Matthew L. Jockers

Text Analysis With R

for Students of Literature

September 2, 2013

Springer

For my mother,
who prefers to follow the instructions

Preface

This book provides an introduction to computational text analysis using the open source programming language `R`. Unlike other very good books on the use of `R` for the statistical analysis of linguistic data[1] or for conducting quantitative corpus linguistics,[2] this book is meant for students and scholars of literature, for humanists wishing to extend their methodological tool kit to include quantitative and computational approaches to the study of literature. This book is also meant to be short. `R` is a complex program that no single textbook can demystify. The focus here is on making the technical palatable and more importantly making the technical useful and immediately rewarding! Here I mean rewarding not in the sense of satisfaction one gets from mastering a programming language, but rewarding specifically in the sense of quick return on your scholarly investment. You will begin doing things with text right away and each chapter will walk you through a new technique or process.

Computation provides access to information in texts that we simply cannot gather using our traditionally qualitative methods of close reading and human synthesis. The reward comes in being able to access that information at both the micro and macro scale. If this book succeeds, you finish it with a foundation, with a broad exposure to core techniques and a basic understanding of the possibilities. The real learning will begin when you put this book aside and build a project of your own. My aim is to give you enough background so that you can begin that project comfortably.

When discussing my work as a computing humanist, I am frequently asked whether the methods and approaches I advocate succeed in bringing new knowledge to our study of literature. My answer is strong and resounding "yes." At the same time, that strong yes must be qualified a bit; not everything that text analysis reveals is a breakthrough discovery. A good deal of computational work is specifically aimed at testing, rejecting, or reconfirming the knowledge that we think we already possess. During a lecture about macro-patters of literary style in the 19th century novel, I used an example from *Moby Dick*. I showed how *Moby Dick* is a

[1] Baayen, H. A. *Analyzing Linguistic Data: A Practical Introduction to Statistics Using R*. Cambridge UP, 2008.

[2] Gries, Stefan Th. *Quantitative Corpus Linguistics with R: A Practical Introduction*. New York: Routledge, 2009.

statistical mutant among a corpus of 1000 other 19th century American novels. A colleague raised his hand and pointed out that literary scholars already know that *Moby Dick* is an aberration, so why, he asked, bother computing an answer to a question we already know?

My colleague's question was good; it was also revealing. The question said much about our scholarly traditions in the humanities. It is, at the same time, an ironic question. As a discipline, we have tended to favor a notion that literary arguments are never closed: but do we really know that *Moby Dick* is an aberration? Maybe *Moby Dick* is only an outlier in comparison to the other twenty or thirty American novels that we have traditionally studied alongside *Moby Dick*? My point in using *Moby Dick* was not to pretend that I had discovered something new about the position of the novel in the American literary tradition, but rather to bring a new type of evidence and a new perspective to the matter and in so doing fortify (in this case) the existing hypothesis.

If a new type of evidence happens to confirm what we have come to believe using far more speculative methods, shouldn't that new evidence be viewed as a good thing? If the latest Mars rover returns more evidence that the planet could have once supported life, that new evidence would be important. Albeit, it would not be as shocking or exciting as the first discovery of microbes on Mars, or the first discovery of ice on Mars, but it would be important evidence nevertheless, and it would add one more piece to a larger puzzle. So, computational approaches to literary study can provide complementary evidence, and I think that is a good thing.

The approaches outlined in this book also have the potential to present contradictory evidence, evidence that challenges our traditional, impressionistic, or anecdotal theories. In this sense, the methods provide us with some opportunity for the kind of falsification that Karl Popper and post-positivism in general offer as a compromise between strict positivism and strict relativism. But just because these methods *can* provide contradiction, we must not get caught up in a numbers game where we only value the testable ideas. Some interpretations lend themselves to computational or quantitative testing; others do not, and I think that is a good thing.

Finally, these methods can lead to genuinely new discoveries. Computational text analysis has a way of bringing into our field of view certain details and qualities of texts that we would miss with just the naked eye.[3] Naturally, I think this is a good thing too.

This is all I have to say regarding a theory for or justification of text analysis. In my other book, I'm a bit more polemical.[4] The mission here is not to defend the approaches but to share them.

Lincoln, Nebraska, *Matthew L. Jockers*
August, 2013

[3] See Flanders, Julia. "Detailism, Digital Texts, and the Problem of Pedantry." *TEXT Technology*, 2:2005, 41-70.

[4] Jockers, Matthew. *Macroanalysis: Digital Methods and Literary History*. UIUC Press, 2013.

Acknowledgements

For many years I taught text analysis courses using a combination of tools and different programming languages. For text parsing, I taught students to use `Perl`, `Python`, `php`, `Java`, and even `XSLT`. For analysis of the resulting data, we often used Excel. In about 2005, largely at the prompting of Claudia Engel and Daniella Witten who offered me some useful, beginner level advice, I began using `R`, instead of Excel. For too long after that I was still writing a lot of text analysis code in `Perl`, `Python` or `php` and then importing the results into `R` for analysis. In 2008 I decided this *workflow* was unsustainable. I was spending way too much time moving the data from one environment to another. I decided to go cold turkey and give up everything in favor of `R`. Since making the transition, I've rarely had to look elsewhere.

Luckily, just as I was making the transition to `R` so too were thousands of other folks; the online community of `R` programmers and developers was expanding at exactly the moment that I needed them. Today's online `R-help` resources are outstanding, and I could not have written this book without them. There are some phenomenal `R` programmers out there making some incredibly useful packages. Only a small handful of these packages are mentioned in this book (this is, after all, a beginner's guide) but without folks such as Stéfan Th. Gries, Harald Baayen and Hadley Wickham, this book and the `R` community would be very much the poorer. I'm amazed at how helpful and friendly the online `R` community has become; it wasn't that way in the early years, so I want to thank all of you who write packages and contribute code to the `R` project and also all of you who offer advice on the `R` forums, especially on the `R-help` list and at stackoverflow.com.

This book began as a series of exercises for students I was teaching at Stanford; they saw much of this content in a raw and less polished form. There are too many of them to thank individually, so a collective thanks to all of my former and current students for your patience and feedback. This book, whatever its faults and shortcomings, is a far better book than it might have been without you.

I first compiled the material from my classes into a manuscript in 2011, and since then I have shared bits and pieces of this text with a few colleagues. Stéfan Sinclair test drove an early version of this book in a course he taught at McGill. He and his

students provided valuable feedback. Maxim Romonov read most of this manuscript in early 2013. He provided encouragement and feedback and ultimately convinced me to convert the manuscript to `LaTeX` for better typesetting. This eventually led me to `Sweve` and `Knitr`; two `R` packages that allowed me to embed and run `R` code from within this very manuscript. So here again, my thanks to Maxim, but also to the authors of `Sweve` and `Knitr`, Friedrich Leisch and Yihui Xie.[5]

Finally, I want to thank you for grabbing a copy of this pre-print e-text and working your way through it. I hope that you will offer feedback and that you will help me make the final print version as good as possible. And when you do offer that feedback, I'll add your name to a list of contributors to be included in the print and online editions. If you offer a substantial contribution, I'll acknowledge it specifically.

I did not learn `R` by myself, and there is still a lot about `R` that I have to learn. I want to acknowledge both of these facts directly and specifically by citing all of you who take the time to contribute to this manuscript and to make `R` world a better place.

[5] Friedrich Leisch. Sweave: Dynamic generation of statistical reports using literate data analysis. In Wolfgang Härdle and Bernd Rönz, editors, Compstat 2002 - Proceedings in Computational Statistics, pages 575-580. Physica Verlag, Heidelberg, 2002. ISBN 3-7908-1517-9. Yihui Xie (2013) knitr: A Comprehensive Tool for Reproducible Research in R. In Victoria Stodden, Friedrich Leisch and Roger D. Peng, editors, Implementing Reproducible Computational Research. Chapman and Hall/CRC. ISBN 978-1466561595

Contributors

Your Name Here!

Contents

Part I
Microanalysis

Chapter 1
R Basics

Abstract

This chapter explains how to download and install `R` and `RStudio`. Readers are introduced to the `R` console and shown how to execute basic commands.

1.1 Introduction

There is a tradition in the teaching of programming languages in which students write a script to print out (to their screen) the words *hello world*. Though this book is about programming in `R`, this is not a programming book. Instead this text is designed to get you familiar with the `R` environment while engaging with, exploring, and even addressing some real literary questions. If you are like me, you probably launched `R` and started typing in commands a few hours ago. Maybe you got stuck, hit the wall, and are now turning to this book for a little shove in the right direction. If you are like me, you probably headed to the index first and tried to find some function or keyword (such as "frequency list") to get you rolling. You are ready to jump in and start working, and if you've ever done any coding before, you may be wondering (maybe dreading) if this is going to be another one of those books that grinds its way through all sorts of data type definitions and then finally "teaches" you how to write an elegant little snippet of code with a tight descriptive comment.

This is not that type of book–not that there is anything wrong with books that are like that! This book is simply different; it is designed for the student and scholar of literature who doesn't already know a programming language, or at the very least does not know the `R` language, and more importantly is a person who has come to `R` because of some literary question or due to some sense that computation might offer a new or particularly useful way to address, explore, probe, or answer some literary question. You are not coming to this book because you want to become a master programmer. You are a student or scholar in the humanities seeking to learn just enough to gain entry into the wide world of humanities computing.

If you want to become a master R programmer, or someone who delivers shrink-wrapped programs, or R packages, then this is not the book for you; there are other books, and good ones, for that sort of thing.[1] Here, however, I'll take my queues from best practices in natural language instruction and begin with a healthy dose of full immersion in the R programming language. In the first section, *Microanalysis*, I will walk you through the steps necessary to complete some basic text analysis of a single book. In the second part of the book, *Mesoanalysis*, we'll move from analysis of one or two texts to analysis of a small corpus. In the final section, *Macroanalysis*, we'll take on a larger corpus. Along the way there will be some new vocabulary and even some new or unfamiliar characters for your alphabet. But all in good time. For now, let's get programming. . .

1.2 R and RStudio

If you thrive on the command line and like the Spartan world of a UNIX-like environment, download the current version of R: instructions follow below under the heading "Download and Install R." If you'd like to work in a more developed GUI-like environment, follow the same instructions and then go directly to the section titled "Download and Install RStudio." Throughout this text I'll assume that you are working in the RStudio environment. You can work directly from the R console or even from your computer's terminal if you prefer, but RStudio is my recommended platform for writing and running R code, this is especially true for newbies.

1.3 Download and Install R

Download the current version of R (at the time of this writing version 3.0.1) from the CRAN website by clicking on the link that is appropriate to your operating system (see http://cran.at.r-project.org):

- If you use MS Windows, click on "base" and then on the link to the executable (i.e. ".exe") setup file (currently http://cran.at.r-project.org/bin/windows/base/R-3.0.1-win.exe).
- If you are running Mac OSX, choose the link to R-3.0.1.pkg (http://cran.at.r-project.org/bin/macosx/R-3.0.1.pkg)
- If you use Linux, choose your distribution and then the installer file.

Follow the instructions for installing R on your system in the standard or "default" directory. You will now have the base installation of R on your system.

[1] See, for example Venables, William N. and David M. Smith. *An Introduction to R*. Network Theory Ltd., 2004; Braun W. John and Duncan J. Murdoch. *A First Course in Statistical Programming with R*. Cambridge University Press, 2008; or any of the books in Springer's *Use R* Series: http://www.springer.com/series/6991?detailsPage=titles

- If you are using a Windows or Macintosh computer, you will find the R application in the directory on your system where Programs (Windows) or Applications (*Macintosh*) are stored. If you want to launch R, just double click the icon to start the R GUI
- If you are on a Linux/Unix system, simply type "R" at the command line to enter the R program environment.

1.4 Download and Install RStudio

The R application is fine for a lot of simple programming, but RStudio is an application that offers an organized user environment for writing and running R programs. RStudio is an IDE, that's "Integrated Development Environment" for R. RStudio runs happily on Windows, Mac, and Linux. After you have downloaded R (by following the instructions above in section 1.3) you can (and probably should) download the "Desktop" version (i.e. not the Server version) of RStudio from http://www.rstudio.com. Follow the installation instructions and then launch RStudio just like you would any other program/application.

1.5 Download the Supporting Materials

Now that you have R or R and RStudio installed and running on your system, you will also need to download the directory of files used for the exercises and examples in this book. The materials consist of a directory titled TextAnalysisWithR that includes the following sub-directories: a directory titled data containing a sub-directory of sample texts in plain text and another for texts in XML format, a directory labeled code for saving your R code, and a directory titled results for saving your derived results. You can download the supporting materials as a compressed zip file from the companion website:
http://www.matthewjockers.net/wp-content/uploads/2013/08/TextAnalysisWithR.zip.

Unzip the file and save the resulting directory (aka "folder") to a convenient location on your system: If you are using a Mac, the file path to this new directory might look something similar to the following:

```
~/Documents/TextAnalysisWithR
```

It does not matter where you keep this new directory as long as you remember where you put it. In the R code that you write, you will need to include information that tells R where to find these materials. All of the examples in this book place the TextAnalysisWithR directory inside of the main /Documents directory.

1.6 RStudio

In case it is not already clear, I'm a big fan of RStudio. When you launch the program you will see the default layout which includes four *quadrants* or *panes* and within each of the panes you can have multiple *tabs*. You can customize this pane/tab layout in RStudio's preferences area. I set my layout up a bit different from the default: I like to have the script editing pane in the upper right and the R console pane in the lower right. You will discover what is most comfortable for you as you begin to spend more time in the program.

The important point to make right now is that RStudio's four window panes each provide something useful and you should familiarize yourself with at least two of these panes right away. These are the script editing pane and the console pane. The former provides an environment in which you can write R programs. This pane works just like a text editor but with the added benefit that it offers *syntax highlighting* and some other shortcuts for interacting with R. As you become a more experienced coder, you will learn to love the highlighting. RStudio's script editor understands the syntax of the R programming language and helps you read the code you write by highlighting variables in one color, literal characters, comments, and so on in other colors. If this does not make sense to you at this point, that is fine. The benefits of syntax highlighting will become clear to you as we progress. A second point about the script editing pane is that anything you write in that pane can be saved to file. When you run commands in the R console, those commands do not get saved into a file that you can reuse.[2] When you write your code in the script editor, you intentionally save this code as a ".R" file. You can then close and reopen these files to run, revise, copy, etc.

Along with the scripting pane, RStudio provides a console pane. If you were simply running R by itself, then this is all you would get: a simple console. In RStudio you can interact with the console just as you would if you had only launched R. You can enter commands at the R *prompt* (represented by a > symbol at the left edge of the console), hit return and see the results.

Because the scripting and console panes are integrated in RStudio you can write scripts in the one pane and run them in the other without having to copy and paste code from the editor into the console. RStudio provides several shortcuts for running code directly from the script editing pane. We'll discuss these and other shortcuts later. For now just know that if you are working in the scripting pane, you can hit `command + return` to send the active line of code (i.e. where your cursor is currently active) directly to the console.

Throughout this book, I will assume that you are writing all of your code in the script editing pane and that you are saving your scripts to the `code` sub directory of the main TextAnalysisWithR directory you downloaded from my website. To help get you started, I'll provide specific instructions for writing and saving your files in

[2] This is not entirely true. RStudio does save your command history and, at least while your session is active, you can access that history and even save it to a file. Once you quit a session, however, that history may or may not be saved.

this first chapter. After that, I'll assume you know what you are doing and that you are saving your files along the way.[3]

1.7 Let's Get Started

If you have not already done so, launch RStudio (or launch R via the console application or by opening your terminal application and issuing the command "R" at your terminal prompt).

The first thing to do is to set TextAnalysisWithR as the "working directory." If you are using RStudio, you can save yourself some typing by going to the "Session" menu in the menu bar and selecting "Set Working Directory" and then "Choose Directory" (as in Figure 1.1). You can then navigate to the TextAnalysisWithR directory using your computer's file system browser. Once you have set the working directory in this manner, you can avoid entering full paths to the various directories and resources in the main directory.

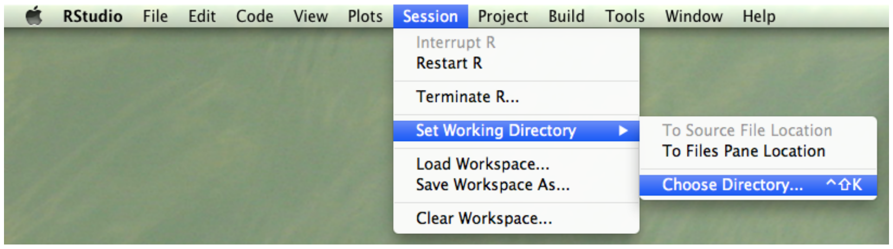


Fig. 1.1 Setting the Working Directory

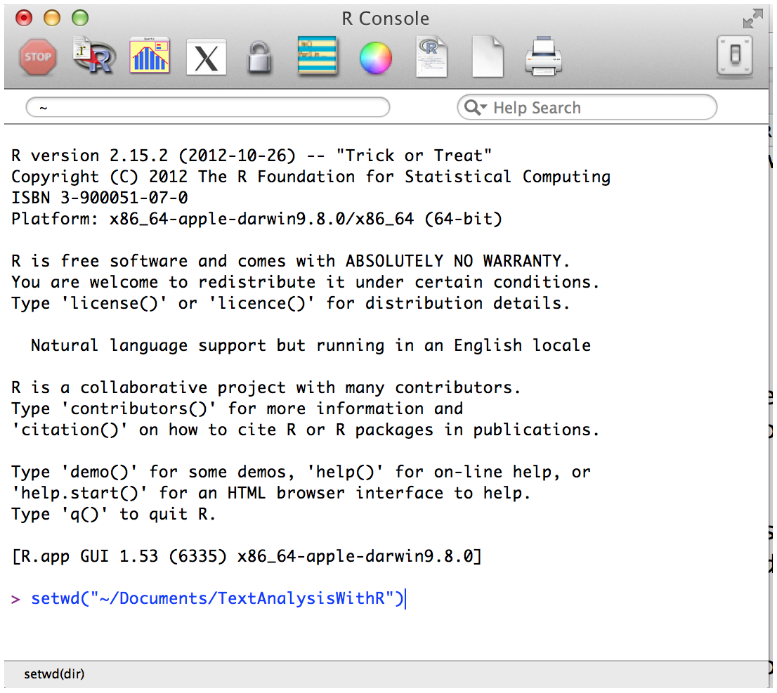
If you are determined to avoid using the RStudio IDE, you can use the setwd function to tell R where to look for the files we will use in this book. Open the R console application (Figure 1.2) and type the expression below (replacing my /Documents/TextAnalysisWithR with the actual path to your directory) into the R console[4].

```
> setwd("~/Documents/TextAnalysisWithR")
```

The > symbol you see here is the "R prompt," the place where you will enter your code (that is, you don't actually enter the >). In future code examples, I'll not include the R prompt >, which will make it easier for you to copy (and paste if you are using the e-book) code exactly as it appears here into your own R files. After entering the command, hit the return key. If you are working directly from the command line in your computer's terminal see Figure 1.3. From here on out all instructions and images will be relative to RStudio.

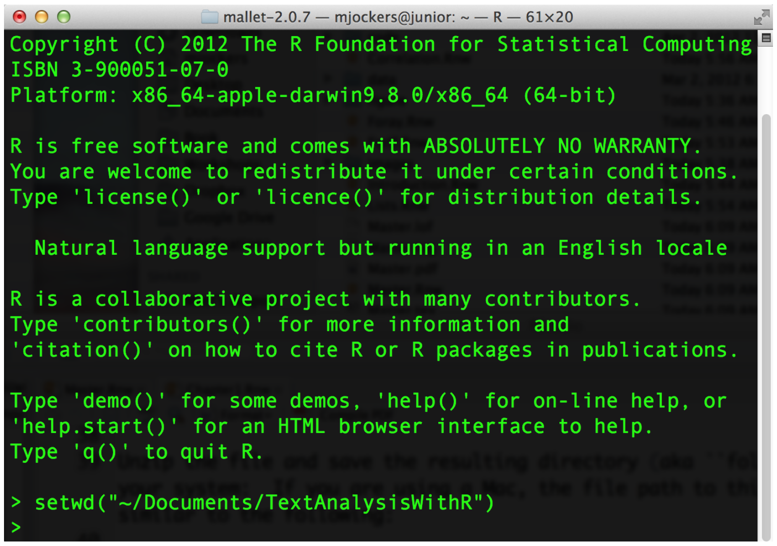
[3] If you do forget to save your file, you can always grab the textbook code from my online code repository or copy from Appendix C.

[4] Paths in R use forward slashes whether you are using Windows, Mac, or Linux



```
R version 2.15.2 (2012-10-26) -- "Trick or Treat"
Copyright (C) 2012 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: x86_64-apple-darwin9.8.0/x86_64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[R.app GUI 1.53 (6335) x86_64-apple-darwin9.8.0]

> setwd("~/Documents/TextAnalysisWithR")
```

Fig. 1.2 The R GUI Console



```
Copyright (C) 2012 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: x86_64-apple-darwin9.8.0/x86_64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> setwd("~/Documents/TextAnalysisWithR")
>
```

Fig. 1.3 R in the Terminal

Even though `RStudio` makes path setting and other common commands easy, you should remember that these are just GUI shortcuts that allow you to avoid typing commands directly into the `R` console. You can still always type directly into the `RStudio` *console* pane and get the same result. To set a working directory path in `RStudio` without using the GUI menu interface, just find the console pane and look for the > prompt. At the prompt, use the `setwd` command as described above, and you will have set the working directory without using `RStudio`'s more user-friendly GUI. It is good to know that you can always run `R` commands directly in `RStudio`'s console pane; `RStudio` gives you all the functionality of the command line as well as all the benefits of an IDE. Cool!

Practice

In order to gain some familiarity with the way that commands are entered into the `R` console, open a new `R` script file. In `RStudio` go to the *File* menu and choose *New* and then *R Script* (i.e. File > New > R Script). Save this file to your code directory as "exercise1.R.". Begin editing this new file by adding a call to the `setwd` function as described above. Now type each of the following exercise problems into your script. To compute the answers, you can either copy and paste each question into the console, or use the *command* + *return* shortcut to interact with the console pane from the script editor.

1.1. Simple Addition and Subtraction

```
10+5
10-5
```

1.2. An asterisk is used for multiplication

```
10*1576
```

1.3. A forward slash is used for division

```
15760/10
```

1.4. `R` has some values that are preset.

```
10+pi
10/pi
```

1.5. `R` uses the carat (^) for exponents

```
10^2
```

1.6. `R` uses the the less than symbol followed by the hyphen as an assignment operator. (The two symbols form an icon that looks like a left facing arrow.)

```
x <- 10  # Assign 10 to the variable "x"
x - 3 # subtract 3 from x
```

1.7. Beware of how `R` implements order in complex mathematical operations. The following expression correctly evaluates to 12. Can you explain why?

```
x <- 10
x - 3 + 10 / 2
```

1.8. If x <- 10 and x - 3 + 10 / 2 = 12 How would you rewrite the expression to = 8.5?

1.9. `R` has some built in mathematical functions:

```
sqrt(12)
abs(-23)
round(pi)
```

1.10. `R` can easily create sequences of numbers:

```
1:10
12:37
```

Chapter 2
First Foray into Text Analysis with R

Abstract
In this chapter readers learn how to load, tokenize, and search a text. Several methods for exploring word frequencies and lexical makeup are introduced. The exercise at the end introduces the `plot` function.

2.1 Loading the First Text File

If you have not already done so, set the working directory to the location of the supporting materials directory.

```
setwd("~/Documents/TextAnalysisWithR")
```

You can now load the first text file using the `scan` function:

```
text.v<-scan("data/plaintext/melville.txt", what="character", sep="\n")
```

Type this command into a new `R` script file and then run it by either copying and pasting into the console or using `RStudio's command + return` shortcut. This is as good a place as any to mention that the `scan` function can also load files from the internet. If you have an internet connection, you can enter a url in place of the file path and load a file directly from the web, like this:

```
text.v<-scan("http://www.gutenberg.org/cache/epub/2701/pg2701.txt",
             what="character", sep="\n")
```

Whether you load the file from your own system–as you will do for the exercises in this book–or from the internet, if the code has executed correctly, you should see the following result:

```
Read 18874 items
>
```

Remember that the > symbol seen here is simply a new `R` prompt indicating that `R` has completed its operation and is ready for the next command. At the new prompt, enter:

```
> text.v
```

You will see the entire text of *Moby Dick* flash before your eyes. Now try:

```
> text.v[1]
```

You will see the contents of the first item in the `text.v` variable, as follows:[1]

```
[1] "The Project Gutenberg EBook of Moby Dick;
or The Whale, by Herman Melville"
```

When you used the `scan` function, you included an *argument* (`sep`) that told the `scan` function to separate the file using `\n`. `\n` is a *regular expression* or *meta-character* (that is, a kind of computer shorthand) representing (or standing in for) the newline (carriage return) characters in a text file. What this means is that when the `scan` function loaded the text, it broke the text up into chunks according to where it found newlines in the text.[2] These chunks were then stored in what is called a *vector*, or more specifically a *character vector*.[3] In this single `R` expression, you invoked the `scan` function and put the results of that invocation into a new object named `text.v`, and the `text.v` object is an object of the type known as a character vector.[4] Deep breath.

It is important to understand that the data inside this new `text.v` object is *indexed*. Each line from the original text file has its own special container inside the `text.v` object, and each container is numbered. You can access lines from the original file by referencing their *container* or *index* number it within a set of square brackets. Entering

```
text.v[1]
```

returns the contents of the first container, in this case, the first line of the text, that is, the first part of the text file you loaded up to the first newline character. If you enter `text.v[2]`, you'll retrieve the second chunk, that is, the chunk between the first and second newline characters.[5]

[1] From this point forward, I will not show the `R` prompt in the code examples.

[2] Be careful not to confuse newline with "sentence" break or even with paragraph. Also note that depending on your computing environment, there may be differences between how your machine interprets *\n*, *\r* and *\n\r*.

[3] If you have programmed in another language, you may be familiar with this kind of data structure as an *array*.

[4] In this book and in my own coding I have found it convenient to append suffixes to my variable names that indicate the type of data being stored in the variable. So, for example, *text.v* has a *.v* suffix to indicate that the the variable is a vector: *v* = *vector*. Later you will see *data frame* variables with *.df* extensions and lists with a *.l*, and so on

[5] Those who have programming experience with another language may find it disorienting that `R` (like `FORTRAN`) begins indexing at `1` and not `0` like `C`, `JAVA` and many other languages.

2.2 Separate Content from Metadata

The electronic text that you just loaded into `text.v` is the plain text version of *Moby Dick* that is available from *Project Gutenberg*. Along with the text of the novel, however, you also get a lot of *Project Gutenberg* boilerplate metadata. Since you do not want to analyze the boilerplate, you need to determine where the novel begins and ends. As it happens, the main text of the novel begins at chunk `text.v[408]` and ends at `text.v[18576]`. One way to figure this out is to visually inspect the output you saw (above) when you typed the object name `text.v` at the `R` prompt and hit return. If you had lightning fast eyes, you might have noticed that chunk `text.v[408]` contained the text string: "CHAPTER 1. Loomings." and `text.v[18576]` contained the string "orphan." Instead of removing all this boilerplate text in advance, I opted to leave it in so that we might explore ways of accessing the chunks of a character vector.

Let's assume that you did not already know about chunks `408` and `18576`, but that you did know that the first chunk of the book contained "CHAPTER 1. Loomings" and that the last chunk of the book contained "orphan." We could use this information, along with `R`'s `which` function to isolate the main text of the novel. To do this on your own, enter the following code, but be advised that in `R`, and any other programming language for that matter, accuracy counts. The great majority of the errors you will encounter as you write and test your first programs will be the results of careless typing. [6]

```
start.v<-which(text.v == "CHAPTER 1. Loomings.")
end.v<-which(text.v == "orphan.")
```

In reality, of course, you are not likely to know in advance the exact contents of the chunks that `scan` created by locating the *newline* characters in the file, and the `which` function requires that you identify an exact match. Later on I'll show a better way of handling this situation by using the `grep` function. For now just pretend that you know where all the chunks begin and end. You can now see the chunk numbers (the vector indices) returned by the `which` function by entering the following at the prompt[7]:

```
start.v
end.v
```

You should now see the following returned from `R`:

```
start.v
## [1] 408
end.v
## [1] 18576
```

In a moment we will use this information to separate the main text of the novel from the metadata, but first a bit more about character vectors. . .

[6] In these expressions, the two equal signs serve as a comparison operator. You cannot use a single equals sign because the equals sign can also be used in `R` as an *assignment* operator. I'll clarify this point later, so for now just know that you need to use two equals signs to compare values.

[7] In `R` you can enter more than one command on a single line by separating the commands with a semi-colon. Entering `start.v;end.v` would achieve the same result as what you see here.

When you loaded *Moby Dick* into the `text.v` object, you asked `R` to divide the text (or delimit it) using the carriage return or *newline* character, which was represented using `\n`. To see how many newlines there are in the `text.v` variable, and, thus, how many chunks, use the `length` function:

```
length(text.v)
## [1] 18874
```

You'll see that there are 18874 lines of text in the file. But not all of these lines, or *strings*,[8] of text are part of the actual novel that you wish to analyze. *Project Gutenberg* files come with some baggage, and so you will want to remove the non-*Moby Dick* material and keep the story: everything from "Call me Ishmael..." to "...orphan." You need to reduce the contents of the `text.v` variable to just the lines of the novel proper. Rather than throwing out the metadata contained in the *Project Gutenberg* boilerplate, save it to a new variable called `metadata.v`, and then keep the text of the novel itself in a new variable called `novel.lines.v`. Enter the following four lines of code:

```
start.metadata.v<-text.v[1:start.v -1]
end.metadata.v<-text.v[(end.v+1):length(text.v)]
metadata.v<-c(start.metadata.v, end.metadata.v)
novel.lines.v<- text.v[start.v:end.v]
```

The first line takes lines 1 through 407 from the `text.v` variable and puts them into a new variable called `metadata.v`. If you are wondering where the `407` came from, remember that the `start.v` variable you created earlier contains the value `408` and refers to a place in the text vector that contains the words "`CHAPTER 1. Loomings.`" Since you want to keep that line of the text (that is, it is not metadata but part of the novel) you must subtract `1` from the value inside the `start.v` variable to get the `407`.

The second line does something similar; it grabs all of the lines of text that appear after the end of the novel by first adding one to the value contained in the `end.v` variable then spanning all the way to the last value in the vector, which can be calculated using `length`.

The third line then combines the two using the `c` or *combine* function. As it happens, I could have saved a bit of typing by using a short cut. The same result could have been achieved in one expression, like this:

```
metadata.v<-c(text.v[1:(start.v-1)], text.v[(end.v+1):length(text.v)])
```

Sometimes this kind of short-cutting can save a lot of extra typing and extra code. At the same time, you will want to be careful about too much of this function *embedding* as it can also make your code harder to read later on.

