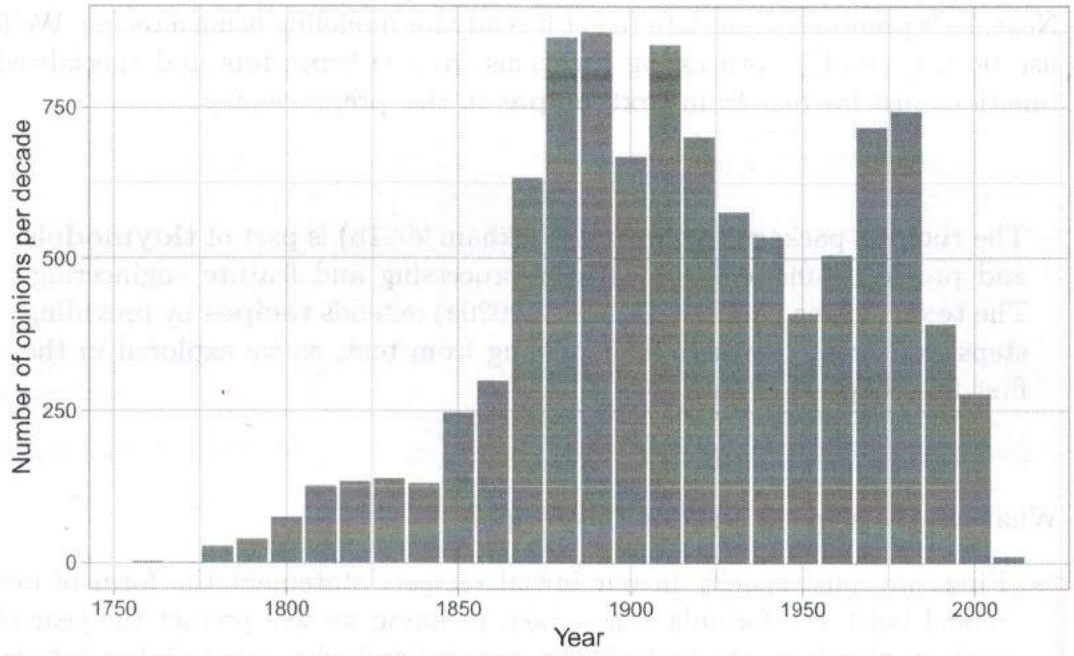



**FIGURE 6.1:** Supreme Court opinions per decade in sample

### 6.1.1 Building our first regression model

Our first step in building a model is to split our data into training and testing sets. We use functions from **tidymodels** for this; we use `initial_split()` to set up *how* to split the data, and then we use the functions `training()` and `testing()` to create the data sets we need. Let's also convert the year to a numeric value since it was originally stored as a character, and remove the ' character because of its effect on one of the models[1] we want to try out.

```
library(tidymodels)
set.seed(1234)
scotus_split <- scotus_filtered %>%
  mutate(year = as.numeric(year),
         text = str_remove_all(text, "'")) %>%
  initial_split()

scotus_train <- training(scotus_split)
scotus_test <- testing(scotus_split)
```

[1]The random forest implementation in the **ranger** package, demonstrated in Section 6.3, does not handle special characters in columns names well.

Next, let's preprocess our data to get it ready for modeling using a recipe. We'll use both general preprocessing functions from **tidymodels** and specialized functions just for text from **textrecipes** in this preprocessing.

The **recipes** package (Kuhn and Wickham 2021b) is part of **tidymodels** and provides functions for data preprocessing and feature engineering. The **textrecipes** package (Hvitfeldt 2020a) extends **recipes** by providing steps that create features for modeling from text, as we explored in the first five chapters of this book.

What are the steps in creating this recipe?

- First, we must specify in our initial `recipe()` statement the form of our model (with the formula `year ~ text`, meaning we will predict the year of each opinion from the text of that opinion) and what our training data is.
- Then, we tokenize (Chapter 2) the text of the court opinions.
- Next, we filter to only keep the top 1000 tokens by term frequency. We filter out those less frequent words because we expect them to be too rare to be reliable, at least for our first attempt. (We are *not* removing stop words yet; we'll explore removing them in Section 6.4.)
- The recipe step `step_tfidf()`, used with defaults here, weights each token frequency by the inverse document frequency.
- As a last step, we normalize (center and scale) these tf-idf values. This centering and scaling is needed because we're going to use a support vector machine model.

```
library(textrecipes)

scotus_rec <- recipe(year ~ text, data = scotus_train) %>%
  step_tokenize(text) %>%
  step_tokenfilter(text, max_tokens = 1e3) %>%
  step_tfidf(text) %>%
  step_normalize(all_predictors())

scotus_rec
```

```
#> Data Recipe
#>
#> Inputs:
#>
#>       role #variables
#>    outcome           1
#>  predictor           1
#>
#> Operations:
#>
#> Tokenization for text
#> Text filtering for text
#> Term frequency-inverse document frequency with text
#> Centering and scaling for all_predictors()
```

Now that we have a full specification of the preprocessing recipe, we can `prep()` this recipe to estimate all the necessary parameters for each step using the training data and `bake()` it to apply the steps to data, like the training data (with `new_data = NULL`), testing data, or new data at prediction time.

```
scotus_prep <- prep(scotus_rec)
scotus_bake <- bake(scotus_prep, new_data = NULL)

dim(scotus_bake)
```

```
#> [1] 7500 1001
```

For most modeling tasks, you will not need to `prep()` or `bake()` your recipe directly; instead you can build up a tidymodels `workflow()` to bundle together your modeling components.

In **tidymodels**, the **workflows** package (Vaughan 2021b) offers infrastructure for bundling model components. A *model workflow* is a convenient way to combine different modeling components (a preprocessor plus a model specification); when these are bundled explicitly, it can be easier to keep track of your modeling plan, as well as fit your model and predict on new data.

Let's create a `workflow()` to bundle together our recipe with any model specifications we may want to create later. First, let's create an empty `workflow()` and then only add the data preprocessor `scotus_rec` to it.

```
scotus_wf <- workflow() %>%
  add_recipe(scotus_rec)

scotus_wf
```

```
#> == Workflow ====================================================================
#> Preprocessor: Recipe
#> Model: None
#>
#> -- Preprocessor ----------------------------------------------------------------
#> 4 Recipe Steps
#>
#> * step_tokenize()
#> * step_tokenfilter()
#> * step_tfidf()
#> * step_normalize()
```

Notice that there is no model yet: `Model: None`. It's time to specify the model we will use! Let's build a support vector machine (SVM) model. While they don't see widespread use in cutting-edge machine learning research today, they are frequently used in practice and have properties that make them well-suited for text classification (Joachims 1998) and can give good performance (Van-Tu and Anh-Cuong 2016).

An SVM model can be used for either regression or classification, and linear SVMs often work well with text data. Even better, linear SVMs typically do not need to be tuned (see Section 7.4 for tuning model hyperparameters).

Before fitting, we set up a model specification. There are three components to specifying a model using tidymodels: the model algorithm (a linear SVM here), the mode (typically either classification or regression), and the computational engine we are choosing to use. For our linear SVM, let's use the **LiblineaR** engine (Helleputte 2021).

```
svm_spec <- svm_linear() %>%
  set_mode("regression") %>%
  set_engine("LiblineaR")
```

Everything is now ready for us to fit our model. Let's add our model to the workflow with `add_model()` and fit to our training data `scotus_train`.

```
svm_fit <- scotus_wf %>%
  add_model(svm_spec) %>%
  fit(data = scotus_train)
```

We have successfully fit an SVM model to this data set of Supreme Court opinions. What does the result look like? We can access the fit using `pull_workflow_fit()`, and even `tidy()` the model coefficient results into a convenient dataframe format.

```
svm_fit %>%
  pull_workflow_fit() %>%
  tidy() %>%
  arrange(-estimate)
```

```
#> # A tibble: 1,001 x 2
#>    term                   estimate
#>    <chr>                     <dbl>
#>  1 Bias                     1920.
#>  2 tfidf_text_later            1.50
#>  3 tfidf_text_appeals          1.48
#>  4 tfidf_text_see              1.39
#>  5 tfidf_text_noted            1.38
#>  6 tfidf_text_example          1.27
#>  7 tfidf_text_petitioner       1.26
#>  8 tfidf_text_even             1.23
#>  9 tfidf_text_rather           1.21
#> 10 tfidf_text_including        1.13
#> # ... with 991 more rows
```

The term `Bias` here means the same thing as an intercept. We see here what terms contribute to a Supreme Court opinion being written more recently, like "appeals" and "petitioner."

What terms contribute to a Supreme Court opinion being written further in the past, for this first attempt at a model?

```
svm_fit %>%
  pull_workflow_fit() %>%
  tidy() %>%
  arrange(estimate)
```

```
#> # A tibble: 1,001 x 2
#>    term                 estimate
#>    <chr>                   <dbl>
#>  1 tfidf_text_ought        -2.77
#>  2 tfidf_text_1st          -1.94
#>  3 tfidf_text_but          -1.63
#>  4 tfidf_text_same         -1.62
#>  5 tfidf_text_the          -1.57
#>  6 tfidf_text_therefore    -1.54
#>  7 tfidf_text_it           -1.46
#>  8 tfidf_text_which        -1.40
#>  9 tfidf_text_this         -1.39
#> 10 tfidf_text_be           -1.33
#> # ... with 991 more rows
```

Here we see words like "ought" and "therefore."

### 6.1.2 Evaluation

One option for evaluating our model is to predict one time on the testing set to measure performance.

> The testing set is extremely valuable data, however, and in real-world situations, we advise that you only use this precious resource one time (or at most, twice).

The purpose of the testing data is to estimate how your final model will perform on new data; we set aside a proportion of the data available and pretend that it is not available to us for training the model so we can use it to estimate model performance on strictly out-of-sample data. Often during the process of modeling, we want to compare models or different model parameters. If we use the test set for these kinds of tasks, we risk fooling ourselves that we are doing better than we really are.

Another option for evaluating models is to predict one time on the training set to measure performance. This is the *same data* that was used to train the model, however, and evaluating on the training data often results in performance estimates that are too optimistic. This is especially true for powerful machine learning algorithms that can learn subtle patterns from data; we risk overfitting to the training set.

Yet another option for evaluating or comparing models is to use a separate validation set. In this situation, we split our data *not* into two sets (training and testing) but into three sets (testing, training, and validation). The validation set is used for computing performance metrics to compare models or model parameters. This can be a great option if you have enough data for it, but often we as machine learning practitioners are not so lucky.

What are we to do, then, if we want to train multiple models and find the best one? Or compute a reliable estimate for how our model has performed without wasting the valuable testing set? We can use **resampling**. When we resample, we create new simulated data sets from the training set for the purpose of, for example, measuring model performance.

