Text Analysis and Text Mining | 27

by **Patrick Boily**, with contributions from **Andrew Macfie**

In this chapter, we introduce the basic notions of **text mining** so that we can learn how to extract insight from text data. We also present some elementary applications (drawn from the world of machine learning), including **sentiment analysis**, setting the stage for a more sophisticated treatment of **natural language processing** and **large language models** (see Chapters 31 and 32).

27.1 Introduction

We start our foray into text analysis by discussing a use case for text mining which made the news early in 2017.

27.1.1 Case Study: BOTUS

In 2013, the BBC reported on various ways in which social media giant Twitter was changing the world, detailing specific instances in the fields of business, politics, journalism, sports, entertainment, activism, arts, and law [27].

It is not always clear what influence Twitter users have, if any, on world events or business and cultural trends; it was once thought (perhaps without appropriate evidence) that entertainers, athletes, and celebrities, that is to say, users with extremely high followers to following ratios, wielded more "influence" on the platform than world leaders [4]. Certainly, such users continue to be among the most popular – as of September 13, 2017, Twitter's 40 most-followed accounts tend to belong to entertainers, celebrities, and athletes, with a few exceptions [11].

One account has recently bridged the gap between celebrity and politics in an explosive manner: `@realDonaldTrump`, which belongs to the 45th President of the United States of America, has maintained a very strong presence on Twitter. As of September 13, 2017, the account had 38,205,766 followers, and it was the 26th most-followed account on the planet, producing 35,755 tweets since it was activated in March 2009, and roughly 6 tweets a day in August of 2017 [11].

Titles BOTUS [15], Trump & Dump Bot [42]

Authors Tradeworx (BOTUS), T3 (Trump & Dump)

Date 2017

Sponsor NPR's podcast *Planet Money* (BOTUS)

Methods sentiment analysis, social media monitoring, AI, real-time analysis, simulations

Objective There is some evidence to suggest that tweets from the 45th POTUS may have an effect on the stock market [21]. Can sentiment analysis and AI be used to take real-time advantage of the tweets' unpredictable nature? Let's take a look at bots built for that purpose by NPR's *Planet Money* and by T3 (an Auston advertising agency).

Methodology Tradeworx followed these steps:

1. *Data collection*: tweets from `@realDonaldTrump` are collected for analysis.
2. *Sentiment analysis of tweets*: each tweet is given a sentiment score on the positive/negative axis.
3. *Validation*: the sentiment analysis scoring must be validated by observers: are human-identified positive or negative tweets correctly identified by BOTUS?
4. *Identification of the company in a tweet*: is the tweet even about a company? If so, which one?
5. *Determining the trading universe*: are there companies that should be excluded from the bot's trading algorithms?
6. *Classifying tweets as "applicable" or "unapplicable"*: is a tweet's sentiment strong enough for BOTUS to engage the trading strategy?
7. *Determining a trading strategy*: how soon after a flagged tweet does BOTUS buy a company's stock, and how long does it hold it for?
8. *Testing the trading strategy on past data*: how would BOTUS have fared from the U.S. Presidential Election to April 2017? What are BOTUS' limitations?

T3's Trump and Dump uses a similar process (see Figure 27.1).

Data The data consists of:

- tweets by `@realDonaldTrump` (from around Election Day 2016 through the end of March 2017 for BOTUS; no details are given for T3) (see Figure 27.2 for sample);
- a database of publicly traded companies, such as can be found at [17, 18, 14], although which of these were used, if any, is not specified (no explicit mention is made for BOTUS), and
- stock market data for real-time pricing (Google Finance for T3) and backcasting simulation (for BOTUS, source unknown).

It is not publicly known whether the 2 bots are upgrading their algorithms by including new data as time passes.

Figure 27.1: T3's Trump and Dump process [42].

Strengths and Limitations of Algorithms and Procedure

- In sentiment analysis, an algorithm analyzes documents in an attempt to identify the attitude they express or the emotional response they seek. It presents numerous challenges, mostly related to the richness and flexibility of human languages and their syntax variations, the context-dependent meaning of words and lexemes, the use of sarcasm and figures of speech, and the lack of perfect inter-rater reliability among humans [35]. As it happens, `@realDonaldTrump` is not much of an ironic tweeter – "sad" and "great" are usually meant in their most general sense. This greatly simplifies the analysis.
- The bots have to learn to recognize whether a tweet is directed at a publicly traded company or not. In certain cases, the ambiguity can be resolved relatively easily with an appropriate training set (Apple the company vs. apple the food-item, say), but no easy solutions were found in others (Tiffany the company vs. Tiffany the daughter, for example). Rather than have humans step in and instruct BOTUS when it faces uncertainty (which would go against the purpose of the exercise), a decision was made to exclude these cases from the trading universe. The T3 documentation does not describe such details.

Figure 27.2: Examples of `@realDonaldTrump` tweets involving Delta, Toyota Motor, L.L.Bean, Ford, and Boeing.

- Once the bot knows how to rate `@realDonaldTrump`'s tweets and to identify when he tweets about publicly-traded companies, the next question is to determine what the trading strategy should be. If the tweet's sentiment is negative enough T3 shorts the company's stock.[1] Of course, this requires first purchasing the stock (so that it can be shorted). Planet Money's decision was similar: buy once the tweet is flagged, and sell right away... but what does "right away" mean in this context? There is a risk involved: if the stock goes back up before BOTUS has had a chance to purchase the low-priced stock, it will lose money. To answer that question, Tradeworx simulated the stock market over the last few months, introducing the tweets, and trying out different trading strategies. It turns out that, in this specific analysis, "right away" can be taken to be 30 minutes after the tweet.

1: It sells the stock when the price is high, that is, *before* the tweet has had the chance to bring the stock down, and it repurchases it once the price has been lowered by the tweet, but before the stock has had the chance to recover.

Results, Evaluation and Validation For a trading bot, the validation is in the pudding, as they say – do they make money? T3's president says that their bot is profitable (they donate the proceeds to the ASPCA) [42]: for instance, they netted a return of 4.47% on `@realDonaldTrump`'s Delta tweet (see Figure 27.2); however, he declined to provide specific numbers (and made vague statements about providing monthly reports, which I have not been able to locate) [31].

The BOTUS process was more transparent, and we can point to Planet Money's transcript for a discussion on sentiment analysis validation (comparing BOTUS's sentiment rankings with those provided by human observers, or running multiple simulations to determine the best trading scenario) [15] – but it suffers from a serious impediment: as of roughly 4 months after going online, **it still had not made a single trade** [13]!

The reasons are varied (see Figures 27.3 and 27.4), but the most important setback was that `@realDonaldTrump` had not made a single valid tweet about a public company whose stock BOTUS could trade during the stock market business hours. Undeterred, Planet Money relaxed its trading strategy: if `@realDonaldTrump` tweets during off-hours, BOTUS will short the stock at the market's opening bell.

This is a risky approach, and so far it has not proven very effective: a single trade of Facebook's trade, on August 23rd, which resulted in a loss of 0.30$ (see Figure 27.4).

Figure 27.3: BOTUS reporting on its trades (part 1).

Take-Aways As a text analysis and scenario analysis project, both BOTUS and Trump & Dump are successful – they present well-executed sentiment analyses, and a simulation process that finds an optimal trading strategy. As predictive tools, they are sub-par (as far as we can tell), but for reasons that (seem to) have little to do with data analysis *per se*.

Figure 27.4: BOTUS reporting on its trades (part 2).

Unfortunately, this is not an atypical feature of descriptive data analysis: we can explain what has happened (or what is happening), but the modeling assumptions are not always applicable to the predictive domain.

27.1.2 Text Analysis

It has been said that "the only valid model of the universe might just be the universe itself" [author unknown]. With that maxim in mind, it would appear that there is simply no substitute – in order to get meaning out of documents, we first need to read them in their entirety.

This interpretation is overly simplistic, however. Consider the works of William Shakespeare, for instance, to whom 38 plays (or so) and over 150 sonnets and poems written in the late 1600s and early 1700s have been attributed [48]. It is fairly straightforward to have a go at *Macbeth*, say; one only needs to pick up a printed or digital copy, or sit through any of the numerous stagings of the Scottish play, and voilà! – instant meaning and themes: ambition, lust for power, appearances vs. reality, temptation, and guilt haunting evildoers [6].

Of course, current readers might find Early Modern English verses difficult to follow without annotation, and those themes might only reveal themselves upon repeated readings or viewings. Lovers of English verse might fully enjoy this arduous process, but non-native speakers might wonder if data analysis methods could provide a complete (or near enough) Shakespeare experience without having to go through *The Complete Works of Shakespeare*, or even *The Complete Works of William Shakespeare (Abridged)* [7, 28]? Is there some "essential" Shakespeare-ness that lurks in his plays and sonnets? Common threads, common humour, common structure, common themes?

These questions (and others, such as **authorship questions** [10, 19, 49, 12]) may only be of interest to scholars, but there is a more compelling reason to study automated text analysis, if only as a first pass – in the age of "fake news" [30, 22, 29, 2], social media, and Amazon reviews, when the tweets of high-profile politicians have a definite and measurable influence on the stock market [13, 15, 16, 5, 33, 34] or when live analysis of panic conversations can drive automated emergency responses [38], we simply produce too much text data for any group of individuals to analyze and understand without technological assistance.[2]

2: In other words, the genie is out of the bottle – what can we do to make sure we understand what it's *really* saying?

27.1.3 Text Mining vs. Natural Language Processing

Text mining is the collection of processes by which we can extract useful insights from text. Inherent in this definition is the idea of **automated data reduction**: useful insights (whether in the form of summaries, sentiment analyses, or word counts) ought to be "smaller" and "more organized" (from a data point of view) than the original text.

For short texts, however, the benefits of text analysis may not always be evident. Consider, for instance, the following excerpt from a lawn mowing instruction manual:

> Before starting your mower inspect it carefully to ensure that there are no loose parts and that it is in good working order.

This is a fairly **concise** and **structured** way to convey a message. It could be further shortened and organized, perhaps, but it's not clear that one would gain much from the process. In more complex case, the process is less obvious; we discussed data reduction in a more general context previously and encourage readers of this chapter to first take a look at Section 23.1 (*Data Reduction for Insight*).

Ted Kwartler suggests the following **text mining workflow** [26]:

1. problem definition and goals;
2. identify text to be collected;
3. text organization;
4. feature extraction;
5. analysis, and
6. reach an insight, conclusion, or output.

In this chapter, we will further take the position that text mining is the application of **data science and machine learning tasks to text documents**, such as:

- **supervised learning** (classification and class probability estimation, value estimation);

Figure 27.5: A poutine (on the left); an abomination in the eyes of all right-thinking sentient beings (on the right).

- **unsupervised learning** (association rules and hypothesis discovery, similarity matching, clustering);
- **semi-supervised learning** (profiling, link prediction, data reduction, etc.), and
- **visualization and representation**.

Typical applications include **authorship questions** (classification), **sentiment analysis** (value estimation), **taxonomy creation** and **topic modeling** (clustering), **text description**, and **text visualization**.[3]

3: In order to take full advantage of the underlying data science machinery, documents may first need to be mapped into numerical or categorical features, via **kernel transformations**.

We will explore the data preparation process and simple text mining models in Sections 27.2 and 27.3, respectively.

Natural language processing (NLP), in contrast, has a long history of lofty goals, which more or less boil down to developing machines that **react "appropriately" while interacting with (natural) human languages**.[4] The focus of NLP tasks tends towards **"understanding" languages**; with common tasks including:

4: Think of ChatGPT, as a recent example.

- **syntax** (lemmatization, part-of-speech tagging, parsing, terminology extraction, sentence boundary disambiguation, stemming, word segmentation, etc.);
- **semantics** (machine translation, language generation, named entity recognition, optical character recognition, questions and answers, sentiment analysis, textual entailment, topic segmentation, word sense disambiguation, etc.);
- **discourse** (coreference resolution, discourse analysis, summarization, etc.), and
- **speech** (recognition, segmentation, text-to-speech, etc.). [47]

Most natural human languages rules are **dynamic**, and usage may change drastically in space and time – a *poutine* is not the same dish in New Brunswick as it is in Québec, for instance (see Figure 27.5). For another example, consider the meaning of the word *awful*, which drifted from

> "commanding profound respect or reverential fear"

to

> "frightful, very ugly, monstrous"

from 1000 AD onward.

Other issues arise from dialect variations and individual-specific speech patterns, either due to **linguistic drift**, **influence from other languages**, **sarcasm**, **idioms**, **figures of speech**, and so forth. The intended meaning is often clear to experienced human speakers based on the specific context, but it is believed that natural language understanding is **AI-hard** – a

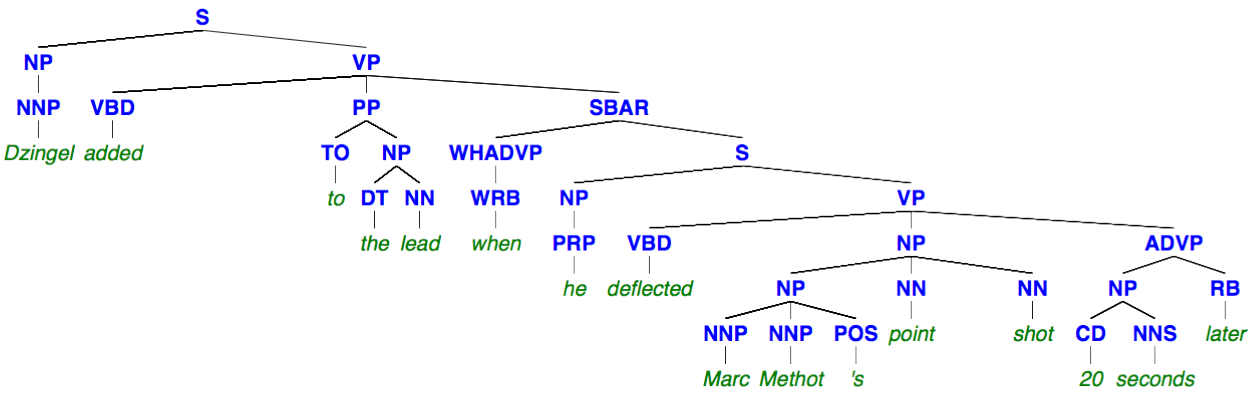


Figure 27.6: Syntactic parsing of a sentence using the Stanford parser [41].

complete resolution of the issues would require the ability to make computers as "smart" as humans [20], in ways that we haven't seen from our machines yet.[5] Thankfully, we rarely need a resolution at the most general level; in many areas, the current state of the art produces results which are acceptable to a large class of users.

5: Scratching the surface of large language models shows how far we remain, in spite of recent progresses.

Since the 1990s, the NLP community has adopted a machine learning paradigm, which has provided advantages over the classical hard-coded hand-produced rules. Statistical machine translation, for instance, can take advantage of domain constraints and formal language habits to reduce the space of outputs and produce accurate translations of technical documents [45]. We will take a more detailed look at NLP concepts and tasks in Chapter 32.

The distinction between text mining and NLP may seem spurious; most researchers and practitioners do not get bogged down in such details. When we focus on the **data science** side of the equation, we'll refer to text analysis as **text mining**; when we focus on **language analysis and understanding**, we'll refer to it as **natural language processing**.

27.2 Basics of Text Analysis

Let's take a look at two schools of thought regarding text mining: semantic parsing and bag-of-words mining.

Semantic Parsing In this view of text mining, word order and word type play a crucial role. The idea is to use a large number of hand-parsed sentences to train a model that outputs the most likely grammatical analysis of a sentence. Words are tagged along a tree structure, and may have multiple features. This information can then be used to extract insights about the sentence or document.

For instance, consider the sentence

> (S1) Dzingel added to the lead when he deflected Marc Methot's point shot 20 seconds later [37],

a syntactic parsing of which is shown in Figure 27.6.[6]

6: The output of the Stanford parser [41].

Another display is provided below:

```
(ROOT
 (S
  (NP (NNP Dzingel))
  (VP (VBD added)
    (PP (TO to)
      (NP (DT the) (NN lead)))
    (SBAR
      (WHADVP (WRB when))
      (S
        (NP (PRP he))
        (VP (VBD deflected)
          (NP
            (NP (NNP Marc) (NNP Methot) (POS ’s))
            (NN point) (NN shot))
          (ADVP
            (NP (CD 20) (NNS seconds))
            (RB later)))))))
```

From the tree diagram, a human observer can clearly see that “Marc Methot” is correctly parsed as a *noun phrase* (NP), that the “’s” is correctly identified as a *possessive marker* (POS), and that “Marc Methot’s point shot” is correctly shown as a NP (built from 2 *singular proper nouns*, NNP), but the parser fails to recognize “point shot” as an NP.[7]

7: The two displays are, of course, equivalent. A computer program can be used to easily go from one to the other; a human with the right experience would find both as insightful. But it’s certainly easier for a neophyte to comprehend the tree diagram. Why is that? Is it simply because we are a visual species? Or because most of us have parsed sentence fragments in our native languages as youths?

In another parsing (using the Enju parser, see Figure 27.7), “Marc Methot’s point shot 20 seconds later” is tagged as a *simple declarative clause* (S), but “Marc Methot’s point” and “shot 20 seconds later” are wrongly identified as a NP and a *verb phrase* (VP), respectively, underscoring the importance of parsing to our understanding of a sentence.

The **part-of-speech tagging** for the sentence is shown in the table below:

Word	Tag	Word	Tag
Dzingel	NNP	Marc	NNP
added	VBD	Methot	NNP
to	TO	’s	POS
the	DT	point	NN
lead	NN	shot	NN
when	WRB	20	CD
he	PRP	seconds	NNS
deflected	VBD	later	RB

The meaning of common tags are provided in Tables 27.1 and 27.2. Notice how relational insight between the parts-of-speech has gotten lost (or is not displayed, at the very least).

The Stanford parser provides a list of **universal dependencies**:

- nsubj(added-2, Dzingel-1)
- root(ROOT-0, added-2)
- case(lead-5, to-3)
- det(lead-5, the-4)
- nmod(added-2, lead-5)

- advmod(deflected-8, when-6)
- nsubj(deflected-8, he-7)
- advcl(added-2, deflected-8)
- compound(Methot-10, Marc-9)
- nmod:poss(shot-13, Methot-10)
- case(Methot-10, ’s-11)
- compound(shot-13, point-12)
- dobj(deflected-8, shot-13)
- nummod(seconds-15, 20-14)
- nmod:npmod(later-16, seconds-15)
- advmod(deflected-8, later-16)

For instance, “he” (the 7th token in the sentence) is he *nominal subject* (nsubj) of “deflected” (the 8th token), “point shot” is recognized as a *compound*, and “shot” (the 13th token) is the *direct object* (dobj, the second most core argument of a verb after the subject) of “deflected” (the 8th token). A list of codes and meanings for UD (v2) can be found in Tables 27.3 and 27.4, on pp. 1718-1719.

Bag of Words In this view of text mining, only the words are important – it is frequency (and relative frequency) that wins the day. In semantic parsing, the words have attributes depending on their position and role in the document’s sentences; in bag of words analysis, **the words themselves are attributes of the document**. Our sentence S1 is simply a collection of words, arranged here alphabetically:

> ’s, 20, added, deflected, Dzingel, he, later, lead, Marc, Methot, point, seconds, shot, the, to, when.

The fact that “point shot” is a noun phrase is not significant, but the fact that “point” and “shot” appear in the list is significant – it is the **relative frequencies** of the terms that provide information about the document or collection of documents (such as intent and themes).

In the rest of the section, we will take a look at the fundamental concepts underlying text preparation.[8] Concrete illustrations of these notions are provided in Section 27.4.

8: Some of the topics will be revisited in Chapter 32.

27.2.1 Text Collection

Nowadays, text data is typically collected from the Web, either through **web scraping** or with the help of a specialized **application programming interface** (API), as we discussed in Chapter 16. **Manual collection** is another option (although strongly discouraged when faced with more than a few dozen documents to collect).

Optical character readers (OCR) can also digitize scanned images and the technology has improved tremendously over the last 20 years; **manual entry of non-digital text data** can be used as a last resort, but it is tedious and likely to introduce infelicities and transcription errors.

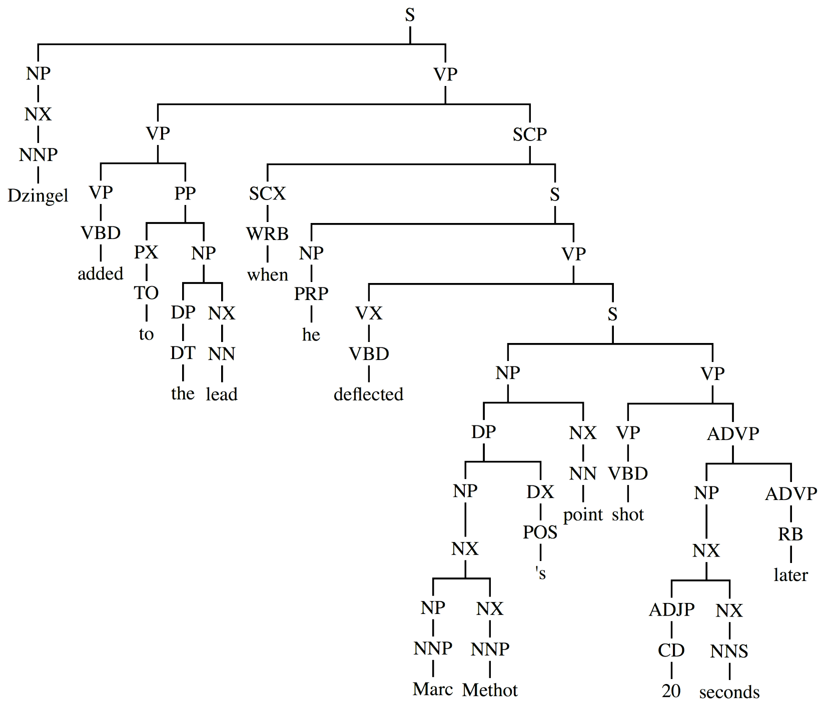


Figure 27.7: Abridged syntactic parsing of a sentence using the Enju English parser [32].

27.2.2 Text Representation

No matter where data comes from and what analyses we hope to run on it, the crucial first step requires extraction, formatting, and storage to a data structure with appropriate numerical properties [39]:

- a **string** or vector of characters, with language-specific encoding;
- a collection of text documents (with meta information) called a **corpus** ('permanent' when stored on disk; 'volatile' when held in RAM);
- a **document-term matrix** (DTM) – or the transposed **term-document matrix** (TDM) – where the rows are the documents, the columns are the terms (see Figure 27.8), and the entries represent some text statistic;
- a **tidy text dataset** containing one **token** (single word, n-gram, sentence, paragraph) per row.

The DTM/TDM representations are **essential** to any statistical analysis of text data – it is on these entities that machine learning algorithms are unleashed.

27.2.3 Text Processing

As with every form of data, text data requires extensive **cleaning** and **processing**. Cleaning text data is, to put it mildly, even less pleasant a process than cleaning numeric or categorical data. There are challenges due to the nature of the data: how would one go about finding anomalies in the text? Outliers? Is the concept even definable for text data?

part-of-speech tagset

CC	Coordinating conj.	TO	infinitival *to*
CD	Cardinal number	UH	Interjection
DT	Determiner	VB	Verb, base form
EX	Existential there	VBD	Verb, past tense
FW	Foreign word	VBG	Verb, gerund/present pple
IN	Preposition	VBN	Verb, past participle
JJ	Adjective	VBP	Verb, non-3rd ps. sg. present
JJR	Adjective, comparative	VBZ	Verb, 3rd ps. sg. present
JJS	Adjective, superlative	WDT	Wh-determiner
LS	List item marker	WP	Wh-pronoun
MD	Modal	WP$	Possessive *wh*-pronoun
NN	Noun, singular or mass	WRB	Wh-adverb
NNS	Noun, plural	#	Pound sign
NNP	Proper noun, singular	$	Dollar sign
NNPS	Proper noun, plural	.	Sentence-final punctuation
PDT	Predeterminer	,	Comma
POS	Possessive ending	:	Colon, semi-colon
PRP	Personal pronoun	(	Left bracket character
PP$	Possessive pronoun	)	Right bracket character
RB	Adverb	"	Straight double quote
RBR	Adverb, comparative	‘	Left open single quote
RBS	Adverb, superlative	“	Left open double quote
RP	Particle	’	Right close single quote
SYM	Symbol	”	Right close double quote

syntactic tagset

ADJP	Adjective phrase
ADVP	Adverb phrase
NP	Noun phrase
PP	Prepositional phrase
S	Simple declarative clause
SBAR	Subordinate clause
SBARQ	Direct question introduced by *wh*-element
SINV	Declarative sentence with subject-aux inversion
SQ	Yes/no questions and subconstituent of SBARQ excluding *wh*-element
VP	Verb phrase
WHADVP	Wh-adverb phrase
WHNP	Wh-noun phrase
WHPP	Wh-prepositional phrase
X	Constituent of unknown or uncertain category
*	“Understood” subject of infinitive or imperative
0	Zero variant of *that* in subordinate clauses
T	Trace of wh-Constituent

Table 27.1: Penn treebank tagset (part 1) [43].

Character encoding may also produces surprises: a text (or a part of text) that looks completely normal to the human eye may be unreadable to a computer because it was expecting a different encoding system. There are probabilistic ways to detect a document's encoding, and ways to coerce a specified encoding – if you are working with text data and your code balks at doing something it should be able to do and none of the usual fixes apply, look into the encoding situation.

Another issue is that legitimate **spelling mistakes** and **typographical errors** are hard to catch in large documents (even with spell-checkers), to say nothing of:

- **accent representation** (*ya new cah's wicked pissa!*);
- **neologisms** and **portemanteaus** (*ruthfull; can't you tell that I'm planning prevenge?*);
- **poor translations** or **foreign words** (*business goose; llongyfarchiadau*);
- **puns** and **play-on-words** (*they were jung and easily freudened!*);
- **specialized vocabulary** (*clopen; poset*);
- **fictional names** and **places** (*Qo'noS; Kilgore Trout*);
- **slang** and **curses** (*skengfire; #$&#!*);
- **mark-up** and **tags** (*<b>; \includegraphics*);
- **uninformative text information** (page number; ISBN blurb), etc.

functional tagset

Text Categories	
-HLN	headlines and datelines
-LST	list markers
-TTL	titles
Grammatical Functions	
-CLF	true clefts
-NOM	non NPs that function as NPs
-ADV	clausal and NP adverbials
-LGS	logical subjects in passives
-PRD	non VP predicates
-SBJ	surface subject
-TPC	topicalized and fronted constituents
-CLR	closely related - see text
Semantic Roles	
-VOC	vocatives
-DIR	direction & trajectory
-LOC	location
-MNR	manner
-PRP	purpose and reason
-TMP	temporal phrases

pseudo-attachments tagset

ICH	Interpret Constituent Here
PPA	Permanent Predictable Ambiguity
RNR	Right Node Raising
EXP	Expletive

disfluency annotations

Utterances	
/	end of complete utterance
-/	end of incomplete utterance
Non-sentence elements	
F	fillers (*uh*, *um*, *huh*, *oh*, etc.)
E	explicit editing term (*I mean*, *sorry*, etc.)
D	discourse marker (*you know*, *well*, etc.)
C	coordinating conjunction (*and*, *and then*, *but*, etc.)
A	aside
Restarts	
[*RM* + *RR*]	restart with repair (see text)
[*RM*+]	restart without repair

Table 27.2: Penn treebank tagset (part 2) [43].

	Nominals	Clauses	Modifier words	Function Words
Core arguments	nsubj obj iobj	csubj ccomp xcomp		
Non-core dependents	obl vocative expl dislocated	advcl	advmod discourse	aux cop mark
Nominal dependents	nmod appos nummod	acl	amod	det clf case
Coordination	**MWE**	**Loose**	**Special**	**Other**
conj cc	fixed flat compound	list parataxis	orphan goeswith reparandum	punct root dep

Table 27.3: The 37 universal syntactic relations used in Universal Dependencies v2. The upper part of the table follows the main organizing principles of the UD taxonomy such that rows correspond to functional categories in relation to the head (core arguments of clausal predicates, non-core dependents of clausal predicates, and dependents of nominals) while columns correspond to structural categories of the dependent (nominals, clauses, modifier words, function words). The lower part of the table lists relations that are not dependency relations in the narrow sense [44].

Code	Relation
acl	clausal modifier of noun (adjectival clause)
advcl	adverbial clause modifier
advmod	adverbial modifier
amod	adjectival modifier
appos	appositional modifier
aux	auxiliary
case	case marking
cc	coordinating conjunction
ccomp	clausal complement
clf	classifier
compound	compound
conj	conjunct
cop	copula
csubj	clausal subject
dep	unspecified dependency
det	determiner
discourse	discourse element
dislocated	dislocated elements
expl	expletive
fixed	fixed multiword expression
flat	flat multiword expression
goeswith	goes with
iobj	indirect object
list	list
mark	marker
nmod	nominal modifier
nsubj	nominal subject
nummod	numeric modifier
obj	object
obl	oblique nominal
orphan	orphan
parataxis	parataxis
punct	punctuation
reparandum	overridden disfluency
root	root
vocative	vocative
xcomp	open clausal complement

Table 27.4: Universal dependency relations, alphabetical listing [44].

The process can be simplified to some extent with the help of **regular expressions** and **text pre-processing functions** (see Section 27.4):

> Specific pre-processing steps will vary based on the project. For example, the words used in tweets are vastly different than those used in legal documents, so **the cleaning process can also be quite different** [26].

We shall illustrate the pre-processing function with the help of the following string:

> <i>He</i> went to bed at   2 A.M. It\'s way too late! He was only 20% asleep at first, but sleep eventually came.

What can we do with this string?

	Document 1	Document 2	Document 3	…	Document *N*	Sum
Token 1	0	0	1	62	3	66
Token 2	0	1	0	61	2	64
Token 3	1	0	3	101	0	105
…	112	24	38	84	0	258
Token *M*	2	2	0	12	3	19
Sum	115	27	42	320	8	

Figure 27.8: Term-document matrix/ document-term matrix for a hypothetical corpus, with Row Sums and Column Sums.

- Modify every upper case character to its corresponding **lower case** version (avoid if seeking proper nouns and names)

  <i>he</i> went to bed at 2 a.m. it\'s way too late! he was only 20% asleep at first, but sleep eventually came.

- Remove all **punctuation marks** (avoid if seeking emojis):

  iHei went to bed at 2 AM Its way too late He was only 20 asleep at first but sleep eventually cam

- Remove all **numerals** (not ideal when text mining quantities):

  <i>He</i> went to bed at A.M. It\'s way too late! He was only % asleep at first, but sleep eventually came.

- Remove all extraneous **white space**:

  <i>He</i> went to bed at 2 A.M. It\'s way too late! He was only 20% asleep at first, but sleep eventually came.

- Remove characters within **brackets** (and the brackets):

  He went to bed at 2 A.M. It\'s way too late! He was only 20% asleep at first, but sleep eventually came.

- Replace all **numerals with words**:

  <i>He</i> went to bed at two A.M. It\'s way too late! He was only twenty% asleep at first, but sleep eventually came.

- Replace **abbreviations**:

  <i>He</i> went to bed at 2 AM Itś way too late! He was only 20% asleep at first, but sleep eventually came.

- Replace **contractions** (avoid if seeking non-formal speech):

  <i>He</i> went to bed at 2 A.M. It is way too late! He was only 20% asleep at first, but sleep eventually came.

- Replace **symbols with words**:

  <i>He</i> went to bed at 2 A.M. Itś way too late! He was only 20 percent asleep at first, but sleep eventually came.

We typically also remove **stop words** ("a", "an", "the", etc.) and **uninformative words** (which tend to be highly context-dependent), as these unnecessarily increase the number of columns in the DTM.[9] We also usually **stem words** and **complete the stems** to remove unnecessary variation in the text: "sleepful", "sleeping", "sleeps", "slept" all **convey**

9: See the **curse of dimensionality**, Chapter 23.

the meaning of "sleep" and might as well be replaced by the latter term (which is a completed stem or a **lemma**).[10]

10: But there are complications, as we will discuss in Chapter 32: in "operations research", "operating system" and "operative dentistry", the stem "operati" has **different meanings**.

In the BoW approach, then, the text string on which we have been working could be pre-processed to:

> he go bed 2 am way late he 20 percent sleep first sleep eventually come

Note that this is not the only reasonable BoW preparation – as always, **context matters**.

27.2.4 Text Statistics

The problem of how to represent a corpus as BoW DTM is simple to solve, but it requires analysts to make use of their agency.

Consider a **corpus** $\mathcal{C} = \{d_1, \ldots, d_N\}$ consisting of N **documents**, with a **BoW dictionary** $\mathcal{D}_\mathcal{C} = t_1, \ldots, t_M$ consisting of M distinct **terms**.[11] For instance, if the corpus is

11: For $i = 1, \ldots, N$, the ith document's dictionary $\mathcal{D}_{\mathcal{C};i} = \{t_{i;1}, \ldots, t_{i;M_i}\}$ consists of the distinct terms of $\mathcal{D}_\mathcal{C}$ found in d_i.

$$\mathcal{C} = \{\text{"the dogs who have been let out"}, \text{"who did that"}, \text{"my dogs breath smells like dogs food"}\},$$

then $N = 3$, $d_1 =$ "the dogs who have been let out", $d_2 =$ "who did that", and $d_3 =$ "my dogs breath smells like dogs food", $M_1 = 7$, $M_2 = 3$, $M_3 = 7$, $M = 14$, and the BoW dictionary terms are:

- $t_1 =$ "been",
- $t_2 =$ "breath",
- $t_3 =$ "did",
- $t_4 =$ "dogs",
- $t_5 =$ "food",
- $t_6 =$ "have",
- $t_7 =$ "let",
- $t_8 =$ "like",
- $t_9 =$ "my",
- $t_{10} =$ "out",
- $t_{11} =$ "smells",
- $t_{12} =$ "that",
- $t_{13} =$ "the", and
- $t_{14} =$ "who".

We could further pre-process the corpus (remove stopwords, stem the words, etc.), but for the purposes of illustrating text statistics, we will leave the documents as they are.

The purest bag of word information about a term t in a document d is the raw **term frequency count**

$$tf_{t,d} = \#\text{ times } t \text{ occurs in } d,$$

but its relative usefulness is impacted by the documents' sizes.[12]

12: And size variation among documents.

$\mathrm{tf}_{t,d}$		t													
		1 been	2 breath	3 did	4 dogs	5 food	6 have	7 let	8 like	9 my	10 out	11 smells	12 that	13 the	14 who
d	1	1	0	0	1	0	1	1	0	0	1	0	0	1	1
	2	0	0	1	0	0	0	0	0	0	0	0	1	0	1
	3	0	1	0	2	1	0	0	1	1	0	1	0	0	0

The **relative term frequency** (or term proportion)

$$tf^*_{t,d} = \frac{tf_{t,d}}{M_d}$$

typically provides a more useful representation of the BoW.