The forth line of the code block isolates the part of the electronic file that is in between the metadata sections, which is to say, the main novel. In this case I use `(end.v + 1)` because I want to keep the last line of the novel which is accessed at the point in the vector currently stored in the `end.v` variable. If this does not make sense, try entering the following to see just what is what:

```
text.v[start.v]
text.v[start.v-1]
text.v[end.v]
text.v[end.v+1]
```

[8] Sentences, lines of text, etc. are formally referred to as *strings* of text.

You can now compare the size of the original file to the size of the new `novel.lines.v` variable that excludes the boilerplate metadata:

```
length(text.v)
## [1] 18874
length(novel.lines.v)
## [1] 18169
```

If you want, you can even use a little subtraction to calculate how much smaller the new object is: `length(text.v) - length(novel.lines.v)`.

The main text of *Moby Dick* is now in an object titled `novel.lines.v`, but the text is still not quite in the format you need for further processing. Right now the contents of `novel.lines.v` are spread over 18169 line items derived from the original decision to delimit the file using the newline character. Sometimes, it is important to maintain line breaks: for example, some literary texts are encoded with purposeful line breaks representing the lines in the original print publication of the book or sometimes the breaks are for lines of poetry. For our purposes here, maintaining the line breaks is not important, so you will get rid of them using the `paste` function to *join* and *collapse* all the lines into one long string:

```
novel.v<-paste(novel.lines.v, collapse=" ")
```

The paste function with the `collapse` argument provides a way of *gluing* together a bunch of separate *pieces* using a *glue character* that you define as the value for the `collapse` argument. In this case, you are going to glue together the lines (the *pieces*) using a blank space character (the *glue*). After entering this expression, you will have the entire contents of the novel stored as a single string of words, or rather, a string of characters. You can check the size of the novel object by typing

```
length(novel.v)
## [1] 1
```

At first you might be surprised to see that the length is now `1`. The variable called `novel.v` is a vector just like `novel.lines.v`, but instead of having an indexed slot for each line, it has just one slot in which the entire text is held. If you are not clear about this, try entering:

```
novel.v[1]
```

A lot of text is going to flash before your eyes, but if you were able to scroll up in your console window to where you entered the command, you would see something like this:

```
[1] "CHAPTER 1. Loomings. Call me Ishmael. Some years ago..."
```

`R` has dumped the entire contents of the novel into the console. Go ahead, read the book;-)

2.3 Reprocessing the Content

Now that you have the novel loaded as a single string of characters, you are ready to have some fun. First use the `tolower` function to convert the entire text to lowercase characters.

```
novel.lower.v<-tolower(novel.v)
```

You now have a big blob of *Moby Dick* in a single, lowercase string, and this string includes all the letters, numbers, and marks of punctuation in the novel. For the time being, we will focus on the words, so you need to extract them out of the full text string and put them into a nice organized list. R provides an aptly named function for splitting text strings: `strsplit`.

```
moby.words.l<-strsplit(novel.lower.v, "\\W")
```

The `strsplit` function, as used here, takes two arguments and returns what R calls a *list*.[9] The first argument is the object (`novel.lower.v`) that you want to split, and the second argument `\\W` is a *regular expression*. A *regular expression* is a special type of character string that is used to represent a pattern. In this case, the regular expression will match any non-word character.[10] Using this simple *regex*, `strsplit` can detect *word boundaries*.

So far we have been working with vectors. Now you have a list. Both vectors and lists are data types, and R, like other programming languages, has other data types as well. At times you may forget what kind of data type one of your variables is, and since the operations you can perform on different R objects depends on what kind of data they contain, you may find yourself needing the `class` function. R's `class` function returns information about the data type of an object you provide as an argument to the function. Here is an example that you can try:

```
class(novel.lower.v)
## [1] "character"
class(moby.words.l)
## [1] "list"
```

To get even more detail about a given object, you can use R's `str` or *structure* function. This function provides a compact display of the internal structure of an R object. If you ask R to give you the structure of the `moby.words.l` list, you'll see the following:

```
str(moby.words.l)
## List of 1
##  $ : chr [1:253989] "chapter" "1" "" "loomings" ...
```

R is telling you that this object (`moby.words.l`) is a list with one item and that the one item is a character (`chr`) vector with `253,989` items. R then shows you the first few items, which happen to be the first few words of the novel.[11] If you look closely, you'll see that the third item in the `chr` vector is an empty string. We'll deal with that in a moment...

Right now, though, you may be asking, why a list? The short answer is that the `strsplit` function that you used to split the novel into words returns its results as a list. The long answer is that sometimes the object being given to the `strsplit` function is more complicated than a simple character string and so `strsplit` is designed to deal with more complicated situations.

[9] Because this new object is a list, I have appended ".l" to the variable name.

[10] WIKIPEDIA provides a fairly good overview of regular expression and a web search for "regular expressions" will turn up all manner of useful information.

[11] I have adopted a convention of appending a data type abbreviation to the end of my object names. In the long run this saves me from having to check my variables using *class* and *str*.

It is worth mentioning here that anytime you want some good technical reading about the nature of R's functions, just enter the function name proceeded by a question mark, e.g.: ?strsplit. This is how you access R's built in "help" files. Be forewarned that your mileage with R-help may vary. Some functions are very well documented and others are like reading tea leaves.[12] One might be tempted to blame poor documentation on the fact that R is open source, but I think it's more accurate to say that the documentation assumes a degree of familiarity with programming and with statistics. R-help is not geared toward the novice, but, fortunately, R has now become a popular platform, and if the built-in help is not always kind to newbies, the resources that are available online have become increasingly easy to use and newbie friendly.[13] For the novice, the most useful part of the built-in documentation is often found in the code examples that almost always follow the more technical definitions and explanations. Be sure to read all the way down in the help files, especially if you are confused. When all else fails, or even as a first step, consider searching for answers and examples on sites such as http://www.stackexchange.com.

Because you used strsplit, you have a list, and since you do not need a list for this particular problem, we'll simplify it to a vector using the unlist function:

```
moby.word.v<-unlist(moby.words.l)
```

When discussing the str function above, I mentioned that the third item in the vector was an empty character string. Calling str(moby.words.l) revealed the following:

```
## List of 1
##  $ : chr [1:253989] "chapter" "1" "" "loomings" ...
```

As it happens, that empty string between *1* and *loomings* is where the period character used to be. The \\W regular expression that you used to split the string ignored all the punctuation in the file, but then left these little blanks, as if to say, "if I'd kept the punctuation, it'd be right here."[14] Since you are ignoring punctuation, at

[12] *?functionName* is a shortcut for *R*'s more verbose *help(functionName)*. If you want to see an example of how a function is used, you can try *example(functionName)*. *args(functionName)* will display a list of arguments that a given function takes. Finally, if you want to search *R*'s documentation for a single keyword or phrase, try using *??("your keyword")* which is a shorthand version of *help.search("your keyword")*. I wish I could say that the documentation in R is always brilliant; I can't. It is inevitable that as you learn more about R you will find places where the documentation is frustratingly incomplete. In these cases, the internet is your friend, and there is a very lively community of R users who post questions and answers on a regular basis. As with any web searching, the construction of your query is something of an art form, perhaps a bit more so when it comes to R since using the letter *r* as a keyword can be frustrating.

[13] This was not always the case, but a recent study of the R-help user base shows that things have improved considerably. Trey Causey's analysis "Has *R*-help gotten meaner over time? And what does Mancur Olson have to say about it?" is available online at http://badhessian.org/2013/04/has-r-help-gotten-meaner-over-time-and-what-does-mancur-olson-have-to-say-about-it/

[14] There are much better, but more complicated, regular expressions that can be created for doing word tokenization. One downside to \\W is that it treats apostrophes as word boundaries. So the word *can't* becomes the words *can* and *t* and *John's* becomes *John* and *s*. These can be especially problematic if, for example, the eventual analysis is interested in negation or possession. You will learn to write a more sophisticated *regex* in a later chapter.

least for the time being, these blanks are a nuisance. You will want now to identify where they are in the vector and then remove them. Or more precisely, you'll identify where they are not!

First you must figure out which items in the vector are not blanks, and for that you can use the `which` function that was introduced previously.

```
not.blanks.v <- which(moby.word.v!="")
```

Notice how the `which` function has been used in this example. `which` performs a logical test to identify those items in the `moby.word.v` that are not equal (represented by the "`!=`" operator in the expression) to blank (represented by the empty quote marks `""` in the expression). If you now enter "`not.blanks.v`" into `R`, you will get a list of all of the *positions* in `moby.words.v` where there is not a blank. Try it:

```
not.blanks.v
```

If you tried this, you just got a screen full of numbers. Each of these numbers corresponds to a *position* in the `moby.words.v` vector where there is *not* a blank. If you scroll up to the top of this mess of numbers, you will find that the series begins like this:

```
not.blanks.v
[1] 1 2 4
```

Notice specifically that position `3` is missing. That is because the item in the third position was an empty string! If you want to see just the first few items in the not.blanks.v vector, try showing just the first ten items, like this:

```
not.blanks.v[1:10]
[1] 1 2 4 6 7 8 10 11 12 14
```

With the non-blanks identified, you can overwrite `moby.words.v` like this:[15]

```
moby.word.v<- moby.word.v[not.blanks.v]
```

Here only those items in the original `moby.words.v` that are not blanks are retained.[16] Just for fun, now enter:

```
moby.word.v
```

After showing you the first 99,999 words of the novel, `R` will give up and return a message saying something like `[[ reached getOption("max.print") - omitted 204889 entries ]]`. Even though `R` will not show you the entire vector, it is still worth seeing how the word data has been stored in this vector object. As a gut check, you may want to try the following:

```
moby.word.v[1:10]
##  [1] "chapter"  "1"        "loomings" "call"
##  [5] "me"       "ishmael"  "some"     "years"
##  [9] "ago"      "never"
```

[15] Overwriting a object is generally not a great idea, especially when you are writing code that you are unsure about, which is to say code that will inevitably need debugging. If you overwrite your variables, it makes it harder to debug later. Here I'm making an exception because I am certain that I'm not going to need the vector with the blanks in it.

[16] A shorthand version of this whole business could be written as `moby.word.v <- moby.word.v[which(moby.word.v != "")]`

The numbers in the square brackets are the *index* numbers showing you the position in the vector where each of the words will be found. `R` put a bracketed number at the beginning of each row. For instance the word “chapter” is stored in the first (`[1]`) position in the vector and the word “ishmael” is in the 6th (`[6]`) position. An instance of the word “ago” is found in the 9th position and so on. If, for some reason, you wanted to know what the 99986th word in *Moby Dick* is you could simply enter

```
moby.word.v[99986]
## [1] "by"
```

This is an important point (not that the 99986th word is *by*). You can access any item in a vector by referencing its index. And, if you want to see more than one item, you can enter a range of index values using a *colon* such as this:

```
moby.word.v[4:6]
## [1] "call"    "me"      "ishmael"
```

Alternatively, if you know the exact positions, you can enter them directly using the `c` combination function to create a vector of positions or index values. First enter this to see how the `c` works

```
mypositions.v<-c(4,5,6)
```

Now simply combine this with the vector:

```
moby.word.v[mypositions.v]
## [1] "call"    "me"      "ishmael"
```

You can do the same thing without putting the vector of values into a new variable. Simply use the `c` function right inside the square brackets:

```
moby.word.v[c(4,5,6)]
## [1] "call"    "me"      "ishmael"
```

Admittedly, this is only useful if you already know the index positions you are interested in. But, of course, `R` provides a way to find the indexes by also giving access to the contents of the vector. Say, for example, you want to find all the occurrences of *whale*. For this you can use the `which` function and ask `R` to find *which* items in the vector satisfy the condition of being the word *whale*.

```
which(moby.word.v=="whale")
```

Go ahead and enter the line of code above. `R` will return a list of index positions where the word *whale* was found. Now remember from above that if you know the index numbers, you can find the items stored in those index positions. Before entering the next line of code, see if you can predict what will happen.

```
moby.word.v[which(moby.word.v=="whale")]
```

2.4 Beginning the Analysis

Putting all of the words from *Moby Dick* into vector of words (or, more precisely, a *character* vector) provides a handy way of organizing all the words in the novel in chronological order; it also provides a foundation for some deeper quantitative analysis. You already saw how to find a word based on its position in the overall vector (the word *by* was the 99986th word). You then saw how you could use `which` to figure out which positions in the vector contain a specific word (the word *whale*). You might also wish to know how many occurrences of the word *whale* appear in the novel. Using what you just learned, you can easily calculate the number of *whale* tokens using `length` and `which` together:

```
length(moby.word.v[which(moby.word.v=="whale")])
## [1] 1150
```

Perhaps you would now also like to know the total number of tokens (words) in *Moby Dick*? Simple enough, just ask `R` for the `length` of the entire vector:

```
length(moby.word.v)
## [1] 214889
```

You can easily calculate the percentage of *whale* occurrences in the novel by dividing the number of whale *hits* by the total number of word tokens in the book. To divide, simply use the forward slash character.[17]

```
# Put a count of the occurecnes of whale into whale.hits.v
whale.hits.v<-length(moby.word.v[which(moby.word.v=="whale")])

# Put a count of total words into total.words.v
total.words.v<-length(moby.word.v)

# now divide
whale.hits.v/total.words.v
## [1] 0.005352
```

More interesting, perhaps, is to have `R` calculate the number of unique word types in the novel. `R`'s `unique` function will examine all the values in the character vector and identify those that are the same and those that are different. By combining the `unique` and `length` functions, you calculate the number of unique words in Melville's *Moby Dick* vocabulary.

```
length(unique(moby.word.v))
## [1] 16872
```

It turns out that Melville has a fairly big vocabulary: In *Moby Dick* Melville uses 16,873 unique words. That's interesting, but let's kick it up another notch. What we really want to know is how often he uses each of his words and which words are his favorites. We may even want to see if *Moby Dick* abides by Zipf's law regarding the general frequency of words in English.[18] No problem. `R`'s `table` function can be used to build a "contingency" table of word types and their corresponding frequencies.

```
moby.freqs.t<-table(moby.word.v)
```

[17] In these next few lines of code, I have added some comments to explain what the code is doing. This is a good habit for you to adopt; explaining or *commenting* your code so that you and others will be able to understand it later on. In *R* you insert comments into to code by using a *#* symbol before the comment. When processing your code, *R* ignores everything between that *#* and the next full line return.

[18] According to Zipf's law, the frequency of any word in a corpus is inversely proportional to its "rank" or position in the overall frequency distribution. In other words, the second most frequent word will occur about half as often as the most frequent word.

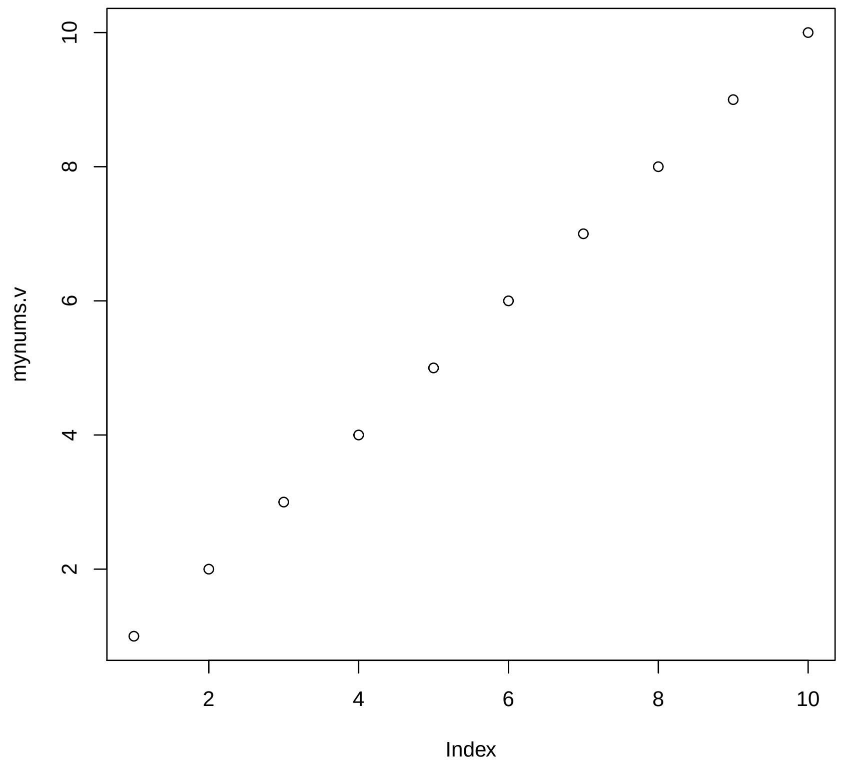
You can view the first few using `moby.freqs.t[1:10]`, and the entire frequency table can be sorted from most frequent to least frequent words using the `sort` function like this:

```
sorted.moby.freqs.t<-sort(moby.freqs.t , decreasing=T)
```

Practice

2.1. Having sorted the frequency data as described in section 2.4, figure out how to get a list of the top ten most frequent words in the novel? If you need a hint, go back and look at how we found the words "call" "me" "ishmael" in moby.word.v. Once you have the words, use R's built in plot function to visualize whether the frequencies of the words correspond to Zipf's law. The plot function is fairly straightforward. To learn more about the plot's complex arguments, just enter: ?plot. To complete this exercise, consider this example:

```
mynums.v<-c(1:10)
plot(mynums.v)
```



You only need to substitute the mynums.v variable with the top ten values from sorted.moby.freqs.t. You do not need to enter them manually!

Chapter 3
Accessing and Comparing Word Frequency Data

Abstract
In this chapter, we derive and compare word frequency data. We learn about vector recycling and the exercises invite you to compare the frequencies of several words in Melville's *Moby Dick* to the same words in Jane Austen's *Sense and Sensibility*.

3.1 Accessing Word Data

While it is no surprise to find that the word *the* is the most frequently occurring word in *Moby Dick*, it is a bit more interesting to see that the ninth most frequently occurring word is *his*. Put simply, there are not a lot of women in *Moby Dick*. In fact, you can easily compare the usage of *he* vs. *she* and *him* vs. *her* as follows: [1]

```
sorted.moby.freqs.t["he"]
##   he
## 1876
sorted.moby.freqs.t["she"]
## she
## 114
sorted.moby.freqs.t["him"]
##  him
## 1058
sorted.moby.freqs.t["her"]
## her
## 330
```

Notice how unlike the original `moby.word.v` in which each word was indexed at a *position* in the vector, here the word types *are* the indexes, and the values are the frequencies, or counts, of those word tokens. When accessing values in `moby.word.v`, you had to first figure out where in the vector those word tokens resided. Recall that you did this in two ways: you found "call," "me," and "ishmael" by entering

[1] This chapter expands upon code that developed in Chapter 2. Before beginning to work through this chapter, clear your workspace and run the *Chapter 3 starter code* found in Appendix C or in the online code repository.

```
moby.word.v[4:6]
```

and you found a whole pod of *whales* using the `which` function to test for the presence of *whale* in the vector.

With the *tabled* data object (`sorted.moby.freqs.t`), however, you get both numerical indexing and *named* indexing. In this way, you can access a value in the table either by its numerical position in the table or by its *name*. Thus, this expression

```
sorted.moby.freqs.t[1]
```

returns the same value as this one:

```
sorted.moby.freqs.t["the"]
```

These each return the same result because the word type *the* happens to be the first (`[1]`) item in the vector.

If you want to know just how much more frequent *him* is than *her*, you can use the / operator to perform division.

```
sorted.moby.freqs.t["him"]/sorted.moby.freqs.t["her"]
##   him
## 3.206
```

him is `3.2` times more frequent than *her*, but, as you'll see in the next code snippet, *he* is `16.5` times more frequent than *she*.

```
sorted.moby.freqs.t["he"]/sorted.moby.freqs.t["she"]
##    he
## 16.46
```

Often when analyzing text, what you really need are not the raw number of occurrences of the word types but the relative frequencies of word types expressed as a percentage of the total words in the file. Relativising in this way allows you to more easily compare the patterns of usage from one text to another. For example, you might want to compare Jane Austen's use of male and female pronouns to Melville's. Doing so requires compensating for the different lengths of the novels, so you convert the raw counts to percentages by dividing by a count of all of the words in each whole text. These are called *relative frequencies*.

As it stands you have a sorted table of raw word counts. You want to convert those raw counts to percentages, which requires dividing each count by the total number of word tokens in the entire text. You already know the total number of words because you used the `length` function on the original word vector.

```
length(moby.word.v)
## [1] 214889
```

It is worth mentioning, however, that you could also find the total by calculating the sum of all the raw counts in the tabled and sorted vector of frequencies.

```
sum(sorted.moby.freqs.t)
## [1] 214889
```

3.2 Recycling

You can convert the raw counts into relative frequency percentages using division and then a little multiplication by `100` (multiplication in `R` is done using the asterisks) to make the resulting numbers easier to read:

```
sorted.moby.rel.freqs.t<-100*(sorted.moby.freqs.t/sum(sorted.moby.freqs.t))
```

The key thing to note about this expression is that `R` understands that it needs to *recycle* the result of `sum(sorted.moby.freqs.t)` and apply that result to each and every value in the `sorted.moby.freqs.t` variable. This recycling also works with definite values. In other words, if you wanted to multiply every value in a vector by ten, you could do so quite easily. Here is a simple example for you to try.

```
num.vector.v <- c(1,2,3,4,5)
num.vector.v * 10
## [1] 10 20 30 40 50
```

Having applied the above calculation to the `sorted.moby.freqs.t` object, you can now access any word type and return its relative frequency as a percentage. Because you have multiplied by `100`, this percentage shows the number of occurrences per every `100` words.

```
sorted.moby.rel.freqs.t["the"]
##   the
## 6.596
```

the occurs `6.6` times for every `100` words in *Moby Dick*. If you want to plot the top ten words by their percentage frequency, you can use the `plot` function as you learned in exercise two. Here I'll add a few more arguments to `plot` in order to convey more information about the resulting image. The shape of the line in this plot is the same as in exercise two, but here the values on the y-axis have been converted from raw counts to counts per hundred.

```
plot(sorted.moby.rel.freqs.t[1:10], type="b",
     xlab="Top Ten Words", ylab="Percentage of Full Text", xaxt ="n")
axis(1,1:10, labels=names(sorted.moby.rel.freqs.t [1:10]))
```

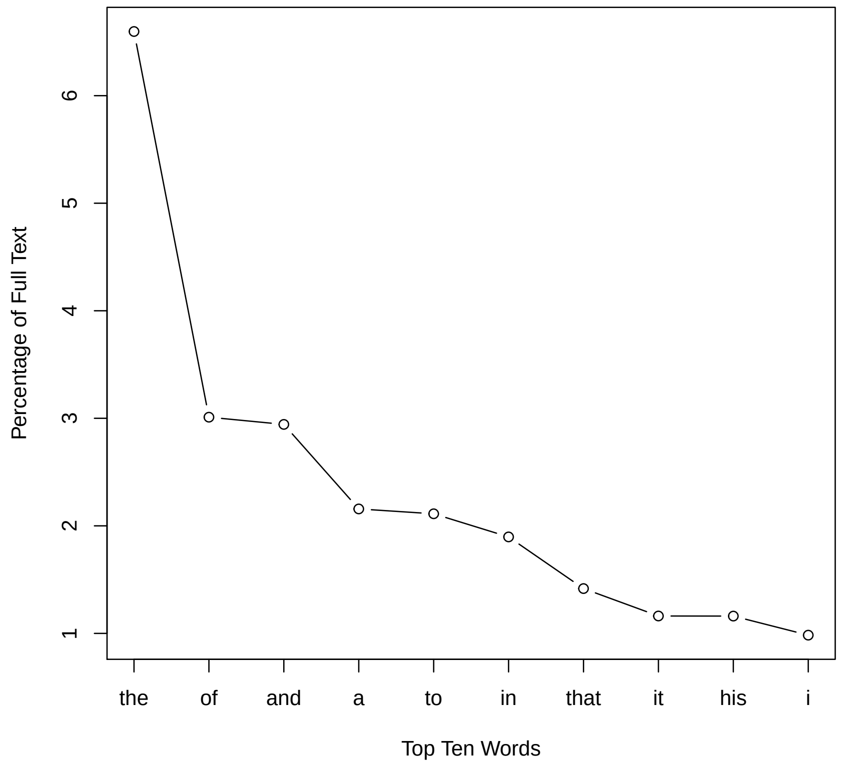


Fig. 3.1 Top Ten Words in *Moby Dick*

Practice

3.1. Top Ten Words in *Sense and Sensibility*
In the same directory in which you found *melville.txt*, locate *austen.txt* and produce a relative word frequency table for Austen's *Sense and Sensibility* that is similar to the one created in Exercise 1 using *Moby Dick*. Keep in mind that you will need to separate out the metadata from the actual text of the novel just as you did with Melville's text. Once you have the relative frequency table (i.e. `sorted.sense.rel.freqs.t`), plot it as above for *Moby Dick* and visually compare the two plots.

3.2. In the previous exercise, you should have noticed right away that the top ten most frequent words in *Sense and Sensibility* are not identical to those found in *Moby Dick*. You will also have seen that the order of words, from most to least frequent, is different and that the two novels share eight of the same words in the top ten. Using the `c` combination function join the names of the top ten values in each of the two tables and then use the `unique` function to show the 12 word types that occur in the combined name list. Hint: look up how to use the functions by entering a question mark followed by function name (i.e. `?unique`):

3.3. The *%in%* operator is a special *matching* operator that returns a logical (as in TRUE or FALSE) vector indicating if a match is found for its left operand in its right operand. It answers the question "is x found in y?" Using the `which` function in combination with the "%in%" operator, write a line of code that will compute which words from the two top ten lists are shared.

3.4. Write a similar line of code to show the words in the top ten of *Sense and Sensibility* that are *not* in the top ten from *Moby Dick*.

Chapter 4
Token Distribution Analysis

Abstract
This chapter explains how to use the positions of words in a vector to create distribution plots showing where words occur across a narrative. Several important `R` functions are introduced including `seq_along`, `rep`, `grep`, `rbind`, `apply`, and `do.call`. `If` conditionals and `for` loops are also presented.

4.1 Dispersion Plots

You've seen how easy it is to calculate the raw and relative frequencies of words. These are *global* statistics that show something about the central tendencies of words across a book as a whole. But what if you want to see exactly where in the text different words tend to occur; that is, where the words appear and how they behave over the course of a novel. At what points, for example, does Melville really *get into* writing about *whales*?

For this analysis you will need to treat the order in which the words appear in the text as a measure of time, *novelistic* time in this case. [1] If you do not already have the `moby.word.v` object loaded, go to the *Chapter 4 starter code* found in Appendix C (and in the online code repository) and regenerate `moby.word.v`.

You now need to create a sequence of numbers from 1 to `n`, where `n` is the position, or index number) of last word in *Moby Dick*. You can create such a sequence using the `seq` (*sequence*) function. For a simple sequence of the numbers, 1 to 10, you could enter:

```
seq(1:10)
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Instead of `one` through `ten`, however, you will need a sequence from `one` to the last word in *Moby Dick*. You already know how to get this number `n` using the

[1] For some very interesting work on modeling narrative time, see Mani, Inderjeet. *The Imagined Moment: Time, Narrative and Computation*. University of Nebraska Press, 2010

`length` function. Putting the two functions together allows for an expression like this:

```
n.time.v<-seq(1:length(moby.word.v))
```

This expression returns an integer vector (`n.time.v`) containing the positions of every word in the book.[2] I have titled this object `n.time.v` because it is a vector (`.v`) that will serve to represent narrative time (`n.time`) in the novel.

Now you need to locate the position of every occurrence of *whale* in the novel, or, more precisely, in the `moby.word.v` object. You have already learned how the `which` function can be used to locate items meeting certain conditions, so you can use `which` to identify the positions in the vector that are an occurrence of *whale* and store them in a new integer vector called `whales.v`:

```
whales.v<-which(moby.word.v == "whale")
```

If you now enter the object name (`whales.v`) into the console and hit enter, `R` will return a list of the numerical positions where it found instances of *whale*.

Ultimately you want to create a dispersion plot where the x-axis is novelistic time. You have those x-axis values in the `n.time.v` object. Another vector containing the values for plotting on the y-axis is now needed, and in this case, the values need only be some numerical reflection of the logical condition of `TRUE` where a *whale* is found and `FALSE` when an instance of *whale* is not found. In `R` you can represent the logical value `TRUE` with a number `1` and `FALSE` with a special character sequence–`NA`– as in "not available." Begin, therefore, by initializing a new vector object called "`w.count.v`" that will be full of `NA` values. It needs to be the same length as the `n.time.v` object, so you can use the `rep` or *repeat* function to repeat `NA` as many times as there are items in the `w.count.v` variable.

```
w.count.v<-rep(NA,length(n.time.v))
```

Now you simply need to reset the `NA` values to 1 in those places in the `moby.word.v` where a *whale* was found. You have those numerical positions stored in the `whales.v` object, so the resetting is simple with this expression:

```
w.count.v[whales.v]<-1
```

With the places where each *whale* was found now set to a value of `1` and everything else set to a value of `NA`, you can crank out a very simple plot showing the distribution of the word *whale* across the novel[3]:

This simple dispersion plot (figure 4.1) shows that the greatest concentration of the word *whale* occurs in what is, roughly, the third quarter of the novel. The first real concentration of *whale* begins just before `50,000` words, and then there is a fairly sustained *pod* of *whale* occurrences from `100,000` to about `155,000`, and then there is a final patch at the end of the novel, just after `200,000`.

[2] Remember that you can find out the data type of any `R` object using the `class` function. E.g. `class(n.time.v)`

[3] If you get an error saying: "`Error in plot.new() : figure margins too large`" you may not have enough screen real estate devoted to the plot pane of `RStudio`. You can solve this problem by increasing the size of the plots pane (just click and drag the frame). Your plot may also appear a lot taller (or thicker) than the one seen here. I have shrunk the plotting pane height in `RStudio` to make the image fit this page better.

```
plot(w.count.v, main="Dispersion Plot of `whale' in Moby Dick",
     xlab="Novel Time", ylab="whale", type="h", ylim=c(0,1), yaxt='n')
```

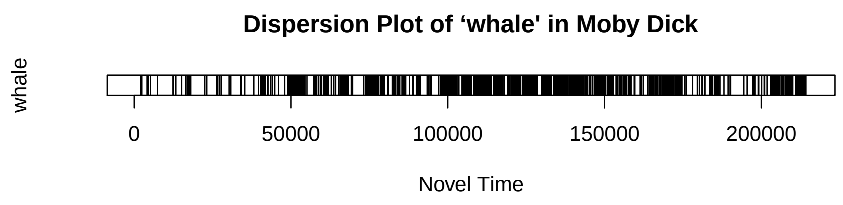


Fig. 4.1 Dispersion Plot of ‘whale’ in *Moby Dick*

By changing just a few lines of code, you can generate a similar plot (figure 4.2) showing the occurrences of *ahab*.