Let's estimate the performance of the linear SVM regression model we just fit. We can do this using resampled data sets built from the training set.

In **tidymodels**, the package for data splitting and resampling is **rsample** (Silge et al. 2021).

Let's create 10-fold cross-validation sets, and use these resampled sets for performance estimates.

```
set.seed(123)
scotus_folds <- vfold_cv(scotus_train)

scotus_folds
```

```
#> #  10-fold cross-validation
#> # A tibble: 10 x 2
#>    splits             id
#>    <list>             <chr>
#>  1 <split [6750/750]> Fold01
#>  2 <split [6750/750]> Fold02
#>  3 <split [6750/750]> Fold03
```

```
#>  4 <split [6750/750]> Fold04
#>  5 <split [6750/750]> Fold05
#>  6 <split [6750/750]> Fold06
#>  7 <split [6750/750]> Fold07
#>  8 <split [6750/750]> Fold08
#>  9 <split [6750/750]> Fold09
#> 10 <split [6750/750]> Fold10
```

Each of these "splits" contains information about how to create cross-validation folds from the original training data. In this example, 90% of the training data is included in each fold for analysis and the other 10% is held out for assessment. Since we used cross-validation, each Supreme Court opinion appears in only one of these held-out assessment sets.

In Section 6.1.1, we fit one time to the training data as a whole. Now, to estimate how well that model performs, let's fit many times, once to each of these resampled folds, and then evaluate on the heldout part of each resampled fold.

```
set.seed(123)
svm_rs <- fit_resamples(
  scotus_wf %>% add_model(svm_spec),
  scotus_folds,
  control = control_resamples(save_pred = TRUE)
)

svm_rs
```

```
#> # Resampling results
#> # 10-fold cross-validation
#> # A tibble: 10 x 5
#>    splits             id     .metrics         .notes           .predictions
#>    <list>             <chr>  <list>           <list>           <list>
#>  1 <split [6750/750]> Fold01 <tibble [2 x 4]> <tibble [0 x 1]> <tibble [750 x 4~
#>  2 <split [6750/750]> Fold02 <tibble [2 x 4]> <tibble [0 x 1]> <tibble [750 x 4~
#>  3 <split [6750/750]> Fold03 <tibble [2 x 4]> <tibble [0 x 1]> <tibble [750 x 4~
#>  4 <split [6750/750]> Fold04 <tibble [2 x 4]> <tibble [0 x 1]> <tibble [750 x 4~
#>  5 <split [6750/750]> Fold05 <tibble [2 x 4]> <tibble [0 x 1]> <tibble [750 x 4~
#>  6 <split [6750/750]> Fold06 <tibble [2 x 4]> <tibble [0 x 1]> <tibble [750 x 4~
#>  7 <split [6750/750]> Fold07 <tibble [2 x 4]> <tibble [0 x 1]> <tibble [750 x 4~
#>  8 <split [6750/750]> Fold08 <tibble [2 x 4]> <tibble [0 x 1]> <tibble [750 x 4~
#>  9 <split [6750/750]> Fold09 <tibble [2 x 4]> <tibble [0 x 1]> <tibble [750 x 4~
#> 10 <split [6750/750]> Fold10 <tibble [2 x 4]> <tibble [0 x 1]> <tibble [750 x 4~
```

These results look a lot like the resamples, but they have some additional columns, like the `.metrics` that we can use to measure how well this model performed and the `.predictions` we can use to explore that performance more deeply. What results do we see, in terms of performance metrics?

```
collect_metrics(svm_rs)
```

```
#> # A tibble: 2 x 6
#>   .metric .estimator   mean     n std_err .config
#>   <chr>   <chr>       <dbl> <int>   <dbl> <chr>
#> 1 rmse    standard   15.6      10 0.216   Preprocessor1_Model1
#> 2 rsq     standard    0.895    10 0.00244 Preprocessor1_Model1
```

The default performance metrics to be computed for regression models are RMSE (root mean squared error) and $R^2$ (coefficient of determination). RMSE is a metric that is in the same units as the original data, so in units of *years*, in our case; the RMSE of this first regression model is 15.6 years.

RSME and $R^2$ are performance metrics used for regression models.

RSME is a measure of the difference between the predicted and observed values; if the model fits the data well, RMSE is lower. To compute RMSE, you take the mean values of the squared difference between the predicted and observed values, then take the square root.

$R^2$ is the squared correlation between the predicted and observed values. When the model fits the data well, the predicted and observed values are closer together with a higher correlation between them. The correlation between two variables is bounded between $-1$ and 1, so the closer $R^2$ is to one, the better.

These values are quantitative estimates for how well our model performed, and can be compared across different kinds of models. Figure 6.2 shows the predicted years for these Supreme Court opinions plotted against the true years when they were published, for all the resampled data sets.

```
svm_rs %>%
  collect_predictions() %>%
  ggplot(aes(year, .pred, color = id)) +
```

```
geom_abline(lty = 2, color = "gray80", size = 1.5) +
geom_point(alpha = 0.3) +
labs(
  x = "Truth",
  y = "Predicted year",
  color = NULL,
  title = "Predicted and true years for Supreme Court opinions",
  subtitle = "Each cross-validation fold is shown in a different color"
)
```

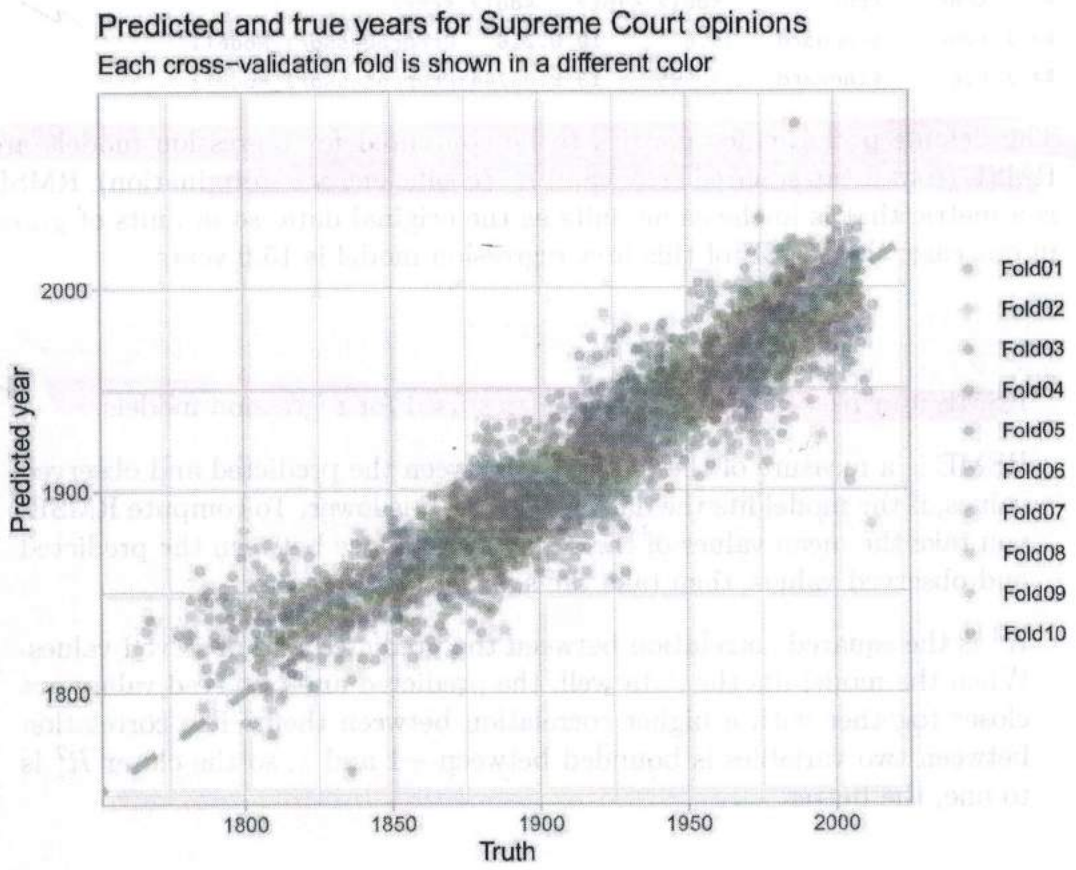



**FIGURE 6.2:** Most Supreme Court opinions are near the dashed line, indicating good agreement between our SVM regression predictions and the real years

The average spread of points in this plot above and below the dashed line corresponds to RMSE, which is 15.6 years for this model. When RMSE is better (lower), the points will be closer to the dashed line. This first model we have tried did not do a great job for Supreme Court opinions from before 1850, but for opinions after 1850, this looks pretty good!

Hopefully you are convinced that using resampled data sets for measuring performance is the right choice, but it can be computationally expensive. Instead of fitting once, we must fit the model one time for *each* resample. The resamples are independent of each other, so this is a great fit for parallel processing. The tidymodels framework is designed to work fluently with parallel processing in R, using multiple cores or multiple machines. The implementation details of parallel processing are operating system specific, so look at tidymodels' documentation for how to get started[2].

## 6.2 Compare to the null model

One way to assess a model like this one is to compare its performance to a "null model."

A null model is a simple, non-informative model that always predicts the largest class (for classification) or the mean (such as the mean year of Supreme Court opinions, in our specific regression case)[3].

We can use the same function `fit_resamples()` and the same preprocessing recipe as before, switching out our SVM model specification for the `null_model()` specification.

```
null_regression <- null_model() %>%
  set_engine("parsnip") %>%
  set_mode("regression")

null_rs <- fit_resamples(
  scotus_wf %>% add_model(null_regression),
  scotus_folds,
```

[2] https://tune.tidymodels.org/articles/extras/optimizations.html

[3] This is sometimes called a "baseline model."

```
  metrics = metric_set(rmse)
)

null_rs
```

```
#> # Resampling results
#> # 10-fold cross-validation
#> # A tibble: 10 x 4
#>    splits             id     .metrics          .notes
#>    <list>             <chr>  <list>            <list>
#>  1 <split [6750/750]> Fold01 <tibble [1 x 4]> <tibble [0 x 1]>
#>  2 <split [6750/750]> Fold02 <tibble [1 x 4]> <tibble [0 x 1]>
#>  3 <split [6750/750]> Fold03 <tibble [1 x 4]> <tibble [0 x 1]>
#>  4 <split [6750/750]> Fold04 <tibble [1 x 4]> <tibble [0 x 1]>
#>  5 <split [6750/750]> Fold05 <tibble [1 x 4]> <tibble [0 x 1]>
#>  6 <split [6750/750]> Fold06 <tibble [1 x 4]> <tibble [0 x 1]>
#>  7 <split [6750/750]> Fold07 <tibble [1 x 4]> <tibble [0 x 1]>
#>  8 <split [6750/750]> Fold08 <tibble [1 x 4]> <tibble [0 x 1]>
#>  9 <split [6750/750]> Fold09 <tibble [1 x 4]> <tibble [0 x 1]>
#> 10 <split [6750/750]> Fold10 <tibble [1 x 4]> <tibble [0 x 1]>
```

What results do we obtain from the null model, in terms of performance metrics?

```
collect_metrics(null_rs)
```

```
#> # A tibble: 1 x 6
#>   .metric .estimator  mean     n std_err .config
#>   <chr>   <chr>      <dbl> <int>   <dbl> <chr>
#> 1 rmse    standard    47.9    10   0.294 Preprocessor1_Model1
```

The RMSE indicates that this null model is dramatically worse than our first model. Even our first very attempt at a regression model (using only unigrams and very little specialized preprocessing) did much better than the null model; the text of the Supreme Court opinions has enough information in it related to the year the opinions were published that we can build successful models.