$\mathrm{tf}^*_{t,d}$		t													
		1 been	2 breath	3 did	4 dogs	5 food	6 have	7 let	8 like	9 my	10 out	11 smells	12 that	13 the	14 who
d	1	1/7	0	0	1/7	0	1/7	1/7	0	0	1/7	0	0	1/7	1/7
	2	0	0	1/3	0	0	0	0	0	0	0	0	1/3	0	1/3
	3	0	1/7	0	2/7	1/7	0	0	1/7	1/7	0	1/7	0	0	0

At a simpler level, we could also look at the **document frequency** df_t, which is to say, the number of documents in which the term t occurs. To compare a term's usage **across different corpora**, however, it might be preferable to compute the **relative document frequency**

$$df^*_t = \frac{df_t}{N}.$$

This text statistic is only of limited usefulness if N is "too small".

df^*_t	t													
	1 been	2 breath	3 did	4 dogs	5 food	6 have	7 let	8 like	9 my	10 out	11 smells	12 that	13 the	14 who
	1/3	1/3	1/3	2/3	1/3	1/3	1/3	1/3	1/3	1/3	1/3	1/3	1/3	2/3

Another approach is to use the **term frequency-inverse document frequency** (tf-idf) of term t in document d:

$$\textit{tf-idf}^*_{t,d} = -tf^*_{t,d} \times \ln df^*_t.$$

This text statistic is a **heuristic**; although it has no solid theoretical backing, it is nevertheless commonly-used.[13]

13: Silge (an early backer of tf-idf) and Schnoebelen suggest an alternative in the form of weighted log odds, which can also be used with non-text data.

The rationalization for its use is that if most of the documents contain the term t, then $df^*_t \approx 1$ and the presence of that term in a document does not provide a lot of information about said document (since it shows up in most documents):

$$\textit{tf-idf}^*_{t,d} \approx -tf^*_{t,d} \times \ln 1 = 0.$$

Furthermore, if the term t does not occur often in a document d for which M_d is large, then $tf^*_{t,d} \approx 0$ and

$$\textit{tf-idf}^*_{t,d} \approx -0 \times \ln df^*_t = 0.$$

In this BoW approach, it is the terms that appear relatively often **only in a small subset** of documents (with large values) that are crucial to understanding those documents in the general context of the corpus.

$\text{tf}-\text{idf}_t^*$		t													
		1 been	**2 breath**	**3 did**	**4 dogs**	**5 food**	**6 have**	**7 let**	**8 like**	**9 my**	**10 out**	**11 smells**	**12 that**	**13 the**	14 who
d	**1**	**0.16**	0	0	0.06	0	**0.16**	**0.16**	0	0	**0.16**	0	0	**0.16**	0.06
	2	0	0	**0.37**	0	0	0	0	0	0	0	0	**0.37**	0	0.14
	3	0	**0.16**	0	0.12	**0.16**	0	0	**0.16**	**0.16**	0	**0.16**	0	0	0

27.2.5 Text Visualization

One of the major differences between **text analysis** and plain numerical **data analysis** is that even though we are able to interpret numerical results with some (often minimal) effort, we can easily interpret text analysis results with **no effort**: we have a built-in **semantics detector** – text is not just a label for data, it has **meaning** (derived from the context) that is automatically available to us.[14]

14: My son Llewellyn, upon learning how to read, complained that he **couldn't help but read** text when he saw it – the blissful ignorance of the past is forever gone.

In truth, we can train ourselves to read numerical data and results, especially with the help of **data visualizations** (see Chapter 18 and [9], for instance). Somewhat paradoxically, we can also visualize text data. Common methods include **barcharts**, **scatterplots**, **word clouds**, and **phrase nets** (see Figure 27.9 and Section 27.4).[15]

15: These can be quite handy when conducting a BoW analysis on a subject of which the analysts know very little, or if the text is in a language that they do not master yet.

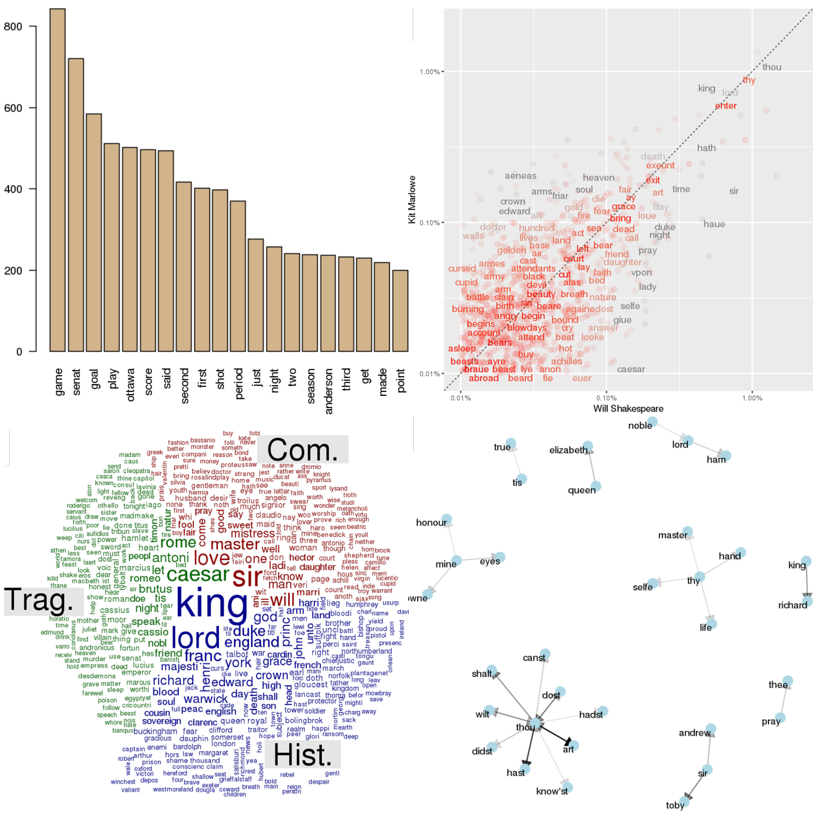


Figure 27.9: Examples of text visualizations: barchart (top left, from Section 27.4.1), scatterplot (top right, from Section 27.4.2), word cloud (from Section 27.4.3), and phrase net (from Section 27.4.2).

27.3 Text Mining Tasks

> "If the computer can successfully tell a joke as well as Henry Youngman, then that's the voice I want." [Roger Ebert, TED Talk, 2011]

We can easily leverage **machine learning** (ML) **techniques** to improve the BoW (this section) and semantics (see Chapter 32) approaches to text analysis.

We have seen that text usually enters the text analysis pipeline in **unstructured** and **unorganized** formats, from a variety of sources. Through pre-processing, text becomes **clean** (yet remains unstructured).

The BoW approach provides a framework for a **structured numerical representation** of text data, either in the form of DTM/TDM or **tidy data** (see Section 27.4 for examples of the latter). It is on these representations that ML algorithms are unleashed.

At the ML stage, it is important to remember where the data comes from and the context in which it applies. The **text mining/NLP pipeline** of Figure 27.10 applies in most (if not all) text analysis situations.

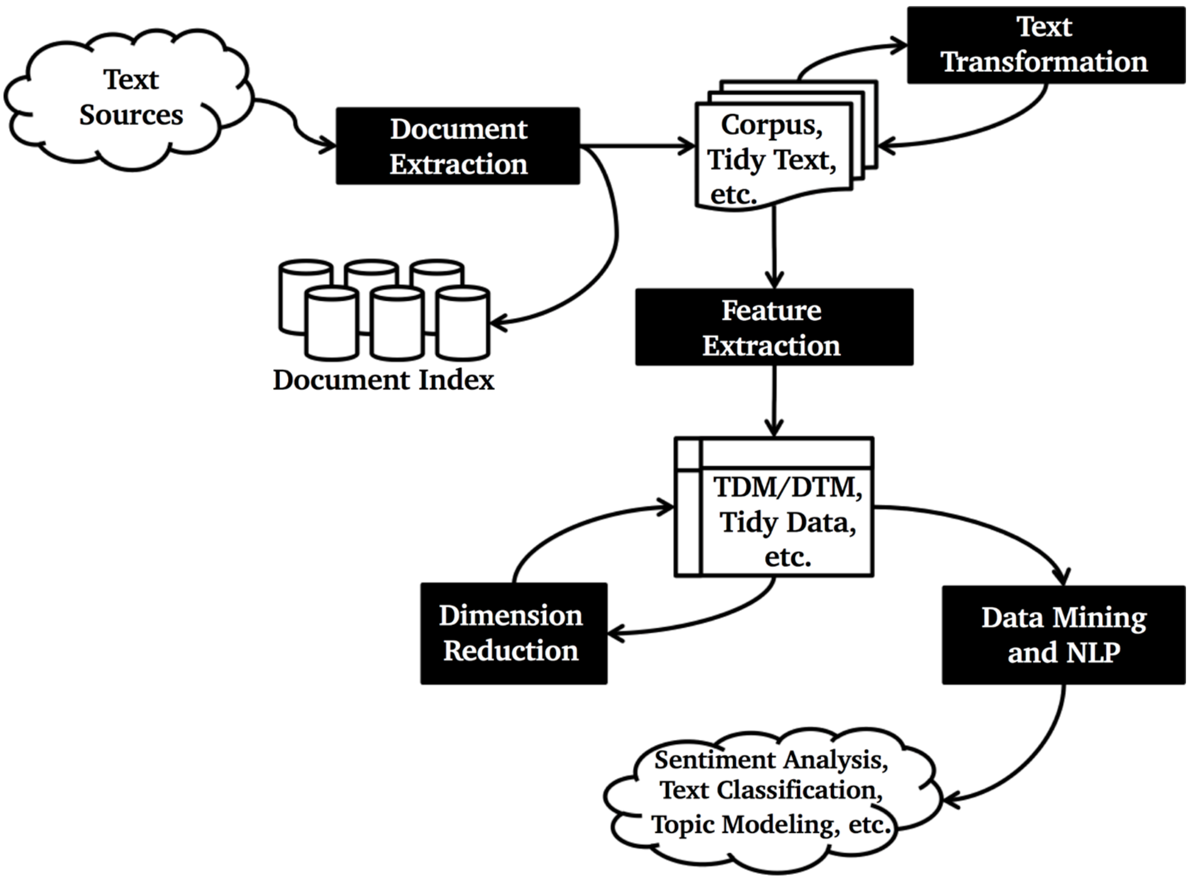


Figure 27.10: Text mining and NLP pipeline [author unknown].

We have discussed **document extraction** in Chapter 16, **feature extraction** in Chapter 23, and **machine learning** in Chapters 19–22.

In **supervised learning** (SL), there is a target/response against which to **train** and **test** the models; typical tasks include **classification** and **class probability estimation**, **regression** and **value estimation**, etc.

In **unsupervised learning** (UL), there is no target; UL could be used to **discover** potential target/response levels that could eventually be used in SL tasks with new data, say; typical tasks include **association rules** and **hypothesis discovery**, **similarity matching** and **clustering**, etc.

Other ML tasks include **profiling**, **link prediction**, and so on. In text mining, the most commonly used ML tasks are **classification** and **clustering**.

27.3.1 Classification

In **classification**, a sample set of data (the **training set**) is used to determine **rules** (or a **model**) that can be used to **divide** the data into **pre-determined groups** (also known as **classes**). The model is then **validated** by examining its performance on a **test set**.

In **text classification**, the data must first be given a **numerical representation** (DTM/TDM/tidy data) – it is on this object that **classifiers** are trained.

For instance, we may wish to answer questions such as:

- Based on the terms that appear in a text, can we determine who its author is likely to be?
- Based on the words that appear in a news story, is it propaganda?
- Is the email that was just received legitimate or malicious?
- Should a city trigger its emergency response system based on social media conversations?
- What is a tweet's main sentiment?
- etc.

Text Classification Workflow In particular, the text classification process should follow the regular classification pipeline, with the exception of the conversion from text data to numerical representation:

1. data collection;
2. data pre-processing;
3. exploration and text visualization;
4. text representation;
5. training the model, and
6. testing and evaluation.

Notes and Comments Remember that in order to train and test a classifier, the true labels have to be known for at least some of the data – this might not be achieved easily.[16]

16: It is usually quite costly to obtain these labels, especially with large text or image datasets.

Classification is also affected by the **No Free-Lunch Theorem** stating, in effect, that no single classifier is always the best option – we have to consider a number of models, on a case-by-case basis.

In situations where at least one of the class labels **occurs rarely** in the training data, the classifiers may be swamped by the frequent labels: how would an e-mail spam filter handle a term it has never encountered before? This hurdle is tricky to overcome, technically-speaking, especially as rare occurrences are often more interesting and/or important in the problem context.[17]

17: We discuss these briefly in Section 21.3.

Since it is recommended that we try out different classifiers, how can we determine if a model is preferable to another? The theory of **performance metrics** is richer for **binary classifiers** than general classifiers: ideally, a good model would have high rates of true positives and true negatives, and low rates both of false positives and false negatives.[18]

18: There are complications, as expected: see Sections 19.4.4 and 21.1.2 for more details.

Multinomial naïve Bayes Classification We will showcase a classification approach which has found quite a bit of success as an e-mail spam filter.[19] **Multinomial naïve Bayes** is a classifier for which the **feature vectors** in each class are assumed to have a **multinomial distribution**.

19: It is a variant of the algorithm presented in Section 21.4.4.

Consider a training set n of email messages – the **records**. Each record has ℓ features, the **frequencies** (or relative frequencies) of ℓ pre-selected terms in the email message body. Each record can then be represented by its signature $\mathbf{x} = (x_1, \ldots, x_\ell)$.

We assume that there are K categories in which a record could be classified.[20] Let $\{C_k \mid k = 1, \ldots, K\}$ denote the categories.

20: Perhaps the class labels are: spam, quarantined, personal, business, etc.

For any incoming e-mail message, the **classification problem** is to determine the **posterior distribution**

$$P(\mathbf{x} \in C_k \mid x_1, \ldots, x_\ell)$$

for each label k. The **predicted class** of $\mathbf{x}$ is the class C_k for which the posterior is largest.

Fix k. From **Bayes' Theorem**, we have

$$P(\mathbf{x} \in C_k \mid x_1, \ldots, x_\ell) \propto P(\mathbf{x} \in C_k) \times P(x_1, \ldots, x_\ell \mid \mathbf{x} \in C_k).$$

The **naïve assumption** is that

$$P(x_1, \ldots, x_\ell \mid \mathbf{x} \in C_k) = P(x_1 \mid \mathbf{x} \in C_k) \times \cdots P(x_\ell \mid \mathbf{x} \in C_k),$$

so that

$$P(\mathbf{x} \in C_k \mid x_1, \ldots, x_\ell) \propto P(\mathbf{x} \in C_k) \times \prod_{i=1}^{\ell} P(x_i \mid \mathbf{x} \in C_k).$$

The **multinomial assumption** is that

$$P(x_i \mid \mathbf{x} \in C_k) = p_{k,i}^{x_i},$$

where $p_{k,i} \in [0, 1]$ for each feature (word) $1 \leq i \leq \ell$.

Combining these assumptions, the posterior "probabilities" are then

$$P(\mathbf{x} \in C_k \mid x_1, \ldots, x_\ell) \propto P(\mathbf{x} \in C_k) \times \prod_{i=1}^{\ell} p_{k,i}^{x_i}.$$

The model can be further **linearized** by taking logarithms on both sides:

$$\log P(\mathbf{x} \in C_k \mid x_1, \dots, x_\ell) \propto b_k + \sum_{i=1}^{\ell} x_i \log p_{k,i}.$$

The classifier is trained by **estimating the parameters** $p_{k,i}$ on a subset of all records and by specifying the "priors" b_k. The predicted class $C(\mathbf{x})$ is the C_k for which $b_k + \sum_{i=1}^{\ell} x_i \log p_{k,i}$ is **maximized**.[21]

21: Recall that this classifier is **not calibrated** – the relative values of the posterior "probabilities" have no intrinsic value in and of themselves.

If a message encounters terms (tokens, words) that were not seen in the training data, it is impossible to predict its most likely class membership using (non-existent) past behaviour. In that case, to avoid divisions by 0, we make use of the **corrected estimate**

$$\hat{p}_{k,i} = \frac{\sum_{\mathbf{x}\in C_k} x_i + 1}{\sum_{\mathbf{x}\in C_k} (x_1 + \cdots + x_\ell) + |v|} = \frac{(\#w_i \in C_k) + 1}{W_k + |v|},$$

where $|v|$ is the size of the vocabulary, $\#w_i \in C_k$ is the **frequency** of the word w_k in the training documents belonging to class C_k, and W_k is the **count of all words** appearing in training documents in class C_k.

As an example, consider the training set and testing set below (raw and processed), describing the sentiment (class: + or −) associated with 6 reviews for a specific phone.

Training set

cl	ID	text
+	i1	I love this phone
+	i2	amazing sound quality
+	i3	Love this great phone
-	i4	i hate it
-	i5	bad quality
-	i6	so bad Hate it

Testing set

cl	ID	text
??	i7	hate hate HATE the phone quality

Processed training set

cl	ID	text
+	i1	love phone
+	i2	amazing sound quality
+	i3	love great phone
-	i4	hate
-	i5	bad quality
-	i6	bad hate

Processed testing set

cl	ID	text
??	i7	hate hate hate phone quality

For the priors of each class, we use the proportions of positive and negative reviews in the training data. In the processed data, there are 8 distinct vocabulary terms, and there are 8 (resp. 5) terms in the positive (resp. negative) reviews. The corrected estimates for the vocabulary word "amazing" are computed below.

$$P(+) = \frac{3}{6} = 0.5; \quad P(-) = \frac{3}{6} = 0.5 \implies b_+ = b_- = \ln 0.5;$$

$$|v| = 8,\ W_+ = 8, W_- = 5;$$

$$\hat{p}_{+,\text{amazing}} = \frac{(\#\text{amazing} \in +) + 1}{W_+ + 8} = \frac{1+1}{8+8} = \frac{1}{8}$$

$$\hat{p}_{-,\text{amazing}} = \frac{(\#\text{amazing} \in -) + 1}{W_- + 8} = \frac{0+1}{5+8} = \frac{1}{13}$$

The corrected estimates for each vocabulary word in the training documents are shown below, as is the signature vector **x** for the lone test record.

Training set

$\hat{p}$	amazing	bad	great	hate	love	phone	quality	sound
+	0.1250	0.6025	0.1250	0.0625	0.1875	0.1875	0.1250	0.1250
−	0.0769	0.2308	0.0769	0.2308	0.0769	0.0769	0.1538	0.0769

Testing set

	amazing	bad	great	hate	love	phone	quality	sound
i7	0	0	0	3	0	1	1	0

Simple computations show that

$$P(+ \mid \mathbf{x})\text{“ =”}\ 2.9 \times 10^{-6} < P(- \mid \mathbf{x}) = 9.7 \times 10^{-6},$$

from which we conclude that the test review is **negative**.[22]

22: Hopefully, that is not much of a surprise in light of the original test review.

Of course, any classification algorithm may be used. Common methods also include support vector machines and artificial neural networks (see Chapter 21), not only multinomial naïve Bayes. We will have more to say on the topic in Section 27.4.4.

27.3.2 Clustering

The aim of **clustering** is to **divide** the data into **latent groups** (also known as **clusters**). Within a cluster, data points are seen as **similar** to one another; between clusters, they are seen as **dissimilar**. As befits the unsupervised learning nature of the task, the cluster labels are **not pre-determined**.

For instance, we may wish to:

- divide existing social media users into subgroups based on the shared characteristics of their posts;
- create (new) taxonomies on the fly, as new items are added to a group of items to ease product navigation;
- cluster terms within a corpus of document (topic modeling);
- cluster documents within a corpus based on their use of terms;
- identify keywords in a document;
- etc.

Text Clustering Workflow The steps are quite similar to those of text classification:

1. data collection;
2. data pre-processing;
3. exploration and text visualization;
4. text representation;
5. run multiple clustering algorithms with parameter variations, and
6. compare and validate the results.

Notes and Comments Conceptually, clustering is relatively intuitive for people: we recognize clusters when we see them.

But there are issues, chiefly:

- there is no agreed-upon definition of what a cluster is;
- there is no "magic" recipe to determine which similarity measure to use;
- the number of cluster is not usually specified, and
- due to the non-deterministic nature of many clustering algorithms, the results are often unreplicable.

And what does it mean to cluster **text data**? With a DTM text representation, we can **cluster the documents with respect to the terms**, which is to say that we look for documents that have **similar term signatures**. With a TDM text representation, we can **cluster the terms with respect to the documents**, which is to say that we look for terms that have **similar document signatures**.

We will have more to say on the topic in Section 27.4.5.

27.3.3 Sentiment Analysis

Most of us have a good native understanding of the emotional intent of words, which leads us to infer **surprise**, **disgust**, **joy**, **pain**, and so on from a text segment.[23] When applied by machines to a block of text, the (somewhat subjective) process of identifying emotions is known as **sentiment analysis** (or **opinion mining**).

23: Although **sarcasm** or **lies** are not always obvious without other contextual cues.

Typical sentiment questions could include:

- Is this movie review positive or negative?
- Is this customer email a complaint?
- How have newspapers' attitudes about the PM changed since the election?
- etc.

Most humans would typically be able to answer these questions when presented with the appropriate text documentation, but there is no guarantee that each individual's reading of the situation would be the same. For text processing machines (even modern LLMs), questions of this nature may be quite difficult to answer.

Challenges Data analysis is not easy, in general, but sentiment analysis is even more complicated, as:

- the **topic** may change halfway through the text;
- the author may be using **rhetorical devices** ;
- we do not always agree on the **emotional content** of text (due to cultural context, or lack of familiarity with the language, or different political affiliations, etc.);
- words may have **different meanings**/**emotional values** depending on the context (stolen goods may be **hot**, in which case we would want nothing to do with them; a new song may be **hot**, in which case we would probably stream it right away);

- **qualifiers** can drastically change a term's emotional value (note the emotional difference between **he was really happy today** vs. **he was not really happy today**, which may only differ in one term, but has a completely different meaning), etc.

Sentiment analysis is a **supervised learning** problem, requiring dictionaries of **emotional content** to have been compiled ahead of time or **sentiment labels** to have been assigned to a training text dataset.[24] The following reviews were taken from Amazon: there is a 5–star, a 3–star, and a 1–star review – but which is which?

24: Related tasks include:

- discarding subjective information (**information extraction**);
- recognizing opinion-oriented questions (**question answering**);
- accounting for multiple viewpoints (**summarization**);
- identifying suitability of videos for kids;
- identifying bias in news sources;
- identifying appropriate content for ad placement, etc.

> **Review 1:** "Love the jeans, price, fit, but even more, love the suppliers. Simple concerns were not only answered immediately, they went beyond any expectations I had! Will definitely be buying through this supplier, highly recommended!"
>
> **Review 2:** "DON'T BUY. Great series aside, this special addition is pathetic. They're basically mass-market paperbacks: small and uncomfortable to hold. The regular paperback versions are far superior for basically the same price."
>
> **Review 3:** "Beginning the second use, the bowl keeps falling out 30 seconds after the mixing starts. A bit disappointed."

This is how the AIPlaybook at a16z.com/ai saw the situation in 2021:

"Love the jeans, price, fit, but even more, love the suppliers. Simple concerns were not only answered immediately, they went beyond any expectations I had! Will definitely be buying through this supplier, highly recommended!"

Scores

API Name	Result	Total request time	API Time
+ Sentiment.JS (node.js library)	very positive (90)	299	0
+ Sentimental (node.js library)	very positive (90)	289	0
+ IBM Alchemy Language API	-1	341	43
+ IBM Watson Developer Cloud	positive (72)	1472	1128
+ Google Cloud APIs	-1	651	351
+ Microsoft Azure Cognitive Services	very positive (93)	789	482

"DON'T BUY. Great series aside, this special addition is pathetic. They're basically mass-market paperbacks: small and uncomfortable to hold. The regular paperback versions are far superior for basically the same price."

Scores

API Name	Result	Total request time	API Time
+ Sentiment.JS (node.js library)	neutral (10)	163	0
+ Sentimental (node.js library)	neutral (10)	157	1
+ IBM Alchemy Language API	-1	240	79
+ IBM Watson Developer Cloud	neutral (-3)	1164	922
+ Google Cloud APIs	-1	436	218
+ Microsoft Azure Cognitive Services	negative (-69)	788	609

"Beginning the second use, the bowl keeps falling out 30 seconds after the mixing starts. A bit disappointed."

Scores

API Name	Result	Total request time	API Time
+ Sentiment.JS (node.js library)	negative (-30)	65	0
+ Sentimental (node.js library)	negative (-30)	66	0
+ IBM Alchemy Language API	-1	379	211
+ IBM Watson Developer Cloud	neutral (-6)	1300	1026
+ Google Cloud APIs	-1	408	281
+ Microsoft Azure Cognitive Services	very negative (-78)	684	486

There is a fair amount of variation, both within and across reviews. What would a modern tool like GPT make of these?

Patrick: Good evening, GPT. I am going to post three product reviews. One is a 5-star review, one is a 3-star review, and one is a 1-star review. I would like you to tell me which review matches with which score.

GPT: Absolutely, that sounds like an interesting exercise! Go ahead and post the three reviews, and I'll do my best to match each review with the correct star rating.

Review 1: "Love the jeans, [...]"

Review 2: "DON'T BUY. [...]"

Review 3: "Beginning [...]"

Based on the content and tone of each review, here's how I would match them with their respective star ratings.

Review 1: This review is overwhelmingly positive, emphasizing satisfaction with both the product (jeans) and the supplier's customer service. The reviewer mentions loving the product and having their expectations exceeded, and they even recommend the supplier. This review aligns with a **5-star rating**.

Review 3: This review expresses disappointment, but it's not entirely negative. The reviewer seems to have had a specific issue with the product (a bowl falling out during use). However, they don't express complete dissatisfaction or advise against purchasing the product. This seems more in line with a **3-star rating** – not completely dissatisfied, but not fully satisfied either.

Review 2: This review is clearly negative. The reviewer strongly advises against purchasing the product, using phrases like "DON'T BUY" and describing the product as "pathetic." They also compare it unfavorably to another version, suggesting the other is a better choice. This tone and content are indicative of a **1-star rating**.

Let me know if you agree with these assessments!

Thank you, I think that this is indeed what intelligent agents would conclude.

For the sake of completeness, I should mention that Review 1 was the 5-star review; Review 2, the 3-star review, and Review 3, the 1-star review. There is nothing you could have done differently to obtain these results... humans are not always logical when it comes to rating things.

Thank you for sharing the actual ratings, and you're absolutely right! Human reviews can be quite subjective and not always follow a logical pattern. People's rating criteria can vary greatly, and what might be a minor inconvenience to one person could be a major drawback to another. It's an interesting reminder of the diversity in human experiences and expectations.

Perhaps GPT's ratings were skewed because it was told that one of the reviews was a 5-star review, and so on. What if we prompt it differently, in a new conversation? As it turns out, GPT comes to the same conclusion, as we can see below.

Patrick: Good evening, GPT. I am going to provide you with 3 product reviews. I would like for you to rate them, on a scale from 1 (poor) to 5 (great).

GPT: Sure, I'd be happy to help with that. Please go ahead and share the three product reviews, and I'll rate each of them on a scale from 1 to 5 based on their content.

Review 1: "Love the jeans, [...]"

Review 2: "DON'T BUY. [...]"

Review 3: "Beginning [...]"

Based on the content of each review, here are the ratings I would assign.

Review 1: This review is very positive. The customer expresses satisfaction with multiple aspects of the product (jeans) and exceptional satisfaction with the supplier's customer service. The enthusiasm and recommendation suggest a high level of satisfaction. I would rate this as a **5 (great)**.

Review 2: This review is highly negative. The customer advises against purchasing the product and describes it as "pathetic," indicating significant dissatisfaction. The comparison with a superior version of the product emphasizes their disappointment. I would rate this as a **1 (poor)**.

Review 3: This review expresses a specific problem and disappointment, but it doesn't completely condemn the product. It seems like a moderate level of dissatisfaction, possibly due to a single but significant issue. I would rate this as a **3 (average)**, assuming the problem is not widespread or indicative of overall poor quality.

These ratings are based on the tone and content of the reviews, reflecting the customers' levels of satisfaction or dissatisfaction with their purchases.

Thank you!

Term-by-Term and Document-by-Document Sentiment Analysis In this chapter, we differentiate 2 types of sentiment analyses:

- **term-by-term** (TBT) sentiment analysis looks at the emotional content of **individual tokens** (terms in the text) and deduces an **emotional score** for passages containing them;
- **document-by-document** (dbd) sentiment analysis looks at **scored passages** and tries to find tokens which **carry the emotional load** to predict how a new passage would score on some emotional spectrum.[25]

25: Note that these terms are not widely-used in the analyst community.

TBT is not a complicated technical task: it only requires the ability to match a **lexicon score** to each term, and to add the scores; DBD is basically a **classification problem**.

Sentiment Lexicons TBT sentiment analysis relies heavily on **lexicons**, which is to say, lists of terms which have been **ranked** (by experts) on some **emotional scale**. Commonly-used lexicons include:

- **AFINN:** words are ranked on a scale from -5 (negative) to 5 (positive);
- **BING:** words are given a binary score (negative/positive);
- **NRC:** words are assigned category(ies) of sentiments;
- **LOUGHRAN:** words are placed in categorical bins.

"abandon"

- AFINN: -2
- BING: NA
- NRC: fear, negative, sadness
- LOUGHRAN: negative

"bad"

- AFINN: -3
- BING: negative
- NRC: anger, disgust, fear, etc.
- LOUGHRAN: negative

"not"

- AFINN: NA
- BING: NA
- NRC: NA
- LOUGHRAN: NA

"egregious"

- AFINN: ?
- BING: ?
- NRC: ?
- LOUGHRAN: ?

What's the best lexicon to use? As always, **context matters**. Is there any reason to expect the various lexicons to give the same scores? Each of these lexicons contains a majority of negative terms (keeping in mind that most words in the English language are **neutral**), so there could at least be some correlation.

Once a lexicon has been selected, TBT is simply a matter of **chunking** the text and computing sentiment scores on each block (every 100 words, every 100 lines, every chapter, etc.). Does the sectioning approach matter? Again, context matters.

We provide examples of TBT and DBD in Sections 27.4.6 and 27.4.8. For a more in-depth discussion on text mining and natural language processing, interested readers are advised to also consult Chapter 32 and [40, 39, 24, 3, 1, 25, 8].

27.4 Examples

Various concepts of text analysis and text mining are illustrated using `R` and `Python` in the following examples, some of which are inspired by the excellent [39] and [26].

27.4.1 NHL Game Recaps

In this example, we introduce the basic notions of text mining using the `tm` (text mining) and `qdap` (quantitative discourse analysis package) libraries in `R`.

The main dataset we work with is the text content of Associated Press game recaps involving the Ottawa Senators during the 2016-2017 NHL season.[26]

26: All of this section's datasets are available at github.com/potrbollvy/Data-Training.