```
ahabs.v<-which(moby.word.v == "ahab") # find `ahab'
a.count.v<-rep(NA,length(n.time.v))
# change `w' to `a' to keep whales and ahabs in seperate variables
a.count.v[ahabs.v]<-1 # mark the occurences with a 1
plot(a.count.v, main="Dispersion Plot of 'ahab' in Moby Dick",
     xlab="Novel Time", ylab="ahab", type="h", ylim=c(0,1), yaxt='n')
```

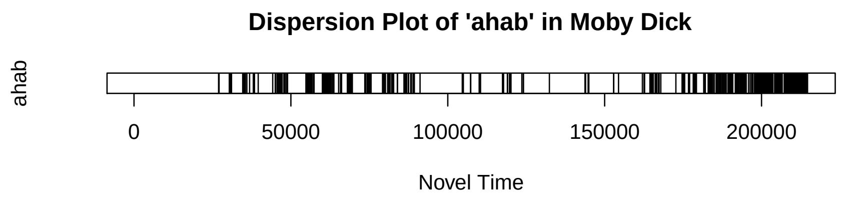


Fig. 4.2 Dispersion Plot of ‘ahab’ in *Moby Dick*

4.2 Searching with grep

Running the analysis for the word *ahab* shows that when *whale* is most present, the appearance of *ahab* is seemingly decreased. In terms of sheer presence, *whale* appears to dominate over *ahab* in the third quarter of *Moby Dick* in particular. Figure 4.3 shows the two plots on top of each other, and while these dispersion plots can be informative, more often than not novels have their own internal structure of chapters, and it can be more productive to honor the author’s own organization of the text.

We should, therefore, devise a method for examining how words appear across the novel's internal chapter structure.

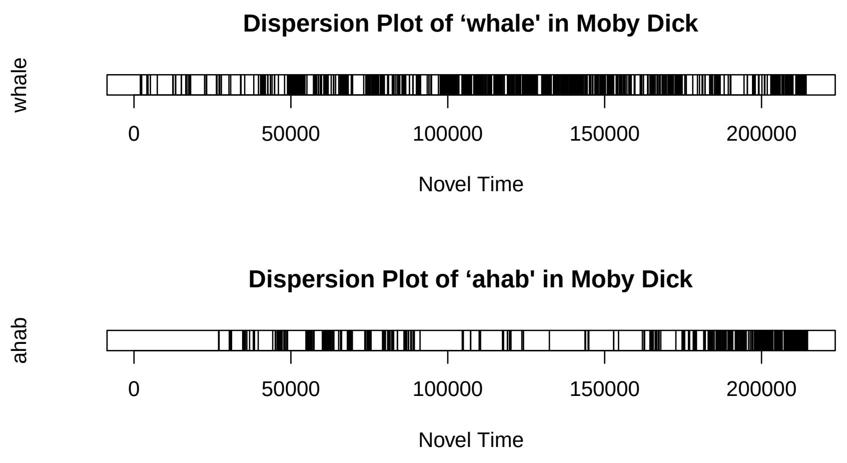


Fig. 4.3 Dispersion Plots for 'whale' and 'ahab' in *Moby Dick*

4.2.1 Cleaning the Workspace

Recall that you have a variable (`novel.lines.v`) containing the entire text from the original *Project Gutenberg* file as a list of lines. If you have had the same `R` session opened for a while, and especially if you have been experimenting with your own code as you work your way through the examples and exercises in this book, it might be a good idea to clear your *workspace*. As you work in a given `R` session, `R` is keeping track of all of your variables in memory. When you switch between projects during the same session, you can avoid a lot of potential variable conflict (and headache) by refreshing or clearing your workspace. Since you are about to do something with chapter breaks instead of looking at the novel as a single string, now is a good time to get a fresh start. In `RStudio` you can do this by selecting *Clear Workspace* from the *Session* menu. Alternatively, you can just enter `rm(list = ls())` into the `R` console. Be aware that both of these commands will delete all of your currently instantiated objects.

Enter the following expression to create a fresh session.

```
rm(list = ls())
```

If you now enter `ls()` you will simply see:

```
character(0)
```

ls is a *list* function that returns a list of all of your currently instantiated objects. This character(0) let's you know that there are no variables instantiated in the session.[4] Now that you have cleared everything, you'll need to reload *Moby Dick* using the same code you learned earlier:

```
text.v<-scan("data/plaintext/melville.txt", what="character", sep="\n")
start.v<-which(text.v == "CHAPTER 1. Loomings.")
end.v<-which(text.v == "orphan.")
novel.lines.v<- text.v[start.v:end.v]
```

You may recall from earlier that you can view the whole text of *Moby Dick* in your R console, newline by newline, by entering:

```
novel.lines.v
```

If you try this now, it might take a few seconds to load, and the results you'll see are not going to be very pretty.[5] One thing you might notice in this long list is that the beginning of each new chapter follows a specific *pattern*. Each new chapter starts with a new line followed by the capitalized word "CHAPTER" and then a space character and then one or more digits. For example,

```
[1] "CHAPTER 1. Loomings."
. . .
[185] "CHAPTER 2. The Carpet-Bag."
```

Because the *Project Gutenberg* text uses this *CHAPTER* convention to mark the chapters, you can *split* the text into chapters by using this character sequence (CHAPTER) as delimiter in a manner similar to the way that you split the text into words using \\W. First you'll need to convert the list to a vector.

```
novel.lines.v<-unlist(novel.lines.v)
```

Now identify the chapter break positions in the list using the grep function. In text analysis grep and its related functions are your ever-loyal friends. Be sure to access the grep help file by typing ?grep at the R prompt. And if you really want to get rolling, do a web search for "regular expressions" or what are known as *regex* for short. grep is an R function for performing regular expression pattern matching. Using the regular expression ^CHAPTER \\d will allow grep to identify lines in the vector that begin (the start of a line is marked by use of the caret symbol ^) with the capitalized letters CHAPTER followed by a space and then any digit (digits are represented using an escaped d, as in \\d). Here is the full expression.

```
chap.positions.v<-grep("^CHAPTER \\d", novel.lines.v)
```

To check your work, enter the next R expression:

```
novel.lines.v[chap.positions.v]
```

If grep and the *regex* did their job, you will now see a character vector containing all 135 of the chapter headings. Here is an abbreviated version showing only the first and last few items:

[4] If you are using RStudio, you can simply check the *workspace* window pane. After running rm(list = ls()) or clearing the workspace from the session menu, it will be blank.

[5] Do not be alarmed if you see a series of backslash characters in the text. These are escape characters that R adds before quotation marks and apostrophes so that they will not be treated as special characters and parsed by R.

```
[1] "CHAPTER 1. Loomings."
[2] "CHAPTER 2. The Carpet-Bag."
[3] "CHAPTER 3. The Spouter-Inn."
...
[133] "CHAPTER 133. The Chase--First Day."
[134] "CHAPTER 134. The Chase--Second Day."
[135] "CHAPTER 135. The Chase.--Third Day."
```

The object `chap.positions.v` holds the positions from the `novel.lines.v` where the search string `^CHAPTER\\d` was found. You must now find a way to collect all of the lines of text that occur *in between* these positions: the chunks of text that make up each chapter.

That sounds simple, but you do not yet have a marker for the *ends* of the chapters, you only know where they *begin*. To get the ends, you can subtract `1` from the known position of the following chapter. In other words, if CHAPTER 10 begins at position 1524, then you know that CHAPTER 9 ends at `1524 - 1` or `1523`.

This technique works perfectly except for the last chapter where there is no following chapter! There are several ways you might address this situation, but a simple solution is to just add one more position to the `chap.positions.v` vector. The position to add will be the position of the last line in `novel.lines.v`, and you find that position easily enough with the `length` function.

But let's slow down so that you can see exactly what is happening:

1. Enter `chap.positions.v` at the prompt to see the contents of the current vector:

```
chap.positions.v
##   [1]     1   185   301   790   925   989  1062
##   [8]  1141  1222  1524  1654  1712  1785  1931
##  [15]  1996  2099  2572  2766  2887  2997  3075
##  [22]  3181  3323  3357  3506  3532  3635  3775
##  [29]  3893  3993  4018  4084  4532  4619  4805
##  [36]  5023  5273  5315  5347  5371  5527  5851
##  [43]  6170  6202  6381  6681  6771  6856  7201
##  [50]  7274  7360  7490  7550  7689  8379  8543
##  [57]  8656  8742  8828  8911  9032  9201  9249
##  [64]  9293  9555  9638  9692  9754  9854  9894
##  [71]  9971 10175 10316 10502 10639 10742 10816
##  [78] 10876 11016 11097 11174 11541 11638 11706
##  [85] 11778 11947 12103 12514 12620 12745 12843
##  [92] 13066 13148 13287 13398 13440 13592 13614
##  [99] 13701 13900 14131 14279 14416 14495 14620
## [106] 14755 14835 14928 15066 15148 15339 15377
## [113] 15462 15571 15631 15710 15756 15798 15873
## [120] 16095 16113 16164 16169 16274 16382 16484
## [127] 16601 16671 16790 16839 16984 17024 17160
## [134] 17473 17761
```

2. Get the last position from the `length` function and add it to the `chap.positions.v` vector using the `c()` function:

```
last.position.v<- length(novel.lines.v)
chap.positions.v <- c(chap.positions.v , last.position.v)
```

3. Enter `chap.positions.v` at the prompt again, this time to see the entire vector but now with a new value (18169) appended to the end:

```
chap.positions.v
##   [1]     1   185   301   790   925   989  1062
##   [8]  1141  1222  1524  1654  1712  1785  1931
##  [15]  1996  2099  2572  2766  2887  2997  3075
##  [22]  3181  3323  3357  3506  3532  3635  3775
##  [29]  3893  3993  4018  4084  4532  4619  4805
##  [36]  5023  5273  5315  5347  5371  5527  5851
##  [43]  6170  6202  6381  6681  6771  6856  7201
##  [50]  7274  7360  7490  7550  7689  8379  8543
##  [57]  8656  8742  8828  8911  9032  9201  9249
##  [64]  9293  9555  9638  9692  9754  9854  9894
##  [71]  9971 10175 10316 10502 10639 10742 10816
##  [78] 10876 11016 11097 11174 11541 11638 11706
##  [85] 11778 11947 12103 12514 12620 12745 12843
##  [92] 13066 13148 13287 13398 13440 13592 13614
##  [99] 13701 13900 14131 14279 14416 14495 14620
## [106] 14755 14835 14928 15066 15148 15339 15377
## [113] 15462 15571 15631 15710 15756 15798 15873
## [120] 16095 16113 16164 16169 16274 16382 16484
## [127] 16601 16671 16790 16839 16984 17024 17160
## [134] 17473 17761 18169
```

The trick now is to figure out how to process the text, that is, the actual *content* of each chapter that appears in between each of these chapter markers. For this we will learn how to use a `for` loop.

4.3 The `for` Loop and `if` Conditional

Most of what follows from here will be familiar to you from what we have already learned about tokenization and word frequency processing. The main difference is that now all of that code will be wrapped inside of a looping function. A `for` loop allows us to do a task over and over again for a set number of iterations. In this case, the number of iterations will be equal to the number of chapters found in the text.

As a simple example, let's say you just want to print (to the screen) the various chapter positions you found using `grep`. Instead of printing them all at once, like you did above by dumping the contents of the `chap.positions.v` variable, you want to show them one at a time. You already know how to return specific items in a vector by putting an index number inside brackets, like this

```
chap.positions.v[1]
## [1] 1
chap.positions.v[2]
## [1] 185
```

Instead of entering the vector indexes (1 and 2 in the example above), you can use a `for` loop to go through the entire vector and *automate* the bracketing of the index numbers in the vector. Here's a simple way to do it using a `for` loop:

```
for(i in 1:length(chap.positions.v)){
  print(chap.positions.v[i])
}
```

Notice the `for` loop syntax; it includes two arguments inside the parentheses: a variable (`i`) and a sequence (`1:length(chap.positions.v)`). These are followed by a set of opening and closing *braces*. These braces contain (or encapsulate) the instructions to perform within each iteration of the loop.[6] Upon the first iteration, `i` gets set to `1`. With `i == 1` the program prints the contents of whatever value is held in the 1st position of `chap.positions.v`. In this case, the value is `1`, which can be a bit confusing. When the program gets to the second iteration, the value printed is `185`, which is less confusing. After each iteration of the loop, `i` is advanced by `1`, and this looping continues until `i` is equal to the length of `chap.positions.v`.

To make this even more explicit, I'll add a `paste` function that will print the value of `i` along with the value of the chapter position, making it all easy to read and understand. Try this now.

```
for(i in 1:length(chap.positions.v)){
  print(paste("Chapter ",i, " begins at position ",
              chap.positions.v[i]), sep="")
}
```

When you run this loop, you will get a clear sense of how the parts are working together.

With that example under your belt, you can now return to the chapter-text problem. As you iterate over the `chap.positions.v` vector, you are going to be grabbing the text of each chapter and performing some analysis. Along the way, you don't want to print the results to the `R` console (as in my example above), so you will need a place to store the results of the analysis during the `for` loop. For this you will create two empty list objects. These will serve as containers in which to store the calculated result of each iteration:

```
chapter.freqs.l<-list()
chapter.raws.l<-list()
```

A list is a special type of object in `R`. You can think of a list as being like a file cabinet. Each drawer is an item in the list and each drawer can contain different kinds of objects. In my file cabinet, for example, I have three drawers full of file folders and one full of old diskettes, CDs and miscellaneous hard drives. You will learn more about lists as we go on.

To summarize, the `for` loop will need iterate over each item in the `chap.positions.v` vector. When it gets to each item, it will use the chapter position information stored in the vector to figure out the *beginning* and the *end* of each chapter. With the chapter boundaries found, the script will then collect the word tokens found within those boundaries and calculate both the *raw* and *relative* frequencies of those word types using the `table` function that you learned about earlier. The frequencies (both the raw counts and the relative frequencies) will then be stored in the two list variables that are instantiated prior to the loop. The processing inside the loop is similar to what was done when plotting the occurrences of *whale* over the course of all the words in the text. The difference is that now you are plotting values by chapter.

Though the tasks are similar, there are one or two complicating factors that must be addressed. The most problematic of these involves what to do when `i` is equal

[6] Using `i` is a matter of convention. You could name this variable anything that you wish: e.g. `my.int`, `x`, etc.

to the length of chap.positions.v. Since there is no text following the last position, you need a way to break out of the loop. For this an if conditional is perfect. Below I have written out the entire loop. Before moving on to my line-by-line explication, take a moment to study this code and see if you can explain each step.

```
for(i in 1:length(chap.positions.v)){
  if(i != length(chap.positions.v)){
    chapter.title<-novel.lines.v[chap.positions.v[i]]
    start<-chap.positions.v[i]+1
    end<-chap.positions.v[i+1]-1
    chapter.lines.v<-novel.lines.v[start:end]
    chapter.words.v<-tolower(paste(chapter.lines.v, collapse=" "))
    chapter.words.l<-strsplit(chapter.words.v, "\\W")
    chapter.word.v<-unlist(chapter.words.l)
    chapter.word.v<-chapter.word.v[which(chapter.word.v!="")]
    chapter.freqs.t<-table(chapter.word.v)
    chapter.raws.l[[chapter.title]]<- chapter.freqs.t
    chapter.freqs.t.rel<-100*(chapter.freqs.t/sum(chapter.freqs.t))
    chapter.freqs.l[[chapter.title]]<-chapter.freqs.t.rel
  }
}
```

Now that you've had a chance to think through the logic of this loop for yourself, here is a line-by-line explication:

1.
```
for(i in 1:length(chap.positions.v)){
```

Initiate a for loop that iterates over each item in chap.positions.v.

2.
```
  if(i != length(chap.positions.v)){
```

As long as the value of i is not equal to the length of the vector, keep iterating over the vector. Here I introduce the conditional if. if allows me to set a condition that will evaluate to either TRUE or FALSE. If the condition is found to be TRUE, then the code inside the curly braces of the if statement will be executed. This has the effect of saying "so long as this condition is met, continue iterating." The condition here is that i not be equal (!=) to the length of the vector. The reason I must set this condition is because there is no chapter text after the last item in chap.positions.v. I do not want to keep the loop going once it gets to the end!

Assuming that the condition stated in the if statement is met, I proceed to the next line. At this stage the program pauses to capture the chapter title which is found at the place in the novel.lines.v indicated by the value held in the chap.positions.v.

3.
```
    chapter.title<-novel.lines.v[chap.positions.v[i]]
```

If this is confusing, try this: In your console, set i to 1.

```
    i<-1
```

Now enter:

```
novel.lines.v[chap.positions.v[i]]
```

When you hit return, you will see:

```
[1] "CHAPTER 1. Loomings."
```

If that is still not clear, you can break it down even further, like this:

```
i<-1
chap.positions.v[i]
[1] 1
novel.lines.v[chap.positions.v[i]]
[1] "CHAPTER 1. Loomings."
i<-2
chap.positions.v[i]
[1] 185
novel.lines.v[chap.positions.v[i]]
[1] "CHAPTER 2. The Carpet-Bag."
```

4.
```
start<-chap.positions.v[i]+1
```

I know that the title of the chapter is at the `i`th line in `novel.lines.v`, so I can add `1` to `i` and get the values of the next line in the vector. `i+1` will give me the position of the first line of the chapter text (i.e. excluding the chapter title).

5.
```
end<-chap.positions.v[i+1]-1
```

What is done here is a bit more subtle. Instead of adding `1` to the value held in the `ith` position of `chap.positions.v`, I must to add `1` to `i` in its capacity as an *index*. Instead of grabbing the value of the `ith` item in the vector, the program is going to grab the value of the *item* in the *next* position beyond `i` in the vector. If this isn't clear, you can break it down like this:

```
i<-1
chap.positions.v[i]
[1] 1
chap.positions.v[i+1]
[1] 185
```

When `i==1`, the value held in `chap.positions.v[i]` will be `1` because `1` happens to be the first value stored in the vector. When `i == i+1`, in this case `2`, `R` will return the value held in the 2nd position in `chap.positions.v`, or `185`. In the next iteration, `i` will be `2` and so `[i+1]` will be `3` and the result will be `301`, which is the third value stored in the vector.
`chap.positions.v[i+1]` will return the next item in the vector, and the value held in that spot is the position for the start of a new chapter. Since I do not want to count the words in the chapter heading, I must subtract `1` from that value in order to get the line number in `novel.lines.v` that comes just before the start of a new chapter. Thus I subtract `1` from the value found in the `[i+1]` position.

6.
```
chapter.lines.v<-novel.lines[start:end]
chapter.words.v<-tolower(paste(chapter.lines.v, collapse=" "))
chapter.words.l<-strsplit(chapter.words.v, "\\W")
chapter.word.v<-unlist(chapter.words.l)
chapter.word.v<-chapter.word.v[which(chapter.word.v!="")]
chapter.freqs.t<-table(chapter.word.v)
```

These lines should be familiar to you from previous sections. With `start` and `end` points defined, you grab the lines, paste them into a single block of text, lowercase everything and then split it all into a vector of words that is tabulated into a frequency count of each word type.

7.
```
      chapter.raws.l[[chapter.title]]<- chapter.freqs.t
```

This next line is where the resulting table of raw frequency counts is stuffed into the list that was created before entering the loop. The double bracketing here is used to assign a *name* or *label* to the list item, and here each item in the list is named with the chapter heading extracted a few lines above. It is not *necessary* to assign labels to list items in this way. If you leave out the label, the list will just be created with numerical indexes. The utility of this labeling will become clear later on.

8.
```
      chapter.freqs.t.rel<-100*(chapter.freqs.t/sum(chapter.freqs.t))
      chapter.freqs.l[[chapter.title]]<-chapter.freqs.t.rel
    }
  }
```

The last two lines simply convert the raw counts to relative frequencies based on the number of words in the chapter. This relative frequency table is then stuffed into the other list object that was created before entering the `for` loop.

4.4 Accessing and Processing List Items

With the two lists now populated with data, a way of accessing the data and putting it into a usable structure that allows for easy comparisons of word frequencies across chapters is needed. For this you will learn to use three functions: `rbind`, `lapply`, and `do.call` and along the way you will learn something more about `vector recycling`.

4.4.1 `rbind`

`rbind` is the simplest of the three functions introduced in this section. As the name suggests, `rbind` is a function for *binding* rows of data together. For `rbind` to work, the rows being bound must have the same number of columns. Enter the following `R` code into your console window:

```
x<-c(1,2,3,4,5)
y<-c(6,7,8,9,10)
```

These expressions create two vectors of five numerical values each. If you now use `rbind` to combine them, you get a matrix object with two rows and 5 columns.

```
rbind(x,y)
##   [,1] [,2] [,3] [,4] [,5]
## x    1    2    3    4    5
## y    6    7    8    9   10
```

Notice, however, what happens when I recreate the y vector so that x and y are not of the same length:

```
y<-c(6,7,8,9,10, 11)
rbind(x,y)

## Warning: number of columns of result is not a multiple of vector length (arg 1)

##   [,1] [,2] [,3] [,4] [,5] [,6]
## x    1    2    3    4    5    1
## y    6    7    8    9   10   11
```

First, R reports a warning message that the vectors were not of the same length. In R, a warning is just a warning; your script did not fail to execute. In fact, you now have a sixth column. Take a moment to experiment with this example and see if you can figure out what R is doing when it has two vectors of different lengths.

4.4.2 More Recycling

What you should have discovered is something called *recycling*. The recycling occurs because you are binding vectors of differing lengths. At some point R discovers that the shorter vector is at its end and that the longer vector still has uncombined elements. So R simply returns to the beginning of the shorter vector and begins *recycling* its elements. R will keep recycling from the longer vector until it reaches the end of the process. The elements of the shorter vector will be reused over and over again until the process is complete. Sometime this recycling is incredibly useful. Say you want to multiply every item in one vector by a value held in some other vector. Here, for example, I multiply each number in the x vector by the number held in the y vector.[7]

```
x<-c(1,2,3,4,5,6)
y<-2
x*y
## [1]  2  4  6  8 10 12
```

This recycling can get a bit confusing when you have more complicated vectors. In the example above, each value in the x vector is multiplied by the value in the y vector. When the y vector contains more than one item then the recycling gets a bit more complicated. Consider this example:

```
x<-c(1,2,3,4,5,6)
y<-c(2, 3)
x*y
## [1]  2  6  6 12 10 18
```

[7] It might seem a bit odd, but in R even objects containing only one item are vectors. So in this example the y object is a vector of one item. If you simply enter y into the console, you'll get a bracketed number 1 [1] followed by the value 2, which is the value held in the first position of the y vector.

Here, the 2 and 3 get recycled over and over again in order such that the first item in the x vector is multiplied by first item in the y vector (the number 2), the second item in the x vector is multiplied by second item in the y vector (the number 3). But then when R gets to the third item in the x vector it recycles the y vector by going back to the first item in the y vector (the number 2) again. Deep breath.

4.4.3 `apply`

lapply is one of several functions in the apply family. lapply (with an “l” in front of “apply”) is specifically designed for working with lists. Remember that you have two lists that were filled with data using a for loop. These are:

```
chapter.freqs.l
chapter.raws.l
```

lapply is similar to a for loop. Like for, lapply is a function for iterating over the elements in a data structure. The key difference is that lapply requires a list as one of its arguments, and it requires the name of some other function as one of its arguments. When run, lapply returns a new list object of the same length as the original one, but it does so after having applied the function it was given in its arguments to *each* item of the original list. Say, for example, that you have the following list called x:

```
x <- list(a = 1:10, b = 2:25, b=100:1090)
```

This is a list of three integer objects each containing a series of numbers. Enter x at the R prompt to look at the contents of the x list. Basically, x is like a file cabinet with three drawers, and each of the drawers contains is an integer vector. If you now enter: lapply(x, mean) R will return a new list in which the function (mean) is applied to each object in the list called x

```
lapply(x, mean)
## $a
## [1] 5.5
##
## $b
## [1] 13.5
##
## $b
## [1] 595
```

R has calculated the mean for each of the integer vectors in the x list.

Now consider the construction of the lists you filled up using the for loop. Each list contains a series of frequency tables. Each item in chapter.raws.l contains a table of raw counts of each word type in each chapter, and each list item in chapter.freqs.l contains a table of the relative frequencies of each word type in a chapter.

If you want to know the relative frequency of the word type whale in the first chapter of *Moby Dick*, you could get the value using bracketed sub setting, like this:

```
chapter.freqs.l[[1]]["whale"]
```

This expression tells `R` that you want to go to the first item in the `chapter.freqs.l` list (list items are accessed using the special double bracket `[[]]` notation), which is a frequency table of all the words from chapter one, i.e.

```
chapter.freqs.l[[1]]
```

But you also instruct `R` to return only those values for the word type `whale`. Try it for yourself:

```
chapter.freqs.l[[1]]["whale"]
##  whale
## 0.1337
```

The result indicates that the word *whale* occurs `0.13` times for every one hundred words in the first chapter. Since you know how to get this data for a single list item, it should not be too hard then to now use `lapply` to grab the `whale` values from the entire list. In fact, you can get that data by simply entering this:

```
lapply(chapter.freqs.l, '[', 'whale')
```

Well, OK, I'll admit that using `[` as the function argument here is not the most intuitive thing to do, and I'll admit further that knowing you can send another argument to the main function is even more confusing. So let's break this down a bit. The `lapply` function is going to handle the iteration over the list by itself. So basically, `lapply` handles calling each chapter from the list of chapters. If you wanted to do it by hand, you'd have to do something like this:

```
chapter.freqs.l[[1]]
chapter.freqs.l[[2]]
. . .
chapter.freqs.l[[135]]
```

By adding `[` as the *function* argument to `lapply`, you tell `lapply` to "apply bracketed sub setting" to each item in the list. Recall again that each item in the list is a table of word counts or frequencies. `lapply` allows us to add another *optional* argument to the function that is being called, in this case the function is "bracketed sub-setting." When you send the key word *whale* in this manner, then behind the scenes `R` executes code for each item that looks like this:

```
chapter.freqs.l[[1]]["whale"]
chapter.freqs.l[[2]] ["whale"]
. . .
chapter.freqs.l[[135]] ["whale"]
```

If you enter a few of these by hand, you'll begin to get a sense of where things are going with `lapply`.

Here is how to put all of this together in order to generate a new list of the *whale* values for each chapter.

```
whale.l<-lapply(chapter.freqs.l, '[', 'whale')
```

Instead of just printing out the values held in this new list, you can capture the results into a single matrix using `rbind`.

One option would be to `rbind` each item in the `whale.l` list object by hand: something like what follows here (but with more than the just the first three list items):

```
rbind(whale.l[[1]], whale.l[[2]], whale.l[[3]])
```

While this method works, it is not very scaleable or elegant. Fortunately `R` has another function for just this kind of problem: the function is `do.call` and is pronounced *do dot call*.

4.4.4 `do.call` (do dot call*)*

Like `lapply`, `do.call` is a function that takes another function as an argument. In this case the other function will be `rbind`. The `do.call` function will take `rbind` as an input argument and call it over the different elements of the list object. Consider this very simple application of the `do.call` function: First create a list called `x` that contains 3 integer vectors.

```
x <- list(1:3,4:6,7:9)
x
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] 4 5 6
##
## [[3]]
## [1] 7 8 9
```

To convert this list into a matrix where each row is one of the vectors and each column is one of the three integers from each of the list items, use `do.call`

```
do.call(rbind,x)
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

Using `do.call` in this way binds the contents of each list item *row-wise*.

The list of *whale* occurrences in *Moby Dick* is fairly similar to the list (`x`) that was used in this example. In some ways `whale.l` is even simpler than `x` because each integer vector only contains one item. You can use `do.call`, therefore, to activate the `rbind` function across the list of *whale* results. Doing this will generate a matrix object of 135 chapter *rows* by `1` *column* of relative frequency values. A *matrix* object is like a very simple spreadsheet; a matrix has rows and columns. One special (or limiting) thing about matrix objects in `R`, however, is that they can only contain one type of data. That is, they cannot contain text values in one column and numerical values in another. `R` has another object for handling mixed data, and we'll cover that later on. For now, just think about a matrix as a simple spreadsheet. I'll call this new matrix `whales.m`:

```
whales.m<-do.call(rbind, whale.l)
```

After you have entered this expression, look at the results by entering `whales.m` at the `R` prompt. Here is an abbreviated version of how the results should look:

```
whales.m
CHAPTER 1. Loomings.  0.13368984
CHAPTER 2. The Carpet-Bag.  0.06882312
CHAPTER 3. The Spouter-Inn.  0.10000000
CHAPTER 4. The Counterpane.  NA
. . .
CHAPTER 133. The Chase--First Day.  0.76965366
CHAPTER 134. The Chase--Second Day.  0.61892131
CHAPTER 135. The Chase.--Third Day.  0.67127746
```

Using what you have learned thus far, you can create another matrix of chapter-by-chapter values for occurrences of *ahab*. The only thing you need to change in the code described already is the keyword: you'll use *ahab* in place of *whale*:

```
ahab.l<-lapply(chapter.freqs.l, '[', 'ahab')
ahabs.m<-do.call(rbind, ahab.l)
```

4.4.5 cbind

With both `whales.m` and `ahabs.m` instantiated in memory, you can easily bind them together *column-wise* using `cbind`. But since these are both very simple matrix objects, you might as well put them into an even simpler vector objects first.