## 6.3 Compare to a random forest model

Random forest models are broadly used in predictive modeling contexts because they are low-maintenance and perform well. For example, see Caruana, Karampatziakis, and Yessenalina (2008) and Olson et al. (2018) for comparisons of the performance of common models such as random forest, decision tree, support vector machines, etc. trained on benchmark data sets; random forest models were one of the best overall. Let's see how a random forest model performs with our data set of Supreme Court opinions.

First, let's build a random forest model specification, using the ranger implementation. Random forest models are known for performing well without hyperparameter tuning, so we will just make sure we have enough `trees`.

```
rf_spec <- rand_forest(trees = 1000) %>%
  set_engine("ranger") %>%
  set_mode("regression")

rf_spec
```

```
#> Random Forest Model Specification (regression)
#>
#> Main Arguments:
#>   trees = 1000
#>
#> Computational engine: ranger
```

Now we can fit this random forest model. Let's use `fit_resamples()` again, so we can evaluate the model performance. We will use three arguments to this function:

- Our modeling `workflow()`, with the same preprocessing recipe we have been using so far in this chapter plus our new random forest model specification
- Our cross-validation resamples of the Supreme Court opinions
- A `control` argument to specify that we want to keep the predictions, to explore after fitting

```
rf_rs <- fit_resamples(
  scotus_wf %>% add_model(rf_spec),
  scotus_folds,
  control = control_resamples(save_pred = TRUE)
)
```

We can use `collect_metrics()` to obtain and format the performance metrics for this random forest model.

```
collect_metrics(rf_rs)
```

```
#> # A tibble: 2 x 6
#>   .metric .estimator   mean     n std_err .config
#>   <chr>   <chr>       <dbl> <int>   <dbl> <chr>
#> 1 rmse    standard   15.0      10 0.264   Preprocessor1_Model1
#> 2 rsq     standard    0.919    10 0.00283 Preprocessor1_Model1
```

This looks pretty promising, so let's explore the predictions for this random forest model.

```
collect_predictions(rf_rs) %>%
  ggplot(aes(year, .pred, color = id)) +
  geom_abline(lty = 2, color = "gray80", size = 1.5) +
  geom_point(alpha = 0.3) +
  labs(
    x = "Truth",
    y = "Predicted year",
    color = NULL,
    title = paste("Predicted and true years for Supreme Court opinions using",
                  "a random forest model", sep = "\n"),
    subtitle = "Each cross-validation fold is shown in a different color"
  )
```

Figure 6.3 shows some of the strange behavior from our fitted model. The overall performance metrics look pretty good, but predictions are too high and too low around certain threshold years.

It is very common to run into problems when using tree-based models like random forests with text data. One of the defining characteristics of text data is that it is *sparse*, with many features but most features not occurring in most observations. Tree-based models such as random forests are often

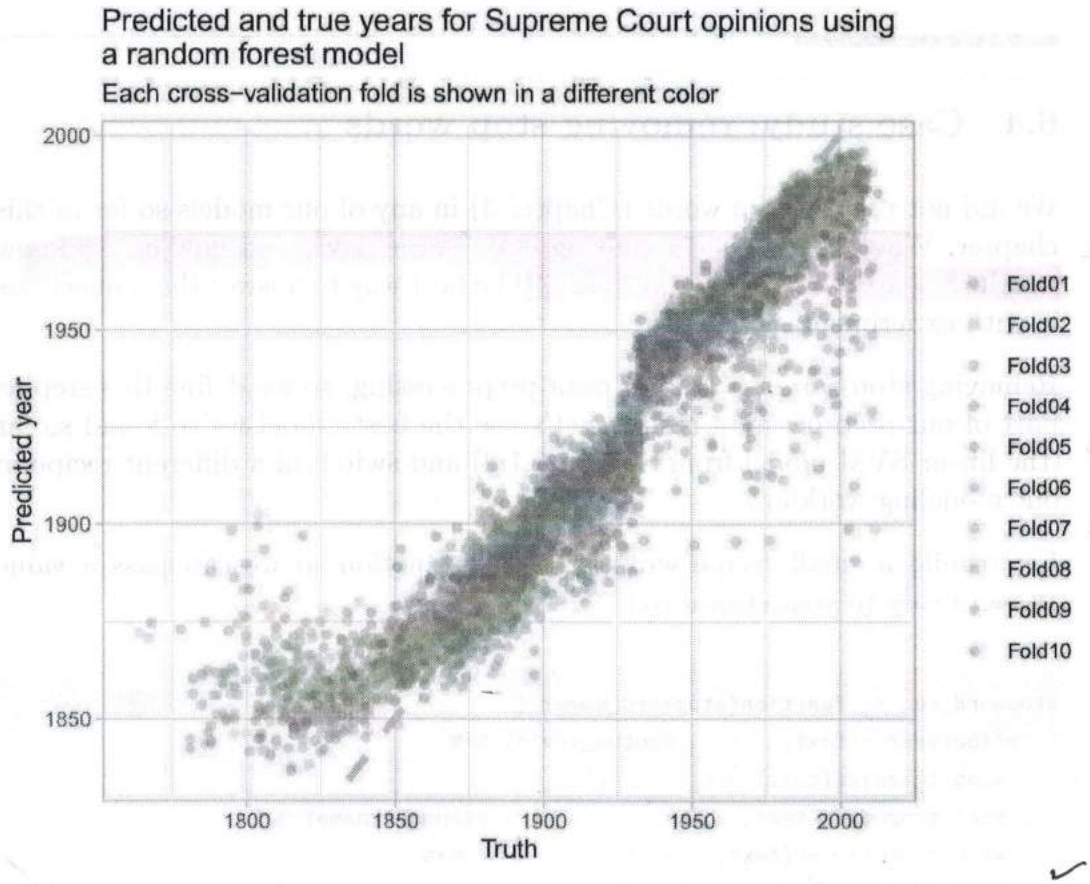



**FIGURE 6.3:** The random forest model did not perform very sensibly across years, compared to our first attempt using a linear SVM model

not well-suited to sparse data because of how decision trees model outcomes (Tang, Garreau, and Luxburg 2018).

Models that work best with text tend to be models designed for or otherwise appropriate for sparse data.

Algorithms that work well with sparse data are less important when text has been transformed to a non-sparse representation, such as with word embeddings (Chapter 5).

## 6.4 Case study: removing stop words

We did not remove stop words (Chapter 3) in any of our models so far in this chapter. What impact will removing stop words have, and how do we know which stop word list is the best to use? The best way to answer these questions is with experimentation.

Removing stop words is part of data preprocessing, so we define this step as part of our preprocessing recipe. Let's use the best model we've found so far (the linear SVM model from Section 6.1.2) and switch in a different recipe in our modeling workflow.

Let's build a small recipe wrapper helper function so we can pass a value `stopword_name` to `step_stopwords()`.

```
stopword_rec <- function(stopword_name) {
  recipe(year ~ text, data = scotus_train) %>%
    step_tokenize(text) %>%
    step_stopwords(text, stopword_source = stopword_name) %>%
    step_tokenfilter(text, max_tokens = 1e3) %>%
    step_tfidf(text) %>%
    step_normalize(all_predictors())
}
```

For example, now we can create a recipe that removes the Snowball stop words list by calling this function.

```
stopword_rec("snowball")
```

```
#> Data Recipe
#>
#> Inputs:
#>
#>       role #variables
#>    outcome          1
#>  predictor          1
#>
#> Operations:
#>
#> Tokenization for text
```

```
#> Stop word removal for text
#> Text filtering for text
#> Term frequency-inverse document frequency with text
#> Centering and scaling for all_predictors()
```

Next, let's set up a new workflow that has a model only, using `add_model()`. We start with the empty `workflow()` and then add our linear SVM regression model.

```
svm_wf <- workflow() %>%
  add_model(svm_spec)

svm_wf
```

```
#> == Workflow ====================================================================
#> Preprocessor: None
#> Model: svm_linear()
#>
#> -- Model -----------------------------------------------------------------------
#> Linear Support Vector Machine Specification (regression)
#>
#> Computational engine: LiblineaR
```

Notice that for this workflow, there is no preprocessor yet: `Preprocessor: None`. This workflow uses the same linear SVM specification that we used in Section 6.1, but we are going to combine several different preprocessing recipes with it, one for each stop word lexicon we want to try.