Initializing the Environment

```
install.packages("tm")
install.packages("qdap")
```

Preliminaries

We start with a simple example to showcase the possibilities.

```
new_text <- "The Ottawa Senators have the Atlantic Division lead in their sights.  Mark
Stone had a goal and four assists, Derick Brassard scored twice in the third period and
the Senators recovered after blowing a two-goal lead to beat the Toronto Maple Leafs 6-3
on Saturday night.  The Senators pulled within two points of Montreal for first place in
the Atlantic Division with three games in hand.  We like where we're at. We're in a good
spot, Stone said. But there's a little bit more that we want. Obviously, there's teams
coming and we want to try and create separation, so the only way to do that is keep
winning hockey games.  Ottawa led 2-0 after one period but trailed 3-2 in the third
before getting a tying goal from Mike Hoffman and a power-play goal from Brassard. Stone
and Brassard added empty-netters, and Chris Wideman and Ryan Dzingel also scored for the
Senators.  Ottawa has won four of five overall and three of four against the Leafs this
season. Craig Anderson stopped 34 shots.  Morgan Rielly, Nazem Kadri and William Nylander
scored and Auston Matthews had two assists for the Maple Leafs. Frederik Andersen allowed
four goals on 40 shots.  Toronto has lost eight of 11 and entered the night with a tenuous
grip on the final wild-card spot in the Eastern Conference.  The reality is we're all big
boys, we can read the standings. You've got to win hockey games, Babcock said. After
Nylander made it 3-2 with a power-play goal 2:04 into the third, Hoffman tied it by rifling
a shot from the right faceoff circle off the post and in. On a power play 54 seconds later,
Andersen stopped Erik Karlsson's point shot, but Brassard jumped on the rebound and put
it in for a 4-3 lead.  Wideman started the scoring in the first, firing a point shot
through traffic moments after Stone beat Nikita Zaitsev for a puck behind the Leafs goal.
Dzingel added to the lead when he deflected Marc Methot's point shot 20 seconds later.
Andersen stopped three shots during a lengthy 5-on-3 during the second period, and the
Leafs got on the board about three minutes later. Rielly scored with 5:22 left in the
second by chasing down a wide shot from Matthews, carrying it to the point and shooting
through a crowd in front.  About three minutes later, Zaitsev fired a shot from the right
point that  sneaked through Anderson's pads and slid behind the net. Kadri chased it down
and banked  it off Dzingel's helmet and in for his 24th goal of the season. Dzingel had
fallen in the crease trying to prevent Kadri from stuffing the rebound in.  Our game plan
didn't change for the third period, and that's just the maturity we're gaining over time,
Senators coach Guy Boucher said. Our leaders have been doing a great job, but collectively,
the team has grown dramatically in terms of having poise, executing under pressure. Game
notes: Mitch Marner sat out for Toronto with an upper-body injury. Marner leads Toronto
with 48 points and is also expected to sit Sunday night against Carolina."
```

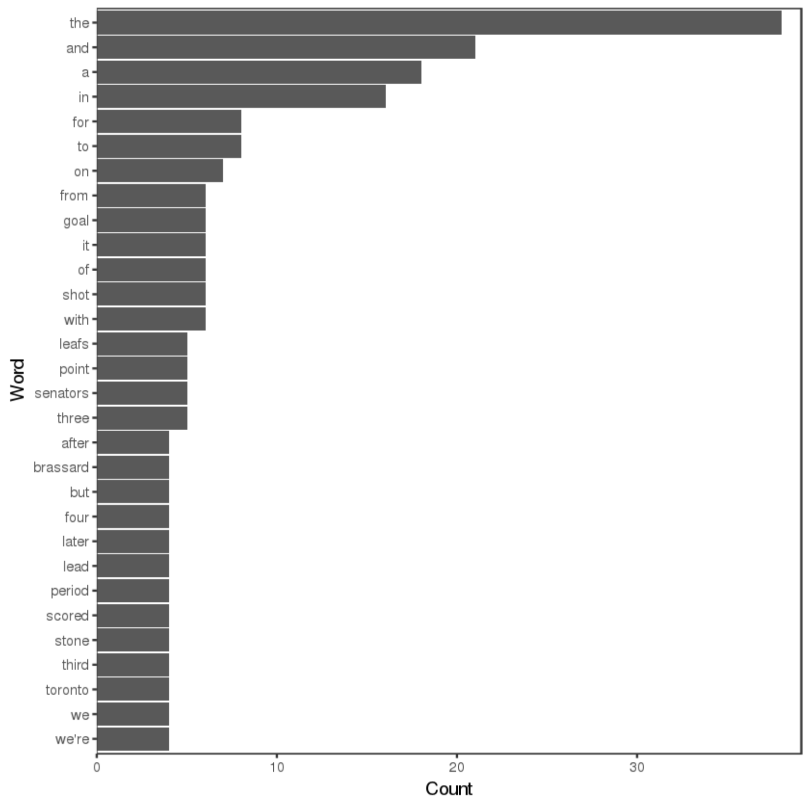
We find the 20 most frequent terms using `qdap`'s `term_count()`.

```
(term_count <- qdap::freq_terms(new_text,20))
```

```
   WORD      FREQ      WORD      FREQ      WORD      FREQ      WORD      FREQ
1  the         38   9  goal         6   17 three        5   25 scored       4
2  and         21   10 it           6   18 after        4   26 stone        4
3  a           18   11 of           6   19 brassard     4   27 third        4
4  in          16   12 shot         6   20 but          4   28 toronto      4
5  for          8   13 with         6   21 four         4   29 we           4
6  to           8   14 leafs        5   22 later        4   30 we're        4
7  on           7   15 point        5   23 lead         4
8  from         6   16 senators     5   24 period       4
```

There are more than 20 entries because of ties (at 4 occurrences apiece). This information can also be displayed as a chart.

```
plot(term_count)
```



Sens Recaps Data

We now import the data for all games during the season.

```
recaps <- read.csv(file="Recap_data.csv", header=TRUE, sep=",", stringsAsFactors=FALSE)
nrow(recaps)
str(recaps)
```

```
[1] 101

'data.frame':  101 obs. of  34 variables:
 $ GP          : int  1 2 3 4 5 6 7 8 9 10 ...
 $ X0_Type     : chr  "1_Regular" "1_Regular" "1_Regular" "1_Regular" ...
 $ Date        : chr  "10/12/2016" "10/15/2016" "10/17/2016" "10/18/2016" ...
 $ Time        : chr  "7:00 PM" "7:00 PM" "7:30 PM" "7:30 PM" ...
 $ X           : chr  "" "" "A" "" ...
 $ Opponent    : chr  "Toronto Maple Leafs" "Montreal Canadiens" "Detroit Red Wings" ...
 $ GF          : int  5 4 1 7 1 3 2 2 2 1 ...
 $ GA          : int  4 3 5 4 4 0 5 0 1 0 ...
 $ Result      : chr  "W" "W" "L" "W" ...
 $ OT_SO       : chr  "OT" "SO" "" "" ...
```

```
 $ W_Record    : int  1 2 2 3 3 4 4 5 6 7 ...
 $ L_Record    : int  0 0 1 1 2 2 3 3 3 3 ...
 $ OL_Record   : int  0 0 0 0 0 0 0 0 0 0 ...
 $ Streak      : chr  "W 1" "W 2" "L 1" "W 1" ...
 $ OTT_S       : int  30 38 32 42 28 28 33 22 32 24 ...
 $ OTT_PIM     : int  13 10 22 14 8 2 4 11 13 20 ...
 $ OTT_PPG     : int  0 0 0 1 0 0 2 0 0 0 ...
 $ OTT_PPO     : int  2 4 3 2 3 1 4 2 2 4 ...
 $ OTT_SHO     : int  0 0 1 1 0 0 0 0 0 0 ...
 $ OPP_S       : int  38 24 25 35 35 22 19 37 33 27 ...
 $ OPP_PIM     : int  11 10 20 6 6 2 8 9 11 22 ...
 $ OPP_PPG     : int  0 1 2 1 2 0 0 0 0 0 ...
 $ OPP_PPO     : int  4 4 4 5 4 1 2 4 3 2 ...
 $ OPP_SHG     : int  0 0 0 0 0 0 0 0 0 0 ...
 $ ATT         : chr  "17,618" "18,195" "20,027" "11,061" ...
 $ LOG         : chr  "2:36" "2:44" "2:33" "2:43" ...
 $ AP_Headline : chr  "Maple Leafs\xcd Matthews has modern record" ...
 $ AP_Recap    : chr  "Auston Matthews needed 40 minutes to get into"| __truncated__ ...
 $ SSS_Author  : chr  "Ross A" "Ary M" "Michaela Schreiter" "Ary M" ...
 $ SSS_Headline: chr  "Auston Matthews Loses 5-4 to Sens in OT" ...
 $ SSS_Recap   : chr  "The NHL.com headline for the game was \xd2Auston Matthews scores"| __truncated__
 $ OPP_Blog    : chr  "Pension Plan Puppet" "Eyes on the Prize" ...
 $ OPP_Title   : chr  "Sens 5, Auston Matthews 4 (OT)" ...
 $ OPP_Recap   : chr  "The first period started exactly the way that Leafs' fans wanted"| __truncated__
```

It is child's play to isolate the text from individual game recaps.

```
AP.recaps <- recaps$AP_Recap
head(AP.recaps,2)
```

```
'Auston Matthews needed 40 minutes to get into the NHL record book. In the highest-scoring debut
  in modern NHL history, Matthews scored four goals for the Toronto Maple Leafs, but Kyle Turris
  scored 37 seconds into overtime to give the Ottawa Senators to a 5-4 victory Wednesday night.
  Matthews got his fourth with 3 seconds left in the second  period, bringing his mother to
  tears in the stands. He called it a \xf1surreal\xee moment, adding that \xf1I couldn\xcdt
  believe that was happening out there.\xee [...] UP NEXT Maple Leafs: Host Boston on Saturday
  night for their home opener. Senators: Host Montreal on Saturday night.'
'Guy Boucher trusted his instincts when selecting skaters for the shootout and it paid off for the
  Ottawa Senators. The Senators\' head coach opted to go with defenseman Erik Karlsson and the
  captain scored the game winner to give Ottawa a 4-3 victory over the Montreal Canadiens on Saturday
  night to open the season with back-to-back wins. \'\'Sometimes it\'s just small things  and you
  follow a gut feeling,\'\' Boucher said. [...]'
```

There are odd characters in the game recaps (xf1, xee, etc.), which highlight some issue with text encoding and formatting. We revisit the last few steps with a slightly different data file.

```
recaps <- read.csv(file="Recap_data_first_pass.csv", header=TRUE,
                   sep=",", stringsAsFactors=FALSE)
AP.recaps <- recaps$AP.recaps
head(AP.recaps,2)
```

```
'Auston Matthews needed 40 minutes to get into the NHL record book. In the highest-scoring debut
  in modern NHL history, Matthews scored four goals for the Toronto Maple Leafs, but Kyle Turris
  scored 37 seconds into overtime to give the Ottawa Senators to a 5-4 victory Wednesday night.
  Matthews got his fourth with 3 seconds left in the second period, bringing his mother to tears
  in the stands. He called it a "surreal" moment, adding that "I couldn\'t believe that was
  happening out there." [...]'
'Guy Boucher trusted his instincts when selecting skaters for the shootout and it paid off for the
  Ottawa Senators. The Senators\' head coach opted to go with defenseman Erik Karlsson and the captain
  scored the game winner to give Ottawa a 4-3 victory over the Montreal Canadiens on Saturday night
  to open the season with back-to-back wins. "Sometimes it\'s just small things and you follow a gut
  feeling," Boucher said. [...]'
```

The results are easier to read, for sure, but for reasons that are too technical to get into here, the encoding of `Recap_data_first_pass.csv` creates issues with `tm` and `qdap` down the road, but the issues disappear when we use a different encoding (UTF-8).

```
recaps <- read.csv(file="Recap_data_first_pass_utf8.csv", header=TRUE, sep=",",
                   stringsAsFactors=FALSE)
AP.recaps <- recaps$AP.recaps
```

VCorpus from Vector with `tm`

The `tm` package makes it easy to work with vector sources and volatile corpora. For instance, we can make a **vector source** as follows.

```
AP.recaps.source <- tm::VectorSource(AP.recaps)
```

This vector source can be converted to a **volatile corpus**.

```
AP.recaps.corpus <- tm::VCorpus(AP.recaps.source)
```

At a fundamental level, the volatile corpus contains the following information.

```
AP.recaps.corpus
```

```
<<VCorpus>>
Metadata:  corpus specific: 0, document level (indexed): 0
Content:  documents: 101
```

This is not entirely useful, to be honest, although we do recover the 101 games played by the Senators during the season. Let's say we wanted more details on the 15th game.

```
AP.recaps.corpus[[15]]
```

```
<<PlainTextDocument>>
Metadata:  7
Content:  chars: 2871
```

There are two entries in the list; the first is the game recap text.

```
AP.recaps.corpus[[15]][1]
```

```
$content = 'For a team playing its third game in four nights, the Minnesota Wild looked plenty
  fresh on Sunday night -- even in overtime. Matt Dumba scored late in the extra session and
  Darcy Kuemper stopped 35 shots, helping Minnesota beat the Ottawa Senators 2-1. The Wild [...]'
```

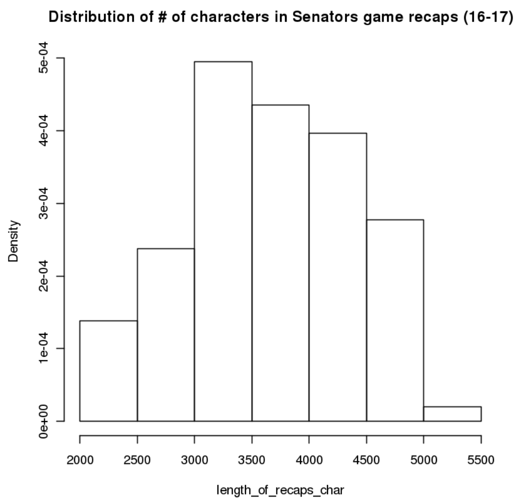
The entry's **metadata** can be queried as follows.

```
AP.recaps.corpus[[15]][2]
```

```
$meta
  author       : character(0)
  datetimestamp: 2019-09-15 13:29:16
  description  : character(0)
  heading      : character(0)
  id           : 15
  language     : en
  origin       : character(0)
```

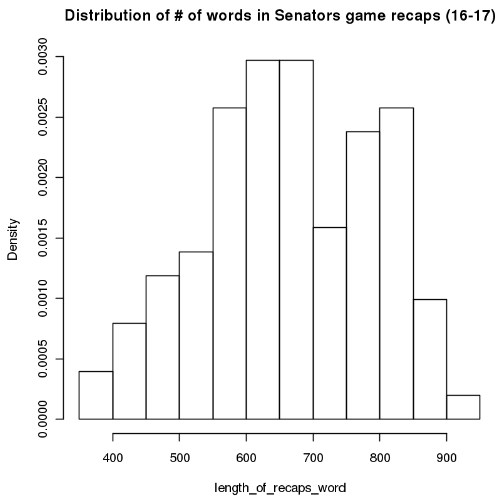
We can also take a look at some basic statistics regarding the **number of characters** and the **number of words** in the game recaps.

```
length_of_recaps_char <- vector(mode="numeric", length=nrow(recaps))
for(j in 1:nrow(recaps)){
    length_of_recaps_char[j]=nchar(AP.recaps.corpus[[j]][1])
}

hist(length_of_recaps_char, freq=F,
     main="Distribution of # of characters in Senators game recaps (16-17)")
summary(length_of_recaps_char)
```




```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   2027    3223    3689    3683    4227    5087
```

```
length_of_recaps_word <- vector(mode="numeric", length=nrow(recaps))
for(j in 1:nrow(recaps)){
    length_of_recaps_word[j]=length(strsplit(gsub(' {2,}',' ',
                                            AP.recaps.corpus[[j]][1]),' ')[[1]])
}

hist(length_of_recaps_word, freq=F,
     main="Distribution of # of words in Senators game recaps (16-17)")
summary(length_of_recaps_word)
```




```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    375     565     665     664     774     921
```

Pre-Processing a Document with `tm`

To get the most of this corpus, we must first transform it into a **bag-of-words** (BoW). We first show how to implement the various text processing functionalities on the text string used in Section 27.2.3.

```
(text <- "<i>He</i> went to bed at       2 A.M. It\'s way too late!  He was only 20%
    asleep at first, but sleep eventually came.")
```

```
[1] "<i>He</i> went to bed at       2 A.M. It\'s way too late!  He was only 20% asleep at first,
but sleep eventually came."
```

All characters can be converted to lower case with the `tolower()` function.

```
tolower(text)
```

```
[1] "<i>he</i> went to bed at       2 a.m. it\'s way too late!  he was only 20% asleep at first,
but sleep eventually came."
```

The output of the following three `tm` functions should be clear from their name.

```
tm::removePunctuation(text)
```

```
[1] "iHei went to bed at      2 AM Its way too late  He was only 20 asleep at first
but sleep eventually came"
```

```
tm::removeNumbers(text)
```

```
[1] "<i>He</i> went to bed at        A.M. It\'s way too late!  He was only % asleep at first,
but sleep eventually came."
```

```
tm::stripWhitespace(text)
```

```
[1] "<i>He</i> went to bed at 2 A.M. It\'s way too late! He was only 20% asleep at first,
but sleep eventually came."
```

Pre-Processing a Document with `qdap`

Some of the more sophisticated processes are implemente in `qdap`. The functionality should be clear from the function's name (as well as its output).[27]

27: Note that all of these also strip unnecessary spaces in the string.

```
# Remove text within brackets
qdap::bracketX(text)

# Replace numbers with words
qdap::replace_number(text)

# Replace abbreviations
qdap::replace_abbreviation(text)

# Replace contractions
qdap::replace_contraction(text)

# Replace symbols with words
qdap::replace_symbol(text)
```

```
[1] "He went to bed at 2 A.M. It\'s way too late! He was only 20% asleep at first,
but sleep eventually came."

[1] "<i>He</i> went to bed at two A.M. It\'s way too late! He was only twenty% asleep at first,
but sleep eventually came."

[1] "<i>He</i> went to bed at 2 AM It\'s way too late! He was only 20% asleep at first,
but sleep eventually came."

[1] "<i>He</i> went to bed at 2 A.M. it is way too late! He was only 20% asleep at first,
but sleep eventually came."

[1] "<i>He</i> went to bed at 2 A.M. It\'s way too late! He was only 20 percent asleep at first,
but sleep eventually came."
```

Stopwords

Stopwords are those words that do not carry semantic content, partly because they occur too frequently to really be "seen" by speakers/readers, such as the "said" tag in a novel. They are often removed from the text prior to BoW analysis.

```
# List standard English stop words
tm::stopwords("en")

# Print text without standard stop words
tm::removeWords(text,tm::stopwords("en"))
```

```
[1] "<>He</> went  bed        2 A.M. It's way  late!  He   20% asleep  first,
sleep eventually came."
```

Of course, stopwords may vary from one context to the next,[28] and it is possible to add or subtract words fromt he stopwords list.

28: Or from one language to the next.

```
# Add "sleep" and "asleep" to the list: new_stops
new_stops <- c("sleep","asleep",tm::stopwords("en"))

# Remove stop words from text
tm::removeWords(text,new_stops)
```

```
[1] "<>He</> went  bed        2 A.M. It's way  late!  He   20%   first,   eventually came."
```

Putting it All Together

We can combine some pre-processing steps into one call (there are, of course multiple ways to do this) – note that the order of implementation matters: a different order may very well lead to a different outcome.

```
tolower(
  tm::stripWhitespace(
    tm::removeWords(
      tm::removePunctuation(
        qdap::replace_symbol(
          qdap::replace_contraction(
            qdap::replace_abbreviation(
              qdap::bracketX(text)
            )
          )
        )
      )
      ,tm::stopwords("en"))
  )
)
```

```
[1] "he went bed 2 am way late he 20 percent asleep first sleep eventually came"
```

Word Stemming and Stem Completion

Stemming is also implemented in `tm`.

```
# Create sleep
(sleep <- c("sleepful","sleeps","sleeping"))

# Perform word stemming: stem_doc
(stem_doc <- tm::stemDocument(sleep))
```

```
[1] "sleepful" "sleeps"   "sleeping"

[1] "sleep" "sleep" "sleep"
```

```
# Create the completion dictionary: sleep_dict
sleep_dict <- c("sleep")

# Perform stem completion: complete_text
(complete_text <- tm::stemCompletion(stem_doc,sleep_dict))
```

```
  sleep   sleep   sleep
"sleep" "sleep" "sleep"
```

For illustrative purposes, let us take a quick look at a string with more substance.

```
text_data <- "In sleepful nights, Katia sleeps to achieve sleeping."
comp_dict <- c("In","sleep","nights","Katia","to","achieve")

# Remove punctuation
rm_punc <- tm::removePunctuation(text_data)

# Create character vector
n_char_vec <- unlist(strsplit(rm_punc, split = ' '))

# Perform word stemming: stem_doc
(stem_doc <- tm::stemDocument(n_char_vec))

# Re-complete stemmed document: complete_doc
(complete_doc <- tm::stemCompletion(stem_doc,comp_dict))
```

```
[1] "In"     "sleep"  "night"  "Katia"  "sleep"  "to"     "achiev" "sleep"

 In     sleep     night     Katia     sleep        to    achiev     sleep
"In"   "sleep"  "nights"   "Katia"   "sleep"      "to" "achieve"   "sleep"
```

Notice the slight difference between the stemmed string and the completed string.

Pre-Processing a Corpus

In practice, we never work with a single string or with a single document; how would we pre-process an entire corpus of text documents? The function `tm_map` maps the processing step to all documents in the corpus; if the processing function is not implemented in the package `tm`, it must be wrapped by the `content_transformer`.

As an example, consider the following customized pre-processing cleaner, which mixes base, `tm` and `qdap` functions.

```
clean_corpus <- function(corpus){
  corpus <- tm::tm_map(corpus, tm::content_transformer(qdap::replace_abbreviation))
  corpus <- tm::tm_map(corpus, tm::removePunctuation)
  corpus <- tm::tm_map(corpus, tm::removeNumbers)
  corpus <- tm::tm_map(corpus, tm::stemDocument)
  corpus <- tm::tm_map(corpus, tm::content_transformer(tolower))
  corpus <- tm::tm_map(corpus, tm::stripWhitespace)
  corpus <- tm::tm_map(corpus, tm::removeWords, c(tm::stopwords("en")))
  return(corpus)
}
```

We apply it to the Sens game recaps corpus `AP.recaps.corpus`.

```
clean_corp.AP.recaps <- clean_corpus(AP.recaps.corpus)
```

As an example, let's print the cleaned up recap for game 15 (compare with the raw text obtained previously).

```
clean_corp.AP.recaps[[15]][1]
```

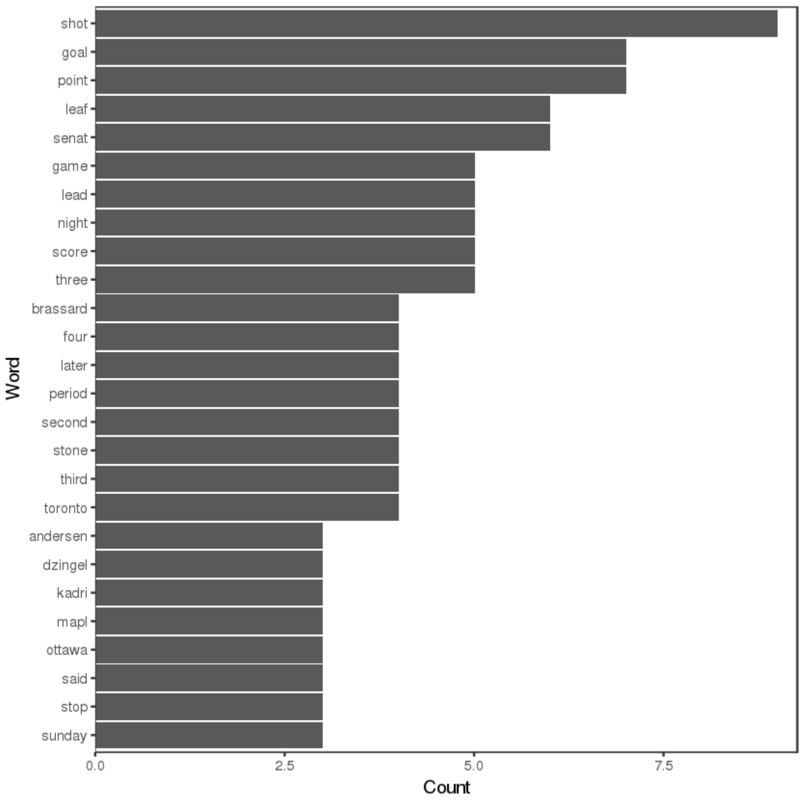
```
$content = 'team play  third game  four night  minnesota wild look plenti fresh  sunday night
  even  overtim matt dumba score late   extra session  darci kuemper stop shot help minnesota
  beat  ottawa senat  wild  come   loss  philadelphia  saturday  beat pittsburgh  thursday    end
  world  play backtoback   thought  held     good job wild coach bruce boudreau said ryan suter
  score  shorthand goal   first period  kuemper help  wild kill  three earli power play    team   [...]'
```

It is obviously not a proper English document, but the "meaning" can be gleamed fairly easily.[29] One important thing to keep in mind: **there is no secret pre-processing formula that will work with all corpora**. Context remains king/queen.

29: In no small part, due to the size of the document; with too few terms, it can be harder to make sense of the processed text.

We can revisit the first game recap we considered (game 56), and look at the new word counts. Originally, the eight most frequent terms ("the", "and", "a", "in", "for", "to", "on", "from") were English stopwords; what are the most frequent terms in the cleaned up corpus?

```
term_count <- freq_terms(clean_corp.AP.recaps[[56]][1],20)
plot(term_count)
```



Document-Term and Term-Document Matrices

The DTM and TDM can also be obtained from `tm`; we show how to create them from the clean game recaps corpus, starting with the DTM.

```
(AP.recaps_dtm <- tm::DocumentTermMatrix(clean_corp.AP.recaps))
```

```
<<DocumentTermMatrix (documents: 101, terms: 3293)>>
Non-/sparse entries: 22187/310406
Sparsity           : 93%
Maximal term length: 15
Weighting          : term frequency (tf)
```

Next, we convert `AP.recaps_dtm` to a matrix.

```
AP.recaps_m <- as.matrix(AP.recaps_dtm)
dim(AP.recaps_m)
```

```
[1]  101 3293
```

We can review a portion of the matrix (keep in mind that the default text statistic is the term frequency tf).

```
AP.recaps_m[79:84, 1005:1010]
    Terms
Docs ferland fewer fibula field fifth fifthround
  79       0     0      0     0     1          0
  80       0     0      0     0     0          0
  81       0     0      0     0     0          0
  82       0     0      0     0     0          0
  83       0     0      0     0     0          0
  84       0     0      0     0     0          0
```

We can do the same thing for the TDM.

```
(AP.recaps_tdm <- tm::TermDocumentMatrix(clean_corp.AP.recaps))
AP.recaps_m <- as.matrix(AP.recaps_tdm)
dim(AP.recaps_m)
AP.recaps_m[1005:1010, 79:84]
```

```
<<TermDocumentMatrix (terms: 3293, documents: 101)>>
Non-/sparse entries: 22187/310406
Sparsity           : 93%
Maximal term length: 15
Weighting          : term frequency (tf)

[1] 3293  101

            Docs
Terms         79 80 81 82 83 84
  ferland      0  0  0  0  0  0
  fewer        0  0  0  0  0  0
  fibula       0  0  0  0  0  0
  field        0  0  0  0  0  0
  fifth        1  0  0  0  0  0
  fifthround   0  0  0  0  0  0
```

Barchart of Frequent Terms with `tm`

These objects can be used to provide a BoW interpretation of the Senators' 2016-2017 season (regular season and playoffs).

We start by computing how often the terms appear in the entire corpus.

```
term_frequency <- rowSums(AP.recaps_m)
```

Next, we sort the term frequencies in descending order.

```
term_frequency <- sort(term_frequency, decreasing=TRUE)
```

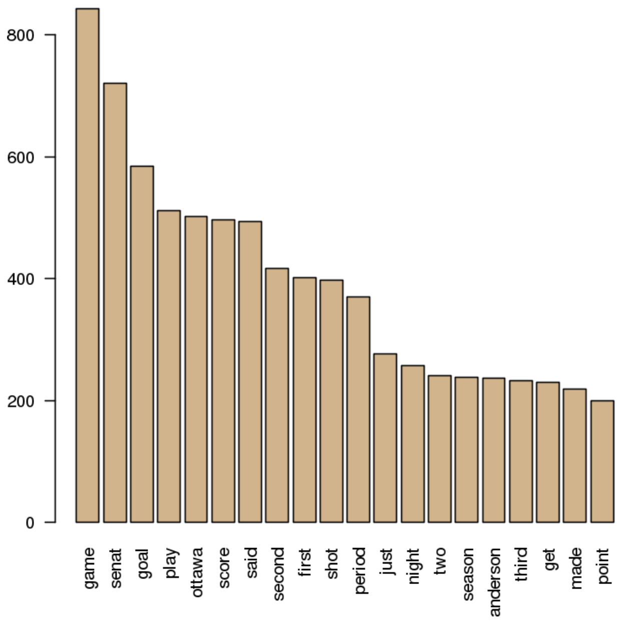
The top 20 most common words in the cleaned corpus are shown below (should we expect ties, as was the case when we looked at a single game recap?).

```
term_frequency[1:20]
```

```
game        score       period      anderson
    843         497         370         237
senat       said        just        third
    720         493         277         233
goal        second      night       get
    584         417         257         230
play        first       two         made
    512         402         240         218
ottawa      shot        season      point
    502         398         238         200
```

We can plot a barchart of the 20 most common words, or a word cloud of the (at most) 100 most common words.

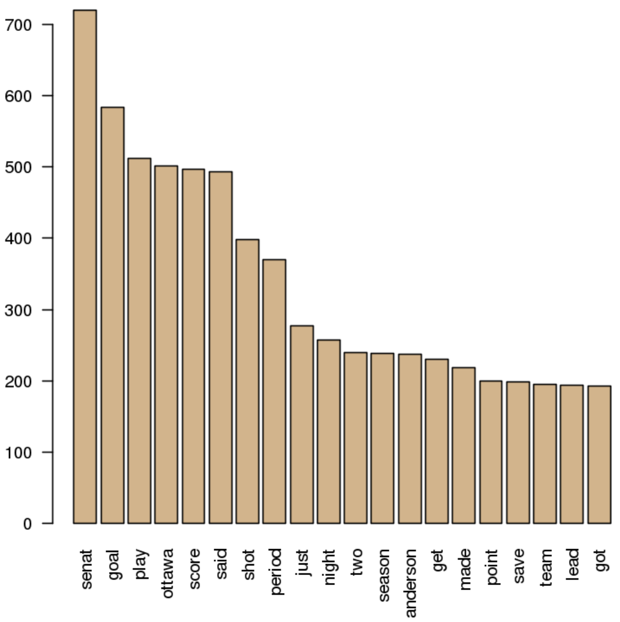
```
barplot(term_frequency[1:20], col = "tan", las = 2)

word_freqs = data.frame(term_frequency)
word_freqs$term = rownames(word_freqs)
word_freqs = word_freqs[,c(2,1)]
colnames(word_freqs)=c("term","num")

wordcloud::wordcloud(word_freqs$term, word_freqs$num, max.words=100, colors="red")
```

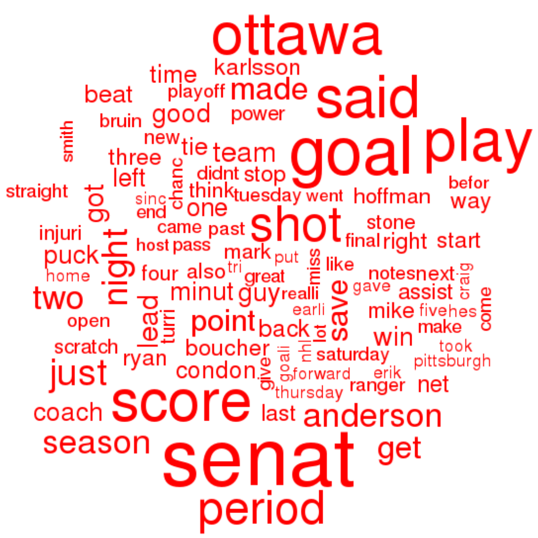


In practice, we already know that the corpus' documents are Ottawa Senators hockey game recaps, so we can remove frequent terms that do not add a lot of information due to the context.

```
clean_corpus_Sens <- function(corpus){
  corpus <- tm::tm_map(corpus, content_transformer(qdap::replace_abbreviation))
  corpus <- tm::tm_map(corpus, tm::removePunctuation)
  corpus <- tm::tm_map(corpus, tm::removeNumbers)
```

```
  corpus <- tm::tm_map(corpus, tm::stemDocument)
  corpus <- tm::tm_map(corpus, tm::content_transformer(tolower))
  corpus <- tm::tm_map(corpus, tm::stripWhitespace)
  corpus <- tm::tm_map(corpus, tm::removeWords, c(tm::stopwords("en"), "game", "first",
                     "second", "third", "Ottawa", "Senators"))
  return(corpus)
}

clean_corp2.AP.recaps <- clean_corpus_Sens(AP.recaps.corpus)
AP.recaps2_tdm <- tm::TermDocumentMatrix(clean_corp2.AP.recaps)
AP.recaps2_m <- as.matrix(AP.recaps2_tdm)

term_frequency2 <- rowSums(AP.recaps2_m)
term_frequency2 <- sort(term_frequency2, decreasing=TRUE)

barplot(term_frequency2[1:20], col = "tan", las = 2)

word_freqs2 = data.frame(term_frequency2)
word_freqs2$term = rownames(word_freqs2)
word_freqs2 = word_freqs2[,c(2,1)]
colnames(word_freqs2)=c("term","num")

wordcloud::wordcloud(word_freqs2$term, word_freqs2$num, max.words=100, colors="red")
```



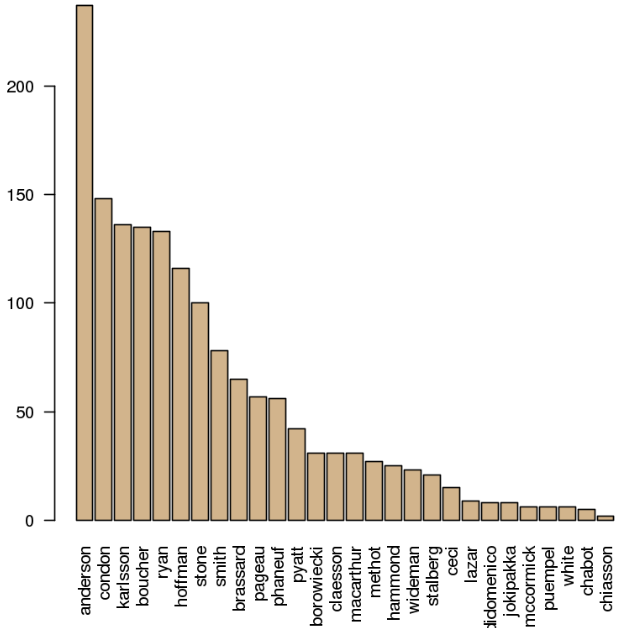


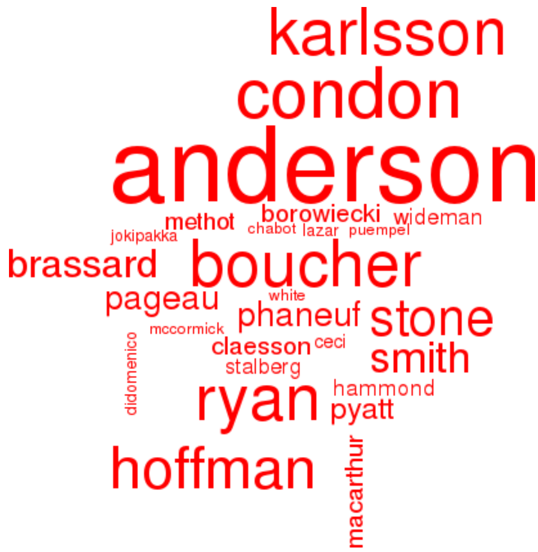
Do we get a better sense for how the season went? Assuming that you knew nothing about how things played out, would you be able to "predict" how close to winning the Stanley Cup the team came?

Finally, we will try to see if the recaps can help us determine the key players in the Senators' season.

```
keep=c("anderson","borowiecki","boucher","brassard","burrows","ceci","chabot","chiasson",
  "claesson","condon","didomenico","drieger","hammond","hoffman","jokipakka","karlsson",
  "lazar","macarthur","mccormick","methot","moore","pageau","phaneuf","puempel","pyatt",
  "ryan","ryans","smith","stalberg","stone","white","wideman","wingels")
```

```
word_freqs3 = word_freqs2[word_freqs2$term %in% keep, ]

barplot(term_frequency2[word_freqs2$term %in% keep], col = "tan", las = 2)
wordcloud::wordcloud(word_freqs3$term, word_freqs3$num, max.words=100, colors="red")
```





The beauty of the BoW approach is that even without any knowledge of the sport, it is rather straightforward to determine the players/personnel who were instrumental to the team's success that year.

27.4.2 Shakespeare vs. Marlowe

In this example, we introduce the basics of **tidy text mining** using the `tidytext` library in `R`, which shares syntax with H. Wickham's popular `tidyverse` suite of packages, which includes `ggplot2`, a powerful graphic library (see Chapter 1 and [9, ch. 13]).

Following [39], we will work with:

- a selection of Shakespeare's plays,
- a selection of Christopher Marlowe's play, and
- the Sens game recaps we work with in the preceding example.

The **tidytext** format (as do the other tidy formats) rely on the programming concept of **pipelines**.

The Pipeline Operator |>

(This section is a repeat of Section 1.4.1)

`R` is a **functional language**, which means that it uses nested parentheses, which can make code difficult to read. The **pipeline operator** `|>` (formerly `%>%`) and the `dplyr` package can be used to remedy the situation. Wickham[30] provided an example to illustrate how it works:

30: See [46] for everything there is to know about pipelines and tidy data.

```
hourly_delay <- filter(
   summarise(
     group_by(
       filter(
         flights,
         !is.na(dep_delay)
       ),
       date, hour
     ),
     delay = mean(dep_delay),
     n = n()
   ),
   n > 10
)
```

Without necessarily knowing how each of the internal functions works, we can still get a sense for what the overall nested structure does, and realize (albeit, with a fair amount of work) that the basic object on which we operate is the `flights` data frame.

The pipeline operator `|>` removes the need for nesting function calls, in favor of passing data from one function to the next:

```
library(dplyr)
hourly_delay <- flights |>
    filter(!is.na(dep_delay)) |>
    group_by(date, hour) |>
    summarise(delay = mean(dep_delay),n = n()) |>
    filter(n > 10)
```

It is now obvious that the flights data frame is the base object, for instance – the **gap** between pseudo-code and "code that runs" is significantly reduced. The beauty of this approach is that the block of code can now be 'read' directly: the `flights` data frame is

1. filtered (to remove missing values of the `dep_delay` variable);
2. grouped by `hours` within `days`;
3. the mean `delay` is calculated within `groups`, and
4. the mean `delay` is returned for those hours with more than `n > 10` flights.