It is easy to convert any column of values into a vector by applying the `as.vector()` function to the matrix column. As noted above, a matrix is like a spreadsheet with rows and columns. You can access any *cell* in the matrix by identifying its row and column number. Here is a simple matrix created by `cbind`-ing several vectors together:

```
x<-c(1,2,3,4,5,6)
y<-c(2,4,5,6,7,8)
z<-c(24,23,34,32,12,10)
test.m<-cbind(x,y,z)
test.m
##      x y  z
## [1,] 1 2 24
## [2,] 2 4 23
## [3,] 3 5 34
## [4,] 4 6 32
## [5,] 5 7 12
## [6,] 6 8 10
```

To access the value held in the second row and third column in this matrix, you use bracketed sub-setting similar to what you have been using when accessing values in a vector. Here, you will need to put both row and column information into the brackets.

```
test.m[2,3] # show the value in the second row third column
##  z
## 23
```

Inside the brackets `2` was entered to indicate the row number. This was followed by a comma and then a `3` to indicate the third column. If you wanted to see an entire row or an entire column, you would just leave the field before or after the comma empty:

```
test.m[2,] # show all values in the second row
##  x  y  z
##  2  4 23
test.m[,1] # show all values in the first column
## [1] 1 2 3 4 5 6
```

It is also worth knowing that if the columns have names, you can access them by name. By default `cbind` names columns based on their original variable name:

```
test.m[,"y"]
## [1] 2 4 5 6 7 8
```

Now that you know how to access specific values in a matrix, you can easily pull them out and assign them to new objects. You can pull out the `whale` and `ahab` values into two new vectors like this:

```
whales.v<-as.vector(whales.m[,1])
ahabs.v<-as.vector(ahabs.m[,1])
```

You can now use `cbind` to bind these vectors into a new, two-column matrix. The resulting matrix will have `135` rows and `2` columns, a fact you can check using the `dim` function.

```
whales.ahabs.m<-cbind(whales.v, ahabs.v)
dim(whales.ahabs.m)
## [1] 135   2
```

Previously I mentioned that by default `cbind` titles columns based on the input variable names. You can reset the column names manually using the `colnames` function in conjunction with the `c` function. To rename the two columns in this example, use this expression:

```
colnames(whales.ahabs.m)<-c("whale", "ahab")
```

Once you have reset the column names, you can plot the results side by side using the `barplot` function.

```
barplot(whales.ahabs.m, beside=T, col="grey")
```

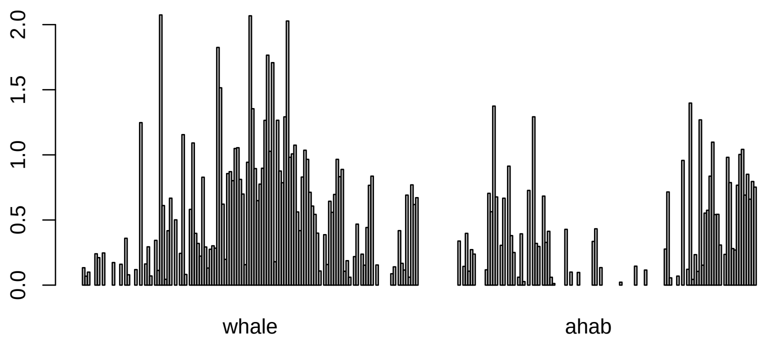


Fig. 4.4 Bar Plot of ‘whale’ and ‘ahab’ Side by Side

Practice

4.1. Write code that will find the value for another word (i.e. queequeg) and then bind those values as a third column in the matrix. Plot the results as in the example code.

4.2. These plots were derived from the list of relative frequency data. Write a script to plot the raw occurrences of *whale* and *ahab* per chapter.

Chapter 5
Correlation

Abstract

This chapter introduces data frames, random sampling, and correlation. Readers learn how to perform permutation tests to assess the significance of derived correlations.

5.1 Introduction

It might be tempting to look at the graphs you have produced thus far and begin forming an argument about the relative importance of Ahab versus the whale in Melville's novel. Occurrences of *whale* certainly appear to occupy the central portion of the book whereas *Ahab* is present at the beginning and at the end. It might also be tempting to begin thinking about the structure of the novel, and this data does provide some evidence for an argument about how the human dimensions of the narrative frame the more naturalistic. But is there, in fact, an inverse relationship?

5.2 Correlation Analysis

Using the frequency data you compiled for *ahab* and *whale*, you can run a correlation analysis to see if there is a statistical connection. A correlation analysis attempts to determine the extent to which there is a relationship, or linear dependence, between two sets of points. Thought of another way, correlation analysis attempts to assess the way that the occurrences of *whale* and *ahab* behave in unison or in opposition to each other over the course of the novel. You can use a correlation analysis to answer a question such as: to what extent does the usage of *whale* change (increase or decrease) in relation to the usage of *ahab*? `R` offers a simple function, `cor`, for calculating the strength of a possible correlation. But before you can employ the cor function on the whales.ahabs.m object, you need to deal with the fact that there are some cells in the matrix that contain the value NA

Not every chapter in *Moby Dick* had an occurrence of *whale* (or *ahab*), so in the previous practice exercise when you ran

```
whale.l <- lapply(chapter.freqs.l, "[", "whale")
```

R found no *hits* for *whale* in some chapters of the novel and recorded an NA, as in *not available* or *missing*. You may recall seeing this NA output when you viewed the contents of whales.ahabs.m matrix:

```
whales.ahabs.m[1:16, ]
##          whale   ahab
##  [1,] 0.13369     NA
##  [2,] 0.06882     NA
##  [3,] 0.10000     NA
##  [4,]      NA     NA
##  [5,]      NA     NA
##  [6,] 0.24067     NA
##  [7,] 0.21097     NA
##  [8,]      NA     NA
##  [9,] 0.24712     NA
## [10,]      NA     NA
## [11,]      NA     NA
## [12,]      NA     NA
## [13,] 0.17341     NA
## [14,]      NA     NA
## [15,]      NA     NA
## [16,] 0.16037 0.3386
```

As you see here, there are no occurrences of *whale* in chapters four or five and no occurrences of *ahab* until chapter sixteen. Because cor is a mathematical function that requires numerical data, you need to replace with these NA values before running the correlation analysis. Since the appearance of an NA in these cases is equivalent to *zero* (there are exactly *zero* occurrences of the keyword in the given chapter), you can safely replace all the occurrences of NA in the whales.ahabs.m matrix with zero. One way to do this is by embedding the conditional is.na function inside a call to the which function as in: which(is.na(whales.ahabs.m)). To set the values to 0, place the entire expression inside the brackets of whales.ahabs.m and assign a 0 to those items that meet the condition:

```
whales.ahabs.m[which(is.na(whales.ahabs.m))]<-0
```

This is the short and easy way to do it, but for the sake of illustration here it is broken down with comments added to explain what is going on:

```
# identify the position of NA values in the matrix
the.na.positions<-which(is.na(whales.ahabs.m))
# set the values held in the found positions to zero
whales.ahabs.m[the.na.positions]<-0
```

With the NAs set to zero, the correlation can be run.

```
cor(whales.ahabs.m)
##          whale    ahab
## whale  1.0000 -0.2411
## ahab  -0.2411  1.0000
```

Because whales.ahabs.m is a matrix of two columns, the result of calling cor is a new matrix containing two rows and two columns. The row and column names are the same and the values held in the cells are the correlation values. It's no surprise to see that *whale* is perfectly correlated with *whale* and *ahab* with *ahab*. The positive `1.000` in these cells is not very informative, which is to say that running `cor` over the entire matrix as I've done here results in a lot of extraneous information. That's because `cor` runs the correlation analysis for every possible combination of columns in the matrix. With a two column matrix like this, it's really overkill. The results could be made a lot simpler by just giving `cor` the two vectors that you really want to correlate:

```
mycor<-cor(whales.ahabs.m[,"whale"], whales.ahabs.m[,"ahab"])
mycor
## [1] -0.2411
```

The resulting number (-0.2411) is a measure of the strength of linear dependence between the values in the *whale* column and the values in the *ahab* column. This result, called the *Pearson Product-moment correlation coefficient*, is expressed as a number between `-1` and `+1`. A negative one (`-1`) coefficient represents perfectly negative correlation; if the correlation between *ahab* and *whale* were `-1`, then we would know that as the usage of *whale* increases, the usage of *ahab* decreases proportionally. Positive one (`+1`) represents perfect positive correlation (as one variable goes up and down the other variable does so in an identical way). Zero (`0`) represents no correlation at all.

The further the coefficient is from zero, the stronger the correlation; conversely the closer the result is to `0`, the less dependence there is between the two variables. Here, with *whale* and *ahab* a correlation coefficient of -0.2411 is observed. This suggests that while there is a slightly inverse relationship, it is not strongly correlated since the result is closer to `0` than to `-1`. Having said that, how one interprets the meaning, or significance, of the correlation indicated by this coefficient is largely dependent upon the context of the analysis and upon the number of observations or data points under consideration. Generally speaking a coefficient between `-0.3` and `-0.1` on the negative side of `0` and between `0.1` and `0.3` on the positive side of `0` is considered quite small. Strong correlation is usually seen as existing at levels less than `-0.5` or greater than `0.5`.

These two data points, for *ahab* and *whale*, appear to show only weak inverse correlation and thus lead us only to further hypotheses and further testing. Unsurprisingly, this correlation test does not lead us to any easy conclusions about the relationship between occurrences of *whale* and occurrences of *ahab*. Obviously, there is much more to be considered.

Consider, for example, how the use of pronouns complicates these results. When *Ahab* is not being referred to by name, he is undoubtedly appearing as either *he* or *him*. The same may be said for the *whale* and the various appellations of *whale* that Melville evokes: *monster, leviathan, etc.* Using the techniques described above, you could investigate all of these and more. But before leaving this seemingly weak correlation, it might be useful to run a few more experiments to see just how significant or insignificant the result really is.

As noted above, the number of samples can be a factor in how the importance of the correlation coefficient is judged, and in this case there are `135` observations for each variable: one observation for each chapter in the novel.

One way of contextualizing this coefficient is to calculate how likely it is that we would observe this coefficient by mere chance alone. In other words, assuming there is no relationship between the occurrences of *whale* and *ahab* in the novel, what are the chances of observing a correlation coefficient of -0.2411? A fairly simple test can be constructed by randomizing the order of the values in either the *ahabs* or the *whales* column and then retesting the correlation of the data.

5.3 A Word about Data Frames

Before explaining the randomization test in detail, I want to return to something mentioned earlier (See Chapter 4, section 5) about the R matrix object and its limitations and then introduce you to another important data object in R: the *data frame*.

Thus far we have not used data frames, but as it happens, data frames are R's bread and butter data type, and they offer us some flexibility that we do not get with matrix objects. Like a matrix, a data frame can be thought of as similar to a table in a database or a sheet in an Excel file: a data frame has rows and some number of columns, and each column contains a specific type of data. A major difference between a matrix and a data frame, however, is that in a data frame, one column may contain character values and another numerical values. To see how this works, enter the following code to create a simple matrix of three rows by three columns:

```
x<-matrix(1, 3, 3)
x
##      [,1] [,2] [,3]
## [1,]    1    1    1
## [2,]    1    1    1
## [3,]    1    1    1
```

If you ask R to return the data type (class) of any one of the values in this matrix, it will return the class *numeric*.

```
class(x[1,2]) # get class of cell in first row second column
## [1] "numeric"
```

Now change the value of one cell in this matrix so that it contains character data instead of a number.

```
x[1,2]<-"Sam I am"
x
##      [,1] [,2]       [,3]
## [1,] "1"  "Sam I am" "1"
## [2,] "1"  "1"        "1"
## [3,] "1"  "1"        "1"
```

You will notice right away that all of the value in the matrix are now shown inside quotation marks. This is because the entire matrix has been converted to character data. Those 1s are no longer numbers, they are the 1 *character*. Among other things, this means that you cannot perform mathematical operations on them any more! If you check the class, R will report the change:

```
class(x[1,2]) # get class of cell in first row second column
## [1] "character"
class(x[1,3]) # get class of cell in first row third column
## [1] "character"
```

To see the difference between a matrix and a data frame, recreate the first matrix example and then convert it to a data frame, like this:

```
x<-matrix(1, 3, 3)
x.df<-as.data.frame(x)
x.df
##   V1 V2 V3
## 1  1  1  1
## 2  1  1  1
## 3  1  1  1
```

You can see right away that a data frame displays differently. Instead of bracketed row and column numbers you now see column headers (V1, V2, V3) and simple row numbers without the brackets. You can now repeat the experiment from above and assign some character data into one of the cells in this data frame.

```
x.df[1,2]<-"Sam I am"
class(x.df[1,2])# get class of cell in first row second column
## [1] "character"
class(x.df[1,3])# get class of cell in first row third column
## [1] "numeric"
x.df
##   V1       V2 V3
## 1  1 Sam I am  1
## 2  1        1  1
## 3  1        1  1
```

When using a matrix, the assignment of character data to any one cell resulted in all the cells in the matrix being converted into character data. Here, with a data frame, only the data in the column containing the target cell are converted to character data, not the entire table of data. The takeaway is that a data frame can have columns containing different types of data. This will be especially useful as your data get more complicated. You may, for example, want a way of storing character based metadata (such as *author gender*, or *chapter title*) along side the numerical data associated with these metadata facets.

Another handy thing about data frames is that you can access columns of data using a bit of `R` short hand. If you want to see all the values in the second column of the `x.df` variable, you can do so using bracketed index references, just as you have done previously with matrix objects. To see the entire second column, for example, you might do this:

```
x.df[, 2]
## [1] "Sam I am" "1"        "1"
```

Alternatively, you can use the fact that the data frame has a header to get column information by referencing the column name, like this:

```
x.df[,"V2"]
## [1] "Sam I am" "1"        "1"
```

And, most alternatively, you can use the shorthand (`$`) to get column data like this:

```
x.df$V2
## [1] "Sam I am" "1"        "1"
```

That is a basic overview of data frames. now we can return to correlating values in *Moby Dick*.

5.4 Testing Correlation with Randomization

In this section you will use your new knowledge of data frames. First convert the matrix object `whales.ahabs.m` into a data frame called `cor.data.df`:

```
# make life easier and convert to a data frame
cor.data.df<-as.data.frame(whales.ahabs.m)
```

As a gut check, you can use the `cor` function on the entire data frame, just as you did with the matrix object. The output should be the same.

```
cor(cor.data.df)
##          whale    ahab
## whale  1.0000 -0.2411
## ahab  -0.2411  1.0000
```

The goal now is to determine if that observed correlation coefficient of `-0.2411` could have been likely to occur by mere chance. To assess this you are going to take the values for one of the two columns in the data frame and *shuffle* them into a random order. Once in random order, you'll run a new correlation test. In this way a chance distribution of the values that is independent of the actual structure of the chapters in the book can be simulated. If the correlation of the shuffled data is similar to the actual (as in *unshuffled*) data, then you'll have to concede that the relationship between *whale* and *ahab* observed in the actual data is really no different from what might be observed if you threw all the occurrences of *whale* and *ahab* up in the air and then created `135` arbitrary piles.

The first step is to randomize the order of the values (the word frequency measurements) in one of the two columns of data in `cor.data.df`. Since the columns contain chapter-by-chapter measurements, this randomizing will have the effect of shuffling the chapter order for one set of measurements and leaving the other set in chronological order. `R` provides a function called `sample` for generating a random shuffling of data. At its most simple, the `sample` function requires a vector of values to shuffle. So, to get a random ordering of the values in the *whale* column of `cor.data.df` you can simply enter:

```
sample(cor.data.df$whale)
##   [1] 0.96567 1.35440 0.30193 1.70807 0.06882
##   [6] 0.00000 0.00000 0.13115 2.02788 0.35971
##  [11] 0.19763 0.71283 0.00000 0.00000 0.87623
##  [16] 0.00000 0.21901 1.09152 0.06079 0.00000
##  [21] 0.27548 0.23791 0.00000 0.76965 0.06079
##  [26] 1.00768 0.83276 0.15829 0.80201 0.00000
##  [31] 0.00000 2.07453 0.00000 0.94340 1.03578
##  [36] 0.39920 0.08749 0.08208 0.00000 0.00000
##  [41] 1.24777 0.89726 1.04895 0.38722 0.54306
##  [46] 0.41841 0.50125 0.98160 0.88832 0.69124
##  [51] 0.24375 0.82988 0.46838 0.82840 0.61100
##  [56] 0.64401 0.00000 0.41841 2.06782 0.00000
##  [61] 0.85653 1.05485 0.22272 0.00000 0.29412
##  [66] 0.10000 0.00000 0.77566 0.00000 0.17341
##  [71] 0.13369 0.14035 0.32017 0.16722 0.24712
##  [76] 0.66760 0.18762 0.58168 0.55866 0.00000
##  [81] 0.00000 1.26506 1.15546 0.00000 0.69930
##  [86] 0.07949 0.00000 0.67128 0.16037 0.00000
##  [91] 0.69620 0.39761 1.02740 0.16260 1.26582
##  [96] 0.81169 0.87131 0.78616 0.56191 0.04448
## [101] 0.64879 0.28392 0.00000 0.34364 0.10638
## [106] 0.29297 0.62176 0.00000 0.11581 0.07047
## [111] 1.29199 1.76565 0.15314 1.82482 0.89485
## [116] 0.44248 0.11287 0.00000 0.00000 0.21097
## [121] 0.15723 0.00000 0.17943 1.51515 0.24067
## [126] 0.96562 0.61892 0.10858 0.76628 1.07469
## [131] 0.41793 0.60716 0.11926 0.15486 0.83682
```

Go ahead and try entering this a few times and you will see that each time `sample` randomly shuffles the order of the values from the *whale* column.

With the ability to randomize the values, you now need to correlate these randomized values against the ordered values in the unshuffled *ahab* column. Using the dollar sign to reference the columns in the data frame, the expression can be written as simply as this:

```
cor(sample(cor.data.df$whale), cor.data.df$ahab)
```

In my first test of this code, `R` returned a correlation coefficient of -0.0869.[1] I copied and pasted the code ten more times and observed the following correlation coefficients for the various shuffles of the data.

```
cor(sample(cor.data.df$whale), cor.data.df$ahab)
[1] 0.122331
cor(sample(cor.data.df$whale), cor.data.df$ahab)
[1] 0.00818978
cor(sample(cor.data.df$whale), cor.data.df$ahab)
[1] -0.01610114
cor(sample(cor.data.df$whale), cor.data.df$ahab)
[1] -0.1289073
cor(sample(cor.data.df$whale), cor.data.df$ahab)
[1] 0.05115036
cor(sample(cor.data.df$whale), cor.data.df$ahab)
[1] 0.0443622
cor(sample(cor.data.df$whale), cor.data.df$ahab)
[1] 0.08513762
cor(sample(cor.data.df$whale), cor.data.df$ahab)
[1] -0.1019796
cor(sample(cor.data.df$whale), cor.data.df$ahab)
[1] 0.07842781
cor(sample(cor.data.df$whale), cor.data.df$ahab)
[1] 0.04410211
```

As you see, in this small sample of ten randomizations, the highest positive correlation coefficient was `0.122331` and as lowest negative correlation coefficient was `-0.1289073`. Remember that the actual correlation coefficient before we began shuffling anything was `-0.2411`. In other words, the actual data seems to be quite a bit below (i.e. further from `0`) what is observed when shuffling the data and simulating a chance distribution of values. Still, `10` randomizations is not very many. Instead of copying and pasting the code over and over again, you can develop a more programmatic way of testing the correlation using a `for` loop and `10,000` iterations!

With a `for` loop you can repeat the randomization and correlation test process multiple times and at each iteration capture the result into a new vector. With this new vector of `10,000` correlation values, it will be easy to generate some statistics that describe the distribution of the random data and offer a better way of assessing the significance of the actual observed correlation coefficient in the unshuffled data.

[1] Your results will be different given the sampling.

The code required for this is simple. Begin by creating an empty container variable called `mycors.v`, and then create a `for` loop that iterates a set number of times (`10,000` in my example). Within the curly braces of that loop, you'll add code for shuffling and then correlating the vectors. At each step in the loop, you will capture the correlation coefficient by adding it to the `mycors.v` vector using the `c` function. Here is how I wrote it:

```
mycors.v<-NULL
for(i in 1:10000){
  mycors.v<-c(mycors.v, cor(sample(cor.data.df$whale), cor.data.df$ahab))
}
```

once run, you can now use some basic `R` functions such as `min, max, range, mean` and `sd` to get a general sense of the results. Here is what my randomization tests returned; your results will be similar but not identical:

```
min(mycors.v)
## [1] -0.2772
max(mycors.v)
## [1] 0.3129
range(mycors.v)
## [1] -0.2772  0.3129
mean(mycors.v)
## [1] -0.0001494
sd(mycors.v)
## [1] 0.08685
```

What these descriptive statistics reveal is that our actual observed value is more typical of the extremes than the norm. A low `standard deviation` suggests that most of the values recorded are close to the `mean`, and here the `mean` is very close to zero (-1.4941×10^{-4}), which you will recall from above can be interpreted as meaning very little correlation. A high `standard deviation` would indicate that the values are spread out over a wide range of values. So even though the `min` value of `-0.2772` is slightly less than our actual observed value of `-0.2411`, that -0.2772 is very *atypical* of the randomized data. In fact, using a bit of additional code that I will not explain here, I can generate a plot showing the distribution of all the values in `mycors.v` (Figure 5.1).

The plot reveals, in dramatic fashion, just how much the data clusters around the `mean`, which as you recall from above is nearly `0`. It also dramatizes the outlier status of the actual value (`-0.2411`) that was observed. In `10,000` random iterations, only `10` correlation coefficients were calculated to be less than the actual observed value and the actual observed value was nearly `3` (`2.8`) `standard deviations` away from the `mean`. In short, the probability of observing a random value as extreme as the actual value observed (-0.2411) is just 0.4%.[2]

[2] Another way to test the significance of a correlation coefficient is to use the `cor.test` function. Use `?cor.test` to learn about this function and then run it using the `method="pearson"` argument. To make more sense out of the results, consider consulting http://en.wikipedia.org/wiki/P-value on p-values and HTTP://EN.WIKIPEDIA.ORG/WIKI/T-TEST on t-tests.

```
h<-hist(mycors.v, breaks=100, col="grey",
        xlab="Correlation Coefficient",
        main="Histogram of Random Correlation Coefficients\n
        with Normal Curve",
        plot=T)
xfit<-seq(min(mycors.v),max(mycors.v),length=1000)
yfit<-dnorm(xfit,mean=mean(mycors.v),sd=sd(mycors.v))
yfit <- yfit*diff(h$mids[1:2])*length(mycors.v)
lines(xfit, yfit, col="black", lwd=2)
```

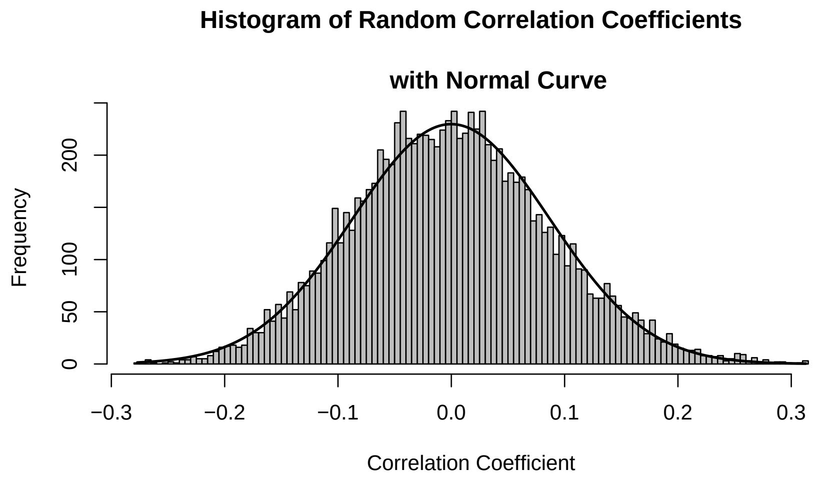


Fig. 5.1 Histogram Plot of Random Correlation Coefficients

Practice

5.1. Add two more columns to the matrix with data for the words *i* and *my* and then rerun the `cor` function. What does the result tell you about the usage of the words *i* and *my*?

5.2. Calculate the correlation coefficient for *i* and *my* and run a randomization test to evaluate whether the results are significant.

Part II
Mesoanalysis

Chapter 6
Measures of Lexical Variety

Abstract
In this chapter we'll begin to transition from microanalysis to macroanalysis. We'll leave behind the study of single terms and begin to explore two global measures of lexical variety: mean word frequency and type-token ratios.

6.1 Lexical Variety and the Type-Token Ratio

Moby Dick is a complicated book with a complex vocabulary. Readers of the book, inevitably remember chapter 32. This is the cetology chapter in which Melville offers a zoological and pseudo-scholarly, pseudo-comical account of whale history and physiology. Students frequency complain that this section of the novel is more complex or *difficult*. One way to measure the complexity of the language is to calculate a measure of vocabulary richness. Such a measure can be represented as a mean word frequency or as a relationship between the number of unique words used (i.e. the working lexicon) and a count of the number of word tokens in the document. Using either measure as a proxy for lexical complexity or variety, you can compare the lexical variety in the cetology chapter to the other chapters of he novel.

Vocabulary richness is more commonly expressed as a percentage or ratio of unique word types to the total number of word tokens. A *type-token* ratio, or *TTR* as it is generally called, is calculated by dividing the total number of unique word types by the total number of word tokens. The result is then typically multiplied by `100` so as to end with a percentage. As you can surmise, a lower type-token ratio (TTR) is suggestive of less lexical variety.

In the previous chapters, you learned how to use a `for` loop to generate two list objects containing tables of words and their frequencies for each chapter of *Moby Dick*. The list titled `chapter.raws.l` contains the raw counts of each word token and `chapter.list.freqs` contains the relative frequencies of each word token in the given chapter. To calculate the mean word frequency and TTR values for each chapter of *Moby Dick*, you will need `chapter.raws.l`. You can use your existing code, or go to the code repository for this chapter.

6.2 Mean Word Frequency

To calculate mean word frequency on a chapter by chapter basis, you'll first get the total number of word tokens in each chapter by summing the raw frequency counts in each, and then you'll calculate the number of unique word types in each chapter. These are two very simple calculations that you can derive from the `chapter.raws.l` list object. It is worth taking a moment to recall just exactly what this list contains. The first thing you'll want to know is the *size* or *length* of the list.

```
length(chapter.raws.l)
## [1] 135
```

The `length` function reveals that there are `135` items in the list. As you will recall, each list item corresponds to a chapter in the novel. If you want to see the chapter titles, you can find them in another part of the list object that is accessed via the `names` function.

```
names(chapter.raws.l)
[1] "CHAPTER 1. Loomings."
[2] "CHAPTER 2. The Carpet-Bag."
. . .
[135] "CHAPTER 135. The Chase.--Third Day."
```

Any item in a list may also be accessed and examined individually. Like so much in `R`, such access is facilitated through bracketed sub setting. In this case, since you have also stored the chapter titles as *names* for each list item, you can also use the `$` shortcut character in the way that you did with data frames in the previous chapter. The `chapter.raws.l` contains a series of *table* objects, which you can check by calling the `class` function on one item of the list:

```
class(chapter.raws.l$"CHAPTER 1. Loomings.")
## [1] "table"
```

or, if you do not know the exact name, you can achieve the same result using the numerical index:

```
class(chapter.raws.l[[1]])
## [1] "table"
```

As you can see, the first item in the list is a table object. To access the word frequency table for this first chapter, just remove the call to the `class` function using either this expression

```
chapter.raws.l$"CHAPTER 1. Loomings."
```

or this

```
chapter.raws.l[[1]]
```

to return a table of word types and the corresponding counts of those words in the first chapter.

You have already learned how to find useful information about R objects using the class function. For even more detailed information, you saw how to use str, the *structure* function. The class function tells us something general about the kind of data an R object contains. For example, entering class(chapter.raws.l) shows us that the object is a list. Alternatively, str(chapter.raws.l) returns additional information about the size (135 items) and construction of the list, including a long string of output containing information about each of the 135 items in the list.

str also reveals that the first item of the main list is a table object with 854 items. The line of output that begins with the word *table* shows the counts for the first 10 items in the table. The *int* that you see tells us that the items are stored in an *integer* vector. Two lines below that is another line showing a series of words: "a" "abandon" etc. These are prefaced with the chr marker. chr indicates that these values are stored as a *character* vector. These, of course, are the actual word types from the chapter, and the integers above them are the counts of the occurrences of these word types in the chapter. The word *a*, for example, occurs 69 times in the first chapter. The fourth word in the vector, *about*, occurs 7 times in the chapter and so on.

With an understanding of how and where the data are stored, it is a fairly routine matter to calculate the mean word frequency for each chapter. Summing the integer values held in each chapter *table* will give you a count of all word tokens in the chapter, and you can use the length function to return the total number of word types.

```
sum(chapter.raws.l[[1]])
## [1] 2244
length(chapter.raws.l[[1]])
## [1] 854
```

Using these two figures, the mean word frequency can be calculated by dividing the total number of tokens by the total number of unique word types.

```
sum(chapter.raws.l[[1]])/length(chapter.raws.l[[1]])
## [1] 2.628
```

The result shows that each word type in the first chapter is used an average of 2.6 times. An much simpler way of doing this is to just use R's built in mean function.

```
mean(chapter.raws.l[[1]])
## [1] 2.628
```

6.3 Extracting Word Usage Means

Since the chapters are already in a list object, all that you need now is a method for extracting the frequency data from all of the chapters at once. For this you can employ the lapply function. lapply is an alternative to for that in some sense simply hides the operations of a for loop. In another sense, lapply simplifies the code needed by automatically generating a new list object for a result. That is, we do not need to create an empty list outside of a loop and then fill it with each iteration of the loop. `lapply` takes two arguments: a list object and a function to apply to the items in the list. To get the `mean` word usage for each chapter in *Moby Dick*, for example, you could use the following command:

```
lapply(chapter.raws.l,mean)
$CHAPTER 1. Loomings.
[1] 2.628
$CHAPTER 2. The Carpet-Bag.
[1] 2.34
$CHAPTER 3. The Spouter-Inn.
[1] 3.72
. . .
$CHAPTER 135. The Chase.--Third Day.
[1] 3.44
```

Calling `lapply` in this way generates a new `list` object. In the example here, I have just printed the results to the screen instead of saving the output into a new object.

Since a list is not very handy for further manipulation, you can wrap this `lapply` expression inside a `do.call` function, which will take the list output from `lapply` and apply another function (`rbind`) to the results. This has the effect of putting all of the results into neat rows in a matrix object. Since you want to be able to plot this data, you can direct the results of all of this into a new object called `mean.word.use.m`.

```
mean.word.use.m<-do.call(rbind, lapply(chapter.raws.l,mean))
```

The dimensions of the resulting matrix can be obtained using the `dim` function (for *dimensions*):

```
dim(mean.word.use.m)
## [1] 135   1
```

`dim` reports that the matrix has `135` rows and `1` column. But there is a bit more information stored in this matrix object, and you can get a hint of that content by using the `str` function discussed above. During the creation of this matrix, the individual chapter names were retained and assigned as *rownames*. Entering `rownames(mean.word.use.m)` returns the names.

At this point, you can plot the values and visualize the mean word usage pattern across the novel. Calling `plot` returns a simple barplot (Figure 6.1), in which chapters with higher bars are, in one manner of speaking, *less rich*.

```
plot(mean.word.use.m, type="h")
```

In the chapters with high values, individual word types are used more often; there is more repetition of the same word types. Alternatively, in chapters where the bar is low, each word type has a lower overall usage frequency. In the chapters with high bars, the reader can expect to see the same words repeated rather frequently, in the lower bar chapters the reader is treated to a collection of words that might give the impression of greater variety for being repeated less often.