Now we can put this all together and fit these models that include stop word removal. We could create a little helper function for fitting like we did for the recipe, but we have printed out all three calls to `fit_resamples()` for extra clarity. Notice for each one that there are two arguments:

- A workflow, which consists of the linear SVM model specification and a data preprocessing recipe with stop word removal
- The same cross-validation folds we created earlier with 3 random seeds

```
set.seed(123)
snowball_rs <- fit_resamples(
  svm_wf %>% add_recipe(stopword_rec("snowball")),
```

```
  scotus_folds
)

set.seed(234)
smart_rs <- fit_resamples(
  svm_wf %>% add_recipe(stopword_rec("smart")),
  scotus_folds
)

set.seed(345)
stopwords_iso_rs <- fit_resamples(
  svm_wf %>% add_recipe(stopword_rec("stopwords-iso")),
  scotus_folds
)
```

After fitting models to each of the cross-validation folds, these sets of results contain metrics computed for removing that set of stop words.

```
collect_metrics(smart_rs)
```

```
#> # A tibble: 2 x 6
#>   .metric .estimator   mean     n std_err .config
#>   <chr>   <chr>       <dbl> <int>   <dbl> <chr>
#> 1 rmse    standard   17.2      10 0.199   Preprocessor1_Model1
#> 2 rsq     standard    0.876    10 0.00261 Preprocessor1_Model1
```

We can explore whether one of these sets of stop words performed better than the others by comparing the performance, for example in terms of RMSE as shown Figure 6.4. This plot shows the five best models for each set of stop words, using `show_best()` applied to each via `purrr::map_dfr()`.

```
word_counts <- tibble(name = c("snowball", "smart", "stopwords-iso")) %>%
  mutate(words = map_int(name, ~length(stopwords::stopwords(source = .))))

list(snowball = snowball_rs,
     smart = smart_rs,
     `stopwords-iso` = stopwords_iso_rs) %>%
  map_dfr(show_best, "rmse", .id = "name") %>%
  left_join(word_counts, by = "name") %>%
  mutate(name = paste0(name, " (", words, " words)"),
```

```
        name = fct_reorder(name, words)) %>%
ggplot(aes(name, mean, color = name)) +
geom_crossbar(aes(ymin = mean - std_err, ymax = mean + std_err), alpha = 0.6) +
geom_point(size = 3, alpha = 0.8) +
theme(legend.position = "none") +
labs(x = NULL, y = "RMSE",
     title = "Model performance for three stop word lexicons",
     subtitle = "For this data set, the Snowball lexicon performed best")
```

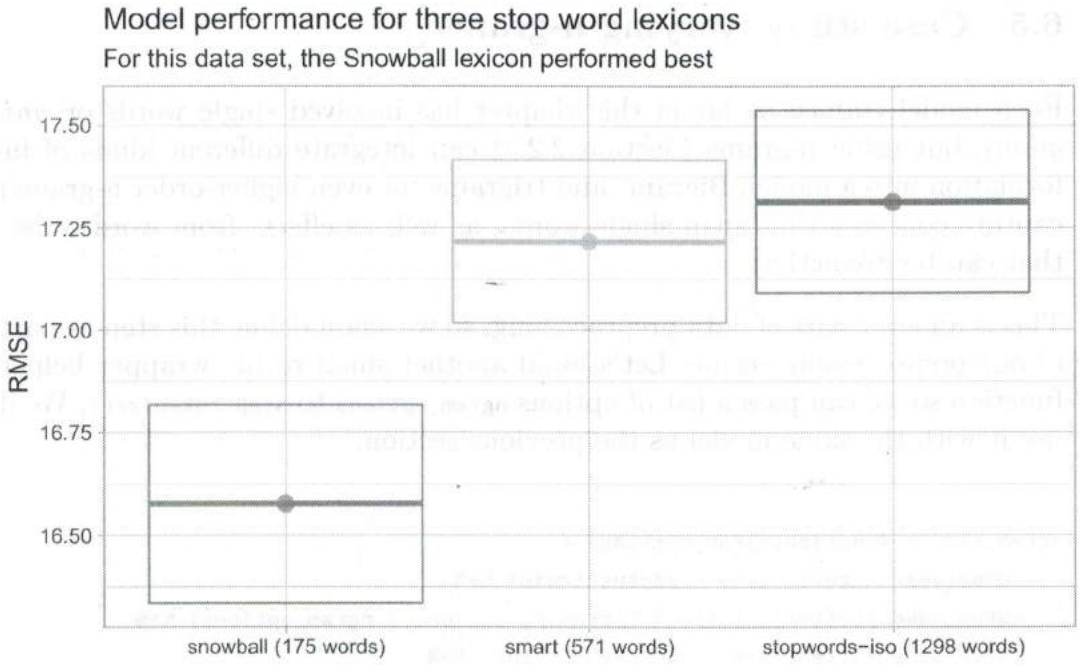



**FIGURE 6.4:** Comparing model performance for predicting the year of Supreme Court opinions with three different stop word lexicons

The Snowball lexicon contains the smallest number of words (see Figure 3.1) and, in this case, results in the best performance. Removing fewer stop words results in the best performance.

This result is not generalizable to all data sets and contexts, but the approach outlined in this section **is** generalizable.

This approach can be used to compare different lexicons and find the best one for a specific data set and model. Notice how the results for all stop word lexicons are worse than removing no stop words at all (remember that the

RMSE was 15.6 years in Section 6.1.2). This indicates that, for this particular data set, removing even a small stop word list is not a great choice.

When removing stop words does appear to help a model, it's good to know that removing stop words isn't computationally slow or difficult so the cost for this improvement is low.

## 6.5 Case study: varying n-grams

Each model trained so far in this chapter has involved single words or *unigrams*, but using n-grams (Section 2.2.3) can integrate different kinds of information into a model. Bigrams and trigrams (or even higher-order n-grams) capture concepts that span single words, as well as effects from word order, that can be predictive.

This is another part of data preprocessing, so we again define this step as part of our preprocessing recipe. Let's build another small recipe wrapper helper function so we can pass a list of options `ngram_options` to `step_tokenize()`. We'll use it with the same model as the previous section.

```
ngram_rec <- function(ngram_options) {
  recipe(year ~ text, data = scotus_train) %>%
    step_tokenize(text, token = "ngrams", options = ngram_options) %>%
    step_tokenfilter(text, max_tokens = 1e3) %>%
    step_tfidf(text) %>%
    step_normalize(all_predictors())
}
```

There are two options we can specify, `n` and `n_min`, when we are using `engine = "tokenizers"`. We can set up a recipe with only `n = 1` to tokenize and only extract the unigrams.

```
ngram_rec(list(n = 1))
```

We can use `n = 3, n_min = 1` to identify the set of all trigrams, bigrams, *and* unigrams.

```
ngram_rec(list(n = 3, n_min = 1))
```

Including n-grams of different orders in a model (such as trigrams, bigrams, plus unigrams) allows the model to learn at different levels of linguistic organization and context.

We can reuse the same workflow `svm_wf` from our earlier case study; these types of modular components are a benefit to adopting this approach to supervised machine learning. This workflow provides the linear SVM specification. Let's put it all together and create a helper function to use `fit_resamples()` with this model plus our helper recipe function.

```
fit_ngram <- function(ngram_options) {
  fit_resamples(
    svm_wf %>% add_recipe(ngram_rec(ngram_options)),
    scotus_folds
  )
}
```

We could have created this type of small function for trying out different stop word lexicons in Section 6.4, but there we showed each call to `fit_resamples()` for extra clarity.

With this helper function, let's try out predicting the year of Supreme Court opinions using:

- only unigrams
- bigrams and unigrams
- trigrams, bigrams, and unigrams

```
set.seed(123)
unigram_rs <- fit_ngram(list(n = 1))

set.seed(234)
bigram_rs <- fit_ngram(list(n = 2, n_min = 1))

set.seed(345)
trigram_rs <- fit_ngram(list(n = 3, n_min = 1))
```

It's good that we consistently check model performance, no matter what model we use.

These sets of results contain metrics computed for the model with that tokenization strategy.

```
collect_metrics(bigram_rs)
```

```
#> # A tibble: 2 x 6
#>   .metric .estimator   mean     n std_err .config
#>   <chr>   <chr>       <dbl> <int>   <dbl> <chr>
#> 1 rmse    standard   15.9      10 0.225   Preprocessor1_Model1
#> 2 rsq     standard    0.892    10 0.00240 Preprocessor1_Model1
```

We can compare the performance of these models in terms of RMSE as shown Figure 6.5.

```
list(`1` = unigram_rs,
     `1 and 2` = bigram_rs,
     `1, 2, and 3` = trigram_rs) %>%
  map_dfr(collect_metrics, .id = "name") %>%
  filter(.metric == "rmse") %>%
  ggplot(aes(name, mean, color = name)) +
  geom_crossbar(aes(ymin = mean - std_err, ymax = mean + std_err), alpha = 0.6) +
  geom_point(size = 3, alpha = 0.8) +
  theme(legend.position = "none") +
  labs(
    x = "Degree of n-grams",
    y = "RMSE",
    title = "Model performance for different degrees of n-gram tokenization",
    subtitle = "For the same number of tokens, unigrams performed best"
  )
```

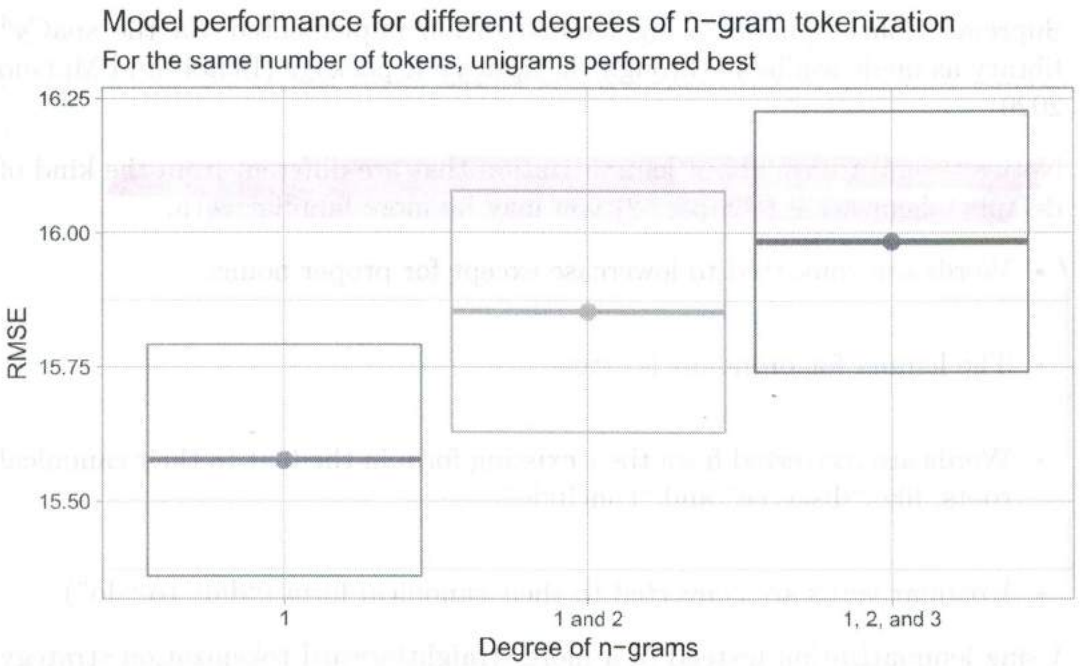



**FIGURE 6.5:** Comparing model performance for predicting the year of Supreme Court opinions with three different degrees of n-grams

Each of these models was trained with `max_tokens = 1e3`, i.e., including only the top 1000 tokens for each tokenization strategy. Holding the number of tokens constant, using unigrams alone performs best for this corpus of Supreme Court opinions. To be able to incorporate the more complex information in bigrams or trigrams, we would need to increase the number of tokens in the model considerably.