The **pipeline rules** are simple – the object immediately to the left of the pipeline is passed as the first argument to the function immediately to its right:

- `data |> function` is equivalent to `function(data)`
- `data |> function(arg=value)` is equivalent to `function(data, arg=value)`

For instance:

```
library(dplyr)
swiss |> summary()
```

```
   Fertility      Agriculture     Examination      Education
 Min.   :35.00   Min.   : 1.20   Min.   : 3.00   Min.   : 1.00
 1st Qu.:64.70   1st Qu.:35.90   1st Qu.:12.00   1st Qu.: 6.00
 Median :70.40   Median :54.10   Median :16.00   Median : 8.00
 Mean   :70.14   Mean   :50.66   Mean   :16.49   Mean   :10.98
 3rd Qu.:78.45   3rd Qu.:67.65   3rd Qu.:22.00   3rd Qu.:12.00
 Max.   :92.50   Max.   :89.70   Max.   :37.00   Max.   :53.00

    Catholic       Infant.Mortality   threshold
 Min.   :  2.150   Min.   :10.80    Min.   :0.0000
 1st Qu.:  5.195   1st Qu.:18.15    1st Qu.:1.0000
 Median : 15.140   Median :20.00    Median :1.0000
 Mean   : 41.144   Mean   :19.94    Mean   :0.9362
 3rd Qu.: 93.125   3rd Qu.:21.70    3rd Qu.:1.0000
 Max.   :100.000   Max.   :26.60    Max.   :1.0000
```

The magrittr vignette provides additional information on the magrittr package, on which dplyr is based.

Tidy Text Structure

Tidy data has specific structure:[31]

- each column represents a unique variable;
- each row represents a unique observation;
- each table represents a unique type of observational unit.

31: See 1.4 for more information.

Tidy text is a table with one **token** (single word, $n-$gram, sentence, paragraph) per row, assuming that words have been tokenized to commonly-used **units of text**.

As an example, consider the following haiku by master Matsuo Basho.

```
haiku <- c('In the twilight rain',
            'these brilliant-hued hibiscus -',
            'A lovely sunset')
haiku
```

```
[1] "In the twilight rain"   "these brilliant-hued hibiscus -"
[3] "A lovely sunset"
```

We turn it into a data frame.

```
(haiku.df <- data.frame(text=haiku,
                        stringsAsFactors = FALSE))
```

```
                             text
1            In the twilight rain
2 these brilliant-hued hibiscus -
3                 A lovely sunset
```

In the data.frame() call above, the last parameter is important as we want to be able to separate the text into constituents tokens (words). We can **unnest** the tokens of the haiku as follows.

```
library(tidytext)
haiku.df |> unnest_tokens(word,text)
```

```
        word
1         in
2        the
3   twilight
4       rain
5      these
6  brilliant
7       hued
8   hibiscus
9          a
10    lovely
11    sunset
```

The `tidytext` function `unnest_token()` separates the tokens (words, in this example), strips away the punctuation, and converts to lowercase.

Tidy Text Flow

In the tidy text framework, we generally:

1. start with text data;
2. unnest the tokens to produce the first iteration of tidy text;
3. clean the tidy text as required;
4. summarize the tidy text into a first iteration of summarized text;
5. clean and analyze the summarized text, and
6. visualize and present the text mining results.

Tidy Text Analysis

We illustrate the flow with the help of some of Shakespeare's plays, available at the Gutenberg Project (Project ID – *Romeo and Juliet*: 1112; *Hamlet*: 1524; *Macbeth*: 2264; *A Midsummer Night's Dream*: 2242, etc.).

```
library(gutenbergr)
will_shakespeare <- gutenberg_download(c(1790,2240,2242,
                          2243,2246,2250,2251,2253,2262,
                          2264,2267,2268,23042,23046))
head(will_shakespeare,20)
```

```
# A tibble: 20 x 2
   gutenberg_id text
          <int> <chr>
 1         1790 "*******************************************************************"
 2         1790 "THIS EBOOK WAS ONE OF PROJECT GUTENBERG'S EARLY FILES PRODUCED AT A"
 3         1790 "TIME WHEN PROOFING METHODS AND TOOLS WERE NOT WELL DEVELOPED. THERE"
 4         1790 "IS AN IMPROVED EDITION OF THIS TITLE WHICH MAY BE VIEWED AS EBOOK"
 5         1790 "(#1528) at https://www.gutenberg.org/ebooks/1528"
 6         1790 "*******************************************************************"
```

```
 7          1790 ""
 8          1790 ""
 9          1790 "Troilus and Cressida, World Library edition, several typos fixed."
10          1790 ""
11          1790 "This Etext file is presented by Project Gutenberg, in"
12          1790 "cooperation with World Library, Inc., from their Library of the"
13          1790 "Future and Shakespeare CDROMS.  Project Gutenberg often releases"
14          1790 "Etexts that are NOT placed in the Public Domain!!"
15          1790 ""
16          1790 "*This Etext has certain copyright implications you should read!*"
17          1790 ""
18          1790 "<<THIS ELECTRONIC VERSION OF THE COMPLETE WORKS OF WILLIAM"
19          1790 "SHAKESPEARE IS COPYRIGHT 1990-1993 BY WORLD LIBRARY, INC., AND IS"
20          1790 "PROVIDED BY PROJECT GUTENBERG WITH PERMISSION.  ELECTRONIC AND"
```

Not the most stirring literature, to be sure. We now produce (and clean) the corresponding tidy text dataset.

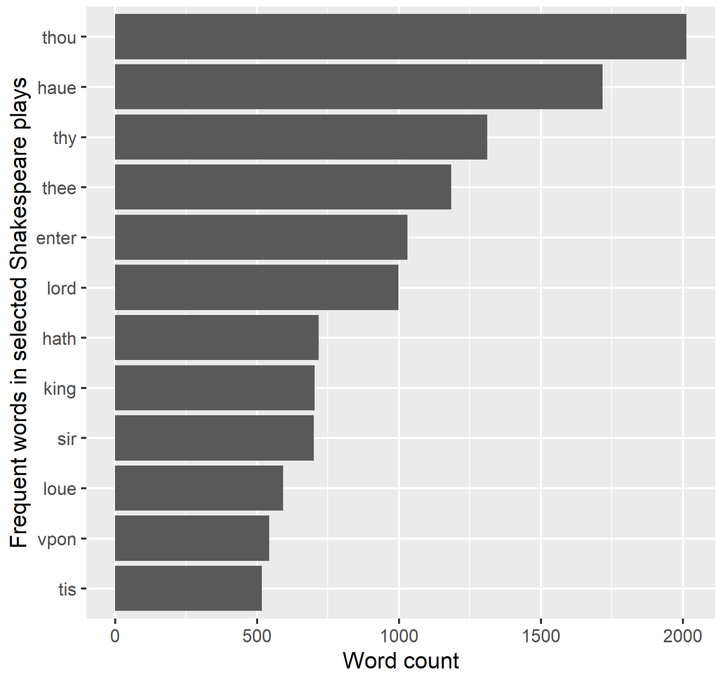
```
library(stringr) # necessary to use str_extract

tidy_ws <- will_shakespeare |>
  unnest_tokens(word,text) |>
  dplyr::mutate(word = str_extract(word,"[a-z']+")) |> # removing stray punctuation, etc.
  dplyr::anti_join(stop_words) |>  # removing the heading business
  na.omit() # removing NAs

head(tidy_ws)
```

```
# A tibble: 6 x 2
  gutenberg_id word
         <int> <chr>
1         1790 ebook
2         1790 project
3         1790 gutenberg's
4         1790 files
5         1790 produced
6         1790 time
```

We can easily produce a word count for this data frame.

```
library(ggplot2)
tidy_ws |> dplyr::count(word, sort=TRUE)

tidy_ws |> dplyr::count(word, sort=TRUE) |>
  dplyr::filter(n > 500) |>
  dplyr::mutate(word=reorder(word,n)) |>
  ggplot(aes(word,n)) +
    geom_col() +
    xlab("Frequent words in selected Shakespeare plays") +
    ylab("Word count") +
    coord_flip()
```

```
# A tibble: 20,297 x 2
   word      n
   <chr> <int>
 1 thou   2014
 2 haue   1718
 3 thy    1311
 4 thee   1185
 5 enter  1030
 6 lord    999
 7 hath    717
 8 king    703
 9 sir     700
10 loue    593
# ... with 20,287 more rows
```



Shakespeare and Marlowe

We can do the same for Christopher Marlowe, a contemporary of Shakespeare.

```
kit_marlowe <-gutenberg_download(c(901,1094,1496,1589,
                                   16169,18781,20288))

tidy_km <- kit_marlowe |>  unnest_tokens(word,text) |>
  dplyr::mutate(word = str_extract(word,"[a-z']+")) |>
  dplyr::anti_join(stop_words) |>
  na.omit() # remove NAs

head(tidy_km)
```

```
# A tibble: 6 x 2
  gutenberg_id word
         <int> <chr>
1          901 jew
2          901 malta
3          901 christopher
4          901 marlowe
5          901 edited
6          901 rev
```

Next, we look at both of these datasets **simultaneously**. In order to do so, we build a word count data set with the help of the pipeline operator. One of its advantages is that we can build the query sequentially and easily see the output at various stages.

We start by binding `tidy_ws` and `tidy_km` into a single data frame.

```
word_count <- dplyr::bind_rows(dplyr::mutate(tidy_ws,author="WillShakespeare"),
                               dplyr::mutate(tidy_km,author="KitMarlowe"))
head(word_count)
tail(word_count)
```

```
# A tibble: 6 x 3
  gutenberg_id word        author
         <int> <chr>       <chr>
1         1790 ebook       WillShakespeare
2         1790 project     WillShakespeare
3         1790 gutenberg's WillShakespeare
4         1790 files       WillShakespeare
5         1790 produced    WillShakespeare
6         1790 time        WillShakespeare

# A tibble: 6 x 3
  gutenberg_id word      author
         <int> <chr>     <chr>
1        20288 mine      KitMarlowe
2        20288 eyes      KitMarlowe
3        20288 witness   KitMarlowe
4        20288 grief     KitMarlowe
5        20288 innocency KitMarlowe
6        20288 exeunt    KitMarlowe
```

Next, we execute a word count for each of the authors (note the sorting of the outputs, and the new field `n`).

```
word_count <- dplyr::bind_rows(dplyr::mutate(tidy_ws,author="WillShakespeare"),
                               dplyr::mutate(tidy_km,author="KitMarlowe")) |>
              dplyr::count(author,word) |>  dplyr::group_by(author)

head(word_count)
tail(word_count)
```

```
# A tibble: 6 x 3
# Groups:   author [1]
  author     word          n
  <chr>      <chr>     <int>
1 KitMarlowe abandon       3
2 KitMarlowe abandon'd     1
3 KitMarlowe abandons      1
4 KitMarlowe abate         2
5 KitMarlowe abated        1
6 KitMarlowe abb           1

# A tibble: 6 x 3
# Groups:   author [1]
  author          word        n
  <chr>           <chr>   <int>
1 WillShakespeare zeal        2
2 WillShakespeare zeale       9
3 WillShakespeare zeales      1
4 WillShakespeare zealous     1
5 WillShakespeare zenith      1
6 WillShakespeare zip         1
```

In order to use the tidy approach, we need `word_count` to have a unique value for each word for each author. Note that the size of each of the author datasets is different, as we are using a higher number of Shakespeare plays. Rather than look at **raw counts** (which would naturally favour the Bard's output), we consider proportions:

$$\frac{\text{number of occurrences of a specific term in an author's dataset}}{\text{total number of terms in an author's dataset}}.$$

```
word_count <- dplyr::bind_rows(dplyr::mutate(tidy_ws,author="WillShakespeare"),
                               dplyr::mutate(tidy_km,author="KitMarlowe")) |>
              dplyr::count(author,word) |>  dplyr::group_by(author) |>
              dplyr::mutate(proportion = n / sum(n))

head(word_count)
tail(word_count)
```

```
# A tibble: 6 x 4
# Groups:   author [1]
  author     word          n proportion
  <chr>      <chr>     <int>      <dbl>
1 KitMarlowe abandon       3  0.0000513
2 KitMarlowe abandon'd     1  0.0000171
3 KitMarlowe abandons      1  0.0000171
4 KitMarlowe abate         2  0.0000342
5 KitMarlowe abated        1  0.0000171
6 KitMarlowe abb           1  0.0000171

# A tibble: 6 x 4
# Groups:   author [1]
  author          word        n proportion
```

```
  <chr>           <chr>   <int>      <dbl>
1 WillShakespeare zeal        2 0.0000136
2 WillShakespeare zeale       9 0.0000611
3 WillShakespeare zeales      1 0.00000679
4 WillShakespeare zealous     1 0.00000679
5 WillShakespeare zenith      1 0.00000679
6 WillShakespeare zip         1 0.00000679
```

We can now remove the raw counts and focus solely on the proportions.

```
word_count <- dplyr::bind_rows(dplyr::mutate(tidy_ws,author="WillShakespeare"),
                               dplyr::mutate(tidy_km,author="KitMarlowe")) |>
              dplyr::count(author,word) |>  dplyr::group_by(author) |>
              dplyr::mutate(proportion = n / sum(n)) |> dplyr::select(-c(n))

word_count
```

```
# A tibble: 31,489 x 3
# Groups:   author [2]
      author               word    proportion
      <chr>                <chr>        <dbl>
    1 KitMarlowe       abandon    0.0000513
    2 KitMarlowe       abandon'd  0.0000171
    3 KitMarlowe       abandons   0.0000171
    4 KitMarlowe       abate      0.0000342
    5 KitMarlowe       abated     0.0000171
    6 KitMarlowe       abb        0.0000171
      ...
31484 WillShakespeare  zeal       0.0000136
31485 WillShakespeare  zeale      0.0000611
31486 WillShakespeare  zeales     0.00000679
31487 WillShakespeare  zealous    0.00000679
31488 WillShakespeare  zenith     0.00000679
31489 WillShakespeare  zip        0.00000679
```

Next, we reshape `word_count` to facilitate the analysis: each word is now represented by a row, and the proportion of the time it appears in each author's writings is shown in the corresponding column.

```
word_count <- dplyr::bind_rows(dplyr::mutate(tidy_ws,author="WillShakespeare"),
                               dplyr::mutate(tidy_km,author="KitMarlowe")) |>
              dplyr::count(author,word) |>  dplyr::group_by(author) |>
              dplyr::mutate(proportion = n / sum(n)) |> dplyr::select(-c(n)) |>
              tidyr::spread(author,proportion)

word_count
```

```
# A tibble: 25,020 x 3
       word   KitMarlowe WillShakespeare
       <chr>       <dbl>           <dbl>
 1 a'kin      NA              0.00000679
 2 a'th       NA              0.0000814
```

```
 3 a'that     NA               0.00000679
 4 abandon     0.0000513       0.0000136
 5 abandon'd   0.0000171       0.0000136
 6 abandons    0.0000171      NA
 7 abash'd    NA               0.00000679
 8 abate       0.0000342       0.0000475
 9 abated      0.0000171      NA
10 abates     NA               0.00000679
# ... with 25,010 more rows
```

We can easily see what proportion of each author's output is not found in the other's.

```
# % of Shakespeare's output terms not in Marlowe's
(WS_nKM <- sum(word_count$WillShakespeare[is.na(word_count$KitMarlowe)]))
# % of Marlowe's output terms not in Shakespeare's
(KM_nWS <- sum(word_count$KitMarlowe[is.na(word_count$WillShakespeare)]))
```

```
[1] 0.3092499
[1] 0.2497649
```

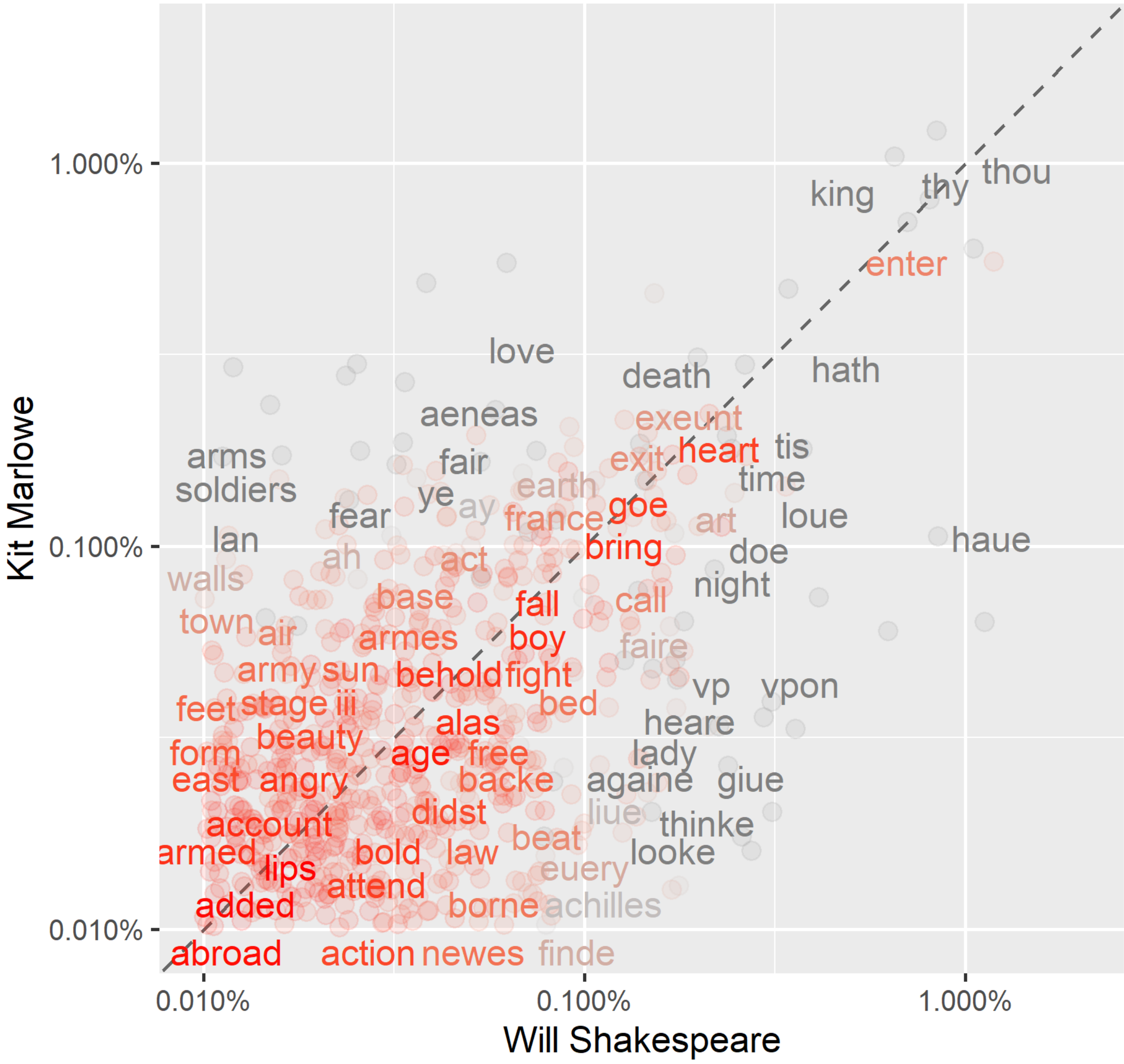
Do these proportions seem high, given that they were contemporaries? Finally, we re-organize the table for use with `ggplot()` (strictly-speaking, this step is not mandatory, but the charts we produce will look nicer).

```
word_count <- dplyr::bind_rows(dplyr::mutate(tidy_ws,author="WillShakespeare"),
                               dplyr::mutate(tidy_km,author="KitMarlowe")) |>
              dplyr::count(author,word) |>  dplyr::group_by(author) |>
              dplyr::mutate(proportion = n / sum(n)) |> dplyr::select(-c(n)) |>
              tidyr::spread(author,proportion) |>
              tidyr::gather(author, proportion, `WillShakespeare`)
```

Here is a logarithmic scale scatterplot of word usage by both authors (for words that were used by both).

```
library(scales)
ggplot(word_count, aes(x = proportion, y = `KitMarlowe`,
                       color = abs(`KitMarlowe` - proportion))) +
  geom_abline(color = "gray40", lty = 2) +
  geom_jitter(alpha = 0.1, size = 2.5, width = 0.3, height = 0.3) +
  geom_text(aes(label = word), check_overlap = TRUE, vjust = 1.5) +
  scale_x_log10(labels = percent_format(),limits=c(0.0001,0.02)) +
  scale_y_log10(labels = percent_format(),limits=c(0.0001,0.02)) +
  scale_color_gradient(limits = c(0, 0.001), low = "red", high = "gray75") +
  theme(legend.position="none") +
  labs(y = "Kit Marlowe", x = "Will Shakespeare")
```

```
Warning messages:
1: Removed 24363 rows containing missing values
(geom_point).
2: Removed 24205 rows containing missing values
(geom_text).
```



Words near the straight line are used with roughly the same frequency by both authors. For instance: "king", "thou", and "thy" in the high-frequency spectrum, and "angry", "alas", and "behold" in the low-frequency spectrum.

Words away from the straight line are used more frequently by one of the authors: "lady" and "achilles" seem to be used relatively more often by Shakespeare than by Marlowe, while "aeneas" is in the opposite situation (these terms are specific to plays).

The colour is related to the (real) distance between the relative frequencies of a term for each author (red is near, gray is far) – the logarithmic scales of both axes explain the shape of the red cloud (large at the bottom, thin at the top).[32]

Note the presence of "armes", "arms", and "armed", or of "love" and "loue" – what does that tell us about the text (and the English used)? Should we be surprised about the prevalence of terms like "enter", "exit", and "exeunt"?

32: Do you see why this needs to be the case?

Finally, let's see if we can quantify the similarity in word usage.

```
cor.test(data = word_count, ~ proportion + ‘KitMarlowe‘)
```

```
  Pearson’s product-moment correlation

data:  proportion and KitMarlowe
t = 89.335, df = 6467, p-value < 2.2e-16
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.7321162 0.7539396
sample estimates:
      cor
0.7432256
```

There's a fairly strong correlation (0.74) between the relative term frequencies for the two wordsmiths (among those terms which are found in both text outputs – recall `KM_nWS` and `WS_nKM`). That should not be entirely unexpected, since they were contemporaries: one would naïvely predict that the depth of their vocabulary and the way they deployed it would be linked, to some extent.

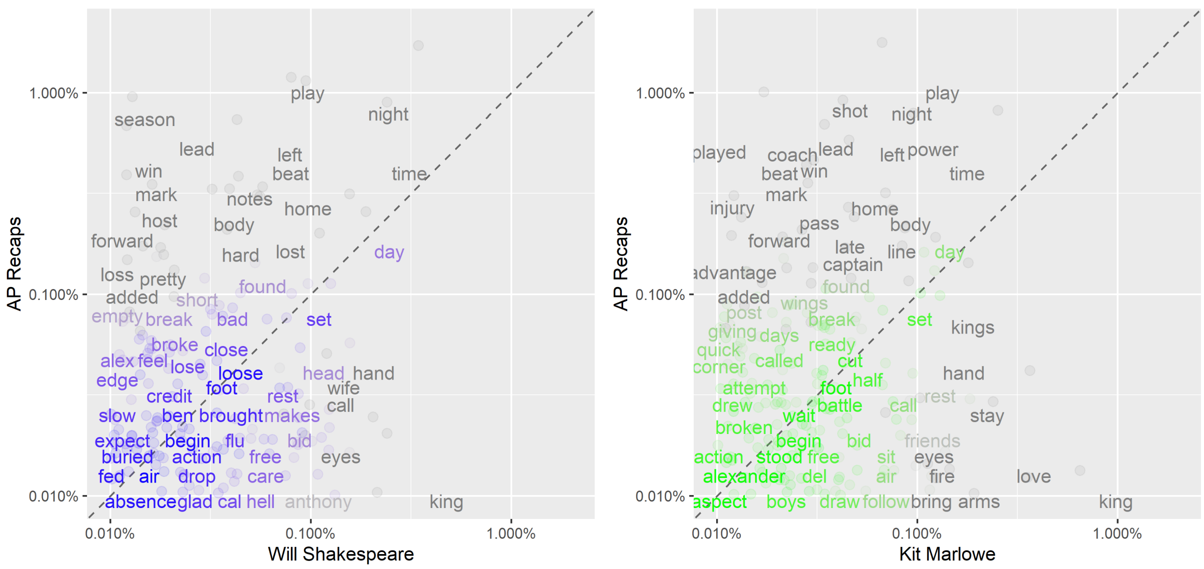
But without comparisons to other texts, it is difficult to really put this value in perspective.

Shakespeare and Sens Game Recaps

Let's see how Shakespeare and Marlowe compare to a modern body of work, the NHL Senators' game recaps from the previous section.

```
recaps <- read.csv(file="Recap_data.csv", header=TRUE, sep=",", stringsAsFactors=FALSE)

AP.recaps <- recaps$AP_Recap

recaps.df <- data.frame(text=AP.recaps, stringsAsFactors = FALSE)

tidy_AP <- recaps.df |>
  tidytext::unnest_tokens(word,text) |>
  dplyr::mutate(word = str_extract(word,"[a-z’]+")) |>
  dplyr::anti_join(stop_words) |>
  na.omit() # remove NAs

head(tidy_AP) # inspect

word_count_2 <- dplyr::bind_rows(dplyr::mutate(tidy_ws,author="WillShakespeare"),
                                 dplyr::mutate(tidy_AP,author="AP_recaps")) |>
                dplyr::count(author,word) |>
                dplyr::group_by(author) |>
                dplyr::mutate(proportion = n / sum(n)) |>
                dplyr::select(-c(n)) |>
                tidyr::spread(author,proportion) |>
                tidyr::gather(author, proportion, ‘WillShakespeare‘)
```

```
word_count_3 <- dplyr::bind_rows(dplyr::mutate(tidy_km,author="KitMarlowe"),
                                 dplyr::mutate(tidy_AP,author="AP_recaps")) |>
                dplyr::count(author,word) |>
                dplyr::group_by(author) |>
                dplyr::mutate(proportion = n / sum(n)) |>
                dplyr::select(-c(n)) |>
                tidyr::spread(author,proportion) |>
                tidyr::gather(author, proportion, `KitMarlowe`)

ggplot(word_count_2, aes(x = proportion, y = `AP_recaps`,
                         color = abs(`AP_recaps` - proportion))) +
  geom_abline(color = "gray40", lty = 2) +
  geom_jitter(alpha = 0.1, size = 2.5, width = 0.3, height = 0.3) +
  geom_text(aes(label = word), check_overlap = TRUE, vjust = 1.5) +
  scale_x_log10(labels = percent_format(),limits=c(0.0001,0.02)) +
  scale_y_log10(labels = percent_format(),limits=c(0.0001,0.02)) +
  scale_color_gradient(limits = c(0, 0.001), low = "blue", high = "gray75") +
  theme(legend.position="none") +
  labs(y = "AP Recaps", x = "Will Shakespeare")

ggplot(word_count_3, aes(x = proportion, y = `AP_recaps`,
                         color = abs(`AP_recaps` - proportion))) +
  geom_abline(color = "gray40", lty = 2) +
  geom_jitter(alpha = 0.1, size = 2.5, width = 0.3, height = 0.3) +
  geom_text(aes(label = word), check_overlap = TRUE, vjust = 1.5) +
  scale_x_log10(labels = percent_format(),limits=c(0.0001,0.02)) +
  scale_y_log10(labels = percent_format(),limits=c(0.0001,0.02)) +
  scale_color_gradient(limits = c(0, 0.001), low = "green", high = "gray75") +
  theme(legend.position="none") +
  labs(y = "AP Recaps", x = "Kit Marlowe")
```



We can see the proportion of terms not found in the other corpora.

```
# % of Shakespeare's terms not in the game recaps
(WS_nAP <- sum(word_count_2$proportion[is.na(word_count_2$AP_recaps)]))
```

```
# % of game recaps' terms not in the selected Shakespeare plays
(AP_nWS <- sum(word_count_2$AP_recaps[is.na(word_count_2$proportion)]))
# % of Shakespeare's terms not in the game recaps
(KM_nAP <- sum(word_count_3$proportion[is.na(word_count_3$AP_recaps)]))
# % of game recaps' terms not in the selected Marlowe plays
(AP_nKM <- sum(word_count_3$AP_recaps[is.na(word_count_3$proportion)]))
```

```
[1] 0.8026031
[1] 0.4804179
[1] 0.7732845
[1] 0.5588203
```

The proportions are much higher than when comparing Marlowe's and Shakespeare's outputs. In light of the differences in terms of topics, style, and linguistic drift over the centuries, should any of these be surprising?

The corresponding correlations are shown below.

```
cor.test(data = word_count_2, ~ proportion + `AP_recaps`)
cor.test(data = word_count_3, ~ proportion + `AP_recaps`)
```

```
  Pearson's product-moment correlation

data:  proportion and AP_recaps
t = 2.8059, df = 1377, p-value = 0.005089
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.02270094 0.12767835
sample estimates:
       cor
0.07539856

  Pearson's product-moment correlation

data:  proportion and AP_recaps
t = 1.2599, df = 1142, p-value = 0.208
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 -0.02074664  0.09501030
sample estimates:
       cor
0.03725681
```

Again, it should come as no surprise that the recap writers do not use English in the same BoW way that Shakespeare and Marlowe did.

It would be interesting to see if these results are stable under a different subset of Shakespeare and Marlowe plays.

n−Grams

Up to this point, we have been using **word**, **term**, **token**, **unit** interchangeably when analyzing text, as befits the BoW approach.

It's not too difficult to think of applications where the basic numerical unit is not the relative frequency (or tf-idf) of **single words**, however, but the **links between 2 or more words**, in succession or in co-occurrence. Rather than tokenize some text by words, we can tokenize it by series of n **consecutive words** (also called n−**grams**).

In what follows, we focus on $n = 2$. Are there interesting **bigrams** in Shakespeare's plays? What would we expect his common bigrams to be?

```
tidy_ws.2 <- will_shakespeare |>
  tidytext::unnest_tokens(bigram,text,token="ngrams",n=2) |>
  dplyr::mutate(bigram = stringr::str_extract(bigram,"[0-9a-zA-Z'\ ]+")) |>
  dplyr::count(bigram,sort=TRUE) |>
  na.omit() # produce a count and sort on decreasing frequency

tidy_ws.2
```

```
# A tibble: 143,344 x 2
   bigram      n
   <chr>   <int>
 1 i am      671
 2 in the    636
 3 my lord   604
 4 i will    584
 5 of the    581
 6 to the    528
 7 i haue    517
 8 it is     424
 9 that i    319
10 and the   305
11 to be     304
12 and i     287
13 is the    284
14 i would   256
15 of my     253
16 i know    244
17 i do      240
18 you are   234
19 if you    222
20 is not    220
# ... with 143,324 more rows
```

There are a **lot more** bigrams than there were individual terms, which makes sense from a combinatorial perspective. At first glance, among the top 10 most frequent bigrams, only one conveys even a sliver of information: "my lord". Everything else is stopword material.

However, what about the 11th most frequent bigram? In a general context, "to be" is a stopword bigram – but there is at least a few specific instance in the Shakespearean context where that specific bigram is emphatically not just a "stopword".[33]

33: "To be, or not to be, that is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune,
Or to take Arms against a Sea of troubles,
And by opposing end them: to die, to sleep
No more; and by a sleep, to say we end
The heart-ache, and the thousand natural shocks
That Flesh is heir to?
– *Hamlet* (Act 3, Scene 1)

Removing bigram stopwords is simple, although not as straigthforward as in the unigram case:

1. split the two members of the bigrams into 2 columns;
2. verify if each, separately, is a regular stopword, and
3. remove the bigrams for which one of the components is a stopword.

For the sake of this exercise, let's also remove words related to the printing business, and theatre terms.

```
word = c("gutenberg","shakespeare","","etext","1990","1993","public","print","copies"
        ,"membership","commercial","commercially","electronic","download","distribution"
        ,"ff","f1","f2","f3","f4","NA","collier","ms","cap","txt","zip"
        ,"library","printed", "text","editions"
        ,"executive", "pobox", "fees", "million", "ascii", "legal", "61825", "2782"
        ,"director", "machine","readable","carnegie","mellon","university"
        ,"exit", "exeunt", "enter", "scene", "act", "folio", "dramatis"
        ,"mine","tis", "thine","thy", "thou","art","hast", "shalt","dost","thee"
        ,"act_4","act_1","act_2","act_3","act_5","sc_1","sc_2","sc_3","sc_4","sc_5"
        ,"sc_6","sc_7","sc_8","sc_9","sc_10","sc_11")
lexicon = rep("modern",length(word)) # let's call it the modern lexicon
addition = data.frame(word,lexicon)
stop_words_ws = rbind(stop_words,addition)

tidy_ws.2_cleaned <- tidy_ws.2 |>
  tidyr::separate(bigram, c("FirstTerm","SecondTerm"), sep=" ") |>
  dplyr::filter(!FirstTerm %in% stop_words_ws$word) |>
  dplyr::filter(!SecondTerm %in% stop_words_ws$word)

tidy_ws.2_cleaned <- tidy_ws.2_cleaned[!is.na(tidy_ws.2_cleaned$FirstTerm) &
                                       !is.na(tidy_ws.2_cleaned$SecondTerm), ]
tidy_ws.2_cleaned <- tidy_ws.2_cleaned |>
                     tidyr::unite(bigram,FirstTerm,SecondTerm, sep=" ")
tidy_ws.2_cleaned
```

```
# A tibble: 31,721 x 2
   bigram          n
   <chr>       <int>
 1 haue beene     32
 2 sir iohn       30
 3 ha ha          27
 4 om pope        27
 5 anon conj      23
 6 haue heard     23
 7 haue lost      22
 8 haue seene     22
 9 noble lord     21
10 hath beene     17
# ... with 31,711 more rows
```

Other bigram and $n-$gram ideas can be found in Section 27.5.

27.4.3 The Play's the Thing

In this section, we will take a more in-depth look at **text visualizations**, which play a role just as important in text analysis as visualizations do in numerical data science. We are somewhat hampered by the lack of numerical values, but there are **workarounds**.

We will work with a set of Shakespearean plays, categorized into **comedies**, **tragedies**, and **histories**. We will use the `tm` and `qdap` libraries in `R`, among others.