To be more interpretable, you may want to consider normalizing these values across the entire text. `R` provides a `scale` function for normalizing or *scaling* data. In such scaling, the overall mean for all of the chapters is first calculated and then subtracted from each individual chapter means. This method has the effect of subtracting away the expected value (expected as calculated by the overall mean) and then showing only the deviations from the mean. The result is a vector of values that

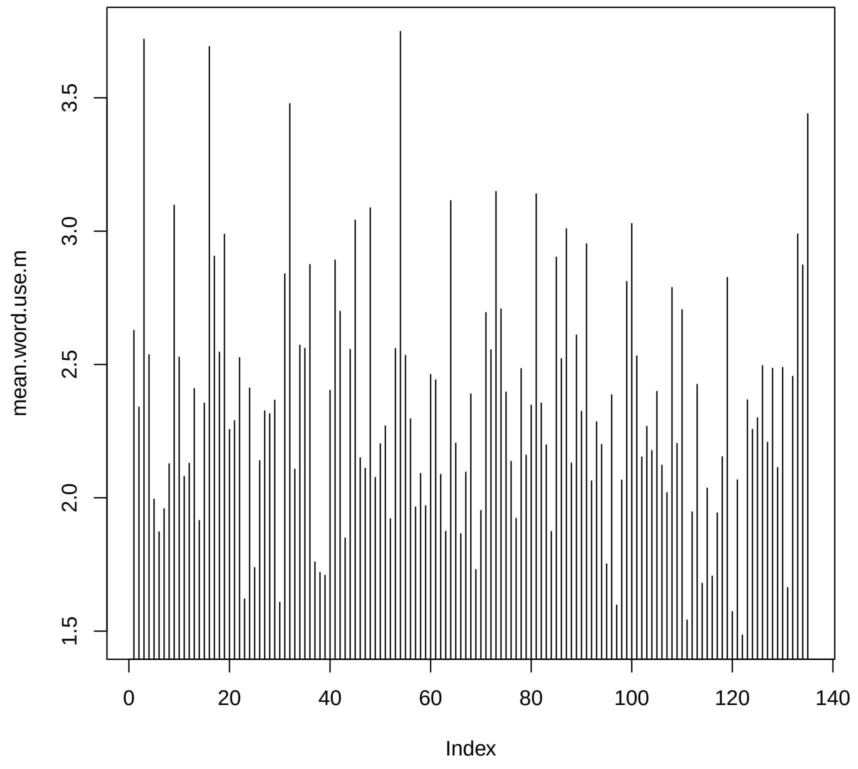


Fig. 6.1 Barplot of Mean Word Use

includes both positive and negative numbers. You can look at the scaled values by entering:

```
scale(mean.word.use.m)
```

Instead of just studying the numbers, however, it might be better to visualize the results as a barplot similar to figure 6.1. In the resulting plot, `0` on the y-axis will correspond to the mean across the entire novel. You will only see the deviations from the mean (Figure 6.2).

By this measure of mean word use, the cetology chapter, which readers so often remember as being one of denser, richer vocabulary is not exceptional at all. Words in the cetology chapter are repeated fairly often. In fact, each unique word type is used an average of `3.5` times.

```
plot(scale(mean.word.use.m), type="h")
```

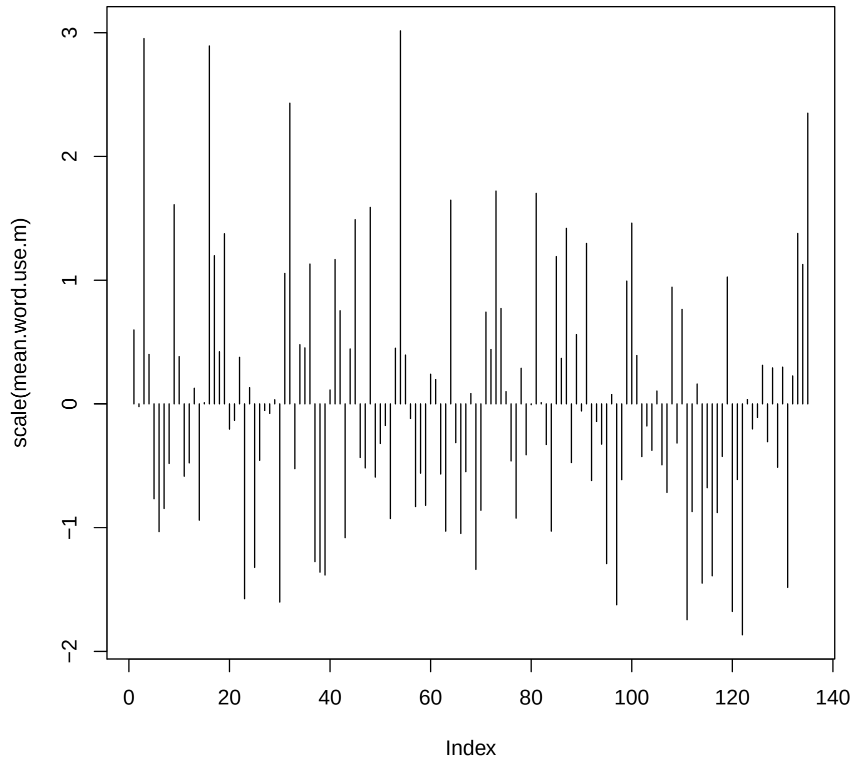


Fig. 6.2 Mean Word Usage Plot

6.4 Ranking the Values

To see where the cetology chapter ranks in terms of average word use, you can employ the `order` function to arrange the data in decreasing rank order. Beware, however, that the `order` function can be confusing. If you enter:

```
order(mean.word.use.m)
```

`R` will return a vector of numbers corresponding to the ranked *positions* of each item in the `mean.word.use.m` vector, like this:

```
order(mean.word.use.m)
##   [1] 122 111 120  97  30  23 131 114 116  39  38
##  [12]  69  25  95  37  43  66   6  63  84  14  52
##  [23]  77 117 112  70   7  57  59   5 107 115  92
##  [34]  98 121  49  11  62  58  67  33  47 129 106
##  [45]   8  12  88  76  26  46 102 118  79 104  83
##  [56]  94  50 109  65 127  20 124 103  51  93  21
##  [67]  56 125  28  90  27   2  80  15  82  29 123
##  [78]  96  68  75 105  40  13  24 113  61 132  60
##  [89]  78 128 130 126  86  22  10 101  55   4  18
## [100]  72  44  53  35  34  89   1  71  42 110  74
## [111] 108  99 119  31 134  36  41  85  17  91  19
## [122] 133  87 100  45  48   9  64  81  73 135  32
## [133]  16   3  54
```

What this vector reveals is that the first item in the `mean.word.use.m` object, the *mean* of the word usage in chapter one of the novel, is the 122nd in rank when the means are sorted in increasing order, from smallest to largest.

If you want to order them according to decreasing rank, you need to set the *decreasing* argument of `order` to *TRUE*.

```
order(mean.word.use.m, decreasing=TRUE)
```

Again, just to emphasize this point, `order` does not order the means; it returns a vector of ranks in which the vector positions correspond to the positions in the vector being ranked. The vector of means can then be reordered, *sorted*, using this new vector of *rank positions* inside the brackets:

```
mean.word.use.m[order(mean.word.use.m, decreasing=TRUE),]
```

Here is an abbreviated look at the results:

```
CHAPTER 54. The Town-Hos Story.
3.748727
CHAPTER 3. The Spouter-Inn.
3.719777
CHAPTER 16. The Ship.
3.692105
CHAPTER 32. Cetology.
3.477622
```

After sorting you will see that the cetology chapter has the *fourth* largest mean of the `135` chapters. Only three other chapters recycle words at the rate of the cetology chapter! By this measure, it is not an especially interesting chapter at all.

6.5 Calculating the TTR inside `lapply`

The last few sections demonstrated how use `R's` built in `mean` function with `lapply` to calculate the mean word frequency of each chapter. Mean word usage is one way of thinking about lexical variety. A Type-Token Ratio provides a similar value for assessing lexical richness, but since `R` does not already have a function for calculating TTR, you will need to modify the arguments given to `lapply`. Instead of using `mean`, you will create your own *function*.

Above, you saw that you could calculate the mean for one chapter using this expression:

```
sum(chapter.raws.l[[1]])/length(chapter.raws.l[[1]])
## [1] 2.628
```

You can calculate the TTR using a similar expression in with the numerator and denominator are revered.

```
length(chapter.raws.l[[1]])/sum(chapter.raws.l[[1]])*100
## [1] 38.06
```

To run a similar calculation for all of the chapters as part of a call to `lapply` a generalized version of this calculation needs to be provided to `lapply` as the function argument. As you have seen, `lapply` takes a function argument such as `mean` or `sum`, etc. Since there is no function for TTR, you can provide `lappy` with your own custom function. In place of an existing function such as `mean` you can insert a function definition using a variable `x` to stand in for each item in the main list.

```
ttr.l<-lapply(chapter.raws.l, function(x) {length(x)/sum(x)*100})
```

Within the parentheses of `lapply` the TTR function is defined as follows: `function(x) length(x)/sum(x)*100`. When executed, `lapply` will treat each item in the `chapter.raws.l` list as the value for `x` (much in the same way that we have been using `i` inside of a `for` loop). The calculations will be performed on each item and the results returned in a new list object. You can then run `do.call` with `rbind` just as you did when calculating the means.

```
ttr.m<-do.call(rbind, ttr.l)
```

Now order the results:

```
ttr.m[order(ttr.m, decreasing=TRUE),]
```

or plot them:

```
plot(ttr.m, type="h")
```

6.6 A Further Use of Correlation

Unfortunately, measures such as the mean word frequency and TTR are not terribly useful because text length, or chapter length n this case, can be a strong determiner in the rate at which words get recycled. As chapter length increases you can generally expect more new words to be introduced. At the same time, many of the existing words will see repeated use because they provide the necessary structure or scaffolding for the introduction of new words. The practice exercises that follow, provide you an opportunity to test these assertions.[1]

[1] In addition to the two measures of lexical variety offered in this chapter, and another approach offered in the next, readers may wish to consider *Yule's K* (see Yule G. Udny. *The Statistical Study of Literary Vocabulary*. Cambridge University Press, 1994). Yule attempts to compensate for text length and provide a stable measure of lexical variety in what he called the *K characteristic*. A function for computing Yule's characteristic constant `K` can be found in the `languageR` package.

Practice

6.1. To test the assertion that document length is a strong determiner in the rate at which words get recycled, measure the strength of correlation between chapter length and TTR. For this you need two vectors of data. You already have the TTR values in the `ttr.m` matrix. Convert that data to a vector using `as.vector`. You now need another vector of chapter lengths. For this you can use `lapply` with the `sum` function instead of using *mean*. Once you have the two vectors, run a correlation analysis similar to the correlation you did previously with occurrences of *whale* and *ahab*. Write up your code and an analysis of the result.

6.2. Run a similar correlation test using the values in the `mean.word.use.m` instead of the TTR values. Write up your code and an interpretation of the result?

6.3. Use randomization to test the likelihood that the correlation coefficient observed with the TTR values could have been the result of chance. Explain the outcome of the test.

6.4. Explain the difference between the results derived in practice exercise 6.1 and 6.2.

Chapter 7
Hapax Richness

Abstract
This chapter expands the analysis of vocabulary by focusing on words that occur very infrequently. Readers learn how to use `sapply` and to create another simple function.

7.1 Introduction

Another way of thinking about vocabulary richness and the *experience* of reading a particular text is to consider the fact that many words appear quite infrequently or even just once. These words that occur just once are sometimes referred to as *singletons* or even *one-zies*, but they are more formally called *hapax legomena*. *Hapax* (for short) may provide a different way of assessing the lexical richness of a given segment of prose. In this chapter you will learn how to calculate the total number of *hapax* and see if there is a correlation between the number of *hapax* and the length of a chapter. The working hypothesis will be that as chapter length increases, you would expect to see an increase in the number of *hapax legomena*.

7.2 sapply

For this analysis, you must return to the `chapter.raws.l` list. Instead of extracting a count of all word tokens, you will compute a `sum` of all of the word types *that appear only once* in each chapter. To extract a count of the *hapax*, you can use the `sapply` function in combination with an argument that identifies the values that are equal to `1`. `sapply` is a simplified, or, as the `R` documentation calls it, a *user-friendly* version of `lapply`. The main difference is that `sapply` returns a *vector* instead of a *list*. The arguments that you provide to `sapply` are going to be very similar to those given to `lapply`, but here you are going to add some additional conditions in the form of a *custom function* that calculates a `sum` of only those values that meet the condition of being equal to `1`. Your function is going to count how many words in the vector are only used once.

7.3 A Mini-Conditional Function

Instead of using built in functions such as `mean` or `sum`, for this task you need to construct your own function using the `function(x)` argument of `sapply` followed by a definition, or declaration, of that function. This is similar to what you did in chapter six in order to compute the TTR of each chapter. In this case the custom function will enclose a conditional expression that sums all the values in the raw counts table that are equal to one. It is important to emphasize here that to express equivalence, `R` expects the use of two *equal signs* (==). As you recall, this is done to avoid confusing the use of a single equals sign, which can be used in `R` as an assignment operator.[1] The code required for counting the number of *hapax* in each chapter, therefore, looks like this:[2]

```
chapter.hapax.v <- sapply(chapter.raws.l, function(x) sum(x == 1))
```

Translating this code block into plain English, we might say something like this: "For each item in `chapter.raws.l`, return the `sum` of the values that are equal to one." Since the *values* in this case are all `one`, `sum` is in essence returning a *count* of the words that occur just once in each chapter. If you print the contents of `chapter.hapax.v` to the console, you will see the *hapax* counts for each chapter.

```
chapter.hapax.v
CHAPTER 1. Loomings.
605
CHAPTER 2. The Carpet-Bag.
433
. . .
CHAPTER 135. The Chase.--Third Day.
903
```

This is a start, but now you need to divide the number of *hapax* in each chapter by the total number of words in each chapter. As it happens, you already have these values in the `chapter.lengths.m` variable from your work in the last chapter. Since `R` easily facilitates matrix division (that is, `R` allows you to divide one matrix of values by the corresponding values in another matrix) the code is simple. Instead of having to perform the division on one value at a time, like this

```
chapter.hapax.v[1] / chapter.lengths.m[1]
CHAPTER 1. Loomings.
0.2696078
chapter.hapax.v[2] / chapter.lengths.m[2]
CHAPTER 2. The Carpet-Bag.
0.2980041
```

[1] In `R` values can be assigned to an object using either <- or =. Throughout this book I use <- because it is the most common convention among users of `R`, and it avoids the whole = vs. == confusion.

[2] Remember that you can get the start up code for this chapter from the appendix or the code repository.

You can do it all at once, like this:

```
hapax.percentage <- chapter.hapax.v / chapter.lengths.m
```

This expression returns a new matrix containing the chapter names and the percentage of *hapax* in each chapter. These values can then be plotted, so that you can visualize the chapter-by-chapter *hapax richness*.

```
barplot(hapax.percentage, beside=T,col="grey",
        names.arg = seq(1:length(chapter.raws.l)))
```

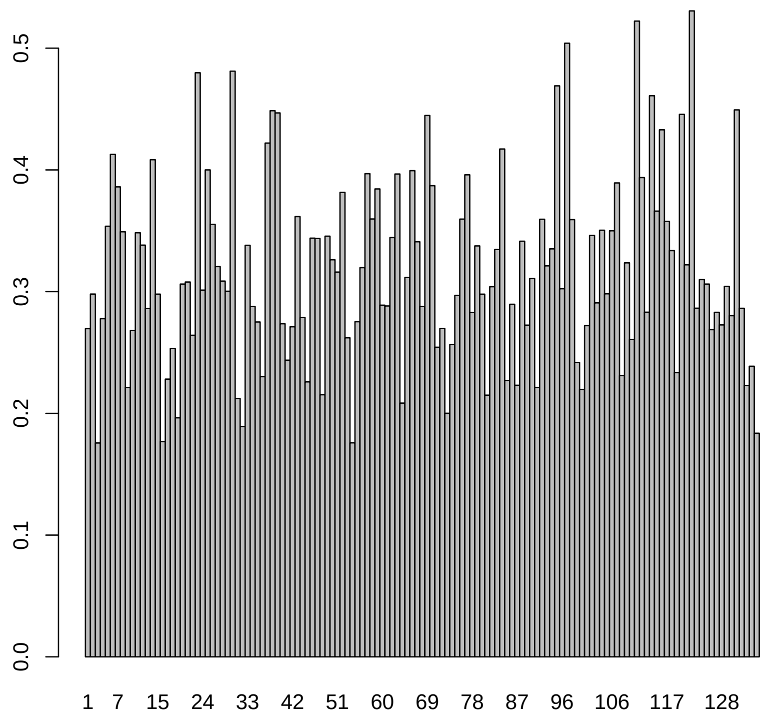


Fig. 7.1 *Hapax* Percentage Plot

Practice

7.1. First calculate the extent of the correlation between chapter length and the number of *hapax*. Offer a brief interpretation of the result (similar to what you did in Exercise 6.1).

7.2. Use `order` to rank the values in `hapax.percentage`. How does the ranks of the Cetology chapter compare to the others.

7.3. The correlation statistic found in Exercise 7.1 is not incredibly useful in and of itself. It becomes much more interesting when compared to another author's work, such as Jane Austen's *Sense and Sensibility* that you will find in the same folder as you found *Moby Dick*: data/plainText. First write the code necessary to get the chapter-by-chapter counts of the *hapax* in Austen's *Sense and Sensibility*. Save these in a list object titled `sense.raws.l`. From this object calculate the number of *hapax* and the chapter lengths for each chapter of *Sense and Sensibility*. Compute the correlation and describe how the correlation results for Melville and Austen compare and what they tell us about the two writers in terms of vocabulary usage habits.

Chapter 8
Do It KWIC

Abstract
In the last chapter a simple function was created within a call to the `sapply` function. In this chapter we explore user-defined functions more broadly and write a function for producing a keyword in context (KWIC) list.

8.1 Introduction

KWIC or *Keyword in Context* searches are a standard way of addressing Rupert Firth's observation that you will know a word's meaning or sense by looking at the other words around it, that is, by its context.[1] In this section (including Exercises 8.1 and 8.2), you will learn how to build a flexible KWIC tool in `R`. You will also be introduced to some `R` functionality that will allow you access and analyze multiple texts at once.

Begin as usual by setting a path to a working directory:

```
setwd("~/Documents/TextAnalysisWithR")
```

Unlike previous chapters where you loaded a single file using `scan`, you are now going to access a collection of files in a directory. Begin by defining a variable that contains the relative path to the directory of text files.

```
input.dir<-"data/plainText"
```

You can now use `R`'s `dir` function with this path variable to retrieve the names of all of the files in the directory. In addition to the path argument, which you have now stored in the `input.dir` object, the `dir` function can take an optional *search pattern string* written using a regular expression. Since you only want `dir` to return the files in the directory that have a *.txt* file extension, use the wildcard character "`.`" (period) followed by an "`*`" (an asterisk meaning *any number of*) to create a *pattern* (in regular expression speak) that will *match any string of characters that is followed by* `.txt`. The expression looks like this:

[1] See Firth, John Rupert. "A Synopsis of Linguistic Theory, 1930-1955." In *Studies in Linguistic Analysis*. Oxford: Blackwell.

```
files.v<-dir(input.dir, ".*txt")
```

Having run this, you should now be able to type `files.v` into `R` and return a vector of file names; in this example, just two files.

```
files.v
## [1] "austen.txt"   "melville.txt"
```

8.2 Custom Functions

This console display of the content of the `files.v` vector is not very pretty and once you get a corpus containing many files, it can become difficult to read through this display. To better understand the idea of custom functions, you will write code to make the contents of the `files.v` vector display in a more organized and *reader-friendly* fashion.

Functions are, primarily, reusable chunks of code. You have already learned about many of `R`'s built in functions, and you have used and reused them many times. You can also create your own custom functions and call *them* over and over again. If you were baking a cake for a friend's birthday, you'd buy some ingredients, pull out some pots and pans, and bake the cake. If, on the other hand, you decided to go into the cake baking business, you'd probably invest some time (and money) setting up a cake baking system that would take a certain set of ingredients and pump out a cake on the other end. That's what functions are; they are ingredient assembly systems.

In this section you will write a simple function called `show.files` that you can use for displaying file names. The specific purpose of the function in this example will be to display the contents of the `files.v` vector in an easy to read format. In `R` you begin a new function by giving it a name (`show.files`) and then using the `function`, *function*. Inside parentheses, you will define the arguments (ingredients) that the function requires: in this case, just one argument, a vector of file names.[2]

```
show.files <-function(file.name.v)
```

Here is the basic outline of the function:

```
show.files <-function(file.name.v){
  # some code goes here
}
```

The parenthetical arguments section is followed by a set of opening and closing *curly braces* that surround the inner workings of the function. This inner section,

[2] It is worth noting that the name(s) we assign to the arguments inside the parentheses have a scope that is limited to the function; these variable names do not persist or exist outside of the function, and they will not overwrite a variable with the same name that may exist outside of the function. That said, it is best to avoid duplication of variable names. Notice that I have named the argument `file.name.v` instead of `files.v`. For an example in code, see Appendix B.

inside the curly braces, is where the instructions (recipe) for how to assemble the ingredients will be defined.

The objective with this function is to provide some sort of *pretty printed* list of the files in the `files.v` variable. You already know that you can use a `for` loop to iterate over the items in a vector, so add code for a `for` loop inside the curly braces here:

```
show.files <-function(file.name.v){
  for(i in 1:length(file.name.v)){
    # some code goes here
  }
}
```

This `for`-looping should be old hat for you by now, but just to review, the `for` loop will be used to iterate over each of the items stored in the vector of file names, one item at a time.

For printing lines into the `R` console (do not confuse this sense of the word *print* with *printing* on paper to your printer), `R` has several functions you could tap, but let's use the `cat` function here. `cat` is a function for concatenation (joining) and printing. Here you want to join the file name with the index of the file name in the vector and then add a *line return* (using the backslash escape character and an *n* to mean *newline*). To achieve this, you will be joining three items: the vector index *key*, the contents or *value* of the item in the vector at that index position, and a *newline* character. To join these pieces requires a bit of *glue*, so `cat` asks us to define a *separator* using the `sep` argument. You can use a space character for the *glue*, and the final function looks like this:

```
show.files <-function(file.name.v){
  for(i in 1:length(file.name.v)){
    cat(i, file.name.v[i], "\n", sep=" ")
  }
}
```

Before taking the function for a test drive (cake walk?), there is one more thing to do. Just as it is very easy to write complicated functions, it is also very easy to forget what they do! So add a little comment to the function so that when you come back a week later you can be reminded of what it does:

```
# Function to print a vector of file names in user
# friendly format
show.files <-function(file.name.v){
  for(i in 1:length(file.name.v)){
    cat(i, file.name.v[i], "\n", sep=" ")
  }
}
```

If you have not already done so, set your working directory and enter the following code to load the data and the new function.[3]

```
input.dir<-"data/plainText"
files.v<-dir(input.dir, ".*txt")
# Function to print a vector of file names in user
# friendly format
show.files <-function(file.name.v){
  for(i in 1:length(file.name.v)){
    cat(i, file.name.v[i], "\n", sep=" ")
```

[3] Tip: In `RStudio`, you can run a whole block of code by selecting it in the editing window and then hitting the `command + return` keys at the same time.

```
  }
}
```

With all of this done, you can now send the `files.v` object to the `show.files` function and see the numbered result as output.

```
show.files(files.v)
## 1 austen.txt
## 2 melville.txt
```

Mission accomplished, your first function!

8.3 A Word List Making Function

In previous chapters, you learned how to generate a word vector from a text file. Since you are now an expert function maker, you'll build a function that will do this task for a whole bunch of files, and you'll write the function to store all of the results in a `list` so that you can easily access the word data from any file in your corpus. Unlike the `show.files` function from above, this new function will take two arguments: a vector of file names and a directory path telling the function where to find the files on your system. It is always a good idea to give your functions names that make sense; call this one `make.file.word.v.l` and begin it with a comment:

```
# Function takes a vector of file names and a directory path and
# returns a list in which each item in the list is an ordered
# vector of words from one of the files in the vector of file names
make.file.word.v.l<-function(files.v, input.dir){

}
```

Here I have entered a comment describing what I want this function to do. Articulating an objective in advance can be a great way to guide your coding. The definition says that I want to return a `list` object, so the first thing to do is to instantiate one inside the function, an empty one in this case. I'll call it `text.word.vector.l`:

```
make.file.word.v.l<-function(files.v, input.dir){
  text.word.vector.l<-list()
  # more code needed here to iterate over the input vector.  For
  # each file we need to load the text and convert it into a word
  # vector
}
```

Notice that here again is another comment to help keep track of what is happening at each stage in the process. It turns out that everything required in that comment is code that you have already written in previous sections and/or exercises. By recycling code from your prior work you can produce the following:

```
make.file.word.v.l<-function(files.v, input.dir){
  #set up an empty container
  text.word.vector.l<-list()
  # loop over the files
  for(i in 1:length(files.v)){
    # read the file in (notice that it is here that we need to know the input
    # directory
    text.v<-scan(paste(input.dir, files.v[i], sep="/"),
                 what="character", sep="\n")
    #convert to single string
    text.v<-paste(text.v, collapse=" ")
    #lowercase and split on non-word characters
    text.lower.v <-tolower(text.v)
    text.words.v<-strsplit(text.lower.v, "\\W")
    text.words.v<-unlist(text.words.v)
    #remove the blanks
    text.words.v<- text.words.v[which(text.words.v!="")]
    #use the index id from the files.v vector as the "name" in the list
    text.word.vector.l[[files.v[i]]]<-text.words.v
  }
  return(text.word.vector.l)
}
```

You now have code that will open each file in the input directory, and for each file it will create a word vector and store that word vector as an item in a `list` object. The only thing that remains to do, once the inner `for` loop has completed its cycle, is to return the `list` object back to the main script. For this, you need to call the `return` function, which is a way of telling the function to take all the cakes it has built and send them back to the head baker in a nice box. So the last line of code before the closing curly brace reads:[4]

```
  return(text.word.vector.l)
```

If you have not yet done so, write out this function and then load it in `R` by either copying and pasting it into the console or using the `RStudio` shortcut (`command + return`). You can then call the function with the `files.v` and `input.dir` arguments and put the returned result into the new variable `my.corpus.l`:

```
  my.corpus.l<-make.file.word.v.l(files.v, input.dir)
```

If everything worked, you should have gotten the following message from R:

```
  Read 10906 items
  Read 18874 items
```

Assuming that all is well, you will now have a new list object called `my.corpus.l`. Use the `class` and `str` functions to investigate its contents.

8.4 Finding Words and Their Neighbors

Now the fun begins. Consider that for each file (each item in the `my.corpus.l` list), you now have an ordered vector of the words. If you were to enter

```
  my.corpus.l[[1]][1:100]
```

you would get the first 100 words of Jane Austen's novel, one word at a time.[5] At this point, I hope that you are already one step ahead of me and thinking to yourself,

[4] In `R` you do not always have to explicitly call `return`. By default `R` will return whatever object is in the last line of the function. Explicitly calling `return`, however, often makes it easier to read and debug your function code.

[5] If you try this now, you'll see that you are actually getting *Project Gutenberg*'s boilerplate words, but you get the idea.

"hey, if I have all the words in order, I can find any word in the text and return its position in the text using a which statement." You already did this when you found the occurrences of *whale* in *Moby Dick*. It is a little bit trickier here because you are working with a list, but consider this next R expression in which the double brackets after the my.corpus.l object provide a way of accessing the first item in the list:

```
positions.v<-which(my.corpus.l[[1]][]=="gutenberg")
```

After indicating a desire to access the vector held as the first item in the list using the double-bracketed number 1, the remainder of the expression seeks out which of those values is a match for the keyword: *gutenberg*. This expression will return the *positions* of every instance of the word *gutenberg* in the file that is indexed as the first item in my.corpus.l. Go ahead and enter this now, and see what you get. The result should be something like this:

```
positions.v
##  [1]      3     47     57     86 120883 120896
##  [7] 120925 121004 121012 121018 121110 121126
## [13] 121155 121168 121179 121191 121206 121259
## [19] 121281 121326 121371 121401 121421 121434
## [25] 121451 121522 121535 121549 121564 121594
## [31] 121677 121709 121728 121740 121750 121758
## [37] 121804 121814 121823 121882 121918 121940
## [43] 121986 122020 122043 122091 122134 122158
## [49] 122163 122215 122244 122278 122302 122327
## [55] 122343 122385 122402 122445 122481 122542
## [61] 122559 122587 122599 122618 122641 122648
## [67] 122731 122740 122750 123133 123154 123192
## [73] 123205 123222 123225 123289 123298 123313
## [79] 123328 123339 123372 123378 123438 123555
## [85] 123560 123778 123819 123833 123857 123869
## [91] 123927 123936 123946
```

These are the positions of every occurrence of *gutenberg* in the file titled *austen.txt*. And if you can find the position of every occurrence of *gutenberg* in the word vector, you can find any other word (i.e. *whale* or *dog*). And, if you can find a word's position, you can also find the items that are next to it: before it, and after it. You can do this by simply adding or subtracting values from the position of the *found* word. Deep breath.

To summarize, you have used the which statement to find all the instances of *gutenberg* and stored those positions in a new vector called positions.v. If you check the length of this positions.v vector, you will get a count of the number of times *gutenberg* occurs in the file

```
length(positions.v)
## [1] 93
```

Now let's say that you want to know the words that come just before and just after the first instance of *gutenberg* in this file. You might begin by specifically identifying the first instance:

```
first.instance<-positions.v[1]
```

Which is to say that you could put the value that is held in the first item in the positions.v vector into a new variable called first.instance. If you look at the full print out above, you'll see that the first value in the positions.v vector is 3. The first instance of *gutenberg* is the third word in the file. With this last R expression, you have put the number 3 into the variable called first.instance.

If you want to check your work, just use that new variable in the original word vector, like this:

```
my.corpus.l[[1]][first.instance]
## [1] "gutenberg"
```

Ta Da! Of course, since you already knew that *gutenberg* is the third word in the file, you could have also done this:

```
my.corpus.l[[1]][3]
## [1] "gutenberg"
```

Ta Da! And, if you want to see the words just before and just after the 3rd word in the file, you could, of course, just do this:

```
my.corpus.l[[1]][2:4]
## [1] "project"   "gutenberg" "ebook"
```

But now consider that another way of getting access to the second and forth positions in the vector is to add and subtract 1 from 3. Since 3 is the value already stored in the first.instance variable you could subtract one from first.instance. With that in mind, you can use the following expression to achieve the same result as above, but without hard coding any of the vector positions.

```
my.corpus.l[[1]][(first.instance-1):(first.instance+1)]
## [1] "project"   "gutenberg" "ebook"
```

If you want to see the results *pretty printed*, just use cat:

```
cat(my.corpus.l[[1]][(first.instance-1):(first.instance+1)])
## project gutenberg ebook
```

Practice

8.1. Using the functions described in this chapter and what you now know about vector indexing, write a script that will produce a *five-word* KWIC list for all occurrences of the word "dog" in both *Moby Dick* and *Sense and Sensibility*.