Keep in mind that adding n-grams is computationally expensive to start with, especially compared to the typical improvement in model performance gained. We can benchmark the whole model workflow, including preprocessing and modeling. Using bigrams plus unigrams takes more than twice as long to train than only unigrams (number of tokens held constant), and adding in trigrams as well takes almost five times as long as training on unigrams alone.

## 6.6 Case study: lemmatization

As we discussed in Section 4.6, we can normalize words to their roots or **lemmas** based on each word's context and the structure of a language. Table 6.1 shows both the original words and the lemmas for one sentence from a

Supreme Court opinion, using lemmatization implemented via the spaCy[4] library as made available through the **spacyr** R package (Benoit and Matsuo 2020).

Notice several things about lemmatization that are different from the kind of default tokenization (Chapter 2) you may be more familiar with.

- Words are converted to lowercase except for proper nouns.
- The lemma for pronouns is `-PRON-`.
- Words are converted from their existing form in the text to their canonical roots, like "disagree" and "conclude."
- Irregular verbs are converted to their canonical form ("did" to "do").

Using lemmatization instead of a more straightforward tokenization strategy is slower because of the increased complexity involved, but it can be worth it. Let's explore how to train a model using *lemmas* instead of *words*.

Lemmatization is, like choices around n-grams and stop words, part of data preprocessing so we define how to set up lemmatization as part of our preprocessing recipe. We use `engine = "spacyr"` for tokenization (instead of the default) and add `step_lemma()` to our preprocessing. This step extracts the lemmas from the parsing done by the tokenization engine.

```
spacyr::spacy_initialize(entity = FALSE)
```

```
#> NULL
```

```
lemma_rec <- recipe(year ~ text, data = scotus_train) %>%
  step_tokenize(text, engine = "spacyr") %>%
  step_lemma(text) %>%
  step_tokenfilter(text, max_tokens = 1e3) %>%
  step_tfidf(text) %>%
  step_normalize(all_predictors())

lemma_rec
```

[4] https://spacy.io/

Lemmatization of one sentence from a Supreme Court opinion

| original word | lemma |
|---|---|
| However | however |
| , | , |
| the | the |
| Court | Court |
| of | of |
| Appeals | Appeals |
| disagreed | disagree |
| with | with |
| the | the |
| District | District |
| Court | Court |
| 's | 's |
| construction | construction |
| of | of |
| the | the |
| state | state |
| statute | statute |
| , | , |
| concluding | conclude |
| that | that |
| it | it |
| did | do |
| authorize | authorize |
| issuance | issuance |
| of | of |
| the | the |
| orders | order |
| to | to |
| withhold | withhold |
| to | to |
| the | the |
| Postal | Postal |
| Service | Service |
| . | . |

```
#> Data Recipe
#>
#> Inputs:
#>
#>       role #variables
#>    outcome           1
#>  predictor           1
#>
#> Operations:
#>
#> Tokenization for text
#> Lemmatization for text
#> Text filtering for text
#> Term frequency-inverse document frequency with text
#> Centering and scaling for all_predictors()
```

Let's combine this lemmatized text with our linear SVM workflow. We can then fit our workflow to our resampled data sets and estimate performance using lemmatization.

```
lemma_rs <- fit_resamples(
  svm_wf %>% add_recipe(lemma_rec),
  scotus_folds
)
```

How did this model perform?

```
collect_metrics(lemma_rs)
```

```
#> # A tibble: 2 x 6
#>   .metric .estimator   mean     n std_err .config
#>   <chr>   <chr>       <dbl> <int>   <dbl> <chr>
#> 1 rmse    standard    14.2      10 0.276   Preprocessor1_Model1
#> 2 rsq     standard     0.913    10 0.00304 Preprocessor1_Model1
```

The best value for RMSE at 14.2 shows us that using lemmatization can have a significant benefit for model performance, compared to 15.6 from fitting a non-lemmatized linear SVM model in Section 6.1.2. The best model using lemmatization is better than the best model without. However, this comes at a cost of much slower training because of the procedure involved in identifying lemmas; adding `step_lemma()` to our preprocessing increases the overall time to train the workflow by over 10-fold.

We can use `engine = "spacyr"` to assign part-of-speech tags to the tokens during tokenization, and this information can be used in various useful ways in text modeling. One approach is to filter tokens to only retain a certain part of speech, like nouns. An example of how to do this is illustrated in this **textrecipes** blogpost[5] and can be performed with `step_pos_filter()`.

## 6.7 Case study: feature hashing

The models we have created so far have used tokenization (Chapter 2) to split apart text data into tokens that are meaningful to us as human beings (words, bigrams) and then weighted these tokens by simple counts with word frequencies or weighted counts with tf-idf. A problem with these methods is that the output space can be vast and dynamic. We have limited ourselves to 1000 tokens so far in this chapter, but we could easily have more than 10,000 features in our training set. We may run into computational problems with memory or long processing times; deciding how many tokens to include can become a trade-off between computational time and information. This style of approach also doesn't let us take advantage of new tokens we didn't see in our training data.

One method that has gained popularity in the machine learning field is the **hashing trick**. This method addresses many of the challenges outlined above and is very fast with a low memory footprint.

Let's start with the basics of feature hashing. First proposed by Weinberger et al. (2009), feature hashing was introduced as a dimensionality reduction method with a simple premise. We begin with a hashing function that we then apply to our tokens.

A hashing function takes input of variable size and maps it to output of a fixed size. Hashing functions are commonly used in cryptography.

[5] https://www.hvitfeldt.me/blog/tidytuesday-pos-textrecipes-the-office/

We will use the `hash()` function from the **rlang** package to illustrate the behavior of hashing functions. The `rlang::hash()` function uses the XXH128 hash algorithm of the xxHash library, which generates a 128-bit hash. This is a more complex hashing function than what is normally used for the hashing trick. The 32-bit version of MurmurHash3 (Appleby 2008) is often used for its speed and good properties.

Hashing functions are typically very fast and have certain properties. For example, the output of a hash function is expected to be uniform, with the whole output space filled evenly. The "avalanche effect" describes how similar strings are hashed in such a way that their hashes are not similar in the output space.

Suppose we have many country names in a character vector. We can apply the hashing function to each of the country names to project them into an integer space defined by the hashing function.

Since `hash()` creates hashes that are very long, let's create `small_hash()` for demonstration purposes here that generates slightly smaller hashes. (The specific details of what hashes are generated are not important here.)

```
library(rlang)
countries <- c("Palau", "Luxembourg", "Vietnam", "Guam", "Argentina",
               "Mayotte", "Bouvet Island", "South Korea", "San Marino",
               "American Samoa")

small_hash <- function(x) {
  strtoi(substr(hash(x), 26, 32), 16)
}

map_int(countries, small_hash)
```

```
#>  [1]   4292706   2881716 242176357 240902473 204438359  88787026 230339508
#>  [8]  15112074  96146649 192775182
```

Our `small_hash()` function uses `7 * 4 = 28` bits, so the number of possible values is `2^28 = 268435456`. This is admittedly not much of an improvement over 10 country names. Let's take the modulo of these big integer values to project them down to a more manageable space.

```
map_int(countries, small_hash) %% 24
```

```
#>  [1] 18 12 13  1 23 10 12 18  9  6
```

Now we can use these values as indices when creating a matrix.

```
#> 10 x 24 sparse Matrix of class "ngCMatrix"
#>
#> Palau          . . . . . . . . . . . . . . . . . . . . | . . . . .
#> Luxembourg     . . . . . . . . . . . | . . . . . . . . . . . . . .
#> Vietnam        . . . . . . . . . . . | . . . . . . . . . . . . . .
#> Guam           | . . . . . . . . . . . . . . . . . . . . . . . . .
#> Argentina      . . . . . . . . . . . . . . . . . . . . . . . . | .
#> Mayotte        . . . . . . . . . | . . . . . . . . . . . . . . . .
#> Bouvet Island  . . . . . . . . . . . | . . . . . . . . . . . . . .
#> South Korea    . . . . . . . . . . . . . . . . . . . . | . . . . .
#> San Marino     . . . . . . . . . | . . . . . . . . . . . . . . . .
#> American Samoa . . . . . | . . . . . . . . . . . . . . . . . . . .
```

This method is very fast; both the hashing and modulo can be performed independently for each input since neither need information about the full corpus. Since we are reducing the space, there is a chance that multiple words are hashed to the same value. This is called a collision and, at first glance, it seems like it would be a big problem for a model. However, research finds that using feature hashing has roughly the same accuracy as a simple bag-of-words model, and the effect of collisions is quite minor (Forman and Kirshenbaum 2008).

Another step that is taken to avoid the negative effects of hash collisions is to use a *second* hashing function that returns 1 and $-1$. This determines if we are adding or subtracting the index we get from the first hashin function. Suppose both the words "outdoor" and "pleasant" hash to the integer value 583. Without the second hashing they would collide to 2. Using signed hashing, we have a 50% chance that they will cancel each other out, which tries to stop one feature from growing too much.

There are downsides to using feature hashing. Feature hashing:

- still has one tuning parameter, and
- cannot be reversed.

The number of buckets you have correlates with computation speed and collision rate, which in turn affects performance. It is your job to find the output that best suits your needs. Increasing the number of buckets will decrease the collision rate but will, in turn, return a larger output data set, which increases model fitting time. The number of buckets is tunable in tidymodels using the **tune** package.

Perhaps the more important downside to using feature hashing is that the operation can't be reversed. We are not able to detect if a collision occurs and it is difficult to understand the effect of any word in the model. Remember that we are left with `n` columns of *hashes* (not tokens), so if we find that the 274th column is a highly predictive feature, we cannot know in general which tokens contribute to that column. We cannot directly connect model values to words or tokens at all. We could go back to our training set and create a paired list of the tokens and what hashes they map to. Sometimes we might find only one token in that list, but it may have two (or three or four or more!) different tokens contributing. This feature hashing method is used because of its speed and scalability, not because it is interpretable.