Loading the Data

We start by loading the data into three corpora.

```
corpus_C <- tm::Corpus(tm::DirSource("ShakespeareComedies/"),
                       readerControl=list(language="lat"))
corpus_T <- tm::Corpus(tm::DirSource("ShakespeareTragedies/"),
                       readerControl=list(language="lat"))
corpus_H <- tm::Corpus(tm::DirSource("ShakespeareHistories/"),
                       readerControl=list(language="lat"))
summary(corpus_C)
summary(corpus_T)
summary(corpus_H)
```

```
# corpus C
                                          Length Class            Mode
A_Midsummer_Nights_Dream_first_pass.txt   2      PlainTextDocument list
Alls_Well_That_Ends_Well_first_pass.txt   2      PlainTextDocument list
As_You_Like_It_first_pass.txt             2      PlainTextDocument list
Cymbeline_first_pass.txt                  2      PlainTextDocument list
Loves_Labours_Lost_first_pass.txt         2      PlainTextDocument list
Measure_for_Measure_first_pass.txt        2      PlainTextDocument list
Much_Ado_About_Nothing_first_pass.txt     2      PlainTextDocument list
Pericles_Prince_of_Tyre_first_pass.txt    2      PlainTextDocument list
Taming_of_the_Shrew_first_pass.txt        2      PlainTextDocument list
The_Comedy_of_Errors_first_pass.txt       2      PlainTextDocument list
The_Merchant_of_Venice_first_pass.txt     2      PlainTextDocument list
The_Merry_Wives_of_Windsor_first_pass.txt 2      PlainTextDocument list
The_Tempest_first_pass.txt                2      PlainTextDocument list
Troilus_and_Cressida_first_pass.txt       2      PlainTextDocument list
Twelfth_Night_first_pass.txt              2      PlainTextDocument list
Two_Gentlemen_of_Verona_first_pass.txt    2      PlainTextDocument list
Winters_Tale_first_pass.txt               2      PlainTextDocument list

# corpus_T
                                    Length Class            Mode
Antony_and_Cleopatra_first_pass.txt 2      PlainTextDocument list
Coriolanus_first_pass.txt           2      PlainTextDocument list
Hamlet_first_pass.txt               2      PlainTextDocument list
Julius_Caesar_first_pass.txt        2      PlainTextDocument list
King_Lear_second_pass.txt           2      PlainTextDocument list
Macbeth_first_pass.txt              2      PlainTextDocument list
```

```
Othello_first_pass.txt             2      PlainTextDocument list
Romeo_and_Juliet_first_pass.txt    2      PlainTextDocument list
Timon_of_Athens_first_pass.txt     2      PlainTextDocument list
Titus_Andronicus_first_pass.txt    2      PlainTextDocument list

# corpus_H
                              Length Class             Mode
Henry_IV_part_1_first_pass.txt 2     PlainTextDocument list
Henry_IV_part_2_first_pass.txt 2     PlainTextDocument list
Henry_V_first_pass.txt         2     PlainTextDocument list
Henry_VI_part_1_first_pass.txt 2     PlainTextDocument list
Henry_VI_part_2_first_pass.txt 2     PlainTextDocument list
Henry_VI_part_3_first_pass.txt 2     PlainTextDocument list
Henry_VIII_first_pass.txt      2     PlainTextDocument list
King_John_first_pass.txt       2     PlainTextDocument list
Richard_II_first_pass.txt      2     PlainTextDocument list
Richard_III_first_pass.txt     2     PlainTextDocument list
```

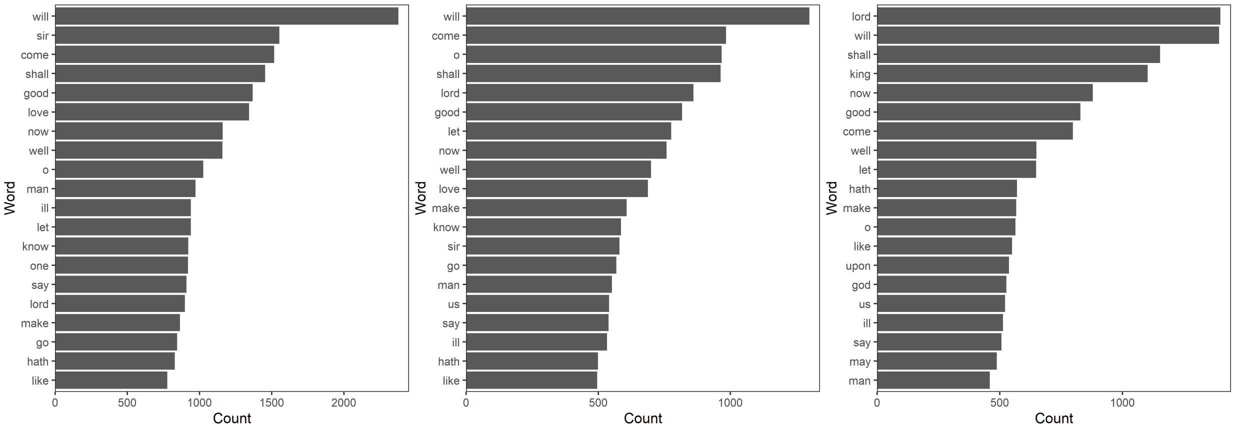
Cleaning the Data

Next, we build a cleaning function for the text and apply it to each corpus.

```
clean_corpus <- function(corpus){
  corpus <- tm::tm_map(corpus, tm::removePunctuation)
  corpus <- tm::tm_map(corpus, tm::removeNumbers)
  corpus <- tm::tm_map(corpus, tm::stemDocument, language="english")
  corpus <- tm::tm_map(corpus, tm::content_transformer(tolower))
  corpus <- tm::tm_map(corpus, tm::stripWhitespace)
  corpus <- tm::tm_map(corpus, tm::removeWords,
                     c(tm::stopwords("english"),
                     c("I", "and", "the", "that", "thou", "thee", "thi", "â")))
  return(corpus)
}

clean_C = clean_corpus(corpus_C)
clean_T = clean_corpus(corpus_T)
clean_H = clean_corpus(corpus_H)
```

We find the 20 most frequent terms in each corpus.

```
term_count_C <- qdap::freq_terms(clean_C,20)
term_count_T <- qdap::freq_terms(clean_T,20)
term_count_H <- qdap::freq_terms(clean_H,20)

plot(term_count_C)
plot(term_count_T)
plot(term_count_H)
```

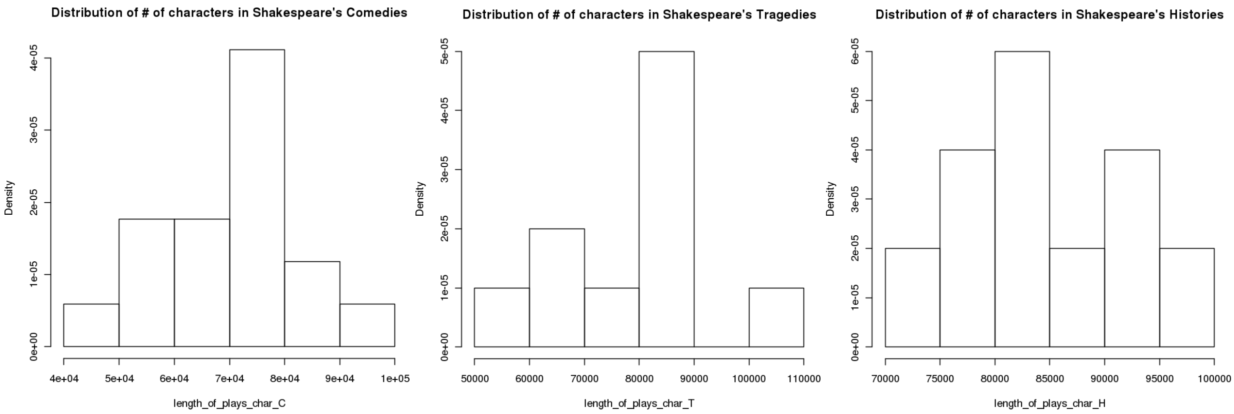


comedies tragedies histories

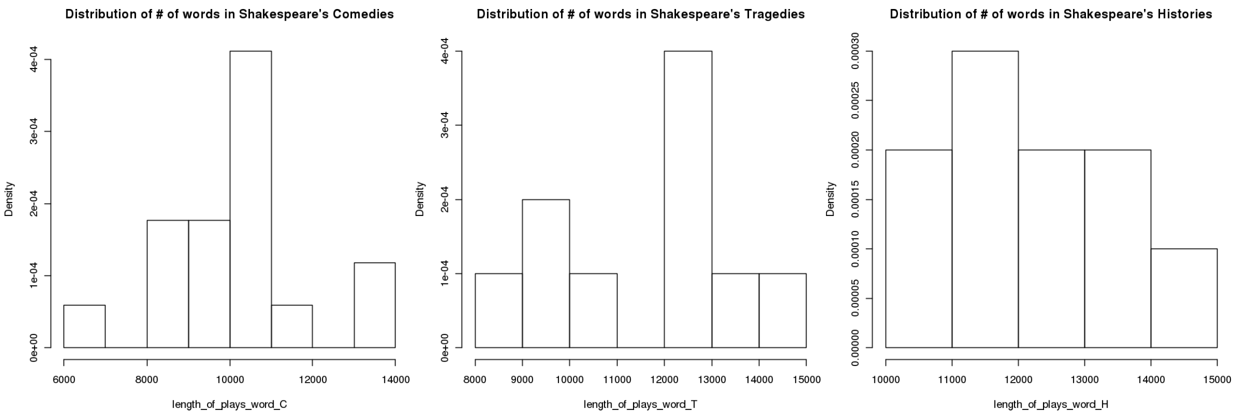
Basic Statistics

We can also take a look at some basic statistics regarding the number of characters (letters, not people) and the number of words in each play.

```
length_of_plays_char_C <- vector(mode="numeric", length=17)
for(j in 1:17){length_of_plays_char_C[j]=nchar(clean_C[[j]][1])}
hist(length_of_plays_char_C, freq=F, main="Distribution of # of char in Shakespeare's Comedies")
summary(length_of_plays_char_C)

length_of_plays_char_T <- vector(mode="numeric", length=10)
for(j in 1:10){length_of_plays_char_T[j]=nchar(clean_T[[j]][1])}
hist(length_of_plays_char_T, freq=F, main="Distribution of # of char in Shakespeare's Tragedies")
summary(length_of_plays_char_T)

length_of_plays_char_H <- vector(mode="numeric", length=10)
for(j in 1:10){length_of_plays_char_H[j]=nchar(clean_H[[j]][1])}
hist(length_of_plays_char_H, freq=F, main="Distribution of # of char in Shakespeare's Histories")
summary(length_of_plays_char_H)

length_of_plays_word_C <- vector(mode="numeric", length=17)
for(j in 1:17){length_of_plays_word_C[j]=length(
               strsplit(gsub(' {2,}',' ',clean_C[[j]][1]),' ')[[1]])}
hist(length_of_plays_word_C, freq=F, main="Distribution of # of words in Shakespeare's Comedies")
summary(length_of_plays_word_C)

length_of_plays_word_T <- vector(mode="numeric", length=10)
for(j in 1:10){length_of_plays_word_T[j]=length(
               strsplit(gsub(' {2,}',' ',clean_T[[j]][1]),' ')[[1]])}
hist(length_of_plays_word_T, freq=F, main="Distribution of # of words in Shakespeare's Tragedies")
summary(length_of_plays_word_T)

length_of_plays_word_H <- vector(mode="numeric", length=10)
for(j in 1:10){length_of_plays_word_H[j]=length(
               strsplit(gsub(' {2,}',' ',clean_H[[j]][1]),' ')[[1]])}
hist(length_of_plays_word_H, freq=F, main="Distribution of # of words in Shakespeare's Histories")
summary(length_of_plays_word_H)
```




```
# summary(length_of_plays_char_C)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  47368   61656   70294   69295   74133   91107

# summary(length_of_plays_char_T)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  62703   67876   82889   79677   87843  100786

# summary(length_of_plays_char_H)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  71811   78630   83089   84155   89740   99953
```




```
# summary(length_of_plays_word_C)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   6930    9056   10198   10090   10849   13260

# summary(length_of_plays_word_T)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   9301   10246   12360   11759   12905   14718

# summary(length_of_plays_word_H)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  10488   11535   12142   12270   13089   14475
```

Term-Document Matrices

We convert the corpora to TDM and remove terms that are **sparse** (too infrequent).

```
# Create TDMs
C_tdm <- tm::TermDocumentMatrix(clean_C)
T_tdm <- tm::TermDocumentMatrix(clean_T)
H_tdm <- tm::TermDocumentMatrix(clean_H)

# Remove sparse terms, with sparsity factor 75%
C_tdm <- tm::removeSparseTerms(C_tdm, 0.75)
T_tdm <- tm::removeSparseTerms(T_tdm, 0.75)
H_tdm <- tm::removeSparseTerms(H_tdm, 0.75)

# Print meta data
C_tdm
T_tdm
H_tdm

# Convert to matrices
C_m <- as.matrix(C_tdm)
T_m <- as.matrix(T_tdm)
H_m <- as.matrix(H_tdm)
```

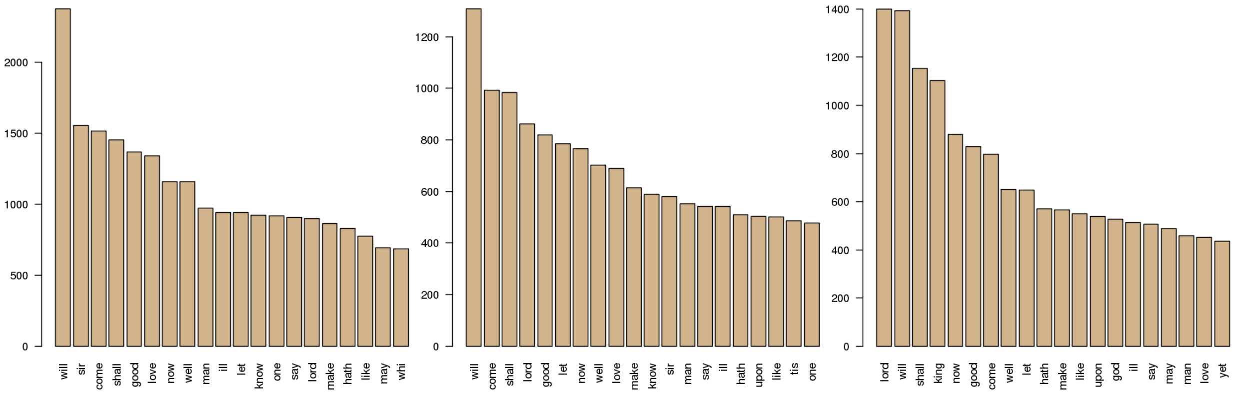
```
<<TermDocumentMatrix (terms: 2945, documents: 17)>>
Non-/sparse entries: 29556/20509
Sparsity           : 41%
Maximal term length: 12
Weighting          : term frequency (tf)

<<TermDocumentMatrix (terms: 3491, documents: 10)>>
Non-/sparse entries: 20699/14211
Sparsity           : 41%
Maximal term length: 12
Weighting          : term frequency (tf)

<<TermDocumentMatrix (terms: 3757, documents: 10)>>
Non-/sparse entries: 22335/15235
Sparsity           : 41%
Maximal term length: 14
Weighting          : term frequency (tf)
```

Barcharts

Next, we produce barcharts of the 20 most-frequent (sparsity-removed) terms in each corpus.

```
term_frequency_C <- rowSums(C_m)
term_frequency_T <- rowSums(T_m)
term_frequency_H <- rowSums(H_m)
```

```
# Sort term_frequency in descending order
term_frequency_C <- sort(term_frequency_C, decreasing=TRUE)
term_frequency_T <- sort(term_frequency_T, decreasing=TRUE)
term_frequency_H <- sort(term_frequency_H, decreasing=TRUE)

# Plot a barchart of the 20 most common words
barplot(term_frequency_C[1:20], col = "tan", las = 2)
barplot(term_frequency_T[1:20], col = "tan", las = 2)
barplot(term_frequency_H[1:20], col = "tan", las = 2)
```



comedies tragedies histories

Word Clouds, Commonality Clouds, and Comparison Clouds

It isn't always easy to read off the terms (we could also list them, of course) or to get a sense for how the corpora differ from one another with barcharts; word clouds (where the size of the word is linked to its frequency in the text) can help.

```
# Create word_freqs
word_freqs_C = data.frame(term_frequency_C)
word_freqs_C$term = rownames(word_freqs_C)
word_freqs_C = word_freqs_C[,c(2,1)]
colnames(word_freqs_C)=c("term","num")

word_freqs_T = data.frame(term_frequency_T)
word_freqs_T$term = rownames(word_freqs_T)
word_freqs_T = word_freqs_T[,c(2,1)]
colnames(word_freqs_T)=c("term","num")

word_freqs_H = data.frame(term_frequency_H)
word_freqs_H$term = rownames(word_freqs_H)
word_freqs_H = word_freqs_H[,c(2,1)]
colnames(word_freqs_H)=c("term","num")

# Create wordclouds
wordcloud::wordcloud(word_freqs_C$term, word_freqs_C$num, max.words=100, colors="red")
wordcloud::wordcloud(word_freqs_T$term, word_freqs_T$num, max.words=100, colors="blue")
wordcloud::wordcloud(word_freqs_H$term, word_freqs_H$num, max.words=100, colors="black")
```

comedies tragedies histories

To create **commonality clouds** and **comparison clouds**, we first create a list of all (cleaned) words in the comedies, tragedies, and histories, from the corpora `clean_C`, `clean_T`, and `clean_H`.

```
all_c = paste(clean_C[[1]][1],clean_C[[2]][1],clean_C[[3]][1],clean_C[[4]][1],
              clean_C[[5]][1],clean_C[[6]][1],clean_C[[7]][1],clean_C[[8]][1],
              clean_C[[9]][1],clean_C[[10]][1],clean_C[[11]][1],clean_C[[12]][1],
              clean_C[[13]][1],clean_C[[14]][1],clean_C[[15]][1],clean_C[[16]][1],
              clean_C[[17]][1],collapse=" ")
all_t = paste(clean_T[[1]][1],clean_T[[2]][1],clean_T[[3]][1],clean_T[[4]][1],
              clean_T[[5]][1],clean_T[[6]][1],clean_T[[7]][1],clean_T[[8]][1],
              clean_T[[9]][1],clean_T[[10]][1],collapse=" ")
all_h = paste(clean_H[[1]][1],clean_H[[2]][1],clean_H[[3]][1],clean_H[[4]][1],
              clean_H[[5]][1],clean_H[[6]][1],clean_H[[7]][1],clean_H[[8]][1],
              clean_H[[9]][1],clean_H[[10]][1],collapse=" ")
```

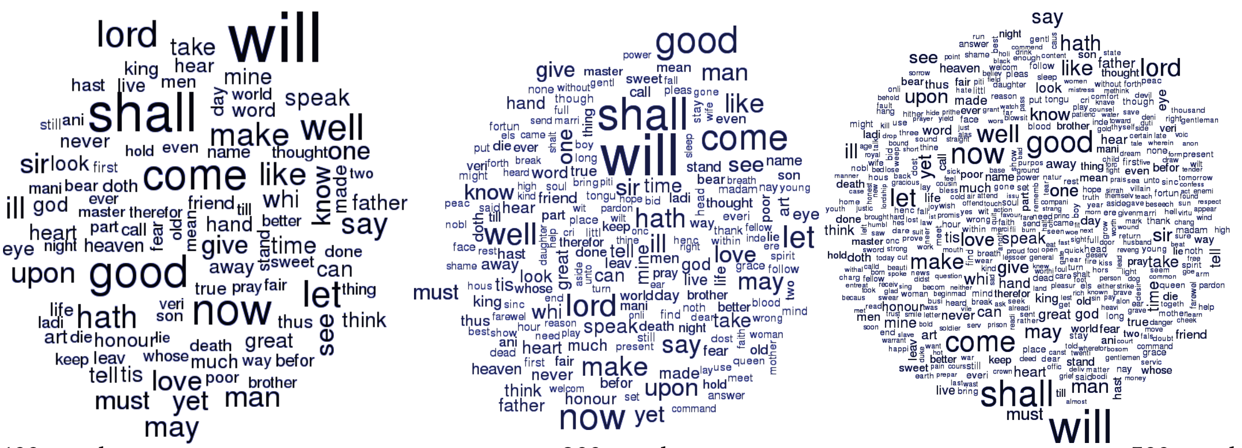
We join the terms as strings and put them into a single corpus.

```
ws_corpus = tm::VCorpus(tm::VectorSource(c(all_c,all_t,all_h))
tm::inspect(ws_corpus)
```

```
<<VCorpus>>
Metadata:  corpus specific: 0, document level (indexed): 0
Content:  documents: 3

[[1]]
<<PlainTextDocument>>
Metadata:  7
Content:  chars: 1178027

[[2]]
<<PlainTextDocument>>
Metadata:  7
Content:  chars: 796779

[[3]]
<<PlainTextDocument>>
Metadata:  7
Content:  chars: 841554
```

Now we create a TDM for this corpus, which we cast as a matrix object before printing the commonality clouds (words shared across the three corpora), with 100, 200, and 500 words.

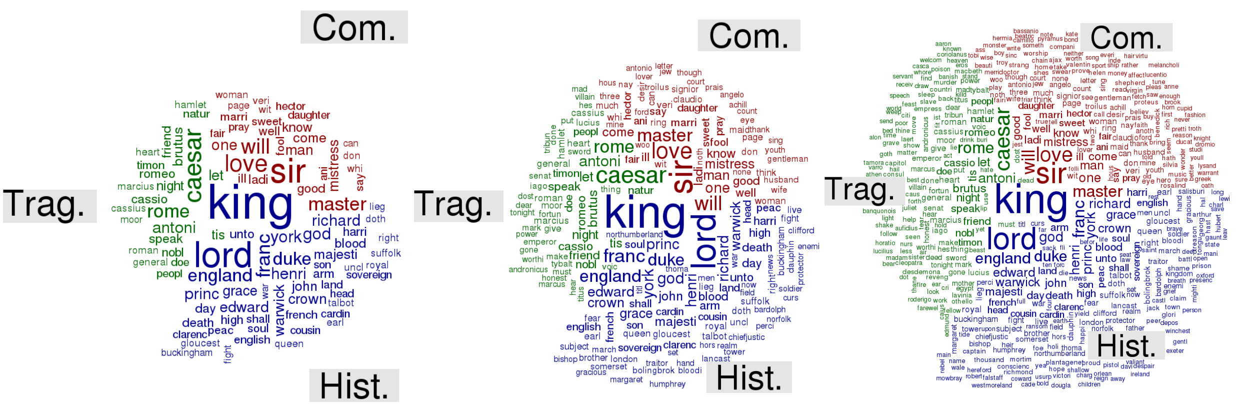
```
ws_tdm = tm::TermDocumentMatrix(ws_corpus)
colnames(ws_tdm) = c("Com.","Trag.","Hist.")
ws_m = as.matrix(ws_tdm)

wordcloud::commonality.cloud(ws_m, colors = "darkblue", max.words = 100)
wordcloud::commonality.cloud(ws_m, colors = "darkblue", max.words = 200)
wordcloud::commonality.cloud(ws_m, colors = "darkblue", max.words = 500)
```



100 words 200 words 500 words

Comparison clouds can be produced using the same syntax.

```
comp.colours = c("darkred","darkgreen","darkblue")
wordcloud::comparison.cloud(ws_m, colors = comp.colours, max.words = 100)
wordcloud::comparison.cloud(ws_m, colors = comp.colours, max.words = 200)
wordcloud::comparison.cloud(ws_m, colors = comp.colours, max.words = 500)
```



100 words 200 words 500 words

Pyramid Plots

We can also produce **pyramid plots** by first finding the terms that are common to any two corpora.

```
common_words_CT = subset(ws_m, ws_m[,1] > 0 & ws_m[,2] > 0)
dim(common_words_CT)
head(common_words_CT)
common_words_CH = subset(ws_m, ws_m[,1] > 0 & ws_m[,3] > 0)
dim(common_words_CH)
head(common_words_CH)
common_words_TH = subset(ws_m, ws_m[,2] > 0 & ws_m[,3] > 0)
dim(common_words_TH)
head(common_words_TH)
```

```
# common_words_CT
[1] 6438    3
         Docs
Terms      Com. Trag. Hist.
  abandon     2     3     0
  abat        5     5     4
  abate       3     2     3
  abbey       7     1     4
  abe         4     5     1
  abhor      14    16     3
```

```
# common_words_CH
[1] 6325    3
           Docs
Terms        Com. Trag. Hist.
  abandond     5     0     1
  abat         5     5     4
  abate        3     2     3
  abbey        7     1     4
  abe          4     5     1
  abet         1     0     1
```

```
# common_words_TH
[1] 5702    3
          Docs
Terms      Com. Trag. Hist.
  abat        5     5     4
  abate       3     2     3
  abbey       7     1     4
  abe         4     5     1
  abhor      14    16     3
  abhorrd     5     4     1
```

The differences in the number of times each token is used in each corpora can be computed as follows.

```
difference_CT = abs(common_words_CT[,1] - common_words_CT[,2])
difference_CH = abs(common_words_CH[,1] - common_words_CH[,3])
difference_TH = abs(common_words_TH[,2] - common_words_TH[,3])
```

Next, we bind these new counts to the respective `common_word` corpora, and order them along the differences.

```
common_words_CT = cbind(common_words_CT,difference_CT)
common_words_CT = common_words_CT[order(common_words_CT[,4],decreasing=TRUE),]
common_words_CH = cbind(common_words_CH,difference_CH)
common_words_CH = common_words_CH[order(common_words_CH[,4],decreasing=TRUE),]
common_words_TH = cbind(common_words_TH,difference_TH)
common_words_TH = common_words_TH[order(common_words_TH[,4],decreasing=TRUE),]
```

If we want to plot the top $n = 30$ words whose usage was the most different in each pair of corpora, we proceed as follows.

```
n=30
top_df_CT = data.frame(x = common_words_CT[1:n,1], y = common_words_CT[1:n,2],
                       labels = rownames(common_words_CT[1:n,]))
top_df_CH = data.frame(x = common_words_CH[1:n,1], y = common_words_CH[1:n,3],
                       labels=rownames(common_words_CH[1:n,]))
top_df_TH = data.frame(x = common_words_TH[1:n,2], y = common_words_TH[1:n,3], l
                       abels=rownames(common_words_TH[1:n,]))

top_df_CT; top_df_CH; top_df_TH
```

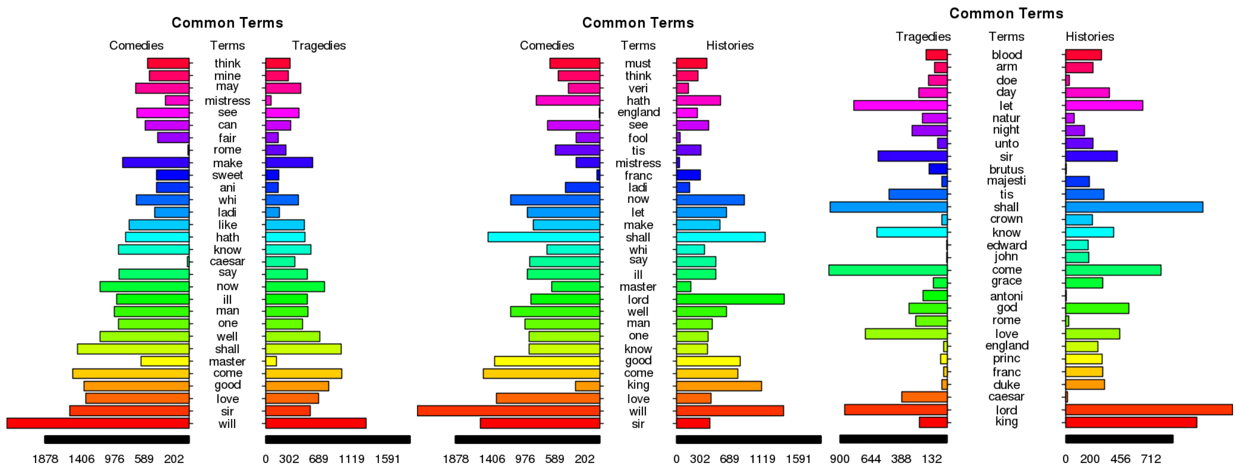
```
# top_df_CT
    x    y   labels
2378 1300     will
1554  581      sir
1343  688     love
1369  818     good
1518  984     come
1454  964    shall
 622  142   master
1159  700     well
 921  476      one
 973  552      man
 940  534      ill
1161  759      now
 909  540      say
  26  380   caesar
 922  587     know
 830  499     hath
 777  496     like
 688  413      whi
 447  180     ladi
 424  164      ani
 865  608     make
 422  170    sweet
 407  158     fair
  13  262     rome
 569  321      can
 676  429      see
 696  452      may
 309   67 mistress
 519  294     mine
 537  316    think
```

```
# top_df_CH
    x    y   labels
1554  434      sir
2378 1394     will
1343  451     love
 317 1103     king
1518  798     come
1369  829     good
 922  404     know
 921  407      one
 973  460      man
1159  650     well
 900 1400     lord
 622  186   master
 940  513      ill
 909  507      say
 688  359      whi
1454 1154    shall
 865  567     make
 940  649      let
1161  880      now
 447  171     ladi
  36  310    franc
 309   37 mistress
 583  319      tis
 306   44     fool
 676  414      see
   9  269  england
 830  570     hath
 412  152     veri
 537  279    think
 652  395     must
```

```
# top_df_TH
   x    y  labels
 231 1103    king
 861 1400    lord
 380   14  caesar
  42  327    duke
  27  310   franc
  54  305   princ
  31  269 england
 688  451    love
 262   25    rome
 318  527     god
 200    1  antoni
 112  309   grace
   4  194    john
 964 1154   shall
 984  798    come
   1  187  edward
 587  404    know
  43  223   crown
 483  319     tis
  43  196 majesti
 152    2  brutus
  77  226    unto
 581  434     sir
 293  154   night
 206   70   natur
 236  368     day
 157   28     doe
 777  649     let
 104  230     arm
 176  298   blood
```

Finally, we produce the pyramid plots themselves for the common terms that had the largest difference in usage for each pair of copora.

```
plotrix::pyramid.plot(top_df_CT$x,top_df_CT$y,labels=top_df_CT$labels,
             gap=500,top.labels=c("Comedies", "Terms", "Tragedies"), main="Common Terms",
             laxlab=NULL, raxlab=NULL, unit=NULL)

plotrix::pyramid.plot(top_df_CH$x,top_df_CH$y,labels=top_df_CH$labels,
             gap=500,top.labels=c("Comedies", "Terms", "Histories"), main="Common Terms",
             laxlab=NULL, raxlab=NULL, unit=NULL)

plotrix::pyramid.plot(top_df_TH$x,top_df_TH$y,labels=top_df_TH$labels,
             gap=500,top.labels=c("Tragedies", "Terms", "Histories"), main="Common Terms",
             laxlab=NULL, raxlab=NULL, unit=NULL)
```



27.4.4 Ham or Spam

In this example, we are going to use a classical SMS dataset where texts have been **classified** as ham/spam in order to build a model which can predict whether an incoming SMS is spam or ham based on its content.

Initializing the Environment

We will use the following `R` libraries:

- `tm` for text mining functions;
- `qdap` for some text processing functions;
- `e1071` for the naive Bayes and support vector machines methods;
- `dplyr` for tidyverse processing;
- `tidytext` for tidyverse analysis;
- `ggplot2` for tidyverse plotting, and
- `psych` for regular plotting.

Importing and Exploring the Data

We load the dataset as usual.

```
ham.spam <- read.csv("SMSSpamCollection.csv", sep=",")
ham.spam$Msg <- as.character(ham.spam$Msg)
```

The dataset consists of 5574 observations and 3 variables/features: SMS messages, length of the messages, and whether they are ham (+) or spam (−).

```
str(ham.spam)
```

```
'data.frame':  5574 obs. of  3 variables:
 $ SpamOrHam: Factor w/ 2 levels "ham","spam": 1 1 2 1 1 2 1 1 2 2 ...
 $ Msg      : chr  "Go until jurong point, crazy.. Available only in bugis n great world la ..." ...
 $ length   : int  111 29 155 49 61 148 77 160 158 154 ...
```

We print a few SMS (one spam and one ham) to get a better idea as to the contents of the text.

```
ham.spam$Msg[12]
ham.spam$Msg[4444]
```

```
[1] "SIX chances to win CASH! From 100 to 20,000 pounds txt>
     CSH11 and send to 87575. Cost 150p/day, 6days, 16+
     TsandCs apply Reply HL 4 info"
[1] "Dear i am not denying your words please"
```

No human would send the first SMS to another person – we all recognize it as spam; the second one seems more legitimate – we have all had conversations of this nature with our partners. The SMS labels confirms the suspicion.

```
ham.spam$SpamOrHam[12]
ham.spam$SpamOrHam[4444]
```

```
[1] "spam"
[1] "ham"
```

What is the distribution of ham/spam messages in the dataset?

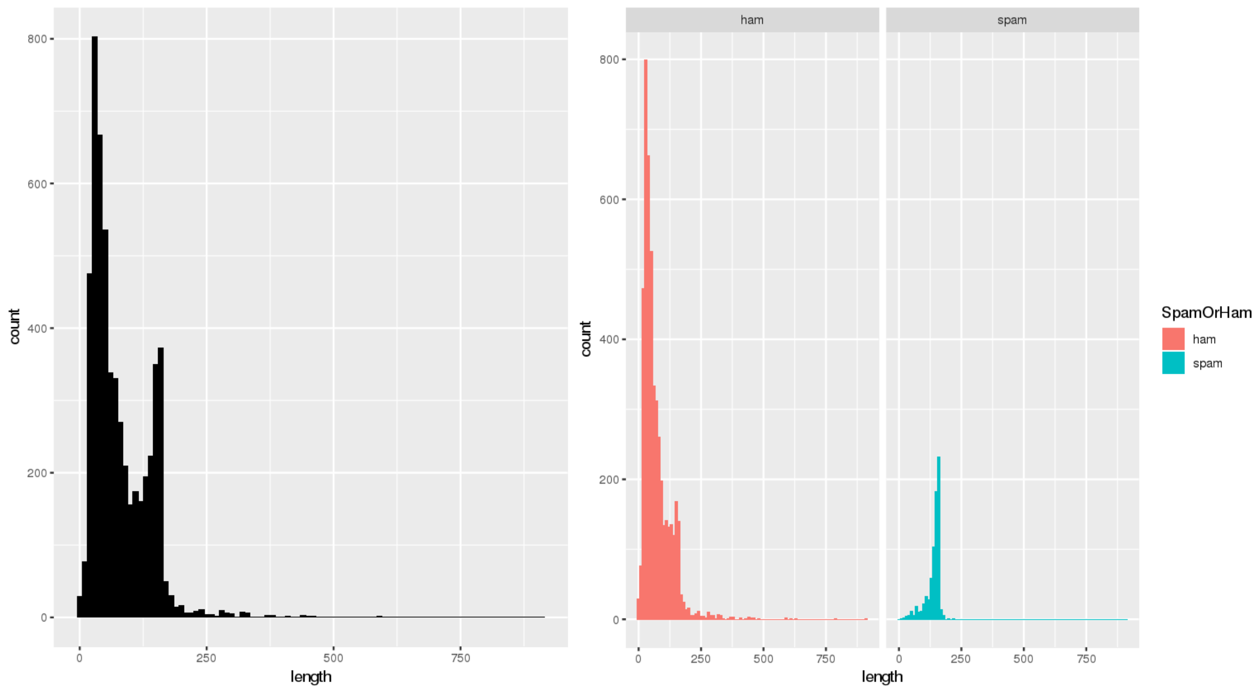
```
table(ham.spam$SpamOrHam)
prop.table(table(ham.spam$SpamOrHam))
```

```
 ham spam
4827  747

      ham      spam
0.8659849 0.1340151
```

Before we dive right into text analysis, can we say anything about the SMS categories just by looking at the lengths of the SMS? Are the examples above representative of spam and ham messages?

Should we expect spam messages to be longer than ham messages, in general? The distribution of message lengths is **bimodal** – does that mean anything?

```
library(ggplot2)
ggplot(ham.spam, aes(length)) +
       geom_histogram(binwidth=10)
ggplot(ham.spam, aes(length, fill = SpamOrHam)) +
       geom_histogram(binwidth=10) +
       facet_wrap(~SpamOrHam)
```



There are more ham messages than spam messages in the dataset, obviously but the height of the distributions is not as important as the shape: in the absence of more information, if the SMS has 150 characters or more, say, it would not be the most unreasonable thing in the world to suspect that it could be spam... but the contents of the message also have to count for something, right?