8.2. KWIC with Cleaner Output
For an even cleaner look, use your new knowledge of the cat function to format your output so that it looks something like this:

```
----------------------- 1 -----------------------
  all over like a newfoundland [dog] just from the water and
----------------------- 2 -----------------------
  a fellow that in the [dog] days will mow his two
----------------------- 3 -----------------------
  was seen swimming like a [dog] throwing his long arms straight
----------------------- 4 -----------------------
  filling one at last down [dog] and kennel starting at the
----------------------- 5 -----------------------
  not tamely be called a [dog] sir then be called ten
```

```
----------------------- 6 -----------------------
  t he call me a [dog] blazes he called me ten
----------------------- 7 -----------------------
  sacrifice of the sacred white [dog] was by far the holiest
----------------------- 8 -----------------------
  life that lives in a [dog] or a horse indeed in
----------------------- 9 -----------------------
  the sagacious kindness of the [dog] the accursed shark alone can
----------------------- 10 -----------------------
  boats the ungracious and ungrateful [dog] cried starbuck he mocks and
----------------------- 11 -----------------------
  intense whisper give way greyhounds [dog] to it i tell ye
----------------------- 12 -----------------------
  to the whale that a [dog] does to the elephant nevertheless
----------------------- 13 -----------------------
  aries or the ram lecherous [dog] he begets us then taurus
----------------------- 14 -----------------------
  is dr bunger bunger you [dog] laugh out why don t
----------------------- 15 -----------------------
  to die in pickle you [dog] you should be preserved to
----------------------- 16 -----------------------
  round ahab and like a [dog] strangely snuffing this man s
----------------------- 17 -----------------------
  lad five feet high hang [dog] look and cowardly jumped from
----------------------- 18 -----------------------
  as a sagacious ship s [dog] will in drawing nigh to
----------------------- 19 -----------------------
  the compass and then the [dog] vane and then ascertaining the
```

Chapter 9
Do It KWIC (Better)

Abstract
This chapter expands upon the previous chapter in order to build an interactive and reusable KWIC application that allows for quick and intuitive KWIC list building. Readers are introduced to interactive `R` functions including `readline` and functions for data type conversion.

9.1 Getting Organized

In the previous chapter, you learned how to find and access a series of index positions in a vector and then how to return values on either side of the found positions. In exercise 8 you hard coded a solution for finding occurrences of the word *dog* in *Sense and Sensibility* and *Moby Dick*. In this section you'll learn how to abstract that code and how to create an interactive and reusable application that will allow you to repeatedly find key words in context within any directory of files, all without having to hard code the search terms.

If you have not already done so, now is the time to get organized. You will be dealing with more and more files as this book continues, and unless you keep your working spaces well-defined and organized things can get complicated. Within your `TextAnalysisWithR` directory, you should already have a sub-directory labeled `code`. This is where you should store all of your `.R` files. You should also have another sub-directory labeled `data` that contains all of your text/corpus files. If you do not already have a sub-directory called "results" create one now because in the last exercise in this chapter you'll be generating a `.csv` file that you can save and then open in `R` or in a spreadsheet application such as Excel or Open Office.

9.2 Separating Functions for Reuse

In the last chapter you created two functions, and in this chapter you will create another. Because you can reuse functions in separate projects, it is convenient to keep them in a separate file so that you can access them from different `R` scripts that you write for different projects. You should begin this chapter, therefore, by copying your functions into a new file that you will title *corpusFunctions.R*. Save this new file inside your `code` sub-directory. Your functions file should include both `show.files` and `make.file.word.l` from the last chapter.

With your functions stored in a separate file, you can now *call* the functions file as part of your working `R` script. Open `R`, or, if it is already open, start a new session by clearing your workspace. Open a new document in your code editor and begin by defining a working directory with the `setwd` function or by choosing a working directory using `RStudio's` menu shortcut. Now enter the following expression that uses `R`'s `source` function to show `R` where to find your functions file.

```
source("code/corpusFunctions.R")
```

When your main script is executed, `R` will load all of the functions in the *corpusFunctions.R* file.

As in chapter 8, you need to show `R` where to find your text files, so next you will define an *input directory* with a relative path to the data/plainText directory

```
input.dir<-"data/plainText"
```

Since you will also be using `R` to create derivative data files, which will need to be saved out to another directory, you'll need to tell `R` where to write these files. Define an output directory like this:

```
output.dir <-"results/"
```

You can now recycle a few lines of code from the prior chapter to load the corpus into memory:

```
files.v<-dir(input.dir, ".*txt")
my.corpus.l<-make.file.word.l(files.v, input.dir)
```

The objective now is to write a KWIC application that will allow you to repeatedly enter different keywords and return back the hits for those terms along with some amount of context on either side of the key term.

9.3 User Interaction

`R` includes a set of built-in functions that, when invoked, require user feedback to complete. Thus far we have been hard-coding file paths in `R`, but we could have been using `R`'s `file.choose` function instead. If you enter `file.choose` at the `R` prompt, you will be prompted with a new popup window that allows you to navigate your file system and locate a file. Here is an example that you can try on your system. Just enter the following expression at the `R` prompt in the console pane and then use your computer's widowing system to locate the file in the exercise directory called "melville.txt."

```
mytext<-scan(file.choose(), what="character", sep="\n")
```

If you did everything correctly, you should see the message:

```
Read 18874 items
```

You will now be able to enter

```
mytext
```

and see all the lines of *Moby Dick*.

9.4 `readline`

There are other functions in `R` that allow for user interaction as well, and one that we will use for this section is called `readline`. `readline` is a function that will print information to the `R` console and then accept input from the user. Enter this expression into the console and hit return:

```
myyear<-readline("What year was Moby Dick Published? \n")
```

When prompted, enter a number (e.g. `1851`) and hit return. If you now type `myyear` at the `R` prompt and hit return, you'll find that `R` has stored the value, that you entered in the `myyear` variable. Here is how it should look:

```
> myyear<-readline("What year was Moby Dick Published? \n")
What year was Moby Dick Published? 1851
> myyear
[1] "1851"
```

9.5 Building a Better KWIC Function

Using the `readline` function, you can write a *KWIC* list function that asks the user (you) for a *file* to search, a *keyword* to find, and an amount of *context* to be returned on either side of the keyword. When you call this function, you will send it your existing `my.corpus.l` list object as the sole argument. Name this function `doitKwic`; a call to it will look like this:

```
doitKwic(my.corpus.l)
```

The only argument that you need to send is the list object you created and stored in `my.corpus.l`. This object already contains the name of every file in the corpus directory as well as all the word vectors for each file. Open your *corpusFunctions.R* file and begin writing this new function like this:

```
doitKwic<-function(named.text.word.vector.l){
# instructions here will ask user for a file to search
# a keyword to find and a "context" number for context
# on either side of the of the keyword
}
```

Keep in mind that the argument name used inside the parentheses of the function does not have to be the same as the name used outside of the function. So here I am defining a function that takes an argument called `named.text.word.vector.l`. This is the same type of object as the `my.corpus.l` object you have already instantiated, but when *inside* the workings of the function, it can be referred to by another name.

You do not have to write your code this way (i.e. using different names when inside or outside of the function), but I find it useful to name my function arguments in a way that is descriptive of their content and a bit more abstract than the names I give my objects within the main script. I may decide to use this function on another project, and several months from now I may have forgotten what `my.corpus.l` means. Using `named.text.word.vector.l` is verbose, but at least it gives me some clues about what kind of object I am dealing with.

As the commented sections of the code suggest, you want the new function to ask the user for input. First it needs to ask which file in the corpus to search in, then what keyword to search for, and finally how much context to display. For the first item, the function should display a list of the files that are inside the `named.text.word.vector.l` object and ask the user to choose one. Luckily you already have a function called `show.files` that does exactly that. You can call this function from inside the new function! Remember too that the `show.files` function is expecting to get a vector of file names as its argument. You can get a vector of file names from the `my.corpus.l` object by calling the `names` function. You can try this by entering

```
show.files(names(my.corpus.l))
## 1 austen.txt
## 2 melville.txt
```

Inside the `doItKwic` function, you will be able to achieve the same result with:

```
show.files(names(named.text.word.vector.l))
```

Having called `show.files`, the `readline` function can then be invoked to ask the user for a number corresponding to a file (i.e. `1` for Austen and `2` for Melville). This user-entered information can be stored in a new object called `file.id`.

```
file.id<- as.numeric(readline(
  "Which file would you like to examine? Enter a file number: \n"))
```

Notice how I have also wrapped the result inside a call to `as.numeric`. Even though the user enters a number, input received from the user with the `readline` function is treated as *character* data and needs to be converted to numeric for use in this function. Something very similar is done for the *context* number that is needed:

```
context<-as.numeric(readline(
  "How much context do you want to see, Enter a number: \n"))
```

Finally, you will want `R` to ask the user for a keyword and you'll make sure it is changed to lowercase:[1]

[1] Obviously you could create a more complex function that would give the user flexibility in terms of capitalization. For simplicity, we are working with a lower-cased text with all of the punctuation stripped out.

```
keyword<-tolower(readline("Enter a keyword: \n"))
```

With these three bits of information, you are now ready to run a KWIC search. You will use `which` to find and return the position indexes of the *hits* for the user's keyword. The search results can be stored in a new object called `hits.v`.

```
hits.v<-which(named.text.word.vector.l[[file.id]] == keyword)
```

Notice that to create and fill the new `hits.v` object, I had to use the value held in both the `file.id` and `keyword` variables that I just got by asking the user to provide them! The rest of the function is simple; it will iterate over the positions in the `hits.v` object and return *context* words to the left and right of the position. Here is the most obvious and simple solution. . .

```
for(h in 1:length(hits.v)){
  start<-hits.v[h]-context
  end<-hits.v[h]+context
  cat(named.text.word.vector.l[[file.id]][start:end])
}
```

9.6 Fixing a Problem

Unfortunately, this simple solution cannot handle all of the possible scenarios that might occur. What if the very first word in the file you are searching in is a hit? In this case the first position in the `hits.v` vector would be `1` and that would cause `start` to be set to `1 - (minus) context`: that is one minus what ever number the user entered for `context`. The result of that subtraction would be a negative number and `R` would choke trying to access a value held at a negative vector index! You can't have that, so you need to add some code to deal with this possibility. Here is one way to deal with the problem using an `if` conditional:

```
for(h in 1:length(hits.v)){
  start<-hits.v[h]-context
  if(start < 1){
    start<-1
  }
  end<-hits.v[h]+context
  cat(named.text.word.vector.l[[file.id]][start:end])
}
```

The `if` conditional tests to see if the value of `start` is set to less than `1`. If it is, then `start` gets reset to `1`. Assuming you implement this solution, the whole function should now look like this:

```
# Function take a list containing word vectors
# from text files and then allows for
# interactive user input to produce KWIC lists
doitKwic<-function(named.text.word.vector.l){
  show.files(names(named.text.word.vector.l))
  # ask the user for three bits of information
  file.id<- as.numeric(readline(
    "Which file would you like to examine? Enter a file number: \n"))
  context<- as.numeric(readline(
    "How much context do you want to see, Enter a number: \n"))
  keyword<- tolower((readline("Enter a keyword: \n")))
  hits.v<-which(named.text.word.vector.l[[file.id]] == keyword)
  if(length(hits.v)>0){
    for(h in 1:length(hits.v)){
      start<-hits.v[h]-context
      if(start < 1){
        start<-1
      }
      end<-hits.v[h]+context
      cat(named.text.word.vector.l[[file.id]][start:end], "\n")
    }
  }
}
```

The result of calling this function and looking for the keyword *dog* in *Sense and Sensibility* looks like this:

```
doitKwic(my.corpus.l)
1 austen.txt
2 melville.txt
Which file would you like to examine? Enter a file number:
  1
How much context do you want to see, Enter a number:
  5
Enter a keyword:
  dog
a fellow such a deceitful dog it was only the last
```

Practice

9.1. In prior exercises and lessons, you have learned how to instantiate an empty `list` object outside of a `for` loop and then how to add new data to that result object during the loop. You have learned how to use `cbind` to add columns of data and `rbind` to add rows. You have also learned how to use `paste` with the `collapse` argument to glue together pieces in a vector of values and how to use `cat` to concatenate items in a vector. And you have used `colnames` to get and set the names of columns in a data frame.

Using all of this knowledge, modify the the function written in this chapter so that the results of a KWIC search are put into a `dataframe` in which each row is single KWIC result. Your data frame should have four columns labeled as follows: *position*, *left*, *keyword*, and *right*. The *position* column will contain the index value showing where in the file the keyword was found. The *left* column will contain the words in the file vector that were found to the left of the keyword. The *keyword* column will contain the keyword, and the *right* column, the context that was found to the right of the keyword. Here is an example of results generated using the keyword *dog* with two words of context in the file "melville.txt."

```
position left  keyword right
[1,] "14555"  "a newfoundland" "dog"  "just from"
[2,] "16376"  "in the"  "dog"  "days will"
[3,] "27192"  "like a"  "dog"  "throwing his"
[4,] "51031"  "last down"  "dog"  "and kennel"
[5,] "51107"  "called a"  "dog"  "sir then"
[6,] "51565"  "me a"  "dog"  "blazes he"
[7,] "73930"  "sacred white"  "dog"  "was by"
[8,] "107614" "in a"  "dog"  "or a"
[9,] "107700" "of the"  "dog"  "the accursed"
[10,] "137047" "and ungrateful" "dog"  "cried starbuck"
```

```
[11,] "137077" "way greyhounds" "dog"  "to it"
[12,] "147004" "that a"  "dog"  "does to"
[13,] "167296" "ram lecherous"  "dog"  "he begets"
[14,] "170197" "bunger you"  "dog"  "laugh out"
[15,] "170577" "pickle you"  "dog"  "you should"
[16,] "171104" "like a"  "dog"  "strangely snuffing"
[17,] "202940" "high hang"  "dog"  "look and"
[18,] "206897" "ship s"  "dog"  "will in"
[19,] "206949" "then the"  "dog"  "vane and"
```

Make a copy of the `doitKwic` function from this chapter and rename it `doitKwicBetter`. Now modify the function to produce a result like that seen above.

9.2. Copy the function you created in exercise 9.1 and modify it to include a feedback loop asking the user if the results should be saved as a .csv file. If the user answers "yes," generate a file name based on the existing user input (keyword, file name, context) and write that file to the results directory using a call to the `write.csv` function, as in this example:

```
write.csv(your.file, paste(output.dir, "your.file.csv", sep=""))
```

Save a copy of this new function in your *corpusFunctions.R* file as `doitKwicBest`.

Chapter 10
Text Quality, Text Variety, and Parsing XML

Abstract
This chapter introduces readers to parsing XML in R with an emphasis on TEI encoded XML.

10.1 Introduction

If you have ever downloaded a digital text from the internet, you already know that there is a great variety when it comes to quality. Some digital texts are available in what is referred to as *dirty OCR*. This means that the texts have been scanned and run through an optical character recognition (OCR) process but not subsequently hand checked and corrected or cleaned up by a human editor (hence the term *dirty*). On the other end of the spectrum, there are digital texts that have been carefully created by double keying and human correction. Double keying involves the use of two typists who each key the entire text into a computer. Once the two versions are completed, they are compared to identify discrepancies. Double keying in not perfect, but it is one of the more reliable methods for deriving a high quality digital version of a text. Somewhere in between double keying and dirty OCR lies corrected OCR. In this case an original document is scanned and then cleaned by a human editor. While this method is still prone to errors, it is a considerable step beyond dirty OCR and frequently good enough for processing and analysis tasks that involve generating global statistics, which is to say a big picture perspective where a single mis-keyed word will have little impact on the overall result.

Anyone working with digital text must at some point assess his or her corpus and form an opinion about its quality and in what ways the quality of the material will impact the analysis of the material. Promising research by Maciej Eder has examined the extent to which OCR errors impact stylometric analysis.[1] This research gives us hope of being able to quantify the margin or error caused by OCR problems.

[1] See Eder, Maciej. "Mind your corpus: systematic errors in authorship attribution." in *Conference Abstracts of the 2012 Digital Humanities Conference*, Hamburg, Germany.

And, make no mistake, this is a very big problem. As the scanning efforts of Google continue and as projects such as the Internet Archive and HathiTrust continue to make more and more dirty OCR text available online, an algorithmic method for dealing with dirty OCR becomes more and more important. Some, myself included, have argued that at the large scale these OCR issues become trivial. That is a hypothesis, however, and one born out of frustration with the reality of our digital corpora. If we want to mine the digital library as it exists today, we need to have a fairly high tolerance for error.

But along side these large and messy archives there are a good number of digital collections that have been carefully curated, and, in some cases, enriched with detailed metadata. Two very fine examples are the *Chadwyck Healey* and *Alexander Street Press* collections. Both of these content providers offer carefully corrected, XML (or SGML) encoded digital texts. The high quality of these texts does, however, come at a price: access to these corpora is available for a fee, and the fee is beyond the budget of a typical scholar. If your institution does not subscribe to these one of these collections, you are more or less out of luck.

Somewhere in between the high quality products of vendors such as *Chadwyck Healey* and *Alexander Street Press* and the dirty OCR of free resources such as *Google Books* and the *Internet Archive*, is *Project Gutenberg*. The texts in *Project Gutenberg* tend to be of fairly high and fairly consistent quality. Having said that, they lack detailed metadata and text provenance is often unclear. If your research does not demand the use of a particular edition, and if you can tolerate some degree of textual error, then *Project Gutenberg* may be a suitable source for digital texts. *Project Gutenberg* texts are frequently available in multiple formats: plain text, html, epub, etc. In many cases, it is possible to convert files in one format into another, and in my own work I have developed scripts for converting *Gutenberg's* plain text into TEI XML.

10.2 The Text Encoding Initiative (TEI)

The Text Encoding Initiative (TEI) offers a document-encoding standard that is commonly used by humanities scholars. The TEI markup scheme provides a way of storing an original text file along side an almost infinite amount of metadata. Since the files are extensible and editable, the amount of metadata available is only limited by the encoder's willingness to modify the documents. Say for example, you are collecting novels written by Irish- and German-American authors. For this project you might have a metadata field in your document where you can indicate the author's national origins. You may have another field where you indicate the author's gender, or birth date, or race, or sexual orientation. Once metadata of this sort is added to the XML files, it can be easily accessed by computer scripts and used, for example, as sorting facet for a particular type of analysis.

http://www.dh2012.uni-hamburg.de/conference/programme/abstracts/mind-your-corpus-systematic-errors-in-authorship-attribution/

In the rest of this book, you will be working with a corpus of texts that are encoded in TEI compliant XML. Unlike the plain text files (*Moby Dick* and *Sense and Sensibility*) that you have processed thus far, these TEI-XML files contain extra-textual information in the metadata of the <teiHeader> element. To proceed, you must be able to parse the XML and extract the metadata while also separating out the actual text of the book from the marked up *apparatus* around the book. You need to know how to parse XML in R.

10.3 Parsing XML with R

An in depth discussion of XML and of the TEI standard is beyond the scope of this book. To understand the way that R parses XML, readers should be familiar with the basic construction of an XML document as an ordered hierarchy of content objects (OHCO) and should have some general familiarity with the structure of a TEI document: its primary divisions into <teiHeader>, <text>, <front>, <body>, and <back>. The R package you will be using in this section (named simply XML) can be used to load an XML file as DOM-tree object based on the standard Document Object Model (DOM).

The DOM is a language-neutral set of objects for representing the nodes (or elements) and structure of an XML document. The topmost element in the DOM tree is called the *document object* (or sometimes the *root*). The beauty of XML and TEI-XML in particular is that it provides us with a structured environment for storing information. In some sense, an XML document can be viewed as a kind of database in which different types of information get stored in different fields. In a database, for example, you might have an *author* table containing fields for *last name*, *first name*, *birthplace*, *gender*, etc. You might then have a *paragraph* table that stores each paragraph of a document associated or *linked* to a specific author. This same kind of information can be easily stored in XML, and the TEI standard gives us some specific rules, or *standards*, for how to organize such documents. Because TEI is an agreed upon set of standards for the encoding of literary texts, you can develop programs in R (and many other languages) for processing these TEI based documents.

To get your feet wet parsing XML in R, you will begin by loading the XML library. In order to do that, however, you'll first need to install the XML package.

10.4 Installing R Packages

R packages are collections of functions developed by programmers in the open source R community. These packages add functionality to R that is not part of the base installation.

There are several ways to install packages in R. The simplest and most direct way is to type the command `install.packages()` at the `R` prompt. The first time you install a package, you will be asked to select a *mirror*, a server location from which to download the software. When asked to select a mirror, choose one that is geographically close to your location. You will then be prompted with a list of all the available packages. If you are using the `R` GUI, you will find a menu option for *Packages & Data*. Under this menu option, you can select *Package Installer*. In `RStudio` you will find an *Install Packages* option under the main *Tools* menu. Once the installer window pops up, type XML (caps sensitive) in the *Packages* field and check the box to *install dependencies*. Checking this box insures that you get any other software that the `XML` package requires.[2]

10.5 Loading and using the `XML` Package

Once the `XML` package (or any package for that matter) is installed, you must call it into the active `R` session. For this you use the expression:

```
library(XML) # note that "XML" here is caps sensitive.
```

Unlike the simple `scan` function that you used to read text files of *Moby Dick* and *Sense and Sensibility*, with `XML` files you'll need some more sophisticated function that can understand the structure of `XML`. If you are just opening `R` be sure to clear your workspace and reset you working directory. Begin by loading the `XML` version of *Moby Dick*:[3]

```
doc<-xmlTreeParse("data/XML1/melville1.xml", useInternalNodes=TRUE)
```

When working with the plain text version of *Moby Dick*, you found the chapter breaks using `grep`; finding the chapter breaks in the `XML` file is a lot easier because the chapters are all *marked up* in the `XML` using a `<div1>` element and a "chapter" attribute. You can gather the chapters easily using an *XPath expression* ("/d:TEI//d:div1[@type='chapter']"). `XPATH` is a language for representing and selecting `XML` nodes, or `elements` in an `XML` document. `XPATH` uses forward slashes to represent the ordered hierarchy of nodes in the document much in

[2] While it is possible to download all of the available packages, doing so would certainly take a long time and would clog up your installation with way too many irrelevant features. The fact is that `R` is a multipurpose platform used in a huge range of disciplines including: bio-statistics, network analysis, economics, data-mining, geography, and hundreds of other disciplines and sub-disciplines. This diversity in the user community is one of the great advantages of `R` and of open-source software more generally. The diversity of options, however, can be daunting to the novice user, and, to make matters even more unnerving, the online `R` user community is notoriously specialized and *siloed* and can appear to be rather impatient when it comes to *newbies* asking simple questions. Having said that, the online community is also an incredible resource that you must not ignore. Because the packages developed for `R` are developed by programmers with at least some amount of *ad hoc* motivation behind their coding, the packages are frequently weak on documentation and generally assume some, if not extensive, familiarity with the academic discipline of the programmer (even if the package is one with applications that cross disciplinary boundaries).

[3] Notice the different path here. The `XML` version of *Moby Dick* is located in a different subdirectory of the main TextAnalysisWithR.

the same way that R and UNIX and other systems and languages use forward slashes to represent the structure of the directories (or folders) in your computer. Send the XML document object (now in a variable named doc) along with this XPath expression to the getNodeSet function, and it will return a new Node Set object. The third argument required by the getNodeSet function is a bit tricky to understand because it has to do with XML *namespaces*, which is not so much about R as it is about XML. The getNodeSet function expects us to identify an XML namespace as an item in a vector, so in what follows I have called that vector d. The d prefix is used as a prefix in each part of the XPath expression.

```
chapters.ns.l<-getNodeSet(doc,
  "/d:TEI//d:div1[@type='chapter']",
  namespace = c(d = "http://www.tei-c.org/ns/1.0"))
```

If you enter class(chapters.ns.l), you'll see that chapters.ns.l is an XMLNodeSet object which is a sub-class of an R list object (which is why I have used the ".ns.l" extension). chapters.ns.l is a special kind of list in that each item in the list is an XML node. What this means is that as you iterate over the list, you must employ XML-based functions to further refine the operations. For example, each XML chapter node encloses a <head> node as a child.[4] This <head> node is where the title of the chapter is stored. Enter the following expression to examine the contents of the first list item.

```
chapters.ns.l[[1]]
```

If you scroll up in the R console, you'll see the beginning of the chapter:

```
chapters.ns.l[[1]]
<div1 type="chapter" n="1" id="\_75784">
  <head>Loomings</head>
  <p rend="fiction">Call me Ishmael. Some years ago-
  never mind how long precisely- having little. . .
```

Notice that the chapter title, *Loomings*, is inside the <head> element. If you enter class(chapters.ns.l[[1]]), you will see that the first item of this R list is an XML node:

```
class(chapters.ns.l[[1]])
## [1] "XMLInternalElementNode"
## [2] "XMLInternalNode"
## [3] "XMLAbstractNode"
```

One way to extract the <head> node of the chapter is to use the xmlElementsByTagName function. To get the <head> node from the first chapter, you might write:

```
chap.title<-xmlElementsByTagName(chapters.ns.l[[1]], "head")
chap.title
## $head
## <head>Loomings</head>
```

But what you really want is not the entire node as an XML object, but rather, the *content* of the node. To get the content, that is, the chapter title *Loomings*, you also need the xmlValue function.[5]

[4] A node inside of another node is often referred to as a "child" node.

[5] Notice that the chap.title object is another type of list, which is why the further bracketed sub-setting is required in order to get at the text contents.

```
xmlValue(chap.title[[1]])
## [1] "Loomings"
```

With a little understanding of R lists from the first part of this book and with some sense of how TEI XML files are structured, you can put all of this together and generate a chapter-by-chapter analysis of *Moby Dick* exactly as you did previously with help from grep.

In TEI the textual data of each chapter is stored inside of <p> elements. What this means is that for each chapter (<div1>) you want to extract both the title of the chapter (found inside of <head>) and the paragraphs (found inside <p>) as two separate items. Ultimately you want to produce a new list in which each item in the list is named with the chapter title and the value of the named list item is a table of words. This is exactly what you did with the plain text files earlier in this book; from the XML file, you create a list object identical to the one created from the plain text version of *Moby Dick*. And, just as you did earlier, here again you will use a for loop to iterate over the items in the chapters.ns.l object.

First create two list objects in which to store the results:

```
chapter.freqs.l<-list()
chapter.raws.l<-list()
```

And now the for loop where all the work gets done. Comments have been added to explain the workings of the loop.

```
for(i in 1:length(chapters.ns.l)){
  # first get the chapter title from the head element
  chap.title<-xmlValue(xmlElementsByTagName(chapters.ns.l[[i]], "head")[[1]])
  # get only the contents of the paragraph tags
  paras.ns<-xmlElementsByTagName(chapters.ns.l[[i]], "p")
  #combine all the words from every paragraph
  chap.words.v<-paste(sapply(paras.ns, xmlValue), collapse=" ")
  # convert to lowercase
  words.lower.v <-tolower(chap.words.v)
  # tokenize
  words.l<-strsplit(words.lower.v, "\\W")
  word.v<-unlist(words.l)
  word.v<- word.v[which(word.v!="")]
  # calculate the frequencies
  chapter.freqs.t<-table(word.v)
  chapter.raws.l[[chap.title]]<- chapter.freqs.t
  chapter.freqs.l[[chap.title]]<-100*(chapter.freqs.t/sum(chapter.freqs.t))
}
```

After execution, you will have two list objects derived from an TEI-XML version of *Moby Dick*. These new lists will be almost identical to the lists created from the plain text version of the novel.[6] You can now run the exact same analysis of *whale* and *ahab* that you ran when working with the plain text:

```
whales<-do.call(rbind, lapply(chapter.freqs.l, '[', 'whale'))
ahabs<-do.call(rbind, lapply(chapter.freqs.l, '[', 'ahab'))
whales.ahabs<-cbind(whales, ahabs)
whales.ahabs[which(is.na(whales.ahabs))]<-0
colnames(whales.ahabs)<-c("whale", "ahab")
```

And naturally, just as you did with the plain text versions, you can run a correlation test to see if *whales* and *ahabs* behave in a correlated manner:

[6] They won't be exactly the same because the come from slightly different sources.

```
barplot(whales.ahabs, beside=T, col="grey")
```

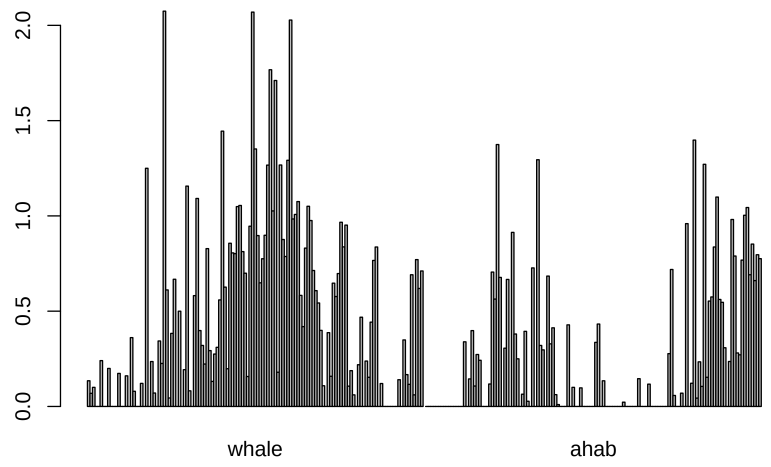


Fig. 10.1 Barplot showing occurrences of *whale* and *ahab*

```
whales.ahabs.df<-as.data.frame(whales.ahabs)
cor.test(whales.ahabs.df$whale, whales.ahabs.df$ahab)
##
##  Pearson's product-moment correlation
##
## data:  whales.ahabs.df$whale and whales.ahabs.df$ahab
## t = -2.798, df = 131, p-value = 0.005915
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  -0.39187 -0.07009
## sample estimates:
##     cor
## -0.2375
```

10.6 Metadata

The real power of processing XML files will become apparent when you are dealing with multiple files. For now, however, it is worth pointing out that with TEI-XML, you have access to a great deal of human-added metadata that is not part of the raw text of the novel.