Feature hashing on tokens is available in tidymodels using the `step_texthash()` step from **textrecipes**. Let's `prep()` and `bake()` this recipe for demonstration purposes.

```
scotus_hash <- recipe(year ~ text, data = scotus_train) %>%
  step_tokenize(text) %>%
  step_texthash(text, signed = TRUE, num_terms = 512) %>%
  prep() %>%
  bake(new_data = NULL)

dim(scotus_hash)
```

```
#> [1] 7500  513
```

There are many columns in the results. Let's take a `glimpse()` at the first 10 columns.

```
scotus_hash %>%
  select(num_range("text_hash00", 1:9)) %>%
  glimpse()
```

```
#> Rows: 7,500
#> Columns: 9
#> $ text_hash001 <dbl> -16, -5, -12, -10, -10, -2, -7, -13, -16, -18, -1, -2, -1~
#> $ text_hash002 <dbl> -1, 1, 3, -2, 0, 0, 5, -1, 1, 6, 0, 2, 0, 0, 0, -3, 1, 2,~
#> $ text_hash003 <dbl> -2, 0, 4, -1, -1, 1, -5, -2, -2, 0, 0, -1, 1, 6, 0, 0, -3~
#> $ text_hash004 <dbl> -2, 0, -1, 0, 0, 0, -14, -14, -4, -2, 0, -10, -1, -2, 0, ~
#> $ text_hash005 <dbl> 0, 0, 0, 0, 0, 0, -2, -1, 2, 1, 0, -1, 0, -1, 0, 0, -1, 0~
#> $ text_hash006 <dbl> 24, 2, 4, 6, 7, 2, 14, 13, 13, 22, 1, 41, 2, 49, 9, 1, 17~
#> $ text_hash007 <dbl> 13, 1, 1, -3, 0, -6, -2, -4, -8, -1, 0, 0, -4, -11, 0, 0,~
#> $ text_hash008 <dbl> -8, 3, 1, 1, 1, 0, -19, 0, 1, 0, 1, -1, 1, 1, -2, 1, -8, ~
#> $ text_hash009 <dbl> -2, 0, -1, 1, 0, 0, 0, -1, -1, -1, 0, -1, -1, -1, 0, 0, --
```

By using `step_texthash()` we can quickly generate machine-ready data with a consistent number of variables. This typically results in a slight loss of performance compared to using a traditional bag-of-words representation. An example of this loss is illustrated in this **textrecipes** blogpost[6].

Future Tina read this!

### 6.7.1 Text normalization

When working with text, you will inevitably run into problems with encodings and related irregularities. These kinds of problems have a significant influence on feature hashing, as well as other preprocessing steps. Consider the German word "schön." The o with an umlaut (two dots over it) is a fairly simple character, but it can be represented in a couple of different ways. We can either use a single character \U00f6[7] to represent the letter with an umlaut. Alternatively, we can use two characters, one for the o and one character to denote the presence of two dots over the previous character \U0308[8].

```
s1 <- "sch\U00f6n"
s2 <- "scho\U0308n"
```

so we have to tell it to take both as the same

These two strings will print the same for us as human readers.

[6] https://www.hvitfeldt.me/blog/textrecipes-series-featurehashing/

[7] https://www.fileformat.info/info/unicode/char/00f6/index.htm

[8] https://www.fileformat.info/info/unicode/char/0308/index.htm

```
s1
```

```
#> [1] "schön"
```

```
s2
```

```
#> [1] "schön"
```

However, they are not equal.

```
s1 == s2
```

```
#> [1] FALSE
```

This poses a problem for the avalanche effect, which is needed for feature hashing to perform correctly. The avalanche effect will result in these two words (which should be identical) hashing to completely different values.

```
small_hash(s1)
```

```
#> [1] 180735918
```

```
small_hash(s2)
```

```
#> [1] 3013209
```

We can deal with this problem by performing **text normalization** on our text before feeding it into our preprocessing engine. One library to perform text normalization is the **stringi** package, which includes many different text normalization methods. How these methods work is beyond the scope of this book, but know that the text normalization functions make text like our two versions of "schön" equivalent. We will use `stri_trans_nfc()` for this example, which performs canonical decomposition, followed by canonical composition, but we could also use `textrecipes::step_text_normalize()` within a tidymodels recipe for the same task.

```
library(stringi)

stri_trans_nfc(s1) == stri_trans_nfc(s2)
```

```
#> [1] TRUE
```

```
small_hash(stri_trans_nfc(s1))
```

```
#> [1] 180735918
```

```
small_hash(stri_trans_nfc(s2))
```

```
#> [1] 180735918
```

Now we see that the strings are equal after normalization.

This issue of text normalization can be important even if you don't use feature hashing in your machine learning.

Since these words are encoded in different ways, they will be counted separately when we are counting token frequencies. Representing what should be a single token in multiple ways will split the counts. This will introduce noise in the best case, and in worse cases, some tokens will fall below the cutoff when we select tokens, leading to a loss of potentially informative words.

Luckily this is easily addressed by using `stri_trans_nfc()` on our text columns *before* starting preprocessing, or perhaps more conveniently, by using `textrecipes::step_text_normalize()` *within* a preprocessing recipe.

## 6.8 What evaluation metrics are appropriate?

We have focused on using RMSE and $R^2$ as metrics for our models in this chapter, the defaults in the tidymodels framework. Other metrics can also be

appropriate for regression models. Another common set of regression metric options are the various flavors of mean absolute error.

If you know before you fit your model that you want to compute one or more of these metrics, you can specify them in a call to `metric_set()`. Let's set up a tuning grid for mean absolute error (`mae`) and mean absolute percent error (`mape`).

```
lemma_rs <- fit_resamples(
  svm_wf %>% add_recipe(lemma_rec),
  scotus_folds,
  metrics = metric_set(mae, mape)
)
```

If you have already fit your model, you can still compute and explore non-default metrics as long as you saved the predictions for your resampled data sets using `control_resamples(save_pred = TRUE)`.

Let's go back to the first linear SVM model we tuned in Section 6.1.2, with results in `svm_rs`. We can compute the overall mean absolute percent error.

```
svm_rs %>%
  collect_predictions() %>%
  mape(year, .pred)
```

```
#> # A tibble: 1 x 3
#>   .metric .estimator .estimate
#>   <chr>   <chr>          <dbl>
#> 1 mape    standard       0.616
```

We can also compute the mean absolute percent error for each resample.

```
svm_rs %>%
  collect_predictions() %>%
  group_by(id) %>%
  mape(year, .pred)
```

```
#> # A tibble: 10 x 4
#>    id     .metric .estimator .estimate
#>    <chr>  <chr>   <chr>          <dbl>
#>  1 Fold01 mape    standard       0.603
```

```
#>  2 Fold02 mape    standard       0.660
#>  3 Fold03 mape    standard       0.596
#>  4 Fold04 mape    standard       0.639
#>  5 Fold05 mape    standard       0.618
#>  6 Fold06 mape    standard       0.611
#>  7 Fold07 mape    standard       0.618
#>  8 Fold08 mape    standard       0.602
#>  9 Fold09 mape    standard       0.604
#> 10 Fold10 mape    standard       0.605
```

Similarly, we can do the same for the mean absolute error, which gives a result in units of the original data (years, in this case) instead of relative units.

```
svm_rs %>%
  collect_predictions() %>%
  group_by(id) %>%
  mae(year, .pred)
```

```
#> # A tibble: 10 x 4
#>    id     .metric .estimator .estimate
#>    <chr>  <chr>   <chr>          <dbl>
#>  1 Fold01 mae     standard        11.5
#>  2 Fold02 mae     standard        12.6
#>  3 Fold03 mae     standard        11.4
#>  4 Fold04 mae     standard        12.2
#>  5 Fold05 mae     standard        11.8
#>  6 Fold06 mae     standard        11.7
#>  7 Fold07 mae     standard        11.9
#>  8 Fold08 mae     standard        11.5
#>  9 Fold09 mae     standard        11.6
#> 10 Fold10 mae     standard        11.6
```

For the full set of regression metric options, see the yardstick documentation[9].

[9] https://yardstick.tidymodels.org/reference/

## 6.9 The full game: regression

In this chapter, we started from the beginning and then explored both different types of models and different data preprocessing steps. Let's take a step back and build one final model, using everything we've learned. For our final model, let's again use a linear SVM regression model, since it performed better than the other options we looked at. We will:

- train on the same set of cross-validation resamples used throughout this chapter,
- *tune* the number of tokens used in the model to find a value that fits our needs,
- include both unigrams and bigrāms,
- choose not to use lemmatization, to demonstrate what is possible for situations when training time makes lemmatization an impractical choice, and
- finally evaluate on the testing set, which we have not touched at all yet.

We will include a much larger number of tokens than before, which should give us the latitude to include both unigrams and bigrams, despite the result we saw in Section 6.5.

### 6.9.1 Preprocess the data

First, let's create the data preprocessing recipe. By setting the tokenization options to `list(n = 2, n_min = 1)`, we will include both unigrams and bigrams in our model.

When we set `max_tokens = tune()`, we can train multiple models with different numbers of maximum tokens and then compare these models' performance to choose the best value. Previously, we set `max_tokens = 1e3` to choose a specific value for the number of tokens included in our model, but here we are going to try multiple different values.

```
final_rec <- recipe(year ~ text, data = scotus_train) %>%
  step_tokenize(text, token = "ngrams", options = list(n = 2, n_min = 1)) %>%
  step_tokenfilter(text, max_tokens = tune()) %>%
  step_tfidf(text) %>%
  step_normalize(all_predictors())

final_rec
```

```
#> Data Recipe
#>
#> Inputs:
#>
#>       role #variables
#>    outcome          1
#>  predictor          1
#>
#> Operations:
#>
#> Tokenization for text
#> Text filtering for text
#> Term frequency-inverse document frequency with text
#> Centering and scaling for all_predictors()
```

### 6.9.2 Specify the model

Let's use the same linear SVM regression model specification we have used multiple times in this chapter, and set it up here again to remind ourselves.

```
svm_spec <- svm_linear() %>%
  set_mode("regression") %>%
  set_engine("LiblineaR")

svm_spec
```

```
#> Linear Support Vector Machine Specification (regression)
#>
#> Computational engine: LiblineaR
```

We can combine the preprocessing recipe and the model specification in a tunable workflow. We can't fit this workflow right away to training data, because the value for `max_tokens` hasn't been chosen yet.

```
tune_wf <- workflow() %>%
  add_recipe(final_rec) %>%
  add_model(svm_spec)

tune_wf
```

```
#> == Workflow ====================================================================
#> Preprocessor: Recipe
#> Model: svm_linear()
#>
#> -- Preprocessor ----------------------------------------------------------------
#> 4 Recipe Steps
#>
#> * step_tokenize()
#> * step_tokenfilter()
#> * step_tfidf()
#> * step_normalize()
#>
#> -- Model ----------------------------------------------------------------------
#> Linear Support Vector Machine Specification (regression)
#>
#> Computational engine: LiblineaR
```

### 6.9.3 Tune the model

Before we tune the model, we need to set up a set of possible parameter values to try.

There is *one* tunable parameter in this model, the maximum number of tokens included in the model.