Creating the Corpus

In order to do text classification, we first need to prepare a document-term matrix. The `tm` functionality from the previous examples does the trick.

```
(SMS.corpus <- tm::VCorpus(tm::VectorSource(ham.spam$Msg)))
```

```
<<VCorpus>>
Metadata:  corpus specific: 0, document level (indexed): 0
Content:  documents: 5574
```

We take a quick peek at the first 4 entries to make sure that everything is as it should be.

```
sapply(SMS.corpus[1:4], function(x){x$content})
```

```
[1] Go until jurong point, crazy.. Available only in bugis n great world la e buffet...
    Cine there got amore wat...
[2] Ok lar... Joking wif u oni...
[3] Free entry in 2 a wkly comp to win FA Cup final tkts 21st May 2005. Text FA to 87121
    to receive entry question(std txt rate)T&C's apply 08452810075over18's
[4] U dun say so early hor... U c already then say...
```

Cleaning the Data

As is **almost always** the case, the next step is to clean the corpus. Note that the order in which the cleaning steps are performed may affect the final form of the cleaned corpus.

```
clean_corpus <- function(corpus){
  corpus <- tm::tm_map(corpus, tm::content_transformer(qdap::replace_abbreviation))
  corpus <- tm::tm_map(corpus, tm::removePunctuation)
  corpus <- tm::tm_map(corpus, tm::removeNumbers)
  corpus <- tm::tm_map(corpus, tm::stemDocument)
  corpus <- tm::tm_map(corpus, tm::content_transformer(tolower))
  corpus <- tm::tm_map(corpus, tm::stripWhitespace)
  corpus <- tm::tm_map(corpus, tm::removeWords, c(tm::stopwords("en")))
  return(corpus)
}

SMS.corpus.clean <- clean_corpus(SMS.corpus)
sapply(SMS.corpus.clean[1:4], function(x){x$content})
```

```
[1] go  jurong point crazi availabl onli  bugi n great world la e buffet cine  got amor wat
[2] ok lar joke wif u oni
[3] free entri   wkli comp  win fa cup final tkts st may text fa   receiv entri questionstd
    txt ratetc appli
[4] u dun say  earli hor u c alreadi  say
```

In the case of spam detection, there could be reasons why we might not want to be too drastic at the cleaning stage: removing the numerals in `08452810075over18's` in the 3rd SMS, for instance, removes an important indicator of spam.

Creating the DTM

We want to classify **documents**, so we need a DTM (and not a TDM) representation of the text dataset on which to apply classifiers.

```
SMS.DTM <- tm::DocumentTermMatrix(SMS.corpus.clean)
tm::inspect(SMS.DTM[1:15,20:30])
```

```
<<DocumentTermMatrix (documents: 20, terms: 11)>>
Non-/sparse entries: 17/203
Sparsity           : 92%
Maximal term length: 11
Weighting          : term frequency (tf)

Sample             :
    Terms
Docs aaniy aaooooright aathilov aathiwher abbey abdomen abeg abelu aberdeen abi abil
  1      0           0        0         0     0       0    0     0        0   0    0
  2      0           0        0         0     0       0    0     0        0   0    0
  3      0           0        0         0     0       0    0     0        0   0    0
  4      0           0        0         0     0       0    0     0        0   0    0
```

```
5       0          0        0         0     0       0    0     0        0   0    0
6       0          0        0         0     0       0    0     0        0   0    0
7       0          0        0         0     0       0    0     0        0   0    0
8       0          0        0         0     0       0    0     0        0   0    0
9       0          0        0         0     0       0    0     0        0   0    0
10      0          0        0         0     0       0    0     0        0   0    0
11      0          0        0         0     0       0    0     0        0   0    0
12      0          0        0         0     0       0    0     0        0   0    0
13      0          0        0         0     0       0    0     0        0   0    0
14      0          0        0         0     0       0    0     0        0   0    0
15      0          0        0         0     0       0    0     0        0   0    0
```

Text Visualization

Before we start with classification proper, let us visualize the frequent terms of both the spam and the ham classes.

```
spam.cloud <- which(ham.spam$SpamOrHam == "spam")
ham.cloud <- which(ham.spam$SpamOrHam == "ham")
wordcloud::wordcloud(SMS.corpus.clean[ham.cloud],
                     min.freq=50,colors="steelblue")
wordcloud::wordcloud(SMS.corpus.clean[spam.cloud],
                     min.freq=50,colors="darkred")
```

ham SMS

spam SMS

We can also identify document-specific high-information terms using the **tf-idf** weighting.

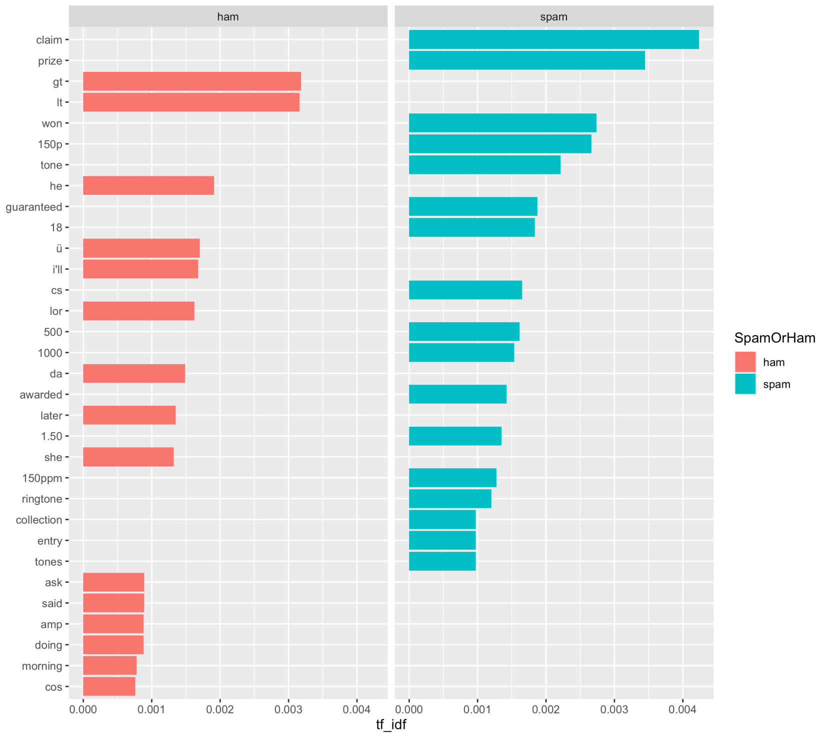
```
SMS_words <- ham.spam |>
  tidytext::unnest_tokens(word,Msg) |>
  dplyr::count(SpamOrHam,word, sort=TRUE) |>
  dplyr::ungroup()
```

```
SMS_distinctive <- SMS_words |>
  tidytext::bind_tf_idf(word,SpamOrHam,n) |>
  dplyr::arrange(desc(tf_idf)) |>
  dplyr::mutate(word = factor(word,
                levels = rev(unique(word)))) |>
  dplyr::group_by(SpamOrHam) |>
  dplyr::top_n(15,tf_idf) |> dplyr::ungroup()

head(SMS_distinctive)
```

```
# A tibble: 6 × 6
  SpamOrHam word      n      tf   idf  tf_idf
  <chr>     <fct> <int>   <dbl> <dbl>   <dbl>
1 spam      claim   113 0.00611 0.693 0.00424
2 spam      prize    92 0.00498 0.693 0.00345
3 ham       gt      318 0.00459 0.693 0.00318
4 ham       lt      316 0.00456 0.693 0.00316
5 spam      won      73 0.00395 0.693 0.00274
6 spam      150p     71 0.00384 0.693 0.00266
```

A simple visual identifies terms that are specific to spam/ham SMS.

```
library(ggplot2)
ggplot(SMS_distinctive, aes(word,tf_idf, fill = SpamOrHam)) +
  geom_col(show.legend=TRUE) + labs(x=NULL, y="tf_idf") +
  facet_wrap(~SpamOrHam) + coord_flip()
```



Training/Testing Data

Classifiers need to be trained on a subset of the data and tested/evaluated on the complement to avoid overfitting the model. There's no steadfast rule, but a 70%/30% split is often applied.[34]

34: When attempting to replicate what follows, remember that we did not set a seed, and so that your results could be somewhat different.

```
ind = sample(1:nrow(ham.spam), size=0.7*nrow(ham.spam))
spam.train = subset(ham.spam[ind,], SpamOrHam == "spam")
ham.train = subset(ham.spam[ind,], SpamOrHam == "ham")
ham.spam.train.labels <- ham.spam[ind,]$SpamOrHam
ham.spam.test.labels <- ham.spam[-ind,]$SpamOrHam
```

We can verify that the training/testing sets are representative of the full dataset (in terms of the target labels, at least).

```
prop.table(table(ham.spam.train.labels))
prop.table(table(ham.spam.test.labels))
prop.table(table(ham.spam$SpamOrHam))
```

```
ham.spam.train.labels
      ham      spam
0.8664445 0.1335555

ham.spam.test.labels
      ham      spam
0.8649133 0.1350867

      ham      spam
0.8659849 0.1340151
```

Next, we convert the training/testing messages to DTM and corpora.

```
SMS.DTM.train <- SMS.DTM[ind,]
SMS.DTM.test <- SMS.DTM[-ind,]
SMS.corpus.clean.train <- SMS.corpus.clean[ind]
SMS.corpus.clean.test <- SMS.corpus.clean[-ind]
```

We also need to select the **features** (in this case, the terms) to include in the model – otherwise, there would be too much information to consider and the **curse of dimensionality** rears is ugly head (see Chapter 23). We can use the list of all terms that appear at least 10 times, say, in the training messages, for instance.[35]

35: Another approach could be to remove **sparse terms**.

```
Freq.Terms <- tm::findFreqTerms(SMS.DTM.train,10)
SMS.DTM.Freq.Terms.train <- SMS.DTM.train[,Freq.Terms]
SMS.DTM.Freq.Terms.test <- SMS.DTM.test[,Freq.Terms]
length(Freq.Terms)
Freq.Terms[1:100]
```

```
[1] 614

[1] 'crazi' 'got' 'great' 'onli' 'point' 'wat' 'world' 'joke' 'lar' 'wif' 'appli' 'entri'
[13] 'final' 'free' 'may' 'receiv' 'text' 'txt' 'win' 'wkli' 'alreadi' 'dun' 'earli' 'say'
[25] 'around' 'dont' 'goe' 'live' 'think' 'though' 'back' 'freemsg' 'fun' 'hey' 'like'
[36] 'now' 'send' 'still' 'week' 'word' 'xxx' 'brother' 'even' 'speak' 'treat' 'caller'
[47] 'copi' 'friend' ...
```

We also need to **categorize** the features so that the data can eventually be fed into a naïve Bayes classifier, say; the absence of a frequent term in a SMS message is denoted by "No", while it's presence is denoted by "Yes".

```
yes.no <- function(x){
    y <- ifelse(x>0,1,0)
    y <- factor(y,levels=c(0,1),labels=c("No","Yes"))
    return(y)
}
```

The (reduced) training/testing sets thus look like:

```
SMS.train <- apply(SMS.DTM.Freq.Terms.train,2,yes.no)
SMS.test <- apply(SMS.DTM.Freq.Terms.test,2,yes.no)
head(SMS.train)
head(SMS.test)
```

```
# SMS.train
     crazi  got great onli point   wat ... auction lei
3151    No   No    No   No    No    No ...      No  No
1572    No  Yes    No   No    No    No ...      No  No
3200    No   No    No   No    No   Yes ...      No  No
1805    No   No    No   No    No    No ...      No  No
1201    No   No    No   No    No    No ...      No  No
534     No   No    No   No    No    No ...      No  No

# SMS.text
     crazi  got great onli point   wat ... auction lei
3       No   No    No   No    No    No ...      No  No
7       No   No    No   No    No    No ...      No  No
9       No   No   Yes   No    No    No ...      No  No
11      No   No    No   No    No    No ...      No  No
13      No   No    No   No    No    No ...      No  No
14      No   No    No   No    No    No ...      No  No
```

Naive Bayes Classifier

We now apply the `naiveBayes()` function from the `R` library `e1071`.[36] Notice the syntax: we apply `naiveBayes` to the training data `SMS.train` and the target variable is the ham/spam label in the training subset (`ham.spam.train.labels`). The `laplace=1` option instructs `naiveBayes()` to look "a little bit harder" into the data, while the `CV=10` option selects 10 cross-validations replicates.

36: See Section 21.4.4 for details.

```
SMS.classifier.NB <- e1071::naiveBayes(SMS.train,ham.spam.train.labels,laplace=1,CV=10)
summary(SMS.classifier.NB)
attributes(SMS.classifier.NB)
```

```
       Length Class  Mode
apriori     2  table  numeric
tables    614  -none- list
levels      2  -none- character
isnumeric 614  -none- logical
call        5  -none- call

$names
    'apriori' 'tables' 'levels' 'isnumeric' 'call'
$class
    'naiveBayes'
```

We can now feed the testing data (`SMS.test`) into the model (`SMS.classifier.NB`) with the help of the `predict()` function.

```
SMS.test.pred.NB <- predict(SMS.classifier.NB,
                            newdata = SMS.test)
table(SMS.test.pred.NB,ham.spam.test.labels)
prop.table(table(SMS.test.pred.NB,ham.spam.test.labels))
```

```
                ham.spam.test.labels
SMS.test.pred.NB  ham spam
            ham  1444   26
            spam    3  200

                ham.spam.test.labels
SMS.test.pred.NB         ham        spam
            ham  0.863120143 0.015540944
            spam 0.001793186 0.119545726
```

These **confusion matrices** are not bad at all! Is there a difference in the representations of mislabeled and sucessfully labeled test SMS?

```
Missed=SMS.test[which(SMS.test.pred.NB!=ham.spam.test.labels),]
Succesful=SMS.test[which(SMS.test.pred.NB==ham.spam.test.labels),]
table(Missed)
table(Succesful)
```

```
Missed
   No   Yes
17653   153

Succesful
     No     Yes
1000706    8710
```

The ratios are basically the same in both instances. We can also take a look at the original text of a few mislabeled and successfully labeled test SMS.

```
head(ham.spam[as.numeric(rownames(Missed)),]$Msg)
head(ham.spam[as.numeric(rownames(Succesful)),]$Msg)
```

```
# some mispredicted SMS
[1] 'England v Macedonia - dont miss the goals/team news. Txt ur national team to 87077 eg
     ENGLAND to 87077 Try:WALES, SCOTLAND 4txt/ú1.20 POBOXox36504W45WQ 16+'
[2] 'U 447801259231 have a secret admirer who is looking 2 make contact with U-find out who
     they R*reveal who thinks UR so special-call on 09058094597'
[3] 'SMS. ac Blind Date 4U!: Rodds1 is 21/m from Aberdeen, United Kingdom. Check Him out
     http://img. sms. ac/W/icmb3cktz8r7!-4 no Blind Dates send HIDE'
[4] 'XCLUSIVE@CLUBSAISAI 2MOROW 28/5 SOIREE SPECIALE ZOUK WITH NICHOLS FROM PARIS.FREE ROSES
     2 ALL LADIES !!! info: 07946746291/07880867867 '
[5] 'Its a valentine game. . . Send dis msg to all ur friends. .. If 5 answers r d same then
     someone really loves u. Ques- which colour suits me the best?rply me'
[6] 'Hi I\'m sue. I am 20 years old and work as a lapdancer. I love sex. Text me live - I\'m
     i my bedroom now. text SUE to 89555. By TextOperator G2 1DA 150ppmsg 18+'

# some correctly predicted SMS
[1] 'Free entry in 2 a wkly comp to win FA Cup final tkts 21st May 2005. Text FA to 87121 to
     receive entry question(std txt rate)T&C\'s apply 08452810075over18\'s'
[2] 'Even my brother is not like to speak with me. They treat me like aids patent.'
[3] 'WINNER!! As a valued network customer you have been selected to receivea £900 prize reward!
     To claim call 09061701461. Claim code KL341. Valid 12 hours only.'
[4] 'I\'m gonna be home soon and i don\'t want to talk about this stuff anymore tonight, k? I\'ve
     cried enough today.'
[5] 'URGENT! You have won a 1 week FREE membership in our £100,000 Prize Jackpot! Txt the word:
     CLAIM to No: 81010 T&C www.dbuk.net LCCLTD POBOX 4403LDNW1A7RW18'
[6] 'I\'ve been searching for the right words to thank you for this breather. I promise i wont
     take your help for granted and will fulfil my promise. You have been wonderful and a blessing
     at all times.'
```

Support Vector Machines

We can use other classifiers – `e1071` also implements support vector machines.[37] The principle is the same, although we do not need to first categorize the features (as we had to do for naïve Bayes.

37: See Section for 21.4.2 for details.

The `cost` parameter is related to the price of allowing **misclassifications**: higher values attempt to minimize such misclassifications, but the resulting model tends to lead to overfitting (a little flexibility in re: misclassfications is a good thing in the long run).

There is some evidence to suggest that the choice of the `"linear"` kernel is preferable when dealing with text data (as opposed to gaussian for some numerical data), but other kernels can also be used.

The steps proceeds as in naïve Bayes (although we demonstrate the `classAgreement` functionality for evaluation the classifier). You can get details on the method by querying the help files.

```
SMS.classifier.SVM <- e1071::svm(as.factor(ham.spam.train.labels) ~.,
                                 data=as.data.frame(as.matrix(SMS.DTM.Freq.Terms.train)),
                                 type="C-classification", cost=10, kernel="linear")
```

```
summary(SMS.classifier.SVM)
attributes(SMS.classifier.SVM)
```

```
Call:
svm(formula = ham.spam.train.labels ~ ., data = as.data.frame(as.matrix(SMS.DTM.Freq.Terms.train)),
    type = "C-classification", cost = 10, kernel = "linear")

Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  linear
       cost:  10
      gamma:  0.001628664

Number of Support Vectors:  424

 ( 299 125 )

Number of Classes:  2


Levels:
 ham spam

$names
    'call' 'type' 'kernel' 'cost' 'degree' 'gamma' 'coef0' 'nu' 'epsilon' 'sparse' 'scaled'
    'x.scale' 'y.scale' 'nclasses' 'levels' 'tot.nSV' 'nSV' 'labels' 'SV' 'index' 'rho' 'compprob'
    'probA' 'probB' 'sigma' 'coefs' 'na.action' 'fitted' 'decision.values' 'terms'

$class
    'svm.formula' 'svm'
```

We can now feed the testing data into the model with the help of the `predict()` function.

```
SMS.test.pred.SVM <- predict(SMS.classifier.SVM,as.data.frame(as.matrix(SMS.DTM.Freq.Terms.test)))
summary(SMS.test.pred.SVM)
```

```
 ham spam
1430  243
```

Of course, it may not be sufficient to know how many SMS are predicted to be ham and/or spam – we might want to know if individual SMS are correctly predicted.

```
(confusion.matrix = table(pred = SMS.test.pred.SVM, true = ham.spam.test.labels))
e1071::classAgreement(confusion.matrix,match.names=TRUE)
```

```
      true
pred    ham spam
  ham  1401   29
  spam   46  197
```

```
$diag
[1] 0.95517

$kappa
[1] 0.81406

$rand
[1] 0.91431

$crand
[1] 0.76592
```

More information on this last function can be obtained by typing in `?e1071::classAgreement` at the prompt.

SVMs with PCA

Finally, we re-visit the SVM model by first reducing the DTM data to its first 4 **principal components** in order to try to mitigate the effects of the curse of dimensionality and to introduce a more complete classification workflow:[38]

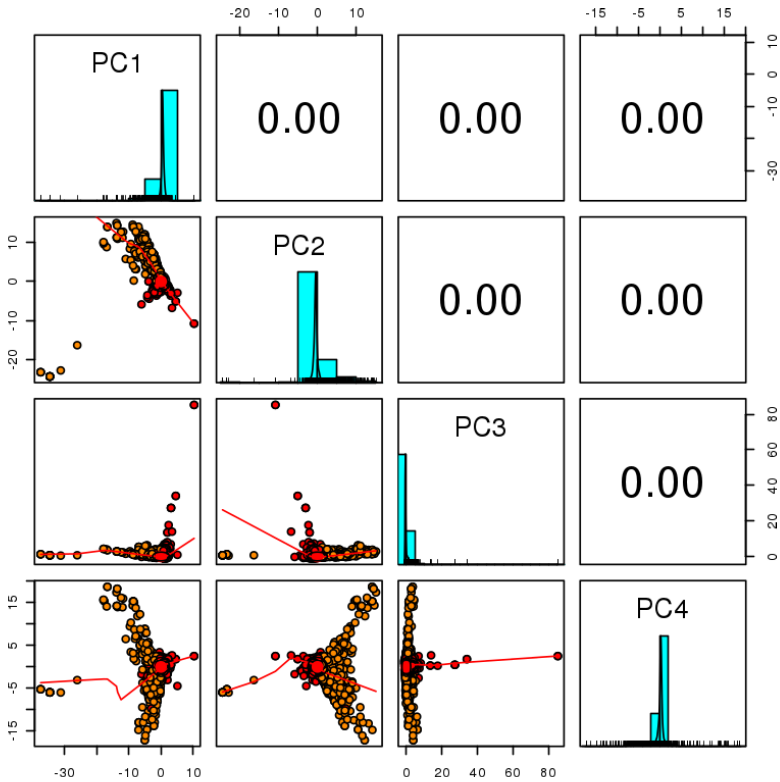
38: See Chapter 23 for more details.

1. compute the principal composition of the training data;
2. set-up the formulas for going back and forth between the original data and the rotated (reduced) data;
3. plot the classes against the first 4 principal components (arbitrary);
4. express the test data in the training PCA universe;
5. tune the SVM model (run some preliminary code to determine the optimal choice of model parameters);
6. train the SVM model;
7. fit the SVM model to testing data, and
8. evaluate the model.

```
# 1: find the PCs of the training data and only keep the first 4, say
SMS.train.pca <- prcomp(as.data.frame(as.matrix(SMS.DTM.Freq.Terms.train)),
       center = TRUE, scale = TRUE)
SMS.train.pca.reduced = SMS.train.pca$x
SMS.train.pca.reduced[,5:ncol(SMS.train.pca.reduced)] = 0   # setting PCs 5-end to 0
```

```
# 2: going back and forth between the original data and the rotated data
SMS.train.recover.full = exp(t(t(SMS.train.pca$x %*% t(SMS.train.pca$rotation)) *
                         SMS.train.pca$scale + SMS.train.pca$center))
SMS.train.recover.pca = exp(t(t(SMS.train.pca.reduced %*% t(SMS.train.pca$rotation)) *
                         SMS.train.pca$scale + SMS.train.pca$center))

# put the data in a data frame
SMS.train.pca.reduced.model = SMS.train.pca.reduced
SMS.train.pca.reduced.model = data.frame(SMS.train.pca.reduced, ham.spam.train.labels)

training.pca = SMS.train.pca.reduced.model[,c(1,2,3,4)]
```

```
# 3: plot the classes in the training data set against the first 4 PCs
psych::pairs.panels(SMS.train.pca$x[,1:4], pch=21,
                    bg=rainbow(11)[unclass(ham.spam.train.labels)])
```




```
# 4: put test data in the PCA context
SMS.test.pca <- scale(as.data.frame(as.matrix(SMS.DTM.Freq.Terms.test)),
      SMS.train.pca$center, SMS.train.pca$scale) %*% SMS.train.pca$rotation

SMS.test.pca.reduced <- SMS.test.pca
SMS.test.pca.reduced[,5:ncol(SMS.train.pca.reduced)] = 0

SMS.test.pca.reduced.model = data.frame(SMS.test.pca.reduced)
```

```
# 5: tune SVM model to find a good choice for the cost parameter C
tuning.pca <- e1071::tune(e1071::svm, train.x=training.pca,
                          train.y=as.factor(ham.spam.train.labels), kernel="linear",
                          ranges=list(cost=10^(-3:3)))
print(tuning.pca)  # optimal parameters
```

```
Parameter tuning of ‘svm’:

- sampling method: 10-fold cross validation

- best parameters:
 cost
   10

- best performance: 0.0302492
```

```
# 6: train the SVM model
SMS.classifier.SVM.pca <- e1071::svm(as.factor(ham.spam.train.labels) ~., data=training.pca,
                                     type="C-classification", cost=10, kernel="linear")
summary(SMS.classifier.SVM.pca)
```

```
Call: svm(formula = ham.spam.train.labels ~ ., data = training.pca, type = "C-classification",
    cost = 10, kernel = "linear")
```

```
Parameters:
   SVM-Type:  C-classification
 SVM-Kernel:  linear
       cost:  10
      gamma:  0.25

Number of Support Vectors:  316

 ( 158 158 )

Number of Classes:  2

Levels:
 ham spam
```

```
# 7: fit SVM to test data
SMS.test.pred.SVM.pca <- predict(SMS.classifier.SVM.pca,SMS.test.pca.reduced)
summary(SMS.test.pred.SVM.pca)   # provide a summary of the predicted values
```

```
 ham spam
1430  243
```

```
# 8: evaluate the fitted model
(confusion.matrix.pca = table(pred = SMS.test.pred.SVM.pca, true = ham.spam.test.labels))
e1071::classAgreement(confusion.matrix.pca,match.names=TRUE)
```

```
      true
pred    ham spam
  ham  1438   40
  spam    9  186

$diag
[1] 0.97071

$kappa
[1] 0.86696

$rand
[1] 0.94310

$crand
[1] 0.83437
```

How does the model on the reduced data compare to the original SVM model?

27.4.5 NHL Game Recaps (Reprise)

In this section, we revisit the NHL game recaps of Section 27.4.1, this time to **cluster** the terms in the documents. Clustering is an unsupervised learning technique that can be used to determine which entities or objects, typically represented by rows in the data, are similar to each other. For text mining, we can use clustering to determine which documents are similar to one another (based on their term signature), for instance.[39]

For the purpose of this example, we will use **hierarchical clustering**, but any other clustering approach would be just as acceptable.[40]

39: We will discuss **topic modeling**, another unsupervised learning application to text data, in Chapter 32.

40: See Chapter 22 for details.

Review of Hierarchical Clustering in R

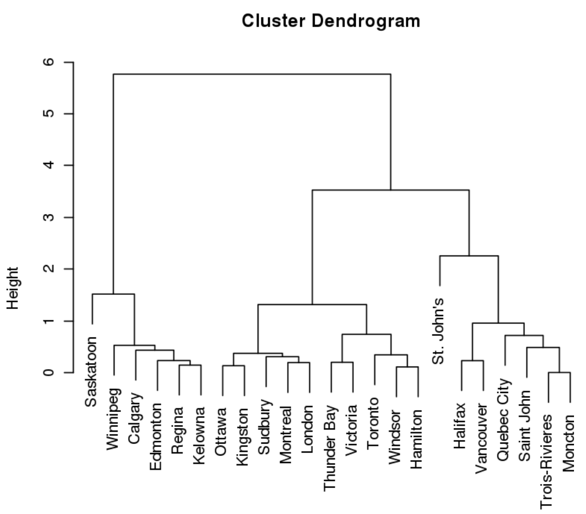
Let's start with a simple example of hierarchical clustering using numeric data. The three lines of data below represent days of rainfall in a number of Canadian cities. Can we use hierarchical clustering to get a sense of which cities are similar to each other?

```
city = c("Montreal","Ottawa","Toronto","Quebec City","Kingston","Trois-Rivieres","Windsor",
         "Hamilton","London","Halifax","Moncton","Saint John","St. John’s","Sudbury",
         "Thunder Bay","Winnipeg","Saskatoon","Regina","Calgary","Edmonton","Kelowna",
         "Vancouver","Victoria")
rainfall = c(1000,920,831,1184,960,1123,935,897,1012,1468,1124,1295,1534,903,684,521,365,
             390,456,419,345,1457,705)
days = c(163,161,145,175,159,161,150,149,168,162,161,158,212,167,143,125,87,118,112,123,
         120,168,148)
```

The code below starts by creating and scaling a data frame for the data. Then `hclust()` builds the clustering information, which `plot()` then uses to display the **clustering dendrogram**.[41]

41: We use `hclust`'s Euclidean dissimilarity and complete linkage defaults.

```
rain.data <- data.frame(city, rainfall,days)
dist.rain <- dist(scale(rain.data[,2:3]))
hc <- hclust(dist.rain) # distances as hc object
plot(hc, labels = rain.data$city)
```



It is not surprising to see that Ottawa and Kingston are very similar when it comes to rainfall totals and number of rainy days, as are Hamilton, Windsor, and Toronto (each group of cities being in the same general region), but it might be surprising to see Halifax and Vancouver as being similar – until we remember that they are both coastal cities.

The Return of the Text-Rich Hockey Dataset

Now that we have seen a simple example of how hierarchical clustering works with numerical data, we turn our attention back to text data, and the hockey dataset of Section 27.4.1.

As we did then, we clean the data (removing stop words, etc.) from the Sens games, at which point we create a **term-document matrix** (TDM) from the cleaned text data.

With the TDM in hand, we can switch over to clustering the data, once we have generated a **distance matrix** (DM),[42] which we use to hierarchically cluster the data, and generate the text data's cluster dendrogram.

42: We could consider either document or term similarity for the DM; here we will use term similarity.

```
recaps <- read.csv(file="Recap_data_first_pass_utf8.csv", header=TRUE, sep=",",
                   stringsAsFactors=FALSE)

# Isolate the text recaps
AP.recaps <- recaps$AP.Recap

# Make a vector source
AP.recaps.source <- tm::VectorSource(AP.recaps)

# Make a volatile corpus
AP.recaps.corpus <- tm::VCorpus(AP.recaps.source)

# Create a customized function to clean the corpus
clean_corpus_Sens <- function(corpus){
  corpus <- tm::tm_map(corpus, content_transformer(qdap::replace_abbreviation))
  corpus <- tm::tm_map(corpus, tm::removePunctuation)
  corpus <- tm::tm_map(corpus, tm::removeNumbers)
  corpus <- tm::tm_map(corpus, tm::stemDocument)
  corpus <- tm::tm_map(corpus, tm::content_transformer(tolower))
  corpus <- tm::tm_map(corpus, tm::stripWhitespace)
  corpus <- tm::tm_map(corpus, tm::removeWords, c(tm::stopwords("en"), "game", "first",
                       "second", "third", "Ottawa", "Senators"))
  return(corpus)
}

# Apply the customized function to the corpus
clean_corp.AP.recaps <- clean_corpus_Sens(AP.recaps.corpus)

# Create a TDM from the corpus
AP.recaps_tdm <- tm::TermDocumentMatrix(clean_corp.AP.recaps)

# Remove sparse terms
AP.recaps_tdm_50 <- tm::removeSparseTerms(AP.recaps_tdm,sparse=.5)
```

```
# What effect did this have on the number of terms?
AP.recaps_tdm_50
```

```
<<TermDocumentMatrix (terms: 82, documents: 101)>>
Non-/sparse entries: 5806/2476
Sparsity           : 30%
Maximal term length: 8
Weighting          : term frequency (tf)
```

```
# Convert AP.recaps_tdm to a matrix: AP.recaps_m
AP.recaps_tdm_50_m <- as.matrix(AP.recaps_tdm_50)

# Save the term document matrix as a data frame
TDM_50.df <- as.data.frame(AP.recaps_tdm_50_m)
TDM_50.df
```

```
         1   2   3   4   5   6   7   8   9  10  ... 92  93  94  95  96  97  98  99 100 101
also     2   0   1   2   1   1   1   1   2   1  ...  4   1   1   1   0   2   0   1   2   0
anderson 3   3   0   2   3   5   1  13   6   1  ...  4   3   5   4   5   2   2   2  10   6
assist   1   0   1   2   0   2   2   0   2   0  ...  4   1   2   1   1   0   2   4   0   2
back     0   3   0   1   0   0   2   1   1   0  ...  4   5   1   1   3   2   1   1   4   1
beat     1   3   1   3   1   1   1   1   1   0  ...  2   1   2   0   0   3   1   3   2   0
befor    1   0   1   4   0   3   2   0   0   1  ...  1   0   0   1   0   2   0   0   0   1
boucher  2   2   0   0   3   2   0   0   2   1  ...  4   3   0   4   1   2   1   3   3   1
came     2   1   0   1   2   2   0   1   0   3  ...  2   2   1   1   2   2   0   1   1   1
chanc    0   2   0   0   0   0   2   1   1   2  ...  2   0   5   2   0   2   0   1   1   0
coach    4   3   0   1   2   1   1   0   3   5  ...  2   2   1   2   1   3   2   2   2   2
come     1   0   0   1   0   1   1   0   1   2  ...  2   1   1   1   0   0   0   0   1   1
craig    1   1   0   1   1   1   1   1   1   1  ...  1   1   1   1   1   1   1   1   1   2
didnt    1   2   0   1   2   2   1   2   0   0  ...  1   1   1   1   3   1   1   1   5   1
end      0   0   0   0   0   0   0   1   2   0  ...  1   1   0   0   4   1   0   0   0   1
erik     2   1   0   1   1   0   1   1   0   2  ...  1   1   1   0   0   1   1   1   1   2
final    0   0   0   1   0   0   0   2   0   1  ...  2   2   4   4   3   2   1   6   3   5
five     1   0   0   1   0   1   3   0   1   2  ...  0   1   0   1   1   0   1   1   0   0
four     2   0   1   1   0   1   0   0   2   1  ...  3   1   0   2   0   2   1   3   2   3
gave     0   1   1   2   0   0   2   1   0   0  ...  0   2   0   2   0   1   1   0   2   0
get      3   2   0   3   2   3   1   0   1   2  ...  2   3   2   5   3   3   3   2   4   2
give     1   1   0   4   1   0   0   1   0   0  ...  0   1   0   3   1   2   0   2   0   2
goal     9   2   9  17   4   4   6   1   3   4  ...  8  13   5   5   1  13   5   4   5   4
good     1   0   0   2   1   4   1   0   3   3  ...  3   1   2   2   1   0   5   3   1   0
got      1   2   0   3   5   0   1   1   0   1  ...  4   2   5   1   0   4   2   1   2   1

        ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...

two      1   2   3   4   0   4   4   3   0   2  ...  9   5   2   2   2   1   6   3   4   3
way      1   3   0   0   1   2   1   1   2   1  ...  0   2   1   2   2   0   1   1   1   5
went     0   0   0   1   0   1   1   0   1   0  ...  1   1   1   1   2   1   3   1   0   0
win      1   4   0   2   0   0   2   2   1   0  ...  3   6   0   1   2   3   0   1   1   1
```

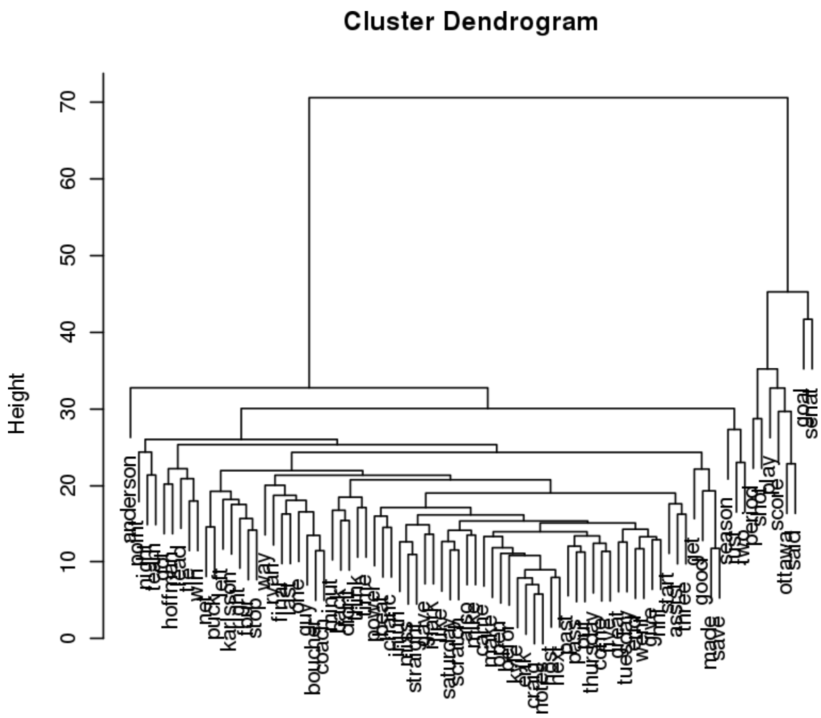
The **document signatures** of the terms have a fair number of non-zero entries, as we would expect since we eliminated sparse terms.

```
# Compute distance matrices
dist_50 = dist(TDM_50.df)
```

```
              also  anderson    assist      back      beat     befor   boucher ...
anderson 28.160256
assist   14.798649 30.594117
back     15.874508 26.095977 18.947295
beat     15.524175 27.313001 18.000000 15.066519
befor    12.529964 30.331502 16.613248 17.464249 14.142136
boucher  14.247807 27.676705 16.431677 16.583124 15.362291 16.613248
came     14.387495 28.879058 16.792856 16.643317 16.552945 13.341664 16.000000 ...
     ... ...       ...       ...       ...       ...       ...       ...       ...
```

With the distance matrix created, the next step is to cluster the terms using the distance matrix.

```
# Build the hc object
hc.50 = hclust(dist_50)

# Plot the dendograms
plot(hc.50)
```



The dendrogram is not easy to read, but we can get the labels directly, if needed.

```
# Build hcd
hcd.50 = as.dendrogram(hc.50)

# labels
labels(hcd.50)
```

```
[1]  ’anderson’ ’point’ ’night’ ’team’ ’got’ ’hoffman’ ’lead’ ’tie’ ’win’ ’net’ ’puck’
[12] ’left’ ’karlsson’ ’right’ ’four’ ’stop’ ’way’ ’ryan’ ’final’ ’last’ ’one’ ’guy’
[23] ’boucher’ ’coach’ ’minut’ ’back’ ’didnt’ ’think’ ’time’ ’power’ ’beat’ ’chanc’
[33] ’injuri’ ’miss’ ’straight’ ’gave’ ’mark’ ’like’ ’saturday’ ’scratch’ ’also’ ’mike’
[43] ’came’ ’make’ ’open’ ’befor’ ’kyle’ ’erik’ ’craig’ ’notes’ ’host’ ’next’ ’past’
[54] ’pass’ ’put’ ’thursday’ ’come’ ’five’ ’great’ ’tuesday’ ’end’ ’went’ ’give’ ’nhl’
[65] ’start’ ’assist’ ’three’ ’get’ ’good’ ’made’ ’save’ ’season’ ’just’ ’two’ ’period’
[76] ’shot’ ’play’ ’score’ ’ottawa’ ’said’ ’goal’ ’senat’
```

Say we are looking for $k = 2$ clusters.

```
result = cutree(hc.50, k=2)

# cluster 1
rownames(TDM_50.df)[result==1]

# cluster 2
rownames(TDM_50.df)[result==2]
```

```
# cluster 1
[1]  ’anderson’ ’point’ ’night’ ’team’ ’got’ ’hoffman’ ’lead’ ’tie’ ’win’ ’net’ ’puck’
...
[65] ’start’ ’assist’ ’three’ ’get’ ’good’ ’made’ ’save’ ’season’ ’just’ ’two’

# cluster 2
[1] ’period’ ’shot’ ’play’ ’score’ ’ottawa’ ’said’ ’goal’ ’senat’
```

The terms found in a cluster are those for whom the document signatures are similar to one another in the corpus.