Here are five examples that use the xpathApply function to find information contained in specific places inside the TEI-XML hierarchy. XPath allows you to use doubles slashes “//” to indicate the stepping down from one level to several levels deeper. Enter the following expression, for example, to find some additional details about the title of *Moby Dick*:

```
xpathApply(doc,
           "/d:TEI//d:fileDesc//d:titleStmt//d:title",
           namespaces=c(d = "http://www.tei-c.org/ns/1.0"))
```

This expression asks R and the XML parser to begin at the root </TEI> and then descend into the document until a <fileDesc> element is found. From there the parser should continue to descend the tree looking for a path that contains a <titleStmt> element. The parser should continue looking deeper for a <title> element that is a descendant of <titleStmt>. After you run this code, the result, an xmlNodeSet, will look like this:

```
## [[1]]
## <title type="main">Moby Dick; Or The Whale</title>
##
## attr(,"class")
## [1] "XMLNodeSet"
```

To refine the result further, and get that title out of the node set, modify the line to reference the item held inside the first ([[1]]) position:

```
xpathApply(doc,
           "/d:TEI//d:fileDesc//d:titleStmt//d:title",
           namespaces=c(d = "http://www.tei-c.org/ns/1.0"))[[1]]
## <title type="main">Moby Dick; Or The Whale</title>
```

And you can even assign the result to a new variable called title:

```
title<-xpathApply(doc,
                  "/d:TEI//d:fileDesc//d:titleStmt//d:title",
                  namespaces=c(d = "http://www.tei-c.org/ns/1.0"))[[1]]
title
## <title type="main">Moby Dick; Or The Whale</title>
```

Notice how I first put a result into a variable called title and then issued a second command to print the contents of title. You can save yourself some time by taking advantage of a handy R shortcut that allows you to wrap any R expression inside parentheses. Doing so instructs R to not only perform the instructions inside the parentheses but to then print the result to the console. Here are a few examples using parentheses to both perform and print a given expression:

```
(title<-xpathApply(doc,
                   "/d:TEI//d:fileDesc//d:titleStmt//d:title",
                   namespaces=c(d = "http://www.tei-c.org/ns/1.0"))[[1]])
## <title type="main">Moby Dick; Or The Whale</title>
(author<-unlist(xpathApply(doc,
                           "/d:TEI//d:author//d:name",
                           namespaces=c(
                             d = "http://www.tei-c.org/ns/1.0")))[[1]])
## <name type="main">Herman Melville</name>
```

In addition to providing access to the contents of the elements themselves, the XML parser gives access to the attributes of the elements. Notice, for example, that the <note> element has two attributes, one called *nation* with a value *American* and another called *gender* with a value of *M*.

```
xpathApply(doc,
           "/d:TEI//d:teiHeader//d:note",
           namespaces=c(d = "http://www.tei-c.org/ns/1.0"))
```

```
## [[1]]
## <note nation="American" gender="M"/>
##
## attr(,"class")
## [1] "XMLNodeSet"
```

To get these attribute values, you can add a `xmlGetAttr` function to the expression. Below are three examples for you to try.

```
(nation<-unlist(xpathApply(doc,
                           "/d:TEI//d:teiHeader//d:note",
                           namespaces=c(d = "http://www.tei-c.org/ns/1.0"),
                           xmlGetAttr, "nation")))
## [1] "American"
(gender<-unlist(xpathApply(doc,
                           "/d:TEI//d:teiHeader//d:note",
                           namespaces=c(d = "http://www.tei-c.org/ns/1.0"),
                           xmlGetAttr, "gender")))
## [1] "M"
(pubdate<-unlist(xpathApply(doc,
                            "/d:TEI//d:teiHeader//d:creation/d:date",
                            namespaces=c(d = "http://www.tei-c.org/ns/1.0"),
                            xmlGetAttr, "value")))
## [1] "1851"
```

Practice

10.1. Using what you have learned about accessing metadata contained in `XML` files, write a single line `R` expression to extract the information contained in the `<respStmt>` element of the `TEI` header.

Part III
Macroanalysis

Chapter 11
Clustering

This chapter moves readers from the analysis of one or two texts to a larger corpus. Machine clustering is introduced in the context of an authorship attribution problem.

11.1 Introduction

This chapter introduces you to document clustering using a rather small corpus. It might be good to think about this experiment as a prototype, or model, for a much larger experiment. Many of the basic tasks will be the same, but instead of working with 4,300 books you'll hone your skills using just 43. Much of the processing done in this section will be familiar to you from the first half of the book where you developed code to compare the vocabulary richness of *Moby Dick* on a chapter by chapter basis. Here, instead of chapters, you'll have entire books to work with. The raw materials and the basic `R` objects will be the same.

11.2 Review

At the `R` prompt, use `setwd` to change the directory path to match the location of the TextAnalysisWithR directory associated with this book. If you prefer to leverage the `RStudio`'s menu system, just select *Set Working Directory* from the *Session* menu. You can then use your computer's windowing system to navigate to the location of the TextAnalysisWithR directory.[1]

[1] If you have been working on other sections of this book or on `R` projects of your own, it might be a good idea to either restart `R` or to clear the `R` workspace. To do the latter, just click on the *Session* menu of the `RStudio` GUI and select *Clear Workspace*. This will remove all `R` objects and functions that you may have been using, wiping the `R` slate clean, as it were.

```
setwd("~/Documents/TextAnalysisWithR")
```

Once this command is executed, the R prompt > will return and you will be able to enter a new command. Check to see that you set the working directory correctly using:

```
getwd()
## [1] "/Users/mjockers/Documents/TextAnalysisWithR"
```

If all is well, you will get a response that looks something like the above:

11.3 Some Oddities in R

You may have noticed here, or elsewhere in this book, that R has prefixed its returned result with a bracketed 1. The fact is that everything in R is vectorized. Even getwd returns its result as a vector. What you are seeing here with the [1] is that the result of calling getwd is a vector with one item. This behavior can feel especially strange to those who are coming to R from another language. The truth is that R has a number of funky behaviors. I have already noted that R indexes vectors beginning with 1; whereas many other languages begin with 0. One of the strangest R features that I am aware of involves the non-value NA. NA is used in R to represent missing values (*non available*). So suppose you had some data about the nationality of different authors:

```
nations<-c(Joyce="Irish", Twain="American", Dickens="English")
```

Now suppose that you found a new author and were not sure of the author's nationality. You could enter that author's nationality as NA:

```
nations<-c(nations, Smith=NA)
```

Notice here that NA is entered without quote marks around it. This is to indicate to R that it is not being entered as a character string but a special set of reserved characters indicating *no value* or *missing value*. You can check on this using the is.na function that tests to see if a value is NA or not:

```
is.na(nations)
##   Joyce   Twain Dickens   Smith
##   FALSE   FALSE   FALSE    TRUE
```

As you can see, the value for Smith is indeed NA and not a character string. The surprising thing about this is that when you then ask R to compute a count of the number of characters in each of the strings in the nations vector, R returns a count of 2 for the NA value for Smith!

```
nchar(nations)
##   Joyce   Twain Dickens   Smith
##       5       8       7       2
```

Irish = 5, yes; *American* = 8, yes; *English* = 7, yes; but *NA* = 2? How could this be since NA is not a string? It is definitely *not* because there are two characters in the string "NA" because NA is specifically not a string in this example! The answer turns out to be an oddity in the way R was designed, and it is only by consulting the help documentation for nchar (e.g. ?nchar) that you learn that *currently* nchar always returns 2 for missing values! Why R was designed in this way is not made clear, but the use of "currently" suggests that at least somewhere along the line someone figured this behavior might be something worth changing.[2]

11.4 Corpus Ingestion

The files you will use in this clustering experiment are all stored in the directory located at data/XMLAuthorCorpus. The first thing you will require is a bit of R that will go to this directory and survey its contents. To keep things neat, put the path to this directory into an R object and call it input.dir.

```
input.dir<-"data/XMLAuthorCorpus"
```

You can now use the dir function to generate a vector containing the names of all the files contained inside input.dir.

```
files.v<-dir(path=input.dir, pattern=".*xml")
```

Notice how in addition to the path argument, I have added a *pattern* argument. This *pattern*, a regular expression, tells dir to return only those files with names matching the regular expression.[3] The files.v variable now contains a vector of character strings from the file names of the 43 XML files found inside the XMLAuthors folder. Here they are:

```
files.v
##  [1] "anonymous.xml"  "Carleton1.xml"
##  [3] "Carleton10.xml" "Carleton11.xml"
##  [5] "Carleton12.xml" "Carleton13.xml"
##  [7] "Carleton14.xml" "Carleton2.xml"
##  [9] "Carleton3.xml"  "Carleton4.xml"
## [11] "Carleton5.xml"  "Carleton6.xml"
## [13] "Carleton7.xml"  "Carleton8.xml"
## [15] "Carleton9.xml"  "Donovan1.xml"
## [17] "Donovan2.xml"   "Driscoll1.xml"
## [19] "Driscoll2.xml"  "Driscoll3.xml"
## [21] "Edgeworth1.xml" "Jessop1.xml"
## [23] "Jessop2.xml"    "Jessop3.xml"
## [25] "Kyne1.xml"      "Kyne2.xml"
## [27] "LeFanu1.xml"    "LeFanu2.xml"
## [29] "LeFanu3.xml"    "LeFanu4.xml"
## [31] "LeFanu5.xml"    "LeFanu6.xml"
## [33] "LeFanu7.xml"    "Lewis.xml"
## [35] "McHenry1.xml"   "McHenry2.xml"
## [37] "Norris1.xml"    "Norris2.xml"
## [39] "Norris3.xml"    "Norris4.xml"
## [41] "Polidori1.xml"  "Quigley1.xml"
## [43] "Quigley2.xml"
```

Notice that the very first of these files is titled *anonymous.xml*. This is the file whose authorship is uncertain. In this chapter, you will use text analysis and unsupervised clustering to compare the word frequency *signal* of the anonymous novel

[2] Patrick Burns has written a 125 page book documenting some of R's unusual behavior. The book is informative and entertaining to read. You can find it online at http://www.burns-stat.com

[3] Enter ?regex at the prompt to learn more about *regex* in R.

to the signals of the others, and then, based on that comparison, you will take a guess at which author in the corpus is the most likely author of this anonymous novel.

With the file names in the `files.v` variable, you must now write a bit of code to iterate over all of these values and at each value, pause to load and process the text corresponding to the value; for each file name in the `files.v` vector, you want the script to perform some other bit of processing related to that file. This is very similar to code you have developed in other chapters. You begin with a `for` loop and a new variable called `i` (for integer):

```
for (i in 1:length(files.v)){
  # Some code here
}
```

This expression tells `R` to begin by setting `i` equal to `1` and to iterate over all of the elements of the vector `files.v` and to stop only when it has finished with the 43rd item, which, in this case, is what is returned by the call to the `length` function.

Remember from chapter two that to parse an `XML` file, you used the `xmlTreeParse` function. This `XML` function takes several arguments, the first of which is the path to the location of the file on your computer. The first question you must address, therefore, is how to give the program the information it requires in order to figure out the file path to each of the files in the corpus directory.

You already know how to access one of the items in the `files.v` object via sub setting, so now consider the following expression that uses the `file.path` function to join together the two objects you have instantiated:

```
file.path(input.dir, files.v)
```

Used in this way, `file.path` returns a series of file path expressions that look like this:

```
##  [1] "data/XMLAuthorCorpus/anonymous.xml"
##  [2] "data/XMLAuthorCorpus/Carleton1.xml"
##  [3] "data/XMLAuthorCorpus/Carleton10.xml"
##  [4] "data/XMLAuthorCorpus/Carleton11.xml"
##  [5] "data/XMLAuthorCorpus/Carleton12.xml"
##  [6] "data/XMLAuthorCorpus/Carleton13.xml"
##  [7] "data/XMLAuthorCorpus/Carleton14.xml"
##  [8] "data/XMLAuthorCorpus/Carleton2.xml"
##  [9] "data/XMLAuthorCorpus/Carleton3.xml"
## [10] "data/XMLAuthorCorpus/Carleton4.xml"
## [11] "data/XMLAuthorCorpus/Carleton5.xml"
## [12] "data/XMLAuthorCorpus/Carleton6.xml"
## [13] "data/XMLAuthorCorpus/Carleton7.xml"
## [14] "data/XMLAuthorCorpus/Carleton8.xml"
## [15] "data/XMLAuthorCorpus/Carleton9.xml"
## [16] "data/XMLAuthorCorpus/Donovan1.xml"
## [17] "data/XMLAuthorCorpus/Donovan2.xml"
## [18] "data/XMLAuthorCorpus/Driscoll1.xml"
## [19] "data/XMLAuthorCorpus/Driscoll2.xml"
## [20] "data/XMLAuthorCorpus/Driscoll3.xml"
## [21] "data/XMLAuthorCorpus/Edgeworth1.xml"
## [22] "data/XMLAuthorCorpus/Jessop1.xml"
## [23] "data/XMLAuthorCorpus/Jessop2.xml"
## [24] "data/XMLAuthorCorpus/Jessop3.xml"
## [25] "data/XMLAuthorCorpus/Kyne1.xml"
## [26] "data/XMLAuthorCorpus/Kyne2.xml"
## [27] "data/XMLAuthorCorpus/LeFanu1.xml"
## [28] "data/XMLAuthorCorpus/LeFanu2.xml"
## [29] "data/XMLAuthorCorpus/LeFanu3.xml"
## [30] "data/XMLAuthorCorpus/LeFanu4.xml"
## [31] "data/XMLAuthorCorpus/LeFanu5.xml"
## [32] "data/XMLAuthorCorpus/LeFanu6.xml"
## [33] "data/XMLAuthorCorpus/LeFanu7.xml"
## [34] "data/XMLAuthorCorpus/Lewis.xml"
## [35] "data/XMLAuthorCorpus/McHenry1.xml"
## [36] "data/XMLAuthorCorpus/McHenry2.xml"
## [37] "data/XMLAuthorCorpus/Norris1.xml"
## [38] "data/XMLAuthorCorpus/Norris2.xml"
## [39] "data/XMLAuthorCorpus/Norris3.xml"
## [40] "data/XMLAuthorCorpus/Norris4.xml"
## [41] "data/XMLAuthorCorpus/Polidori1.xml"
## [42] "data/XMLAuthorCorpus/Quigley1.xml"
## [43] "data/XMLAuthorCorpus/Quigley2.xml"
```

Once wrapped inside a loop, you will be able to easily iterate over these file paths, loading each one in turn. Before you start looping, however, set the new variable `i` equal to `1`. Doing so will allow you to do some prototyping and testing of the code without running the entire loop.

```
i<-1
```

With `i` equal to `1`, reenter the previous expression with the value `i` inside the brackets of the `files.v` variable, like this:

```
file.path(input.dir, files.v[i])
## [1] "data/XMLAuthorCorpus/anonymous.xml"
```

Instead of returning the paths for all of the files, you get just the first file (alphabetically) in the `files.v` object. If you have not already done so, enter the following expression to load the `XML` package:

```
library(XML)
```

With the package loaded and this path name business sorted out, you can now use the `xmlTreeParse` function to ingest, or load, the first `XML` file. Rather than assigning the result of the `file.path` command to another variable, you can just embed one function inside the other, like this:[4]

```
doc<-xmlTreeParse(file.path(input.dir, files.v[i]), useInternalNodes=TRUE)
```

If you enter this expression, and then type `doc` at the `R` prompt, you'll see the entire contents of the file titled *anonymous.xml*. You must now embed this expression within a `for` loop so that you can iterate over all of the `XML` files in the files directory. Do this as follows:

```
for(i in 1:length(files.v)){
  doc<-xmlTreeParse(file.path(input.dir, files.v[i]), useInternalNodes=TRUE)
  # some more code goes here. . .
}
```

This code will consecutively load all of the `XML` files into an object called `doc`. Each time the program starts a new loop, the previous contents of the `doc` object will be overwritten by the new `XML` file. So, before the next iteration begins, you need to process the contents of the `doc` object and store the results in some other variable that will persist beyond the loop. For this, create an empty list called

[4] You can learn more about the `useInternalNodes` argument in the documentation for the `xmlTreeParse` function. Basically, setting it to `TRUE` avoids converting the contents into `R` objects, which saves a bit of processing time.

book.freqs.l before the for loop even begins. This list will serve as a container for the results that will be generated during the processing that takes place inside the loop. Putting it all together, your code should now look like this:

```
book.freqs.l<-list()
for(i in 1:length(files.v)){
  doc<-xmlTreeParse(file.path(input.dir, files.v[i]), useInternalNodes=TRUE)
  #some more code goes here. . .
}
```

11.5 Another Function

Now that you have code for handling the iteration and for loading the XML files, you need to process the loaded files to extract the word frequencies. You have already seen how R's built-in functions work, and you built some simple functions for making *KWIC* lists in earlier chapters. Anytime you know you are going to repeat a task multiple times, it is handy to have a function. For each of the XML files in the corpus, you need to extract, count, and calculate the relative frequency of every word type. This is exactly the process used in the first part of this book, but here you are dealing with XML, so things are a bit different. Since you want to be able to do this task over and over again with any XML file, you can wrap all of this processing in a reusable function called getTEIWordTableList. This function will take just one argument: an XML document object that has been derived by calling the xmlTreeParse function (as we just did above). The function definition begins like this:

```
getTEIWordTableList<-function(doc.object){
  # some code here
}
```

Within the function you will need to add code to extract and process all of the words in the XML files. First grab all of the paragraph nodes using the getNodeSet function from the XML package.

```
paras<-getNodeSet(doc, "/d:TEI/d:text/d:body//d:p",
                  c(d = "http://www.tei-c.org/ns/1.0"))
```

Notice that the second argument of the function is an xpath expression telling the function to only get paragraphs that occur as children of the <body> portion of the XML file. An xpath expression, as you will recall from chapter 10, provides a way of digging down into the structure of an XML file. This expression is used because there might be <front> and <back> matter that is wrapped in <p> tags that is not part of the core text of the novel.[5]

Next you need to join all of the paragraph nodes found by this xpath expression together into a single character string. The getNodeSet function returns a XMLNodeSet object. This object is similar to a list object. The next expression, therefore, combines the sapply and paste functions in order to join all the paragraph node content (found with the xmlValue function) together using a simple blank space as the joining *glue*. You have done this sort of thing before.

[5] See Chapter 10, section 5 for an explanation of the *namespace* argument.

```
words<-paste(sapply(paras,xmlValue), collapse=" ")
```

Next you will need to convert all the words to lowercase and then split the entire string using the `\\W` regular expression. As you saw earlier, this *regex* breaks the text on any *non-word* characters and chunks up the string by word boundaries, splitting the text into word tokens based on any white space or punctuation characters.

```
words.lower <-tolower(words)
words.l<-strsplit(words.lower, "\\W")
```

The result of `strsplit` is a `list`, but you only need a simple vector of words, so use `unlist` to scrap the list elements.

```
word.v<-unlist(words.l)
```

Now you can do a little spring-cleaning. Inevitably there are blank items that get stuck in the vector where punctuation marks would have appeared. You can identify these using a `which` function to locate items with no character content.

```
word.v[which(word.v!="")]
```

This little slight-of-hand expression can be embedded inside another function so that you do not have to instantiate another variable as a temporary storage place for its result. In other words, embed all of this inside a `table` function which will return the final word frequency data you require:

```
book.freqs.t<-table(word.v[which(word.v!="")])
```

With this done, it is simple to calculate the relative word frequencies by dividing the raw counts by the sum of words in the entire book. Optionally, you can multiply the result by 100 to make the percentages easier to read

```
book.freqs.rel.t<-100*(book.freqs.t/sum(book.freqs.t))
```

The last step of this function is to return this final data back to the main script. For that employ the `return` function:

```
return(book.freqs.rel.t)
```

The whole function now looks like this:

```
getTEIWordTableList<-function(doc.object){
  paras<-getNodeSet(doc,
                    "/d:TEI/d:text/d:body//d:p",
                    c(d = "http://www.tei-c.org/ns/1.0"))
  words<-paste(sapply(paras,xmlValue), collapse=" ")
  words.lower <-tolower(words)
  words.l<-strsplit(words.lower, "\\W")
  word.v<-unlist(words.l)
  book.freqs.t<-table(word.v[which(word.v!="")])
  book.freqs.rel.t<-100*(book.freqs.t/sum(book.freqs.t))
  return(book.freqs.rel.t)
}
```

Save this function to your *corpusFunctions.r* file in your code directory and add a line to your main working script to call this supporting file:

```
source("code/corpusFunctions.r")
```

You can now embed a call to this function into the `for` loop that you began building up previously. Your main script should now look like this:

```
source("code/corpusFunctions.r")
book.freqs.l<-list() # a list object to hold the results
for(i in 1:length(files.v)){
  doc<-xmlTreeParse(file.path(input.dir, files.v[i]), useInternalNodes=TRUE)
  worddata<-getTEIWordTableList(doc)
  book.freqs.l[[files.v[i]]]<-worddata
}
```

When run, the values returned from processing each text with the function are added to the `book.freqs.l` list that was created before entering the loop. After processing, all of the data necessary for continuing the clustering experiment will be contained in the single list. Before you go on, however, you might want to inspect this new object. Try a few of these commands, and be sure that the results all make sense to you:

```
class(book.freqs.l)
names(book.freqs.l)
str(book.freqs.l)
```

11.6 Unsupervised Clustering and the Euclidean Metric

Many years of authorship attribution research have taught us that the most effective way to distinguish between the text of one author and another is by comparing the different usages of high frequency features in their writing. High frequency features include words such *the*, *of*, *and*, *to*, etc. as well as, in some studies, marks of punctuation and even common bi-grams, such as *of the*. Here I will assume some familiarity with the concept of distinct stylistic *signals* and jump right into describing a process for comparing the word usage patterns of the writers in the sample corpus.[6]

The technique that I describe here involves a measurement known as *Euclidean distance*. Using the *Euclidean* metric, or what is sometimes called the *Pythagorean* metric, you can calculate each single book's *distance* from every other book in a corpus. Books with a closer distance will have more in common in terms of their feature usage habits, and books with a greater relative distance will be dissimilar. For the sake of illustration, assume that you have just three books and only two features for each book. Call the three books *a*, *b*, and *c*, and the two features *f1* and *f2*. Assume further that the measurements of the two features in each of the book are frequencies per one-hundred words, as follows:

```
my.m
##    f1 f2
## a 10  5
## b 11  6
## c  4 13
```

That is, in book *a*, feature *f1* occurs ten times per one-hundred words and feature *f2* occurs five times per hundred. In book *b* feature *f1* is found eleven times for

[6] For a brief overview of how this work is conducted, Jockers, Matthew L. *Macroanalysis: Digital Methods and Literary History*. University of Illinois Press, 2013. Pages 63-67

every one-hundred words and so on. You can represent this information in an `R` matrix using this code:

```
a<-c(10, 5)
b<-c(11,6)
c<-c(4,13)
my.m<-rbind(a,b,c)
colnames(my.m)<-c("f1", "f2")
```

These feature measurements can in turn be represented as `x` and `y` coordinate values and plotted in a two dimensional space, as in figure 11.1. Once plotted, you

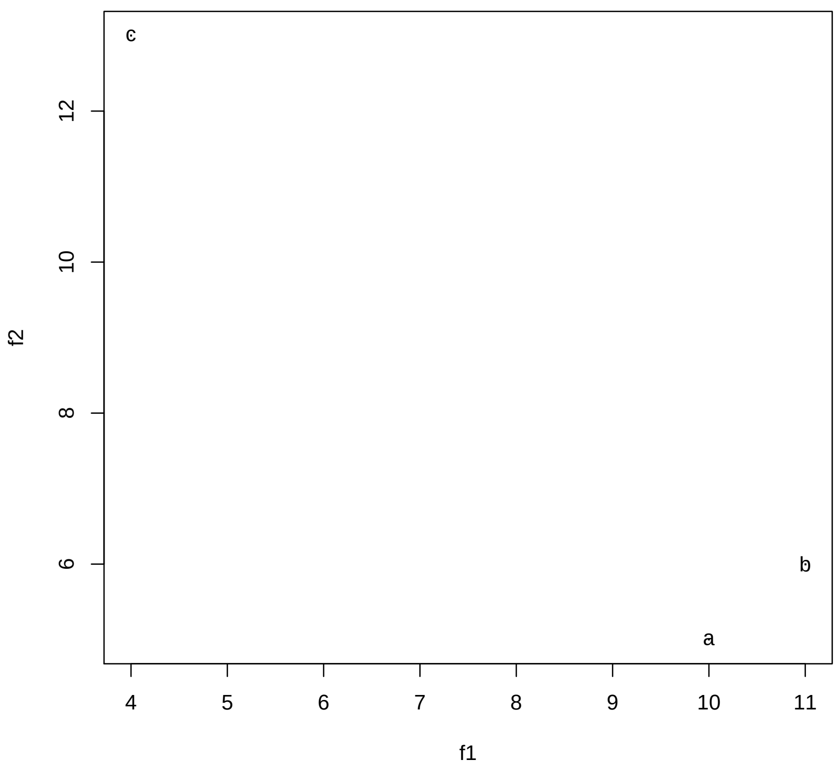


Fig. 11.1 Two-Dimensional Plotting

can measure (as with a ruler) the distance on the grid between the points. In this case, you would find that books *a* and *b* are closest (least distant) to each other. Naturally, you don't want to actually plot the points and then use a ruler, so instead you can employ the `dist` function in `R`.

```
dist(my.m)
##        a      b
## b  1.414
## c 10.000  9.899
```

The result reveals that the *standard* or *ordinary* distance between points *a* and *b* is `1.4`, the distance between *a* and *c* is `10`, and the distance between *b* and *c* is `9.9`. These *distances* provide a way of describing the relative nearness of the points, and, ultimately, the similarity of the documents from which these values were extracted. For convenience, you can think of these distances as *meters*, *feet*, or *miles*; it does not ultimately matter since you are only concerned with the relative closeness of the points. In this example, using only two features (*f1* and *f2*) you would conclude that book *a* and book *b* are the most similar.

When there are only two dimensions (or features) as in this example, the plotting and measuring is fairly simple and straightforward. It becomes more complex when thought of in terms of *fifty* or *five-hundred* features and *twenty* or *forty* books. Nevertheless, the closeness of items in this high dimensional space can still be calculated using the Euclidean metric (which is the default method employed by `R`'s `dist` function. The metric is expressed like this:

$$d(p,q) = \sqrt{(p_1-q_1)^2+(p_2-q_2)+...+(p_i-q_i)2+(p_n-q_n)^2}.$$

Where *d* is the distance and *p* and *q* are two books.

$$p_1$$

is the measure of feature one in book *p* and

$$q_1$$

is the measure of feature one in book *q*, and so on through all of the features.

Assume you have a new data set in which there are four features:

```
a<-c(10, 5, 3, 5)
b<-c(11,6, 5, 7)
c<-c(4,13, 2, 6)
my.m<-rbind(a,b,c)
colnames(my.m)<-c("f1", "f2", "f3", "f4")
```

Using the Euclidean metric, the distances *d* between books (*a, b, c*) are calculated as follows:

$$d(a,b) = \sqrt{(10-11)^2+(5-6)^2+(3-5)^2+(5-7)^2} = 3.162278$$

$$d(a,c) = \sqrt{(10-4)^2+(5-13)^2+(3-2)^2+(5-6)^2} = 10.09950$$

$$d(b,c) = \sqrt{(11-4)^2+(6-13)^2+(5-2)^2+(7-6)^2} = 10.39230$$

To get the same results in `R`, you simply enter:

```
dist(my.m)
##        a      b
## b  3.162
## c 10.100 10.392
```

You see that the distance between *a* and *b* (3.162278) is much smaller than the distance between *a* and *c*. This indicates that *a* and *b* are more similar to each other in terms of these four features. Using R it is a trivial matter to calculate the distances between every book and every other book in the example corpus. Everything you need for doing this calculation is already stored inside the `book.freqs.l` list object. You simply need to get that data out of the list and into a data matrix in which each row is a book and each column is one of the word features.

11.7 Converting an R List into a Data Matrix

Before you can apply the Euclidean metric to the authorship data, you need to get the word frequency information out of the `book.freqs.l` list and into a data matrix in which each row is a book and each column is a word feature. The cells in this matrix will contain the relative frequency values that were calculated using `getTEIWordTableList`. The first step in this process involves converting the `book.freqs.l` into an R data.frame. For this, the `mapply` function is handy. Here you will create a new list object called `freqs.l` that is similar to `book.freqs.l` except that each list item is converted from a `table` object to a `data.frame` object. I'll show the code first and then explain it.

```
freqs.l <- mapply(data.frame,
                  ID=seq_along(book.freqs.l),
                  book.freqs.l, SIMPLIFY=FALSE,
                  MoreArgs=list(stringsAsFactors=FALSE))
```

In this first step, the word frequency tables held in the list items of `book.freqs.l` are converted into individual `data.frame` objects. To compare the difference, type the following code at the R prompt.

```
class(freqs.l[[1]])
## [1] "data.frame"
class(book.freqs.l [[1]])
## [1] "table"
```

The first item in `freqs.l` is a `data.frame` object; whereas, the first item in the original `book.freqs.l` was a `table` object. The `mapply` function used here is in the same family of functions as `lapply` and `sapply` that you have been using elsewhere. `mapply` differs in a couple of ways. First, `mapply` takes a function name (whereas `sapply` and `lapply` have the list object as the first argument). In this case, the function is `data.frame`. Next is a sequence of *ID* numbers corresponding to the books in the corpus. For this, you use `seq_along` to construct the *id* numbers (`ID=seq_along(book.freqs.l)`) from the information contained in the `book.freqs.l` object.[7] In this case, the numbers are `1` through `43`.

[7] `seq_along` is a simple R function for generating a sequence of numbers. Check the `R-help` documentation for details. In this example, I could have just as easily used `1:43` or 1:length(`book.freqs.l`)

The next argument is the actual list object to which the `data.frame` function will be applied. The `SIMPLIFY` argument is set to `FALSE`, so that `mapply` will not try to convert the result into simpler data type (i.e. a vector or matrix). Finally, use the `MoreArgs` argument to let `mapply` know that when building the data frames, you do not want to convert any of the columns into factors.

The result of running `mapply` is that each item in the resulting list (titled `freqs.l`) is a data frame with some number of rows and 3 columns. The three columns are automatically labeled as `ID` (which is taken from the `ID` element given to `mapply`), `Var1` (which is the column of unique word types), and `Freq` (which is the frequency calculated for that word type in the given text). To see the first ten rows of the data frame contained inside first list item (in this case the anonymous text which has been assigned an `ID` of `1` use the following expression.

```
freqs.l[[1]][1:10,]
##    ID Var1      Freq
## 1   1   02 0.0009794
## 2   1   03 0.0009794
## 3   1   05 0.0009794
## 4   1    1 0.0039177
## 5   1   10 0.0019588
## 6   1   11 0.0009794
## 7   1   12 0.0029383
## 8   1   15 0.0009794
## 9   1 1850 0.0009794
## 10  1 1916 0.0009794
```

In this case the word types happen to all be digits because our *regex* expression treats numbers as word characters. To see some more interesting information, try this expression

```
freqs.l[[1]][100:110,]
##     ID           Var1      Freq
## 100  1           aces 0.0009794
## 101  1           ache 0.0019588
## 102  1        achieve 0.0009794
## 103  1    acknowledge 0.0039177
## 104  1   acknowledged 0.0019588
## 105  1   acknowledges 0.0009794
## 106  1 acknowledgment 0.0009794
## 107  1   acquaintance 0.0078354
## 108  1     acquainted 0.0029383
## 109  1     acquiesced 0.0009794
## 110  1        acquire 0.0068560
```

Remember that in order to use the `dist` function, you need to get all of this data into a `matrix` object. The next step, therefore, will be to use `do.call` with an `rbind` argument to *bind* all the data in all the lists into a single 3-column `data.frame`.

```
freqs.df <- do.call(rbind,freqs.l)
```

If you use the `dim` function you can check the size of your data.frame.

```
dim(freqs.df)
## [1] 298792      3
```

You can also inspect the data contained in the data frame using sub-setting. Entering the following expression, for example, will show the data in rows `100` through `110`:

```
freqs.df[100:110,]
##      ID           Var1      Freq
## 100  1            aces 0.0009794
## 101  1            ache 0.0019588
## 102  1         achieve 0.0009794
## 103  1     acknowledge 0.0039177
## 104  1    acknowledged 0.0019588
## 105  1    acknowledges 0.0009794
## 106  1  acknowledgment 0.0009794
## 107  1    acquaintance 0.0078354
## 108  1      acquainted 0.0029383
## 109  1      acquiesced 0.0009794
## 110  1         acquire 0.0068560
```

You will notice when you run this expression, that R provides the column names for this data frame: ID, Var1, and Freq. Under the column titled ID, you are seeing the unique identifiers for the files in the original files.v vector object. In other words, the rows that have a 1 under the ID column are rows derived from the *anonymous.xml* file, which happens to be indexed at the first position in files.v. The items in the Var1 column are the word types, and the Freq column contains the relative frequencies.