Let's include different possible values for this parameter starting from the value we've already tried, for a combination of six models.

```
final_grid <- grid_regular(
  max_tokens(range = c(1e3, 6e3)),
  levels = 6
)
final_grid
```

```
#> # A tibble: 6 x 1
#>   max_tokens
#>        <int>
#> 1       1000
#> 2       2000
#> 3       3000
#> 4       4000
#> 5       5000
#> 6       6000
```

Now it's time for tuning. Instead of using `fit_resamples()` as we have throughout this chapter, we are going to use `tune_grid()`, a function that has a very similar set of arguments. We pass this function our workflow (which holds our preprocessing recipe and SVM model), our resampling folds, and also the grid of possible parameter values to try. Let's save the predictions so we can explore them in more detail, and let's also set custom metrics instead of using the defaults. Let's compute RMSE, mean absolute error, and mean absolute percent error during tuning.

why?

```
final_rs <- tune_grid(
  tune_wf,
  scotus_folds,
  grid = final_grid,
  metrics = metric_set(rmse, mae, mape),
  control = control_resamples(save_pred = TRUE)
)

final_rs
```

```
#> # Tuning results
#> # 10-fold cross-validation
#> # A tibble: 10 x 5
#>    splits             id     .metrics          .notes           .predictions
#>    <list>             <chr>  <list>            <list>           <list>
#>  1 <split [6750/750]> Fold01 <tibble [18 x 5]> <tibble [0 x 1]> <tibble [4,500 ~
#>  2 <split [6750/750]> Fold02 <tibble [18 x 5]> <tibble [0 x 1]> <tibble [4,500 ~
#>  3 <split [6750/750]> Fold03 <tibble [18 x 5]> <tibble [0 x 1]> <tibble [4,500 ~
#>  4 <split [6750/750]> Fold04 <tibble [18 x 5]> <tibble [0 x 1]> <tibble [4,500 ~
#>  5 <split [6750/750]> Fold05 <tibble [18 x 5]> <tibble [0 x 1]> <tibble [4,500 ~
#>  6 <split [6750/750]> Fold06 <tibble [18 x 5]> <tibble [0 x 1]> <tibble [4,500 ~
#>  7 <split [6750/750]> Fold07 <tibble [18 x 5]> <tibble [0 x 1]> <tibble [4,500 ~
#>  8 <split [6750/750]> Fold08 <tibble [18 x 5]> <tibble [0 x 1]> <tibble [4,500 ~
#>  9 <split [6750/750]> Fold09 <tibble [18 x 5]> <tibble [0 x 1]> <tibble [4,500 ~
#> 10 <split [6750/750]> Fold10 <tibble [18 x 5]> <tibble [0 x 1]> <tibble [4,500 ~
```

We trained all these models!

### 6.9.4 Evaluate the modeling

Now that all of the models with possible parameter values have been trained, we can compare their performance. Figure 6.6 shows us the relationship between performance (as measured by the metrics we chose) and the number of tokens.

```
final_rs %>%
  collect_metrics() %>%
  ggplot(aes(max_tokens, mean, color = .metric)) +
  geom_line(size = 1.5, alpha = 0.5) +
  geom_point(size = 2, alpha = 0.9) +
  facet_wrap(~.metric, scales = "free_y", ncol = 1) +
  theme(legend.position = "none") +
  labs(
    x = "Number of tokens",
    title = "Linear SVM performance across number of tokens",
    subtitle = "Performance improves as we include more tokens"
  )
```

Since this is our final version of this model, we want to choose final parameters and update our model object so we can use it with new data. We have several options for choosing our final parameters, such as selecting the numerically best model (which would be one of the ones with the most tokens in our situation here) or the simplest model within some limit around the numerically best result. In this situation, we likely want to choose a simpler model with fewer tokens that gives close-to-best performance.

Let's choose by percent loss compared to the best model, with the default 2% loss.

```
chosen_mae <- final_rs %>%
  select_by_pct_loss(metric = "mae", max_tokens)

chosen_mae
```

```
#> # A tibble: 1 x 9
#>   max_tokens .metric .estimator  mean     n std_err .config              .best .loss
#>        <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>                <dbl> <dbl>
#> 1       5000 mae     standard    10.1    10  0.0680 Preprocessor5_M~  9.98 0.795
```

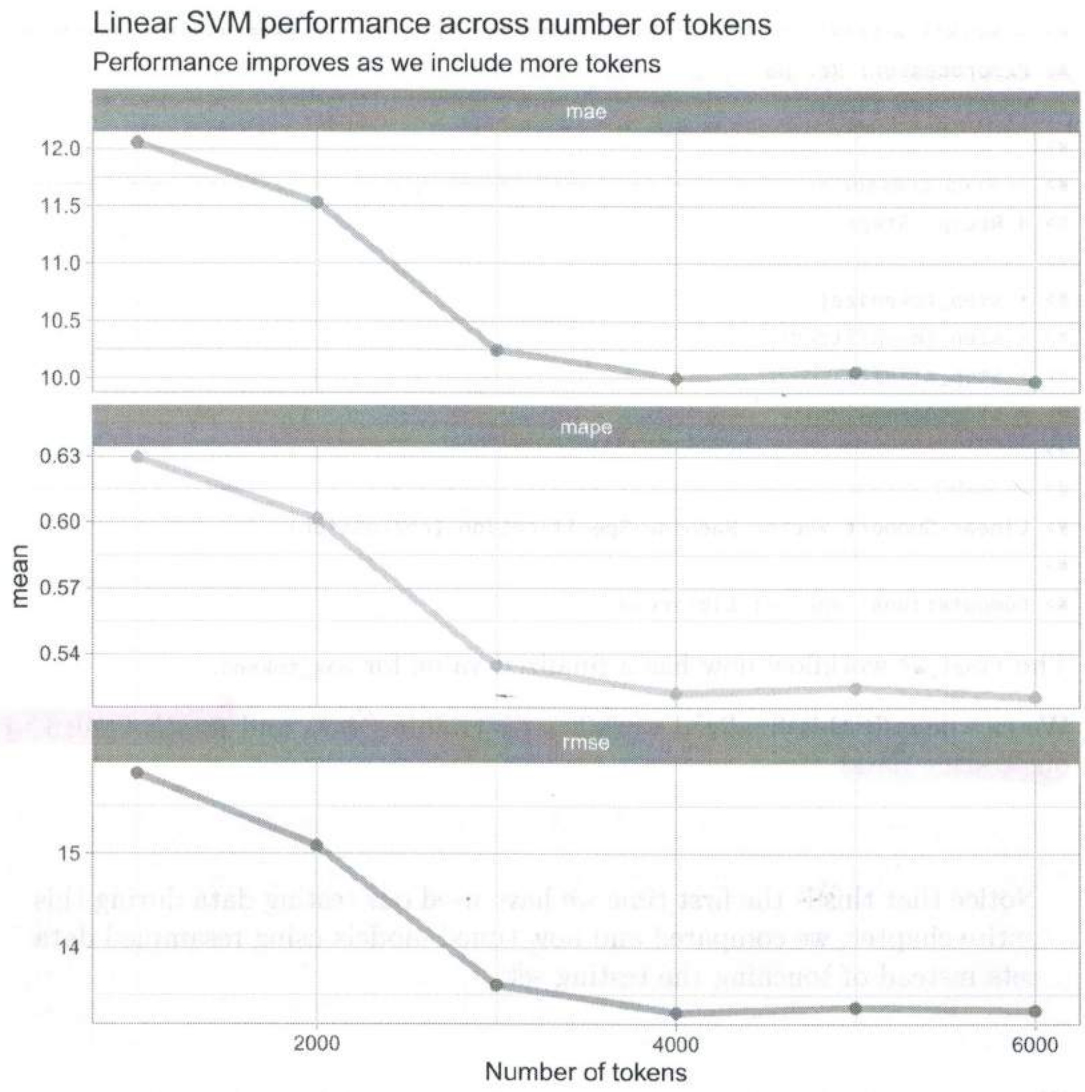



**FIGURE 6.6:** Performance improves significantly at about 4000 tokens

After we have those parameters, `penalty` and `max_tokens`, we can finalize our earlier tunable workflow, by updating it with this value.

```
final_wf <- finalize_workflow(tune_wf, chosen_mae)

final_wf
```

```
#> == Workflow ============================================================
#> Preprocessor: Recipe
#> Model: svm_linear()
#>
#> -- Preprocessor ----------------------------------------------------------
#> 4 Recipe Steps
#>
#> * step_tokenize()
#> * step_tokenfilter()
#> * step_tfidf()
#> * step_normalize()
#>
#> -- Model ----------------------------------------------------------------
#> Linear Support Vector Machine Specification (regression)
#>
#> Computational engine: LiblineaR
```

The `final_wf` workflow now has a finalized value for `max_tokens`.

We can now fit this finalized workflow on training data and *finally* return to our testing data.

Notice that this is the first time we have used our testing data during this entire chapter; we compared and now tuned models using resampled data sets instead of touching the testing set.

We can use the function `last_fit()` to **fit** our model one last time on our training data and **evaluate** it on our testing data. We only have to pass this function our finalized model/workflow and our data split.

```
final_fitted <- last_fit(final_wf, scotus_split)

collect_metrics(final_fitted)
```

```
#> # A tibble: 2 x 4
#>   .metric .estimator .estimate .config
#>   <chr>   <chr>          <dbl> <chr>
#> 1 rmse    standard      13.8   Preprocessor1_Model1
#> 2 rsq     standard       0.921 Preprocessor1_Model1
```

The metrics for the test set look about the same as the resampled training data and indicate we did not overfit during tuning. The RMSE of our final model has improved compared to our earlier models, both because we are combining multiple preprocessing steps and because we have tuned the number of tokens.

The output of `last_fit()` also contains a fitted model (a `workflow`, to be more specific), that has been trained on the *training* data. We can `tidy()` this final result to understand what the most important variables are in the predictions, shown in Figure 6.7.

```
scotus_fit <- pull_workflow_fit(final_fitted$.workflow[[1]])

scotus_fit %>%
  tidy() %>%
  filter(term != "Bias") %>%
  mutate(
    sign = case_when(estimate > 0 ~ "Later (after mean year)",
                     TRUE ~ "Earlier (before mean year)"),
    estimate = abs(estimate),
    term = str_remove_all(term, "tfidf_text_")
  ) %>%
  group_by(sign) %>%
  top_n(20, estimate) %>%
  ungroup() %>%
  ggplot(aes(x = estimate,
             y = fct_reorder(term, estimate),
             fill = sign)) +
  geom_col(show.legend = FALSE) +
  scale_x_continuous(expand = c(0, 0)) +
  facet_wrap(~sign, scales = "free") +
  labs(
    y = NULL,
    title = paste("Variable importance for predicting year of",
                  "Supreme Court opinions"),
    subtitle = paste("These features are the most importance",
                     "in predicting the year of an opinion")
  )
```

The tokens (unigrams or bigrams) that contribute in the positive direction, like "court said" and "constitutionally," are associated with higher, later years; those that contribute in the negative direction, like "ought" and "the judges," are associated with lower, earlier years for these Supreme Court opinions.