Using the TDM to find Associations

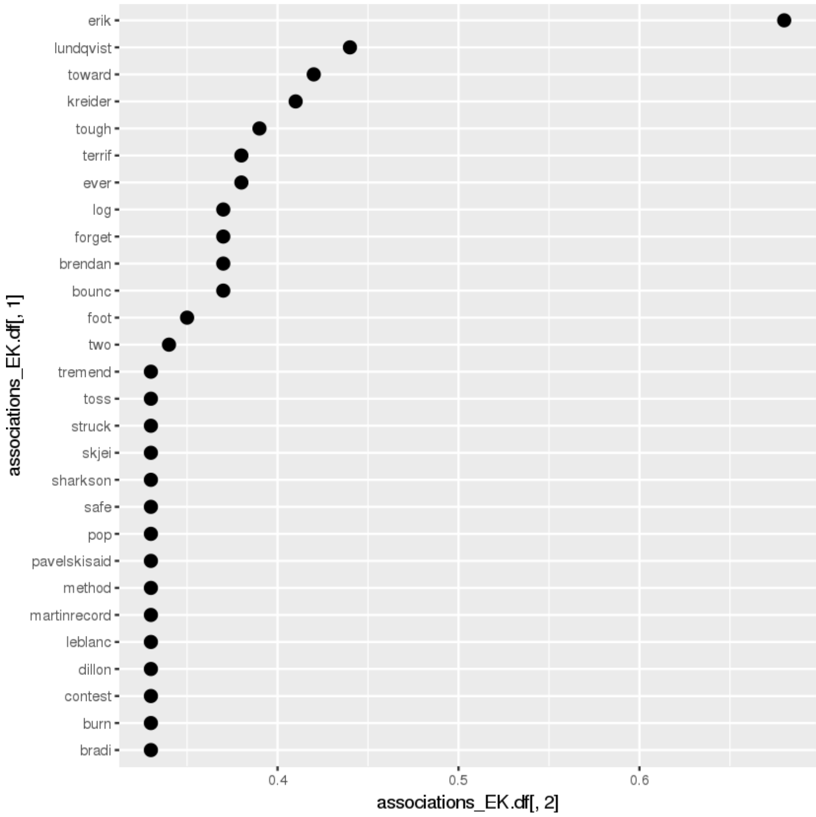
The TDM can also be used to find terms that are **associated with each other** (i.e., terms that appear with each other) across documents. Here we see which other terms are associated with the term ’karlsson’,[43] as visualized using a **dotplot**.

43: Former Senators player Erik Karlsson.

```
(associations_EK <- tm::findAssocs(AP.recaps_tdm, "karlsson", 0.33))  # 0.33: lower correlation limit
```

```
$karlsson =

erik
    0.68
lundqvist
    0.44
toward
    0.42
kreider
    0.41
tough
    0.39
ever
    0.38
terrif
    0.38
bounc
    0.37
brendan
    0.37
forget
    0.37
log
    0.37
foot
    0.35
two
    0.34
bradi
    0.33
burn
    0.33
contest
    0.33
dillon
    0.33
leblanc
    0.33
martinrecord
    0.33
method
    0.33
pavelskisaid
    0.33
pop
    0.33
safe
    0.33
sharkson
    0.33
skjei
    0.33
struck
    0.33
toss
    0.33
tremend
    0.33
```

```
associations_EK.df <- qdap::list_vect2df(associations_EK)[,2:3]

library(ggplot2)
ggplot(associations_EK.df,aes(y=associations_EK.df[,1])) +
    geom_point(aes(x=associations_EK.df[,2]), data=associations_EK.df, size = 3)
```



It is not really surprising that the word with the highest co-occurrence frequency is 'erik' (since that is the player's full name), but some of the other counts are perhaps a bit more surprising: Henrik Lundqvist, a fellow Swede, and Chris Kreider played for the New York Rangers that year, a team the Senators faced 10 times in 2016-2017. This may explain why their last names show up in the dotplot.

27.4.6 The Scottish Play

In this section, we analyze the **emotional content** in Shakespeare's *Macbeth*.[44]

44: This example borrows even more heavily than usual from [39].

Sentiment Analysis Workflow

In general, we conduct **term-by-term** sentiment analysis of text as follows:

1. start with text data;
2. un-nest the tokens to produce a first iteration of **tidy text**;
3. clean and process the tidy text as required by the context;

4. join the tidy text to an appropriate **sentiment lexicon**;
5. summarize the tidy text/sentiment lexicon into a first iteration of **summarized text**;
6. clean and analyze the summarized text, and
7. visualize and present the text mining results.

Sentiment Lexicons

Throughout, we will use the sentiment lexicons included with the `tidytext` package: `AFINN`, `nrc`, `bing`, `loughran`.

```
library(tidytext)
library(textdata) # to obtain the Lexicons

AFINN = get_sentiments("afinn") # words on a scale from -5 (negative) to 5 (positive)
BING = get_sentiments("bing") # binary negative/positive
NRC = get_sentiments("nrc") # assigns categories of sentiments (possibly 1+ to a term)
LOUGHRAN = get_sentiments("loughran")
```

We can take a quick look at the 4 lexicons – the first thing to notice is that they do not all contain the same number of observations.

```
str(AFINN)
table(AFINN$value)
head(AFINN)
tail(AFINN)
```

```
spec_tbl_df [2,477 x 2] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
 $ word : chr [1:2477] "abandon" "abandoned" "abandons" "abducted" ...
 $ value: num [1:2477] -2 -2 -2 -2 -2 -2 -3 -3 -3 -3 ...
 - attr(*, "spec")=
  .. cols(
  ..   word = col_character(),
  ..   value = col_double()
  .. )
 - attr(*, "problems")=<externalptr>

 -5  -4  -3  -2  -1   0   1   2   3   4   5
 16  43 264 966 309   1 208 448 172  45   5
```

```
# A tibble: 6 x 2
  word        value
  <chr>       <dbl>
1 abandon        -2
2 abandoned      -2
3 abandons       -2
4 abducted       -2
5 abduction      -2
6 abductions     -2
```

```
# A tibble: 6 x 2
  word      value
  <chr>     <dbl>
1 youthful      2
2 yucky        -2
3 yummy         3
4 zealot       -2
5 zealots      -2
6 zealous       2
```

```
str(BING)
table(BING$sentiment)
head(BING)
tail(BING)
```

```
tibble [6,786 x 2] (S3: tbl_df/tbl/data.frame)
 $ word     : chr [1:6786] "2-faces" "abnormal" "abolish" "abominable" ..
 $ sentiment: chr [1:6786] "negative" "negative" "negative" "negative" ..

negative positive
    4781     2005
```

```
# A tibble: 6 x 2
  word       sentiment
  <chr>      <chr>
1 2-faces    negative
2 abnormal   negative
3 abolish    negative
4 abominable negative
5 abominably negative
6 abominate  negative
```

```
# A tibble: 6 x 2
  word      sentiment
  <chr>     <chr>
1 zealous   negative
2 zealously negative
3 zenith    positive
4 zest      positive
5 zippy     positive
6 zombie    negative
```

```
str(NRC)
table(NRC$sentiment)
head(NRC)
tail(NRC)
```

```
tibble [13,872 x 2] (S3: tbl_df/tbl/data.frame)
 $ word     : chr [1:13872] "abacus" "abandon" "abandon" "abandon" ...
 $ sentiment: chr [1:13872] "trust" "fear" "negative" "sadness" ...

   anger anticipation      disgust         fear          joy
    1245          837         1056         1474          687
negative     positive      sadness     surprise        trust
    3316         2308         1187          532         1230
```

```
# A tibble: 6 x 2
  word      sentiment
  <chr>     <chr>
1 abacus    trust
2 abandon   fear
3 abandon   negative
4 abandon   sadness
5 abandoned anger
6 abandoned fear
```

```
# A tibble: 6 x 2
  word    sentiment
  <chr>   <chr>
1 zealous trust
2 zest    anticipation
3 zest    joy
4 zest    positive
5 zest    trust
6 zip     negative
```

```
str(LOUGHRAN)
table(LOUGHRAN$sentiment)
head(LOUGHRAN)
tail(LOUGHRAN)
```

```
tibble [4,150 x 2] (S3: tbl_df/tbl/data.frame)
 $ word     : chr [1:4150] "abandon" "abandoned" "abandoning" "abandonment" ...
 $ sentiment: chr [1:4150] "negative" "negative" "negative" "negative" ...

constraining    litigious     negative     positive  superfluous  uncertainty 
         184          904         2355          354           56          297 
```

```
# A tibble: 6 x 2
  word         sentiment
  <chr>        <chr>    
1 abandon      negative 
2 abandoned    negative 
3 abandoning   negative 
4 abandonment  negative 
5 abandonments negative 
6 abandons     negative 
```

```
# A tibble: 6 x 2
  word           sentiment  
  <chr>          <chr>      
1 stratum        superfluous
2 superannuation superfluous
3 theses         superfluous
4 ubiquitous     superfluous
5 wheresoever    superfluous
6 whilst         superfluous
```

At a first glance, it seems that there are more terms in the negative end of the "sentimental spectrum". What kind of an effect might this have on sentiment analysis?[45]

45: Is that the same for every language? French seems to have more room for positive terms, but is that simply confirmation bias at play?

We can also compare how the various lexicons grade specific words – let's take a look at a few possibilities.

```
words = c("abandon","bad","not","cool","egregious","strike")
A.w = AFINN[AFINN$word %in% words,]
A.w$lexicon = "AFINN"
colnames(A.w)[2] = "sentiment"
B.w = BING[BING$word %in% words,]
B.w$lexicon = "BING"
N.w = NRC[NRC$word %in% words,]
N.w$lexicon = "NRC"
L.w = LOUGHRAN[LOUGHRAN$word %in% words,]
L.w$lexicon = "LOUGHRAN"

T.w = rbind(A.w,B.w,N.w,L.w)
dplyr::arrange(T.w,word) |> print(n = Inf)
```

```
# A tibble: 25 x 3
   word      sentiment lexicon 
   <chr>     <chr>     <chr>   
 1 abandon   -2        AFINN   
 2 abandon   fear      NRC     
 3 abandon   negative  NRC     
 4 abandon   sadness   NRC     
 5 abandon   negative  LOUGHRAN
```

```
 6 bad       -3        AFINN
 7 bad       negative  BING
 8 bad       anger     NRC
 9 bad       disgust   NRC
10 bad       fear      NRC
11 bad       negative  NRC
12 bad       sadness   NRC
13 bad       negative  LOUGHRAN
14 cool      1         AFINN
15 cool      positive  BING
16 cool      positive  NRC
17 egregious negative  BING
18 egregious anger     NRC
19 egregious disgust   NRC
20 egregious negative  NRC
21 egregious negative  LOUGHRAN
22 strike    -1        AFINN
23 strike    negative  BING
24 strike    anger     NRC
25 strike    negative  NRC
```

Notes and Comments

- Does it make sense to use a social media lexicon to analyze emotional content in Shakespeare's plays? **Context-specific lexicons** can always be used, instead, but they first need to be **built** (time-consuming) and **validated** (requires domain expertise).
- Beware the no-free lunch theorem: the most suitable lexicon may change from project to project.
- As a rule of thumb, it appears that applying sentiment analysis to any text that is intelligible without a slew of annotations is likely to yield more insight than text that requires annotation, but the latter can still be valuable.
- In general, it seems easier to identify a clear sentiment in a short text than in a long one.
- The **unigram** term "bad" is identified as a negative word, whereas "not" is seen as neutral, but "not bad" would be a mostly positive **bigram**. We have discussed bigrams in Section 27.4.2; there is a lot more to be said on the topic (see Chapter 32).

Matching Sentiments to Words

Let's take a quick look at how we can set-up the matching between sentiments and terms using a lexicon. For the purposes of this example, we will use the NRC lexicon, together with *A Midsummer Night's Dream*, one of the more whimsical of Shakespeare's comedies. We start by creating a custom lexicon for the works of Shakespeare at the Gutenberg Project.

```
word = c("etext", "copyright", "implications", "electronic", "version", "william",
         "shakespeare",  "inc", "gutenberg", "electronic", "machine", "distributed",
         "commercially", "commercial", "distribution", "download", "shareware")
```

```
lexicon = rep("custom",17)
custom = data.frame(word,lexicon)
stop_words_custom_gut = rbind(tidytext::stop_words,custom)
```

Next, we extract the play and put it into a tidy dataset.

```
my_mirror = "http://mirror.csclub.uwaterloo.ca/gutenberg/"
msnd <- gutenbergr::gutenberg_download(c(1514),my_mirror)

tidy_msnd <- msnd |>
  tidytext::unnest_tokens(word,text) |>
  dplyr::mutate(word = stringr::str_extract(word,"[a-z’]+")) |> # removing odd encodings
  dplyr::anti_join(stop_words_custom_gut) |>  # removing the non-play terms
  na.omit() # remove NAs
```

Now, let’s extract the `surprise` words (and attendent frequencies) from *A Midsummer Night’s Dream* (according to the NRC lexicon):

```
nrc_surprise <- NRC |>
  dplyr::filter(sentiment == "surprise")

tidy_msnd |>
  dplyr::inner_join(nrc_surprise) |>
  dplyr::count(word, sort = TRUE) |> print(n = Inf)
```

```
# A tibble: 58 x 2
   word          n
   <chr>     <int>
 1 sweet        48
 2 art          16
 3 leave        15
 4 death        14
 5 pray         13
 6 break         7
 7 lovely        7
 8 marry         7
 9 youth         7
10 catch         5
11 wild          5
12 jest          4
13 kiss          4
14 lose          4
15 spirits       4
16 cheer         3
17 fright        3
18 hope          3
19 judgment      3
20 luck          3
21 perchance     3
22 teach         3
23 chance        2
24 mouth         2
25 musical       2
26 revenge       2
27 stealth       2
28 sun           2
29 trip          2
30 worm          2
31 advance       1
32 angel         1
33 ceremony      1
34 conjure       1
35 frantic       1
36 gift          1
37 illusion      1
38 laughter      1
39 lightning     1
40 marvel        1
41 merriment     1
42 murder        1
43 palpable      1
44 precious      1
45 saint         1
46 scare         1
47 screech       1
48 senseless     1
49 shot          1
50 shout         1
51 shriek        1
52 slip          1
53 smile         1
54 sunny         1
55 tempest       1
56 thief         1
57 tickle        1
58 vanished      1
```

We can do the same for the `anger` terms.

```
nrc_anger <- NRC |>
  dplyr::filter(sentiment == "anger")

tidy_msnd |>
  dplyr::inner_join(nrc_anger) |>
  dplyr::count(word, sort = TRUE) |> print(n = Inf)
```

```
# A tibble: 133 x 2
   word               n
   <chr>          <int>
 1 fear              17
 2 death             14
 3 bear              10
 4 hate               9
 5 scorn              9
 6 lie                7
 7 youth              7
 8 mistress           6
 9 hell               5
10 spite              5
11 strike             5
12 cross              4
13 derision           4
14 force              4
15 lose               4
16 steal              4
17 words              4
18 adder              3
19 beast              3
20 bellows            3
21 bloody             3
22 cruel              3
23 fierce             3
24 hateful            3
25 honest             3
26 ill                3
27 injury             3
28 offend             3
29 wound              3
30 angry              2
31 bully              2
32 confusion          2
33 curse              2
34 dame               2
35 darkness           2
36 delay              2
37 deny               2
38 deserve            2
39 discord            2
40 dreadful           2
41 fight              2
42 grim               2
43 hatred             2
44 hurt               2
45 jealous            2
46 jealousy           2
47 liquor             2
48 loath              2
49 loathe             2
50 lying              2
51 mad                2
52 murderer           2
53 musical            2
54 rage               2
55 revenge            2
56 tyrant             2
57 wasted             2
58 anger              1
59 anguish            1
60 argument           1
61 bark               1
62 battle             1
63 bee                1
64 burial             1
65 complaint          1
66 cruelty            1
67 cur                1
68 damn               1
69 deceive            1
70 defect             1
71 desert             1
72 despise            1
73 detest             1
74 disgrace           1
75 disobedience       1
76 disparage          1
77 dissension         1
78 distracted         1
79 elf                1
80 enmity             1
81 exile              1
82 fee                1
83 foe                1
84 foul               1
85 fury               1
86 gore               1
87 guilty             1
88 harbinger          1
89 harshness          1
90 hit                1
91 hot                1
92 hunting            1
93 injurious          1
94 insufficiency      1
95 killing            1
96 lightning          1
97 lunatic            1
98 madman             1
99 mighty             1
100 miserable         1
101 mislead           1
102 moral             1
103 murder            1
104 odious            1
105 offended          1
106 pare              1
107 prison            1
108 prosecute         1
109 raging            1
110 rail              1
111 riot              1
112 scar              1
113 scare             1
114 senseless         1
115 shiver            1
116 shot              1
117 shout             1
118 shriek            1
119 shun              1
120 sinister          1
121 slay              1
122 stone             1
123 strife            1
124 tempest           1
125 thief             1
126 throttle          1
127 torment           1
128 ungrateful        1
129 unkind            1
130 warrior           1
131 whip              1
132 wrath             1
133 wretch            1
```

Note that there are overlaps: "revenge", for instance, is in both collections. In total, there are 286 occurrences of `anger` terms in the cleaned up text data, and 225 occurrences of `surprise` terms. Does this fit with the nature of the play?

term-by-term Sentiment Analysis of Macbeth

Instead of finding words that express specific sentiments, we are going to compute a **score** for various sections of *Macbeth*.

First, we load a processed version of the text.

```
macbeth = read.csv("Macbeth.csv", header=TRUE, sep=",",
                   stringsAsFactors=FALSE)
str(macbeth)
```

```
'data.frame': 15221 obs. of  6 variables:
 $ Act       : int  1 1 1 1 1 1 1 1 1 1 ...
 $ Scene     : int  1 1 1 1 1 1 1 1 1 1 ...
 $ Speaker   : chr  "First Witch" "First Witch" "Second Witch" "Second Witch" ...
 $ Text      : chr  "When shall we three meet again" "In thunder, lightning, or in rain?" ...
 $ Scene_Line: int  1 2 3 4 5 6 7 8 9 10 ...
 $ Play_Line : int  1 2 3 4 5 6 7 8 9 10 ...
```

The Act and Scene variables could be combined to provide an increasing identifier for the play's sections. For the purpose of this example, we only want to keep information on the text, the line number, and the section.

```
macbeth$section = macbeth$Act*10 + macbeth$Scene
macbeth <- macbeth |>
  dplyr::select(c("Text","Play_Line","section"))
head(macbeth)
```

```
                                Text Play_Line section
1     When shall we three meet again         1      11
2 In thunder, lightning, or in rain?         2      11
3         When the hurlyburly's done,        3      11
4    When the battle's lost and won.         4      11
5   That will be ere the set of sun.         5      11
6                   Where the place?         6      11
```

Next, we unnest the tokens into a tidy format, using word as the basic unit.

```
tidy_macbeth <- macbeth |> tidytext::unnest_tokens(word, Text)
head(tidy_macbeth,20)
```

```
   Play_Line section      word
1          1      11      when
2          1      11     shall
3          1      11        we
4          1      11     three
5          1      11      meet
6          1      11     again
7          2      11        in
8          2      11   thunder
9          2      11 lightning
```

```
10          2      11          or
11          2      11          in
12          2      11        rain
13          3      11        when
14          3      11         the
15          3      11 hurlyburly’s
16          3      11        done
17          4      11        when
18          4      11         the
19          4      11     battle’s
20          4      11        lost
```

At this point, we get a sentiment score for each word using the BING lexicon (words that don’t appear in the lexicon are assumed to be neutral).

```
macbeth_SA <- tidy_macbeth |> dplyr::inner_join(BING)
head(macbeth_SA)
```

```
  Play_Line section word sentiment
1         4      11 lost  negative
2         4      11  won  positive
3        12      11 fair  positive
4        12      11 foul  negative
5        12      11 foul  negative
6        12      11 fair  positive
```

Next, we count the `positive` and `negative` words in each grouping of $L = 30$ lines of text, say.

```
macbeth_SA <- tidy_macbeth |> dplyr::inner_join(BING) |>
  dplyr::count(index = Play_Line %/% 30, sentiment)
head(macbeth_SA)
```

```
  index sentiment  n
1     0  negative 11
2     0  positive 10
3     1  negative 12
4     1  positive 10
5     2  negative 17
6     2  positive 11
```

The counts are stored in the variable n. We reshape the tibble into a **tidy** dataset, one for which each column hosts 1 variable, and each row, 1 observation.

```
macbeth_SA <- tidy_macbeth |> dplyr::inner_join(BING) |>
  dplyr::count(index = Play_Line %/% 30, sentiment) |>
  tidyr::spread(sentiment, n, fill = 0)
head(macbeth_SA)
```

```
  index negative positive
1     0       11       10
2     1       12       10
3     2       17       11
4     3        7        2
5     4        7       14
6     5        8        9
```

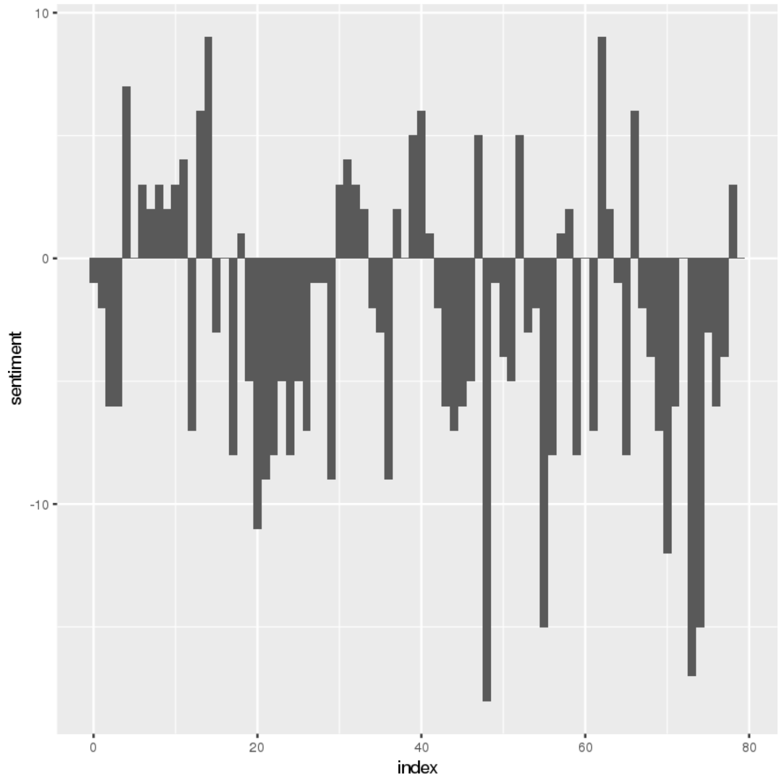
Finally, we compute the **overall sentiment** for each block of lines as the difference between its positive and negative term counts.

```
macbeth_SA <- tidy_macbeth |> dplyr::inner_join(BING) |>
  dplyr::count(index = Play_Line %/% 30, sentiment) |>
  tidyr::spread(sentiment, n, fill = 0) |>
  dplyr::mutate(sentiment = positive - negative)
head(macbeth_SA)
```

```
  index negative positive sentiment
1     0       11       10        -1
2     1       12       10        -2
3     2       17       11        -6
4     3        7        2        -5
5     4        7       14         7
6     5        8        9         1
```

That's it! – although it might be more meaningful to plot the results.

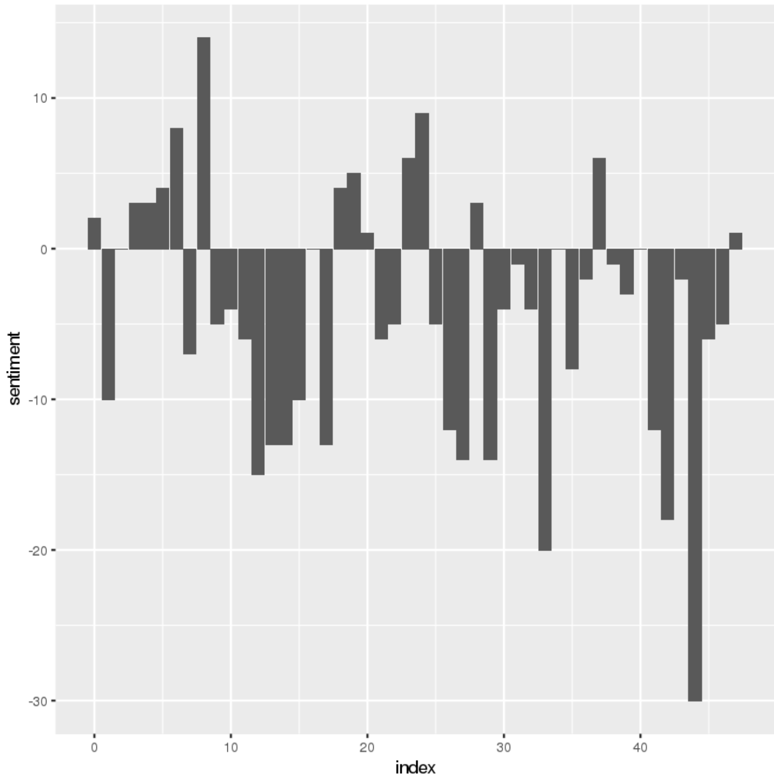
```
library(ggplot2)
ggplot(macbeth_SA, aes(index, sentiment)) + geom_col()
```



The overall picture seems to be somewhat negative – but is that surprising? *Macbeth* is a tragedy, after all, arguably Shakespeare's darkest. Perhaps what we're seeing is an artifact of the way we have **blocked** (grouped) the play, or of the length of the blocks, or even of the sentiment lexicon that we've elected to use. We look into this a little bit more.

Smaller Number of Blocks What happens if we use $L = 50$ instead of $L = 30$?

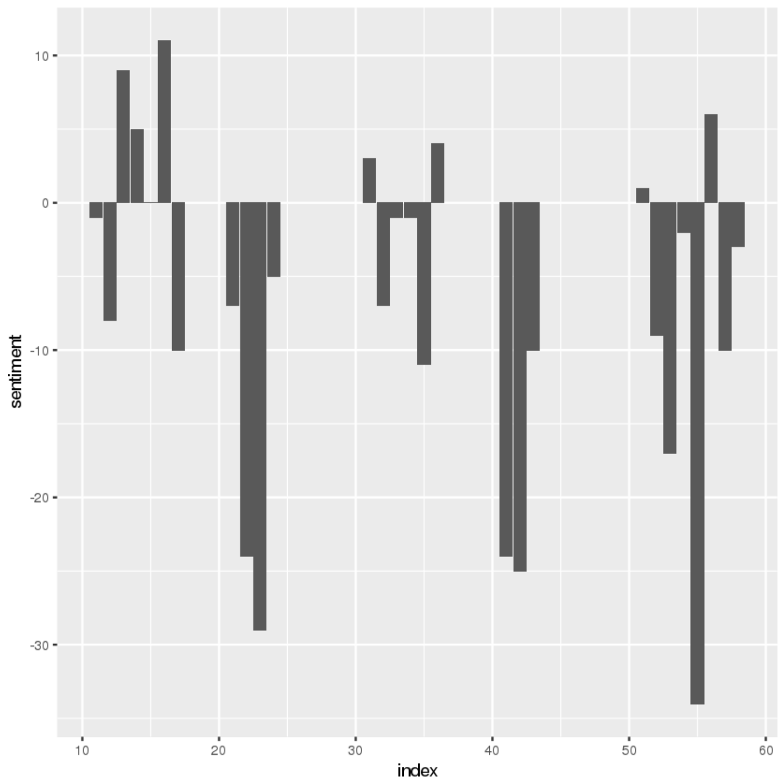
```
macbeth_SA <- tidy_macbeth |> dplyr::inner_join(BING) |>
  dplyr::count(index = Play_Line %/% 50, sentiment) |>
  tidyr::spread(sentiment, n, fill = 0) |>
  dplyr::mutate(sentiment = positive - negative)

ggplot(macbeth_SA, aes(index, sentiment)) + geom_col()
```



There isn't much of a difference.

Different Blocking Mechanism We could use `Act` and `Scene` as separation instead of an arbitrary number of lines L.

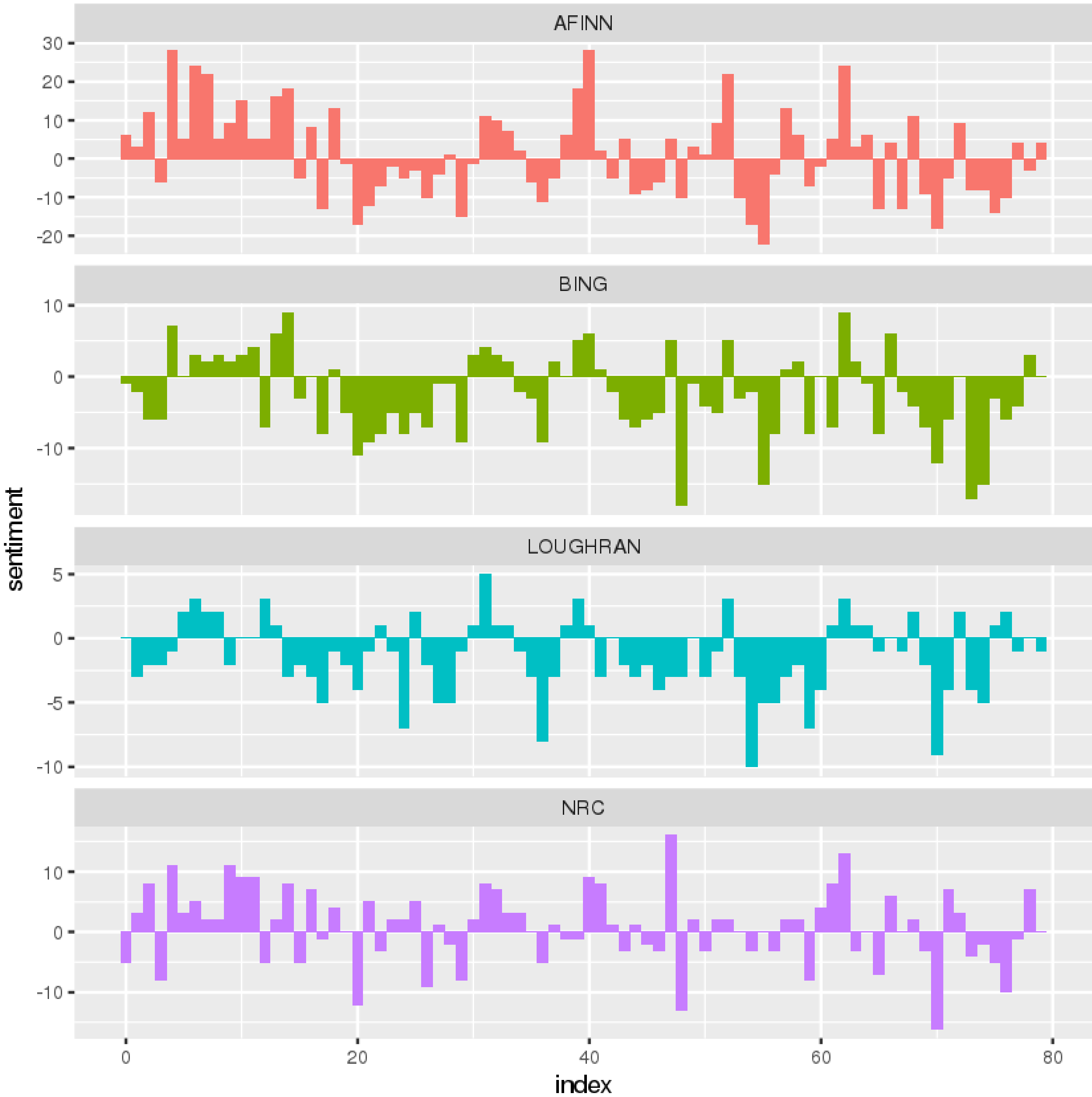
```
macbeth_SA <- tidy_macbeth |> dplyr::inner_join(BING) |>
  dplyr::count(index = section, sentiment) |>
  tidyr::spread(sentiment, n, fill = 0) |>
  dplyr::mutate(sentiment = positive - negative)

ggplot(macbeth_SA, aes(index, sentiment)) + geom_col()
```



Acts II, IV, and V are pretty bleak, seems like...

Different Lexicons We go back to $L = 30$ and run term-by-term sentiment analysis for the four lexicons.

```
afinn_macbeth <- tidy_macbeth |> dplyr::inner_join(AFINN) |>
  dplyr::group_by(index = Play_Line %/% 30) |>
  dplyr::summarise(sentiment = sum(value)) |> dplyr::mutate(method = "AFINN")

bing_nrc_loughran_macbeth <- dplyr::bind_rows(
  tidy_macbeth |>
    dplyr::inner_join(BING) |>
    dplyr::mutate(method = "BING"),
  tidy_macbeth |>
    dplyr::inner_join(NRC |> dplyr::filter(sentiment %in% c("positive","negative"))) |>
    dplyr::mutate(method = "NRC"),
  tidy_macbeth |>
    dplyr::inner_join(LOUGHRAN |> dplyr::filter(sentiment %in% c("positive","negative"))) |>
    dplyr::mutate(method = "LOUGHRAN")) |>
  dplyr::count(method, index = Play_Line %/% 30, sentiment) |>
  tidyr::spread(sentiment, n, fill = 0) |>
  dplyr::mutate(sentiment = positive - negative)

library(ggplot2)
dplyr::bind_rows(afinn_macbeth, bing_nrc_loughran_macbeth) |>
  ggplot(aes(index, sentiment, fill = method)) +
  geom_col(show.legend = FALSE) +
  facet_wrap(~method, ncol = 1, scales = "free_y")
```



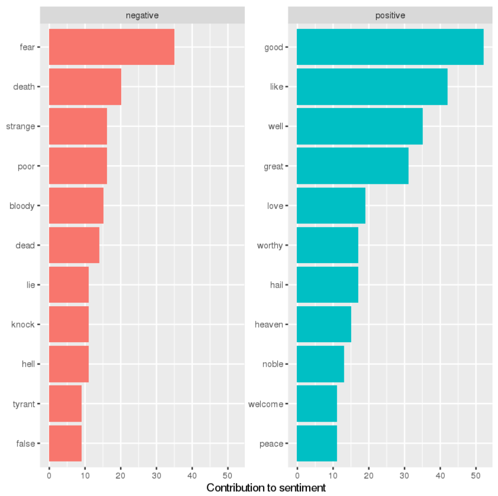
With BING and LOUGHRAN, the "tragedy" of *Macbeth* is "preserved", but that pattern is not as obvious with AFINN and NRC (although there is plenty of negative sentiment in those two as well).

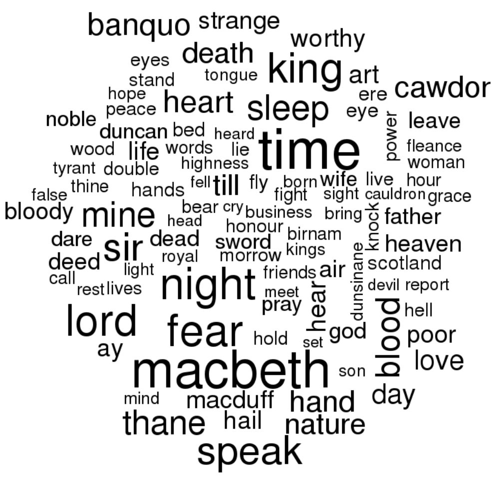
Perhaps we should question the wisdom of using modern lexicons on 450 year old plays?

Text Visualizations Finally, we can also look at how often specific words contribute to positive and negative sentiments in the text of the play, using the BING lexicon.

```
bing_word_counts <- tidy_macbeth |>
  dplyr::inner_join(get_sentiments("bing")) |>
  dplyr::count(word, sentiment, sort = TRUE) |> dplyr::ungroup()
```

```
library(ggplot2)

#bar charts
bing_word_counts |>
  dplyr::group_by(sentiment) |> # will create 2 graphs
  dplyr::top_n(10) |> # pick only the top 10 in each category
  dplyr::ungroup() |> # required to avoid a warning message below
  dplyr::mutate(word = reorder(word, n)) |>
  ggplot(aes(word, n, fill = sentiment)) + # plot a bar chart of word count
  geom_col(show.legend = FALSE) +
  facet_wrap(~sentiment, scales = "free_y") + # there will 2 such bar charts
  labs(y = "Contribution to sentiment",x = NULL) +
  coord_flip() # horizontal bar charts

# wordcloud
word = c("thou", "thy", "thee", "tis", "hath")
lexicon = rep("custom",5)
custom2 = data.frame(word,lexicon)
stop_words_custom_macbeth = rbind(tidytext::stop_words,custom2)
tidy_macbeth |> dplyr::anti_join(stop_words_custom_macbeth) |>
  dplyr::count(word) |>
  with(wordcloud::wordcloud(word, n, max.words = 100))

# comparison cloud
tidy_macbeth |> dplyr::inner_join(BING) |>
  dplyr::count(word, sentiment, sort = TRUE) |>  # counting words for the whole play
  reshape2::acast(word ~ sentiment, value.var = "n", fill = 0) |> # reshaping as a matrix
  wordcloud::comparison.cloud(colors = c("#660000", "#000066"), max.words = 100)
```





Nothing should jump at us as being amiss – which is no guarantee that there's no problem, but it's at least a good sign.

Notes and Comments We see that the choice of lexicon and of the blocking window may have an impact on the sentiment analysis outcome. This is aligned with data anlaysis as we have seen it so far: it's easy to run a sentiment analysis (a few lines of code at most), but it's difficult to pick (or build) the right window and the right lexicon.

27.4.7 Regular Expressions

(This section is a repeat of 16.6.4)

Regular expressions can be used to achieve to extract relevant information from reams of data. Among this mostly unstructured data lurk **systematic elements**, which can be used to help the automation process, especially if quantitative methods are eventually going to be applied to the scraped data.

Systematic structures include numbers, names (countries, etc.), addresses (mailing, e-mailing, URLs, etc.), specific character strings, etc. Regular expressions (**regexps**) are abstract sequences of strings that match concrete recurring patterns in text; they allow for the systematic extraction of the information components from plain text, HTML, and XML.

The examples in this section are based on [23].

Initializing the Environment

The `Python` module for regular expressions is `re`.

```
import re
```

Let us take a quick look at some basics, through the `re` method `match()`. We can try to match a pattern from the beginning of a string, as below:

```
re.match('super','supercalifragilisticexpialidocious')
```

```
<re.Match object; span=(0, 5), match='super'>
```

No such match occurs in the following chunk of code, however.

```
re.match('super','Supercalifragilisticexpialidocious')
```

The regular expression pattern (more on this in a moment) for "word" is `\w+`. The following bit of code would match the first word in a string:

```
w_regex = '\w+'
re.match(w_regex,'Hello World!')
```

```
<re.Match object; span=(0, 5), match='Hello'>
```

Common Regular Expression Patterns

A **regular expression pattern** is a short form used to indicate a type of (sub)string:

- `\w+`: word
- `\d`: digit
- `\s`: space
- `.`: wildcard
- `+` or `*`: greedy match
- `\W`: not word
- `\D`: not digit
- `\S`: not space
- `[a-z]`: lower case group
- `[A-Z]`: upper case group

There are a few `re` functions which, combined with regexps, can make it easier to extract information from large, unstructured text documents:

- `split()`: splits a string on a regexp;
- `findall()`: finds all substrings matching a regexp in a string;
- `search()`: searches for a regexp in a string, and
- `match()`: matches an entire string based on a regexp

Each of these functions takes two arguments: a **regexp** (first) and a **string** (second). For instance, we can split a string on the spaces (and remove them):

```
re.split('\s+','Can  you do the split?')
```

```
['Can', 'you', 'do', 'the', 'split?']
```

The \ in the regexp above is crucial. The following code splits the sentence on the `s` (and removes them):

```
re.split('s+','Can  you do the split?')
```

```
['Can  you do the ', 'plit?']
```

We can also split on single spaces and remove them:

```
re.split('\s','Can  you do the split?')
```

```
['Can', '', 'you', 'do', 'the', 'split?']
```

Alternatively, we can also split on the words and remove them:

```
re.split('\w+','Can  you do the split?')
```

```
['', '  ', ' ', ' ', ' ', '?']
```

Or better yet, split on the non-words and remove them:

```
re.split('\W+','Can you do the split?')
```

```
['Can', 'you', 'do', 'the', 'split', '']
```

Let us take some time to study a silly sentence, saved as a string.

```
test_string = 'Oh they built the built the ship Titanic.
  It was a mistake. It cost more than 1.5 million dollars.
  Never again!'
test_string
```

```
'Oh they built the built the ship Titanic. It was a mistake.
It cost more than 1.5 million dollars. Never again!'
```

In English, only three characters can end a sentence: `.`, `?`, `!`.[46] We create a regexp group (more on those in a moment) as follows:[47]

46: Apparently, nobody's heard of the interrobang...

47: In `Python`, regular expression patterns must be prefixed with an `r` to differentiate between the **raw string** and the **string's interpretation**.

```
sent_ends = r"[.?!]"
```

We could then split the string into its constituent sentences:

```
print(re.split(sent_ends,test_string))
```

```
['Oh they built the built the ship Titanic', ' It was a mistake',
 ' It cost more than 1', '5 million dollars', ' Never again', '']
```

If we wanted to know how many such sentences there were, we simply use the `len()` function:

```
print(len(re.split(sent_ends,test_string)))
```

```
6
```

The regexp range consisting of words with an uppercase initial letter is easy to build:

```
cap_words = r"[A-Z]\w+" # Upper case characters
```

We can find all such words (and how many there are in the string) through:

```
print(re.findall(cap_words,test_string))
print(len(re.findall(cap_words,test_string)))
```

```
['Oh', 'Titanic', 'It', 'It', 'Never']
5
```

The regexp for spaces is:

```
spaces = r"\s+" # spaces
```

We can then split the string on spaces, and count the number of **tokens** (see Chapter 27, *Text Analysis and Text Mining*):

```
print(re.split(spaces,test_string))
print(len(re.split(spaces,test_string)))
```

```
['Oh', 'they', 'built', 'the', 'built', 'the', 'ship', 'Titanic.',
 'It', 'was', 'a', 'mistake.', 'It', 'cost', 'more', 'than', '1.5',
 'million', 'dollars.', 'Never', 'again!']
21
```

The regexp for numbers (contiguous strings of digits) is:

```
numbers = r"\d+"
```

We can find all the numeric characters using:

```
print(re.findall(numbers,test_string))
print(len(re.findall(numbers,test_string)))
```

```
['1', '5']
2
```

The main difference between `search()` and `match()` is that `match()` tries to match from the beginning of a string, whereas `search()` looks for a match anywhere in the string.

Regular Expressions Groups '( )' and Ranges '[ ]' With OR '|'

We can create more complicated regexps using **groups**, **ranges**, and/or "or" statements:

- `[a-zA-Z]+`: an unlimited number of lower and upper case English/French (unaccented) letters;
- `[0-9]`: the digits from 0 to 9;
- `[a-zA-Z'\.\-]+`: any combination of lower and upper case English/French (unaccented) letters, `'`, `.`, and `-`;
- `(a-z)`: the characters `a`, `-`, and `z`;
- `(\s+|,)`: any number of spaces, or a comma;
- `(\d+|\w+)`: words or numerics.

For instance, consider the following text string and regexps groups:

```
text = 'On the 1st day of xmas, my boat sank.'
numbers_or_words = r"(\d+|\w+)"
spaces_or_commas = r"(\s+|,)"
```

This next chunk of code does exactly what one would expect:

```
print(re.findall(numbers_or_words,text))
```

```
['On', 'the', '1', 'st', 'day', 'of', 'xmas', 'my', 'boat', 'sank']
```

What about this one?

```
print(re.findall(spaces_or_commas,text))
```

```
[' ', ' ', ' ', ' ', ' ', ',', ' ', ' ', ' ']
```

27.4.8 Movie Reviews

> "This is a failure of epic proportions. You've got to be a genius to make a movie this bad." (J. Seigel's review of The *Bonfire of the Vanities*)

In this section, we will re-visit sentiment analysis, this time using `Python` and the Natural Language Toolkit (NLTK).[48]

48: See Chapter 32 for more examples of NLTK in action.

Our goal is to develop a sentiment analysis model for movie reviews. The dataset contains 50,000 movie reviews labeled as either **positive** or **negative**. With an accurate sentiment model, we'll have the ability to automatically classify new reviews in order to aggregate review data.

Dataset Information

The `Large Movie Review Dataset v1.0` is described in [36] – the details below are taken verbatim from the same source (the data is also available in the `R` package `textdata`, in the object `dataset_imdb`).

Overview This dataset contains movie reviews along with their associated binary sentiment polarity labels. It is intended to serve as a benchmark for sentiment classification. This document outlines how the dataset was gathered, and how to use the files provided.

Dataset The core dataset contains 50,000 reviews split evenly into 25K train and 25K test sets. The overall distribution of labels is balanced (25K pos and 25K neg). We also include an additional 50,000 unlabeled documents for unsupervised learning.

In the entire collection, no more than 30 reviews are allowed for any given movie because reviews for the same movie tend to have correlated ratings. Further, the training and test sets contain a disjoint set of movies, so no significant performance is obtained by memorizing movie-unique terms and their associated with observed labels. In the labeled train/test sets, a negative review has a score ≤ 4 out of 10, and a positive review has a score ≥ 7 out of 10. Thus reviews with more neutral ratings are not included in the train/test sets. In the unsupervised set, reviews of any rating are included and there are an even number of reviews > 5 and ≤ 5.

Files There are two top-level directories [`train/`, `test/`] corresponding to the training and test sets. Each contains [`pos/`, `neg/`] directories for the reviews with binary labels positive and negative. Within these directories, reviews are stored in text files named following the convention [`[id]_[rating].txt`] where `[id]` is a unique id and `[rating]` is the star rating for that review on a 1-10 scale. For example, the file [`test/pos/200_8.txt`] is the text for a positive-labeled test set example with unique id 200 and star rating 8/10 from IMDb. The [`train/unsup/`] directory has 0 for all ratings because the ratings are omitted for this portion of the dataset.

Preamble

We first need to import the appropriate Python modules. For this exercise, we'll use NLTK.

The `stemmer()` and `tokenize()` functions are used for text processing. The `vader` lexicon is used to analyze the **intensity** of the sentiments.

```
import nltk
from nltk.classify import NaiveBayesClassifier
from nltk.corpus import movie_reviews
from nltk.sentiment import SentimentAnalyzer
from nltk.sentiment.util import mark_negation, extract_unigram_feats
from nltk.tokenize import word_tokenize, sent_tokenize
from nltk.stem.lancaster import LancasterStemmer
stemmer = LancasterStemmer()

import glob
data_path = 'Data/aclImdb/'

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn import metrics
from nltk.sentiment.vader import SentimentIntensityAnalyzer

nltk.download('all')
```

Data Preparation

The reviews are individually stored in text files, and there are four folders for every combination of training/test and positive/negative.

```
train_docs = []
train_labels = []

pos_file_names = glob.glob('{}train/pos/*.txt'.format(data_path))
for file_name in pos_file_names:
    train_docs.append(open(file_name).read())
    train_labels.append(1)
```

```
neg_file_names = glob.glob('{}train/neg/*.txt'.format(data_path))
for file_name in neg_file_names:
    train_docs.append(open(file_name).read())
    train_labels.append(0)
```

The names of the positive reviews are found in an array, just as the names of the negative reviews are.

```
pos_file_names
```

```
['Data/aclImdb/train/pos/0_9.txt',
 'Data/aclImdb/train/pos/10000_8.txt',
 'Data/aclImdb/train/pos/10001_10.txt',
 'Data/aclImdb/train/pos/10002_7.txt',
 ...]
```

We read-in a (random) sample negative review, for the movie *Haunted Boat*, whose file number is `3446_1.txt`. The '`_1`' in the file title lets us know that this is a 1-star review. Does the text support the rating?

```
sample_text_neg = open('{}train/neg/3446_1.txt'.format(data_path)).read()
print(sample_text_neg)
```

```
This film on paper looked like it could possibly be good, after watching though i realised
that this film was completely terrible!! The plot has no meaning, and i think i counted the
best part of 5000 cut scenes each one making the film more annoying boring and ridiculous. I
watched this late night pitch black no noise at all just to add to the SCARINESS of it but
the truth is the only thing that scared me was the music, what they would call tragic music,
they play opera i mean be serious!! This film sums up all of what is not good about this type
of film. To be honest ill say no more but watch at your own risk this film is just complete
rubbish, ENJOY!!
```

Next, we read-in a positive review, now, for a movie called *The Night Listener*, whose file number is `10015_8.txt`. This review is supposed to be an 8-star review – does the text support the rating?

```
sample_text_pos = open('{}train/pos/10015_8.txt'.format(data_path)).read()
print(sample_text_pos)
```

```
Popular radio storyteller Gabriel No one(Robin Williams,scraggy and speaking in hushed,
hypnotic tones) becomes acquainted and friends with a fourteen-year-old boy from Wisconsin
named Pete Logand(Rory Culkin),who has written a book detailing sexual abuse from his parents.
To boot,Pete has AIDS and this compels Gabriel further still,since his partner Jess(Bobby
Cannavale, good)happens to be a survivor of HIV himself. <br /><br />He also acquaints himself
with Pete's guardian,a woman named Donna(Toni Collette, brilliant!)and when Gabriel decides he
wants to meet and talk to the two of them in person and goes to Wisconsin, [...]
```

Bag-of-Words Processing

We will be using a BoW model, so let's explore how we could tokenize (that is, separate) the text into words (the **tokens**).

First, to split a review into sentences we can use the standard `sent_tokenize()` function from NLTK. For instance, the following piece of code will extract the 4th sentence that the tokenizer recognizes (in `Python`, indexing starts with 0).

```
sample_sent = sent_tokenize(sample_text_neg)[3]
print(sample_sent)
```

```
This film sums up all of what is not good about this type of film.
```

We can also try the `word_tokenize()` function to split into words, and do a stemming operation (finding the **roots**) to normalize word forms. The following code will stem all the words in the `sample_sent` sentence from above.

```
sample_words = [ stemmer.stem(word)
                 for word in word_tokenize(sample_sent) ]
print(sample_words)
```

```
['thi', 'film', 'sum', 'up', 'al', 'of', 'what', 'is', 'not',
'good', 'about', 'thi', 'typ', 'of', 'film', '.']
```

One serious problem with a bag of words approach, especially for sentiment analysis, is that the presence of negative/positive words does not imply negative/positive sentiment if the words are negated in the sentence (e.g., "not bad" actually means "good" even though in general an occurrence of "bad" means "bad").

NLTK includes the function `mark_negation()` which takes a tokenized sentence and marks negated words with a '`_NEG`' suffix. Specifically, it marks all words that come after a negation word and before the next punctuation mark. Now 'good' becomes the word '`good_NEG`' so a BoW model can pick up on the **context** of the word.[49]

49: More on this topic in Chapter 32.

```
mark_negation(sample_words)
```

```
['thi',
 'film',
 'sum',
 'up',
 'al',
 'of',
 'what',
 'is',
 'not',
 'good_NEG',
 'about_NEG',
 'thi_NEG',
 'typ_NEG',
 'of_NEG',
 'film_NEG',
 '.']
```

Here is the complete tokenizer function:

1. it tokenizes the text into sentences;
2. for each tokenized sentence, it tokenizes it into words;
3. it keeps only those words of length ≥ 2;
4. it stems the words to only retain the roots, and
5. it marks the negation of certain words.

```
def tokenizer(text):
    sents = sent_tokenize(text)
    tokens = []
    for sent in sents:
        words = word_tokenize(sent)
        words = [ word for word in words if len(word) >= 2 ]
        words = [ stemmer.stem(word) for word in words ]
        words = mark_negation(words)
        tokens += words
    return tokens
```

Now that we have a tokenizer, we can use standard feature extraction methods to get feature vectors for each document.

We use the `scikit-learn` module for the rest of the feature extraction and training; it contains `TfidfVectorizer` class which allows us to define a custom tokenizer and returns a `TFIDF` matrix.[50]

50: We use this class to convert all of the training documents to DTM feature vectors, using the tokenizer defined above (this step can take a few minutes to run).

```
vectorizer = TfidfVectorizer(min_df=1, tokenizer=tokenizer)
train_matrix = vectorizer.fit_transform(train_docs)
```

We can explore the matrix to get an idea of what it contains. It should contain 25,000 documents (as per the introduction), but how many features have been retained?

```
train_matrix.shape
```

```
(25000, 105350)
```

A fair number, as it happens: 25,000 documents and 105,494 features. We can also find the non-zero entries among a subset of the DTM matrix, but that doesn't give us much information at this stage (it will only print the non-zero entries, but we don't know what the features are).

```
print(train_matrix[0:9,0:9])
```

```
  (0, 2) 0.0553631491077
  (1, 3) 0.0250716608331
  (2, 2) 0.0341384144135
  (4, 2) 0.100478224693
  (7, 2) 0.141603801927
  (7, 3) 0.0266220723609
```

Multinomial Naïve Bayes

Multinomial naïve Bayes (`MultinomialNB()`) is one of various classification models in `scikit-learn` (if we wanted to find the best possible classifier, we'd have to try some of the others, but at this stage we just want to show you how the sentiment analysis works).

The `fit` function takes the feature matrix as well as the vector of labels we made when we read the data files.

```
model = MultinomialNB().fit(train_matrix, train_labels)
```

Now that we've trained a model, we can try it out on a 1-star review (but we pick a review in the test set to avoid **overfitting**).

```
neg_sample_text = open(
    '{}test/neg/9999_1.txt'.format(data_path)).read()
print(neg_sample_text)
```

```
When all we have anymore is pretty much reality TV shows with people making fools of
themselves for whatever reason be it too fat or can't sing or cook worth a damn than I
know Hollywood has run out of original ideas. I can not recall a time when anything original
or intelligent came out on TV in the last 15 years. What is our obsession with watching
bums make fools of themselves? I would have thought these types of programs would have
run full circle but every year they come up with something new that is more strange
then the one before. OK so people in this one need to lose weight...most Americans need
to lose weight. I just think we all to some degree enjoy watching people humiliated.
Maybe it makes us feel better when we see someone else looking like a jerk. I don't know
but I just wish something intelligent would come out that did not insult your intelligence.
```

The overall sentiment seems fairly negative. Let's see if our model agrees by computing the class probabilities (negative first, then positive).

```
neg_sample_vec = vectorizer.transform([neg_sample_text])
model.predict_proba(neg_sample_vec)
```

```
array([[ 0.81292637,  0.18707363]])
```

That seems fairly conclusive. Now let's do the same thing for a 10-star movie review in the test set.

```
pos_sample_text = open(
    '{}test/pos/9999_10.txt'.format(data_path)).read()
print(pos_sample_text)
```

```
Although I'm not a golf fan, I attended a sneak preview of this movie and absolutely loved  it. The
historical settings, the blatant class distinctions, and seeing the good and the bad on both sides
of the dividing line held my attention throughout. The actors and their characterizations were
all mesmerizing. And I was on the edge of my seat during the golf segments, which were not only
dramatic and exciting but easy to follow. Toward the end of this movie, "Seabiscuit" came strongly
to mind, although "The Greatest Game Ever Played" is far less complex a story than that film.
In both cases, the fact that the events really happened deepened my interest.
```

We would expect this review to be fairly clearly positive, based on the text alone. What does the model say?

```
pos_sample_vec = vectorizer.transform([pos_sample_text])
model.predict_proba(pos_sample_vec)
```

```
array([[ 0.33913831,  0.66086169]])
```

The class probabilities are closer to one another than with the previous test review, but the positive sentiment is strongest, which is a good sign.

Even though it's not from a review, let's see how the model would deal with a tricky sentence with a "not" in it.

```
stuff = vectorizer.transform(
    ['A ten pound laptop is not a good travel companion.'])
model.predict_proba(stuff)
```

```
array([[ 0.5583131,  0.4416869]])
```

We don't have the greatest confidence in the prediction, but it yields the correct classification.

Performance Evaluation

But it's not enough to try out the sentiment analysis on 1 or 2 reviews: we need to know how well the model performs on the 25,000 test cases?

We'll need to load the test documents before we can compute some evaluation metrics.

```
test_docs = []
test_labels = []

pos_file_names = glob.glob('{}test/pos/*.txt'.format(data_path))
for file_name in pos_file_names:
    test_docs.append(open(file_name).read())
    test_labels.append(1)
neg_file_names = glob.glob('{}test/neg/*.txt'.format(data_path))
for file_name in neg_file_names:
    test_docs.append(open(file_name).read())
    test_labels.append(0)
```

We get the feature vectors for the test data and the model's predictions. Note that we use `transform()` on testing data rather than `fit_transform()`.

```
test_matrix = vectorizer.transform(test_docs)
predicted = model.predict(test_matrix)
```

We can evaluate the **precision**, the **recall**, and the F_1**−score** on the test set. Precision is the fraction of predicted positive results that are actually true positives, whereas recall is the proportion of true positives that are recognized as such by the classification model. Ideally, both of these values would be near 1.

The F1-score is the **harmonic** mean of these quantities.

```
print(metrics.classification_report(
    test_labels, predicted, target_names=['neg', 'pos']))
```

```
             precision    recall  f1-score   support

        neg       0.77      0.87      0.82     12500
        pos       0.85      0.74      0.79     12500

avg / total       0.81      0.81      0.80     25000
```

The performance metrics are actually quite good!

Vader

NLTK comes with a **pre-trained** sentiment analyzer called `vader`. Pre-trained in this context means that it has been trained on a dataset that does not necessarily contain positive and negative movie reviews.

We'll see how it performs on the testing set, but first we'll try it on the trick sentence from above.

```
sia = SentimentIntensityAnalyzer()
sia.polarity_scores('A ten pound laptop is not a good travel companion.')
```

```
{'compound': -0.3412, 'neg': 0.256, 'neu': 0.744, 'pos': 0.0}
```

We see that `vader` wasn't fooled: it recognizes that it's likely to be a neutral sentence, or possibly a negative sentence, but not a positive sentence.

To evaluate on the test data, we find the prediction for each test document, and load them into `classification_report` (remember that it hasn't been trained on the movie review data).

```
vader_predicted = []
for doc in test_docs:
    scores = sia.polarity_scores(doc)
    if scores['pos'] > scores['neg']:
        vader_predicted.append(1)
    else:
        vader_predicted.append(0)
```

We get the following performance metrics.

```
print(metrics.classification_report(
    test_labels, vader_predicted, target_names=['neg', 'pos']))
```

```
             precision    recall  f1-score   support

        neg       0.79      0.52      0.63     12500
        pos       0.64      0.86      0.74     12500

avg / total       0.72      0.69      0.68     25000
```

It is not surprising to see that the results are not quite as good, as the model has not been trained on movie reviews. Still, it is not the worst model in the world.

27.5 Exercises

1. How important are visual cues in communications and business negotiations? How important is context? How easy is it to learn from someone whose context is different from yours (culturally AND professionally)?
2. Conduct a BoW analysis of the Ottawa Senators' 2016-2017 NHL season (as in Section 27.4.1) using the fields `SSS_Recap` and/or `OPP_Recap`.
3. Conduct a BoW analysis of the Ottawa Senators' 2016-2017 NHL season (as in Section 27.4.1) using the fields `AP_Headline`, `SSS_Headline`, and/or `OPP_Title`.
4. Identify the plays linked to each Gutenberg Project ID in Section 27.4.2.
5. Build a Shakespeare word cloud for the Marlowe NAs, and *vice-versa*, as found in Section 27.4.2.
6. Re-run the relevant analyses of Section 27.4.2 after having cleaned the datasets of *theatre* instructions (exit, exeunt, enter, scene, act, folio, dramatis, personae, etc.) and of *copyright/licensing information* and/or *modern contaminating terms*, and with a slightly more restrictive list of early modern english stopwords, removing ancient spelling conventions ("haue" instead of "have", "goode" instead of "good", etc.).
7. Conduct a count of bigrams per play with the Shakespeare corpus of Section 27.4.2.
8. Find the most common title of nobility in each of the Shakespeare plays of Section 27.4.2.
9. Re-run the $n-$gram code of Section 27.4.2 for $n = 3$.
10. Create a classifier for real news/fake news, as in Section 27.4.4, using the `fake_or_real_news1_utf8.csv` file (found in the usual location).
11. Conduct the cluster analysis of Section 27.4.5 using stricter sparsity levels: 90%, 95%, 99%.
12. Conduct a cluster analysis on the data of Section 27.4.5 using the tf-idf weightings (without removing "Ottawa" or "Senators" from the cleaned up recap, using the following as a starting point.

```
# Apply tf-idf weighting
AP.recaps.tfidf_tdm.1 = tmLLTermDocumentMatrix(clean_corp.AP.recaps.1,
                        control=list(weighting=weightTfIdf))

# Remove sparse terms
AP.recaps.tfidf_tdm.1 <- tm::removeSparseTerms(AP.recaps.tfidf_tdm.1,sparse=.7)
```

13. Conduct the cluster analyses of Section 27.4.5 using a DTM instead of a TDM (in essence, finding similar games during the 2016-2017 season).
14. Conduct the cluster analyses of Section 27.4.5 using $k-$means instead of hierarchical clustering.
15. Run a sentiment analysis of *Macbeth* for categories of sentiments other than `positive` or `negative`.
16. Run a sentiment analysis of Trump's tweets in the BOTUS case study.
17. Run a sentiment analysis of the field `SSS_Recap` from the Senators game recap example.

Chapter References

[1] C.C. Aggarwal and C.X. Zhai. *Text Classification*. CRC Press, 2015.

[2] M.M. Alvarez. 'Can Machine Learning and AI Help Solving the Fake News Problem'. In: *The Practical Academic* 12 (2017).

[3] D.C. Anastasia, A. Tagarelli, and G. Karypis. *Document Clustering: The Next Frontier*. CRC Press, 2014.

[4] C. Arthur. 'The big bang visualisation of the top 140 Twitter influencers'. In: *The Guardian* 13 (Sept. 2010).

[5] T. Basu. *NPR's Fascinating Plan to Use A.I. on Trump's Tweets*.

[6] BBC's 60 Seconds Shakespeare. 'Teachers Themes List'.

[7] D. Bevington, ed. *The Complete Works of Shakespeare*. 5th. Longman, 2003.

[8] S. Bird, E. Klein, and E. Loper. *Natural Language Processing with Python*. O'Reilly, 2009.

[9] P. Boily, S. Davies, and J. Schellinck. *The Practice of Data Visualization*. Data Action Lab, 2023.

[10] D. Carnegie and G Taylor, eds. *The Quest for Cardenio: Shakespeare, Fletcher, Cervantes, and the Lost Play*. Oxford University Press, 2012.

[11] Twitter Counter. 'Top 100 Most Followed Twitter Accounts'.

[12] K. Dailey. 'History of Cardenio: Is Shakespeare's Lost Work Recovered?'. In: *BBC News* 12 (2012), p. 2017.

[13] N. Dieker. 'Planet Money's BOTUS Bot Has Yet to Make a Single Stock Trade'. In: 12 (Sept. 2017), p. 2017.

[14] Forbes. 'The World's Biggest Public Companies List - Forbes 2000'.

[15] A. Goldmark. *763: BOTUS*.

[16] S. Greenstone. *When Trump Tweets, This Bot Makes Money*.

[17] InvestorGuide.com. 'List of Publicly Traded Companies'.

[18] 'Company List: NASDAQ, NYSE, & AMEX Companies'. Sept. 2017.

[19] 'Cardenio Play: Overview and Resources'.

[20] N. Ide and J. Veronis. 'Introduction to the special issue on word sense disambiguation: the state of the art'. In: *Computational Linguistics* 24 (1998), pp. 2–40.

[21] M. Ingram. *Here's What a Trump Tweet Does to a Company's Share Price*.

[22] K. Jarmul. *Detecting Fake News with Scikit-Learn*.

[23] K. Jarmul. *Natural Language Processing Fundamentals in Python*.

[24] M.L. Jockers. *Text Analysis with R for Students of Literature*. Springer, 2014.

[25] D. Jurafsky and J.H. Martin. *Speech and Language Processing (2nd ed)*. Pearson, 2009.

[26] T. Kwartler. *Text Mining: Bag of Words*.

[27] D. Lee. *How Twitter Changed the World, Hashtag-by-Hastag*.

[28] A. Long et al. *The Complete Works of William Shakespeare (Abridged): Acting Edition*. Applause, 1987.

[29] G. McIntire. *Fake/Real News Dataset*.

[30] G. McIntire. *On Building a "Fake News" Classification Model*.

[31] K. Mettler. ''Trump and Dump': When POTUS tweets and stocks fall, this animal charity benefits'. In: *Washington Post* 19 (Sept. 2017).

[32] Y. Miyao. *Enju syntactic parser*.

[33] M. Murray. *Topic Modelling Financial News with Natural Language Processing*.

[34] M. Murray. *Topic Modelling Financial News Dataset*.

[35] M. Ogneva. 'How Companies Can Use Sentiment Analysis to Improve Their Business'. In: *Mashable* (2010).

[36] Christopher Potts. 'On the negativity of negation'. In: *Semantics and Linguistic Theory* 20 (2010), pp. 636–659.

[37] *Sens rally after blowing lead, beat Leafs to gain on Habs*. Associated Press. 2017.

[38] P. Shrivastava. 'Text Classification: Applications and Use Cases'. In: *Parallel Dots* (2017).

[39] J. Silge and D. Robinson. *Text Mining with R: a Tidy Approach*. O'Reilly, 2017.

[40] 'Text Mining: Classification, Clustering, and Applications'. In: Chapman and Hall/CRC (2009). Ed. by A.N. Srivastava and M. Sahami.

[41] Stanford Natural Language Processing Group. 'The Stanford Parser: a statistical parser'.

[42] T3. 'Trump & Dump Bot: Analyzes Tweets, Short Stocks'.

[43] A. Taylor, M. Marcus, and B. Santorini. 'The Penn Treebank: An Overview'. In: *Treebanks. Text, Speech and Language Technology*. Ed. by A. Abeillé. Springer, Dordrecht, 2003.

[44] Universal Dependencies. 'Universal Dependencies v2'.

[45] A. Way and N. Gough. 'Comparing Example-Based and Statistical Machine Translation'. In: *Natural Language Engineering* 11 (2005), pp. 295–309.

[46] H. Wickham and G. Grolemund. *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. O'Reilly, Jan. 2017.

[47] Wikipedia. 'Natural Language Processing'.

[48] Wikipedia. 'Shakespeare Bibliography'.

[49] M. Zhou. 'Hamilton: A Text Analysis of the Federalist Papers'. In: *Medium* (2017).