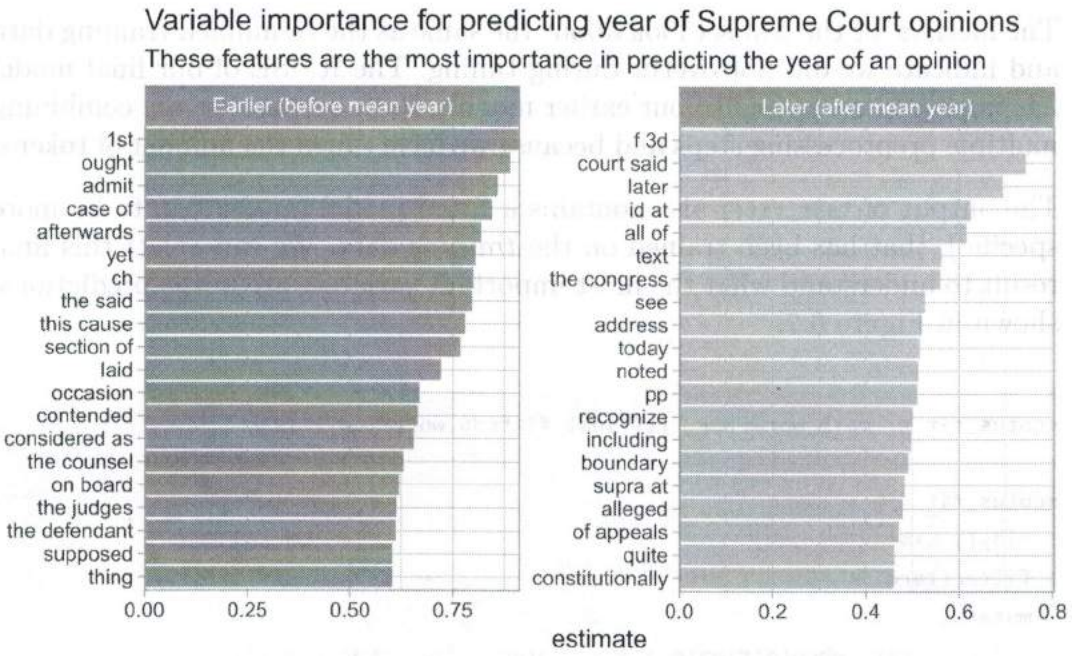



**FIGURE 6.7:** Some words or bigrams increase a Supreme Court opinion's probability of being written later (more recently) while some increase its probability of being written earlier

Some of these features are unigrams and some are bigrams, and stop words are included because we did not remove them from the model.

We can also examine how the true and predicted years compare for the testing set. Figure 6.8 shows us that, like for our earlier models on the resampled training data, we can predict the year of Supreme Court opinions for the testing data starting from about 1850. Predictions are less reliable before that year. This is an example of finding different error rates across sub-groups of observations, like we discussed in the overview to these chapters; these differences can lead to unfairness and algorithmic bias when models are applied in the real world.

```
final_fitted %>%
  collect_predictions() %>%
  ggplot(aes(year, .pred)) +
  geom_abline(lty = 2, color = "gray80", size = 1.5) +
  geom_point(alpha = 0.3) +
  labs(
```

```
    x = "Truth",
    y = "Predicted year",
    title = paste("Predicted and true years for the testing set of",
                  "Supreme Court opinions"),
    subtitle = "For the testing set, predictions are more reliable after 1850"
)
```

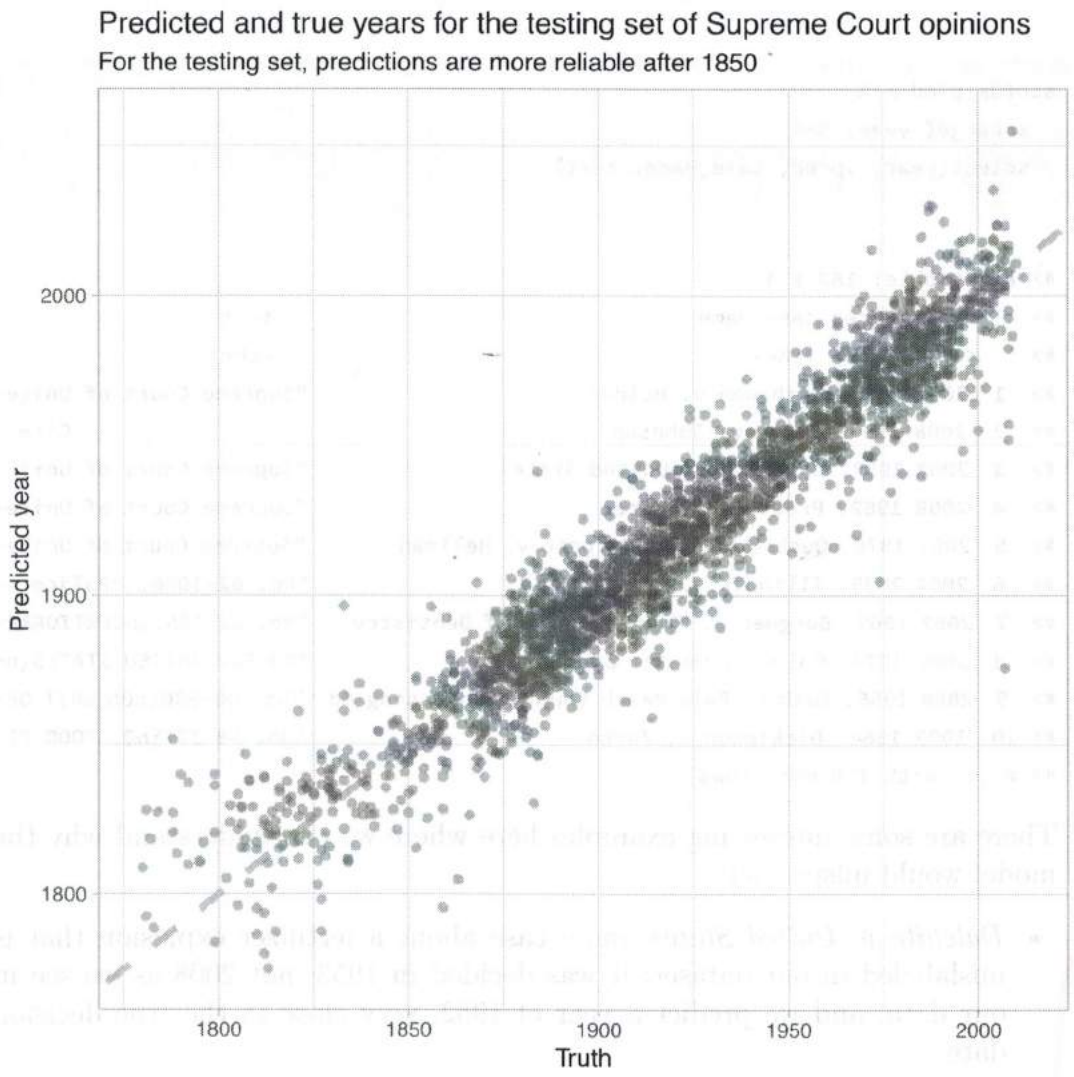



**FIGURE 6.8:** Predicted and true years from a linear SVM regression model with bigrams and unigrams

Finally, we can gain more insight into our model and how it is behaving by looking at observations from the test set that have been *mispredicted*. Let's bind together the predictions on the test set with the original Supreme Court opinion test data and filter to observations with a prediction that is more than 25 years wrong.

```
scotus_bind <- collect_predictions(final_fitted) %>%
  bind_cols(scotus_test %>% select(-year, -id)) %>%
  filter(abs(year - .pred) > 25)
```

There isn't too much training data to start with for the earliest years, so we are unlikely to quickly gain insight from looking at the oldest opinions. However, what do the more recent opinions that were predicted inaccurately look like?

```
scotus_bind %>%
  arrange(-year) %>%
  select(year, .pred, case_name, text)
```

```
#> # A tibble: 168 x 4
#>     year .pred case_name                                    text
#>    <dbl> <dbl> <chr>                                       <chr>
#>  1  2009 2055. Nijhawan v. Holder                          "Supreme Court of Unite~
#>  2  2008 1957. Green v. Johnson                            "                  Cite ~
#>  3  2008 1952. Dalehite v. United States                   "Supreme Court of Unite~
#>  4  2008 1982. Preston v. Ferrer                           "Supreme Court of Unite~
#>  5  2007 1876. Quebec Bank of Toronto v. Hellman           "Supreme Court of Unite~
#>  6  2004 2035. Illinois v. Lidster                         "No. 02-1060.\nPolice s~
#>  7  2002 1969. Borgner v. Florida Board of Dentistry       "No. 02-165.\nCERTIORAR~
#>  8  2000 1974. Ohler v. United States                      "OHLERv.UNITED STATES\n~
#>  9  2000 1955. Bush v. Palm Beach County Canvassing Bd.    "No. 00-836\nON WRIT OF~
#> 10  1999 1964. Dickinson v. Zurko                          "No. 98 377\nQ. TODD DI~
#> # ... with 158 more rows
```

There are some interesting examples here where we can understand why the model would mispredict:

- *Dalehite v. United States* was a case about a fertilizer explosion that is mislabeled in our dataset; it was decided in 1953, not 2008 as we see in our data, and we predict a year of 1952, very close to the true decision date.

- *Bush v. Palm Beach County Canvassing Board* in 2000 was part of the fallout of the 2000 presidential election and dealt with historical issues like the due process clause of the U.S. Constitution; these "old" textual elements push its prediction much earlier than its true date.

Looking at examples that your model does not perform well for is well worth your time, for similar reasons that exploratory data analysis is valuable before you begin training your model.

## 6.10 Summary

You can use regression modeling to predict a continuous variable from a data set, including a text data set. Linear support vector machine models, along with regularized linear models (which we will cover in the next chapter), often work well for text data sets, while tree-based models such as random forest often behave poorly in practice. There are many possible preprocessing steps for text data, from removing stop words to n-gram tokenization strategies to lemmatization, that may improve your model. Resampling data sets and careful use of metrics allow you to make good choices among these possible options, given your own concerns and priorities.

### 6.10.1 In this chapter, you learned:

- what kind of quantities can be modeled using regression
- to evaluate a model using resampled data
- how to compare different model types
- about measuring the impact of n-gram tokenization on models
- how to implement lemmatization and stop word removal with text models
- how feature hashing can be used as a fast alternative to bag-of-words
- about performance metrics for regression models