What you now have in the freqs.df object is a *long form* table. The next step is to reshape the data frame to a *wide* format so that there are only 43 rows (one for each text) and 52,048 columns, one for each unique word type in the whole corpus.

This reshaping can be achieved in a number of different ways. I'll use the xtabs function since it is ideally suited to the conversion task and relatively intuitive.[8]

```
result <- xtabs(Freq ~ ID+Var1, data=freqs.df)
```

xtabs is a function specifically designed for creating *contingency tables*, or *cross tabulations*. You might be more familiar with this concept as a *pivot* table in Excel. The function takes as its first argument a *formula* with the cross-classifying elements joined together using a plus (+) sign on the right hand side of a tilde (~) sign. The items joined by the plus sign are the cross classifying elements that will become the row and column names in the resulting table. The item to the left of the tilde is the element that will be entered into the cells of the resulting table. Used in this way, xtabs converts the long, three column, data frame into a wide format data frame in which each row is a book and each column is a word feature. If you run the previous expression and check the dimensions, you'll see that the data.frame is now 43, 51733.

```
dim(result)
## [1]    43 51733
```

And, if you enter

```
colnames(result)
```

You will get a very long list of all of the word features. For an easier going peek, you might try

[8] Other options include using reshape and expressions that leverage the apply family of functions.

```
colnames(result)[100:110]
##  [1] "aces"            "ache"
##  [3] "achieve"         "acknowledge"
##  [5] "acknowledged"    "acknowledges"
##  [7] "acknowledgment" "acquaintance"
##  [9] "acquainted"      "acquiesced"
## [11] "acquire"
```

You are almost there! Before you can perform the final analysis, you need to clean up the data types. Right now you'll see that the `result` variable is an object of the `xtabs table` variety.

```
class(result)
## [1] "xtabs" "table"
```

To be useful in the next step, you need to convert `result` into a matrix object. As with most things in `R`, there are many roads to the same destination. For efficiency, I use `apply` to convert the columnar data to numeric format and by extension in to a numerical `matrix` object. The `2` in this expression is a reference to the "columns" (a `1` here would serve as a reference to "rows.")

```
final.m<-apply(result, 2, as.numeric)
```

This generates a numeric matrix of 43, 51733.

11.8 Preparing Data for Clustering

While it is certainly the case that you could apply the Euclidean metric to this huge matrix, doing so does not make a lot of sense in the case of authorship attribution. The goal is to figure out which of these texts is most *stylistically* similar to the anonymous text, and you don't want to bias the results by clustering the texts based on the similarity of their themes or content. Say, for example, that two of the books in this corpus were about horses. These two books would likely be drawn together in the clustering because they shared a similar *subject* and not necessarily because they shared a similar *style*. Therefore before clustering it is useful to winnow the data to just those features that are extremely frequent.

There are many ways to do this winnowing; you could, for example, sort the data and keep only the 100 most frequent words in the corpus. I prefer to use a winnowing method based on setting a frequency threshold. In other words, limit the feature list to only those words that appear across the entire corpus with a mean relative frequency of some threshold. For this example, I will set the threshold to `0.25`, one quarter of one percent. Here again the `apply` function becomes useful, and in a single line, I can calculate the column means and generate a subset of `final.m` that consists of only those columns that have a combined mean of at least `0.25`.[9] In the following `R` expression, the number `2` is used to tell `R` to perform the calculations over *columns* rather than *rows*. Rows, as noted above, would be referenced with a `1`.

[9] Remember that the `getTEIWordTableList` function that we built multiplies all the relative frequencies by `100`.

```
smaller.m <- final.m[,apply(final.m,2,mean)>=.25]
```

You can now compare the size of the original matrix to the new, smaller one.

```
dim(final.m)
## [1]    43 51733
dim(smaller.m)
## [1] 43 53
```

Using `0.25` reduced the feature set to the `53` most frequent word features in the corpus. With the data matrix reduced in this way, you can now run the clustering very efficiently.

11.9 Clustering Data

With the hard work of data preparation over, you move to the simple and rewarding work of extracting some results. `R` has a set of wonderful functions for clustering data such as this. In this exercise you will use `dist` (a function that employs the *Euclidean metric* to create a *distance matrix*) and `hclust` (which clusters the data in a distance matrix).[10] As you may have guessed, the function names are shorthand for *distance* and *hierarchical clustering*. Each of these methods has a variety of arguments that you can enter to fine tune the way you want to run the analysis. For our purposes the default values are adequate:

Using the following code:

```
# Create a distance object
dm<-dist(smaller.m)
# Perform a cluster analysis on the distance object
cluster <- hclust(dm)
# Get the book file names to use as labels.
cluster$labels<-names(book.freqs.l)
# Plot the results as a dendrogram for inspection.
plot(cluster)
```

you can produce a cluster dendrogram (figure 11.2) and visually inspect the tree to identify the known authors and texts that are most similar to *anonymous.xml*. If everything went well, you should have found *anonymous.xml* nestled comfortably between *Kyne1.xml* and *Kyne2.xml*. Peter B. Kyne is the author of the *anonymous.xml*!

[10] For details, consult the documentation for the `dist` and `hclust` functions.

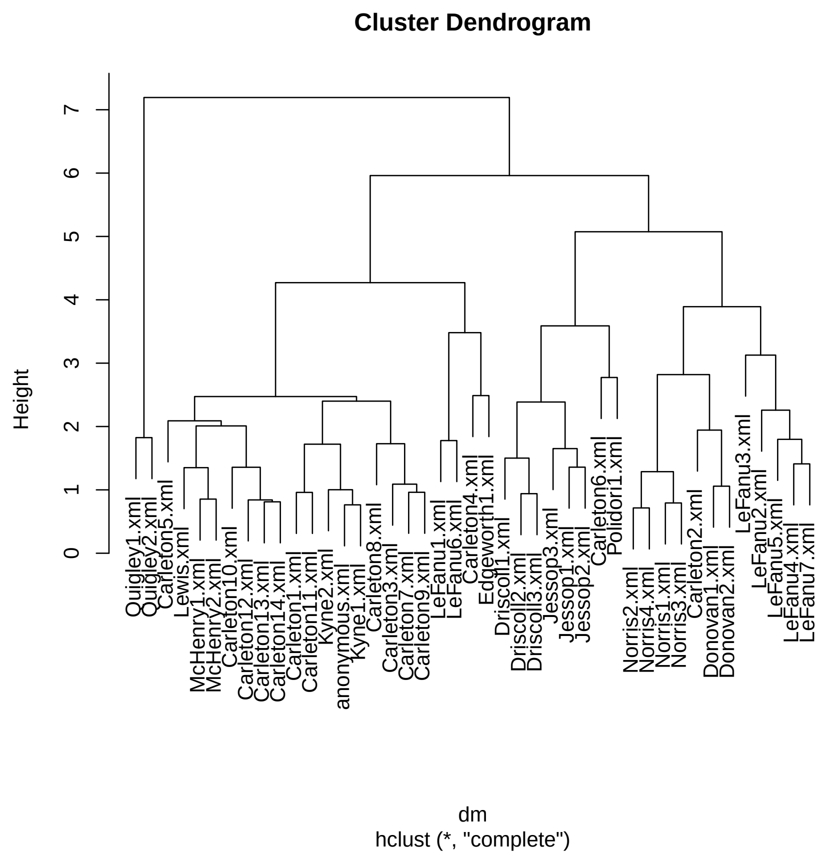


Fig. 11.2 Cluster Dendrogram

Practice

11.1. Now that you have the correct answer, go back to the line of code in which you generated the `smaller.m` matrix. Experiment with different threshold values. Examine how the attribution result changes, or does not, depending upon the number of features that you keep. What is the smallest number of word features you could use in this clustering experiment and still arrive at the same answer?

11.2. As a final experiment, write some code to see what happens if you select a random collection of features. In other words, instead of selecting from among the most high frequency features, write code that uses the sample function to grab a random `sample` of `50` or `100` word features and then see if you still get accurate author clustering.

Chapter 12
Classification

Abstract
This chapter introduces machine classification in the context of an authorship attribution problem. Various methods of text pre-processing are combined here to generate a corpus of `430` text samples. These samples are then used for training and testing a *support vector machines* supervised learning model.

12.1 Introduction

The clustering described in the last chapter is not ideally suited to authorship attribution problems. In fact, clustering is more often used in cases in which the classes are not already known in advance. Clustering is often employed in situations in which a researcher wishes to explore the data and see if there are naturally forming clusters. When the classes are known in advance (i.e. when there is a closed set of possible classes, or *authors* in this case) supervised classification offers a better approach. In addition to providing more information about feature level data, a supervised approach can also provide probabilistic data about the likelihood of a given document being written by one author versus another within the closed set of candidates. Though I will use an authorship attribution example here again, consider that any category of metadata can be inserted into the place held by author. For example, if you wished to gauge the extent to which Irish style differs from British style, you could use *nationality* in place of *author* as the target class.

12.2 A Small Authorship Experiment

For this chapter, you will use the same corpus of novels that was used in the clustering chapter, and you will be able to recycle much of the code you have already written. If you have not already done so, clear your `R` workspace and set the working directory to the location of your TextAnalysisWithR directory. You can now start a new R script with the following code from chapter eleven:

```
setwd("~/Documents/TextAnalysisWithR")
library(XML)
input.dir<-"data/XMLAuthorCorpus"
files.v<-dir(input.dir, ".*xml")
```

12.3 Text Segmentation

Instead of treating every novel as a single text, as was done in the last chapter, here you will first break each text into segments. Instead of having 43 texts for training and testing a classification algorithm, you'll create 430 by first breaking each text into ten equal portions. In the last chapter, you wrote a for loop to load a series of XML files and then send each XML document object to a function (getTEIWordTableList) that would return a table of frequencies. Here you will write a similar function, but before calculating the frequencies and returning the list of tables, this function will first segment each text file into multiple *chunks*.

This new function will take two arguments: an XML document object and a *chunk size* parameter. Let's call the new function getTEIWordSegmentTableList. Just like the previous function you wrote (getTEIWordTableList), this one will also extract the paragraph level content from the XML and then generate a vector of words. Instead of making a table of that entire word vector, however, this new function will first cut the vector into a set number of slices. To get started, you can copy the first few lines from the getTEIWordTableList function and begin a new function:

```
getTEIWordSegmentTableList<-function(doc.object, chunk.size=10){
  paras<-getNodeSet(doc.object,
                    "/d:TEI/d:text/d:body//d:p",
                    c(d = "http://www.tei-c.org/ns/1.0"))
  words<-paste(sapply(paras,xmlValue), collapse=" ")
  words.lower <-tolower(words)
  words.list<-strsplit(words.lower, "\\W|_")
  word.v<-unlist(words.list)
  # . . . some new code here
}
```

The new code that you will now add must calculate the length of the word.v object and divide it by the value of the chunk.size argument, which has been set to 10 as a default.[1]

You already know how to get the length of a vector, so all you need to do here is add a further bit of division :

```
length(word.v)/chunk.size
```

Of course, division like this is never perfect; there are remainders and decimals to deal with. There are several ways you can deal with this, but here is one way that

[1] Inside a function definition, you can define default values for the different arguments. When the function is called from the main script, the default will be used unless you specifically set a value in the function call.

uses the built in `seq_along`, `split` and `ceiling` functions to split the `word.v` vector into a series of chunks:

```
max.length<-length(word.v)/chunk.size
x <- seq_along(word.v)
chunks.l <- split(word.v, ceiling(x/max.length))
```

The resulting `chunks.l` object will be a list in which each item in the list is a character vector. These new character vectors are just slices of the full `word.v` object. They can now be converted into frequency tables in the same way that a table can be generated from the full vector (as you did in chapter 11).

Before you table them, however, you will need remove those pesky blank characters. If you were to peek inside the function at the structure of the `chunks.l` object for a given text, you would see something like this:

```
str(chunks.l)
List of 10
$ 1 : chr [1:7421] "there" "are" "one" "hundred" ...
$ 2 : chr [1:7422] "little" "over" "a" "year" ...
$ 3 : chr [1:7421] "" "for" "" "600" ...
$ 4 : chr [1:7422] "therefore" "predict" "for" "him" ...
$ 5 : chr [1:7421] "there" "is" "no" "good" ...
$ 6 : chr [1:7422] "arrival" "" "he" "engaged" ...
$ 7 : chr [1:7421] "" "if" "such" "a" ...
$ 8 : chr [1:7422] "is" "a" "journal" "of" ...
$ 9 : chr [1:7421] "easily" "made" "in" "california" ...
$ 10: chr [1:7422] "" "" "" "" ...
```

Notice, for example, the first item in the third list item: “”. You need to remove the blanks before making the table, and while there are several ways you could do this (e.g. we used `which` in previous lessons), here I’ll show you how to write a very small, reusable function that can be called iteratively using `lapply`:

```
removeBlanks<-function(x){
  x[which(x!="")]
}
```

Save this function to your *corpusFunction.R* file, and you can add a line to the main `getTEIWordSegmentTableList` function that will send the chunk list object to this new `removeBlanks` function. In this case, since all you are doing is removing the blanks from each character vector, you can overwrite the `chunks.l` object with the new, cleaner version:

```
chunks.l<-lapply(chunks.l, removeBlanks)
```

As seen in previous lessons, the `lapply` function is well-suited for use in combination with the `table` function. You can now table each word vector in the `chunks.l` list using another call to `lapply`.

```
freq.chunks.l<-lapply(chunks.l, table)
```

These raw frequencies may now be converted to relative frequencies, and by now you should know that you can convert a single table of raw counts to relative frequencies like this:

```
freq.chunks.l[[1]]/sum(freq.chunks.l[[1]])
```

Since we do not want to convert each table in the list one at a time, having another function that we can use in a call to `lapply` is handy. Naturally `R` has just such a function: `prop.table`.

```
rel.freq.chunk.l <-lapply(freq.chunks.l, prop.table)
```

In the resulting `rel.freq.chunk.l` object, you will now have a list object containing the relative frequency data for `10` equally sized segments of the text. Here is how your full function should look.

```
getTEIWordSegmentTableList<-function(doc.object, chunk.size=10){
  paras<-getNodeSet(doc.object,
                    "/d:TEI/d:text/d:body//d:p",
                    c(d = "http://www.tei-c.org/ns/1.0"))
  words<-paste(sapply(paras,xmlValue), collapse=" ")
  words.lower <-tolower(words)
  words.list<-strsplit(words.lower, "\\W")
  word.v<-unlist(words.list)
  max.length<-length(word.v)/chunk.size
  x <- seq_along(word.v)
  chunks.l <- split(word.v, ceiling(x/max.length))
  chunks.l<-lapply(chunks.l, removeBlanks)
  freq.chunks.l<-lapply(chunks.l, table)
  rel.freq.chunk.l<-lapply(freq.chunks.l, prop.table)
  return(rel.freq.chunk.l)
}
```

The last line of this function returns a list object back to the main script. If the `chunk.size` argument is left at the default value of `10`, then the resulting list will contain `10` word frequency tables derived from a single book. Save this new function to your *corpusFunctions.R* file.

As you did in chapter eleven, here again you will instantiate a list object outside of a `for` loop as a place to hold the results returned from the new function.

```
book.freqs.l<-list()
for(i in 1:length(files.v)){
  #. . . Some code here
}
```

With each iteration of the loop, an `XML` file will be loaded into a new document object and sent to `getTEIWordSegmentTableList`. The function will return a new list object for each `XML` file. You will then need to add each of these list objects to the larger `book.freqs.l` object. While you could do this all in one single line of code:

```
book.freqs.l[[files.v[i]]]<-getTEIWordSegmentTableList(doc.object, 10)}
```

Your code will be easier to read if you break it into two steps. First

```
chunk.data.l<- getTEIWordSegmentTableList(doc.object, 10)
```

and then

```
book.freqs.l[[files.v[i]]]<- chunk.data.l
```

The `chunk.data.l` list is a temporary storage container. The contents of `chunk.data.l` get inserted into the larger container list `book.freqs.l`. At the same time, you can use double brackets to insert a name for the list that you derive from the file name in the (`files.v`) vector. Remember that the (`files.v`) vector is populated with the file names of the `XML` files inside the `input.dir`. When the script is run, the `book.freqs.l` list will contain `43` list objects (one for each text) and each of these will contain another list of `10` items, each one of these `10` will hold a frequency table corresponding one-tenth of the original text. In other words, you will have a list of lists of tables. Deep breath.

```
book.freqs.l<-list()
for(i in 1:length(files.v)){
  doc.object<-xmlTreeParse(file.path(input.dir, files.v[i]),
                           useInternalNodes=TRUE)
  chunk.data.l<- getTEIWordSegmentTableList(doc.object, 10)
  book.freqs.l[[files.v[i]]]<- chunk.data.l
}
```

Run this code now and then examine the main list with the `length` function, you'll see that it contains `43` records.

```
length(book.freqs.l)
## [1] 43
```

Each of these, however, then contains `10` items which you can ascertain by accessing any single list item via double bracketed sub-setting:

```
length(book.freqs.l[[1]])
## [1] 10
```

With all the necessary data collected from the texts, you now require a way to *munge* it all into a single data frame such that each row is a text chunk and each column is a different word feature. Unlike the clustering experiment in chapter eleven, you now have 430 rows, ten for each novel or one row for each novel segment.

It might help at this point to explore the `book.freqs.l` list object a bit deeper. You can begin with `str(book.freqs.l)` where you will see that `book.freqs.l` is a list of `43` items. Note that each of these `43` items is a *named* list of ten table objects. The names, you will recall, got assigned based on the original file names in the `files.v` object. Since the items in the main list are named, you can inspect them by numeric index

```
str(book.freqs.l[[37]])
```

or by name

```
str(book.freqs.l$Norris1.xml)
```

You will see that the item named *Norris1.xml* in the main `book.freqs.l` list is another list containing ten items (the ten text segments). You will also see that each of these ten items is a table object containing a named vector of words and a corresponding set of values for the relative frequencies.

If you enter `book.freqs.l$Norris1.xml[[1]]`, you will see the entire table of values for the first chunk of the book in the file titled *Norris1.xml*. Here I'll show only the first three words and their corresponding relative frequencies.

```
book.freqs.l$Norris1.xml[[1]][1:3]
##
##         a   ability      able
## 2.827e-02 7.421e-05 7.421e-05
```

If you want to see the first three values in the second text segment, enter

```
book.freqs.l$Norris1.xml[[2]][1:3]
##
##         a  abjuring      able
## 2.682e-02 7.554e-05 7.554e-05
```

As in chapter eleven, you now need a way of converting this list into a matrix. This time, however, things are a bit more complicated because you will need to keep track of ten separate chunks for each text.

12.4 Converting an R List into a Matrix

The goal now is to convert this list, or more precisely, list of lists of tables into a large matrix in which each row is a text chunk (or segment) and each column is a single word feature. The values in the corresponding cells, should be the relative frequencies for the given word column in a given row chunk. Along the way, you will need to keep track of which rows are from which texts.

Some of the R code for doing this will be familiar from the clustering chapter, but with the addition of one more step that allows you to dig one level deeper into the book.freqs.l list. Recall from the clustering chapter that you used mapply to convert each table of word frequencies into a data frame:

```
mapply(data.frame,ID=seq_along(book.freqs.l),
       book.freqs.l,SIMPLIFY=FALSE,
       MoreArgs=list(stringsAsFactors=FALSE))
```

You now have ten tables for each list item, so you can use lapply to iterate over the primary list (book.freqs.l) and send each of the ten table objects to another function that will stitch them all together. For this you will write a short function called my.mapply.

Begin with code for calling lapply with the new function:

```
freqs.l <- lapply(book.freqs.l, my.mapply)
```

Now you just need to write my.mapply. lapply will do the work of calling the new function for each item in the book.freqs.l list. The inner workings of this custom my.mapply function will look very similar to the code you wrote in the clustering chapter. my.mapply will map each of the ten tables together and then convert them into data frames using do.call and rbind. The only real change here is that you have now encapsulated this process inside a function that gets iteratively called using lapply. Here is the function:

```
my.mapply<-function(x){
  my.list<-mapply(data.frame, ID=seq_along(x),
                  x, SIMPLIFY=FALSE,
                  MoreArgs=list(stringsAsFactors=FALSE))
  my.df <- do.call(rbind, my.list)
  return(my.df)
}
```

As you see, my.mapply returns a data frame in which each row contains a chunk reference, a feature type, and a frequency value. You can now call the function and rbind the resulting list object into a long form data frame.

```
freqs.l <- lapply(book.freqs.l, my.mapply)
freqs.df <- do.call(rbind,freqs.l)
```

The resulting data frame is 770818, 3, and using the head function, allows you can preview the first few rows of the results:

```
head(freqs.df)
##                 ID      Var1      Freq
## anonymous.xml.1  1        11 9.459e-05
## anonymous.xml.2  1      1850 9.459e-05
## anonymous.xml.3  1         a 2.724e-02
## anonymous.xml.4  1   abandon 9.459e-05
## anonymous.xml.5  1 abandoned 1.892e-04
## anonymous.xml.6  1    aboard 1.892e-04
```

12.5 Organizing the Data

Examining this output, you will notice that each row begins with a row name corresponding to the original file name in the corpus. You will also see that there are three columns of data with the column names `ID`, `Var1`, and `Freq`. The `ID` is the chunk reference number, `Var1` is the word type (in this case the first two word types are the numbers `11` and `1850` because digits were not *stopped out* during tokenization), and `Freq` is the relative frequency of the word type in the particular chunk of the particular text. Before you can cross tabulate this data and create the wide form data frame you need for the analysis, you will first need to extract some metadata from the row names and `ID` columns. You can do this using a new regular expression and the `gsub` function.

Like `grep`, `gsub` is a pattern matching function. More specifically, `gsub` is a function for finding patterns and replacing them with something else: a global find and replace on steroids. First, to tidy things up a bit, you can use `gsub` to generate a character vector of file names with the ".xml" stripped off (replaced). For arguments, `gsub` takes a *pattern* to search for, a *replacement* value, and a *object* to search in. You can use the regular expression "\\..*" to find the period (".") character followed by any number of other characters in the row names vector.[2] You can replace those matches with nothing ("") and store the resulting strings in a new variable called `bookids.v`.

```
bookids.v<-gsub("\\..*", "", rownames(freqs.df))
```

Using the `paste` function, you can then *glue* each of these new strings to the corresponding segment/chunk reference value in the ID column.

```
book.chunk.ids<-paste(bookids.v, freqs.df$ID, sep="_")
```

This allows you to create a unique *book-with-chunk* identification string. To make these easy to read, I have glued them using an underscore character as the value for the `sep` argument. Now you can replace the existing values in the `ID` column with the new values.

```
freqs.df$ID<-book.chunk.ids
```

If you look at the first few rows, you'll see that the *ID* column values have changed to values expressing both the *file name* and the chunk *ID*.

```
head(freqs.df)
##                          ID      Var1      Freq
## anonymous.xml.1 anonymous_1        11 9.459e-05
## anonymous.xml.2 anonymous_1      1850 9.459e-05
## anonymous.xml.3 anonymous_1         a 2.724e-02
## anonymous.xml.4 anonymous_1   abandon 9.459e-05
## anonymous.xml.5 anonymous_1 abandoned 1.892e-04
## anonymous.xml.6 anonymous_1    aboard 1.892e-04
```

[2] In *regex* the *period* character is used as a special wild card. So in this expression the first period must be escaped using the double backslashes. This tells the *regex* engine to find the *literal* period. The second period in the expression is the period being used as a wild card metacharacter. The asterisk is another special character that is used as a multiplier. So here the asterisk repeats the wild card character indefinitely, until the end of the search string is reached.

12.6 Cross Tabulation

All of this was necessary in order to cross tabulate the data. As you saw in chapter eleven, cross-tabulation works much like a pivot table in Excel. You want to create a single row of data for each text chunk. In the current long form matrix you have a row for each word type. There are several ways of doing cross tabulation in `R`, but here again I will use the `xtabs` function.

```
result.t <- xtabs(Freq ~ ID+Var1, data=freqs.df)
```

The `xtabs` function is given a formula: on the right side of the tilde character, the variables for cross classification: *ID* and *Var1*. On the left side, the word frequency values held in the *Freq* column. In building this new wide format matrix, `R` will treat *ID* and *Var1* as the row name and column name values respectively. It will then insert the *Freq* value into the corresponding cells in the new matrix. The new *xtabbed* object is 430, 53233. To make the data easier to work with, you can convert this `xtab` table object into a data frame:

```
final.df<-as.data.frame.matrix(result.t)
```

If you would like to examine the values for any specific word type, you can do that easily. Here is how to look at the frequencies for the words *of* and *the* in the first ten rows:

```
final.df[1:10, c("of", "the")]
##                    of     the
## anonymous_1  0.02989 0.06356
## anonymous_10 0.02172 0.04765
## anonymous_2  0.02340 0.05042
## anonymous_3  0.02262 0.05632
## anonymous_4  0.02081 0.05105
## anonymous_5  0.02227 0.05765
## anonymous_6  0.02654 0.04727
## anonymous_7  0.02180 0.05031
## anonymous_8  0.02395 0.05602
## anonymous_9  0.02012 0.05569
```

Before you can use any of this in a classification test, however, you still have a bit more preprocessing to do. Since this is an authorship attribution experiment, you probably want to reduce the data frame to include only the very high frequency features, and you will also need a way of keeping track of the metadata, specifically which texts belong to which authors.

12.7 Mapping the Data to the Metadata

First and foremost, you need a way of mapping the word frequency data not just to specific text samples (i.e. the specific chunks) but also to the specific authors. In this corpus you have multiple texts from multiple authors. In fact, excluding the anonymous book, there are twelve authors, forty-two books, and `420` book chunks. What you need right away is an *author* column. Because these files were named with the author's last name, you can extract the necessary metadata from what is now the row name in the `final.df` object.

Begin by deriving a new matrix object (metacols.m) by splitting the row names using that underscore character that was inserted during the paste command above.

```
metacols.m<-do.call(rbind, strsplit(rownames(final.df), "_"))
head(metacols.m)
##      [,1]        [,2]
## [1,] "anonymous" "1"
## [2,] "anonymous" "10"
## [3,] "anonymous" "2"
## [4,] "anonymous" "3"
## [5,] "anonymous" "4"
## [6,] "anonymous" "5"
```

To keep things organized and human readable, reset the column names to something that makes more sense:

```
colnames(metacols.m)<-c("sampletext", "samplechunk")
head(metacols.m)
##      sampletext  samplechunk
## [1,] "anonymous" "1"
## [2,] "anonymous" "10"
## [3,] "anonymous" "2"
## [4,] "anonymous" "3"
## [5,] "anonymous" "4"
## [6,] "anonymous" "5"
```

Using head you can inspect the first few rows for the anonymous text, but remember that for some authors there are multiple books. If you want to see all of the unique values in the sampletext column, the unique function is handy:

```
unique(metacols.m[,"sampletext"])
##  [1] "anonymous"  "Carleton1"  "Carleton10"
##  [4] "Carleton11" "Carleton12" "Carleton13"
##  [7] "Carleton14" "Carleton2"  "Carleton3"
## [10] "Carleton4"  "Carleton5"  "Carleton6"
## [13] "Carleton7"  "Carleton8"  "Carleton9"
## [16] "Donovan1"   "Donovan2"   "Driscoll1"
## [19] "Driscoll2"  "Driscoll3"  "Edgeworth1"
## [22] "Jessop1"    "Jessop2"    "Jessop3"
## [25] "Kyne1"      "Kyne2"      "LeFanu1"
## [28] "LeFanu2"    "LeFanu3"    "LeFanu4"
## [31] "LeFanu5"    "LeFanu6"    "LeFanu7"
## [34] "Lewis"      "McHenry1"   "McHenry2"
## [37] "Norris1"    "Norris2"    "Norris3"
## [40] "Norris4"    "Polidori1"  "Quigley1"
## [43] "Quigley2"
```

As you can see, there are 43 unique texts. You can also see that some texts are by the same authors. You need a way to identify that both books by Quigley (Quigley1 and Quigley2) are by the same author, and likewise for the other authors from whom there are multiple samples. Use gsub again, and another regular expression, that will find instances of one or more *digits* (i.e. the 1 or 2 in *Quigley1* and *Quigley2*) followed by the end of a character string, which you indicate using the dollar ($) symbol. When a match is found, gsub will replace the matched string with nothing, which has the effect of deleting the digits. The result can be saved into a new object called author.v.

```
author.v<-gsub("\\d+$", "", metacols.m[,"sampletext"])
```

You can then check our work using unique.

```
unique(author.v)
##  [1] "anonymous" "Carleton"  "Donovan"
##  [4] "Driscoll"  "Edgeworth" "Jessop"
##  [7] "Kyne"      "LeFanu"    "Lewis"
## [10] "McHenry"   "Norris"    "Polidori"
## [13] "Quigley"
```

With a new vector of author names, you can now create a final data frame that binds this vector as a new column along with the two columns in the `metacols` variable to the existing `final.df`:

```
authorship.df<-cbind(author.v, metacols.m, final.df)
```

12.8 Reducing the Feature Set

At `430` by `53236`, this new data frame contains way too many features for an authorship attribution test. You will need to reduce the number of columns to just those that contain the high frequency features. In chapter eleven, I showed one way of achieving this using `apply` with a conditional expression (see 11.7). Here I show an alternative approach using `colMeans` to select only those feature with a mean relative frequency across the corpus of 0.005.[3]

The task here is to calculate the overall mean of each word type column in the `authorship.df` object. `R` gives us `colMeans` for doing just this. Remember though, that the first three columns in `authorship.df` are *metadata* (containing the author and text information), so you only want to get the means for the columns containing frequency data. To access just these columns, you can use bracketed sub-setting and a sequence vector running from `4` through the number of columns in the `authorship.df` object. To determine that end point, use `ncol` (number of columns), another `R` function that is similar to `length` but specific to data frames and matrices.

```
freq.means.v<-colMeans(authorship.df[,4:ncol(authorship.df)])
```

You can now identify which of these column means is greater than or equal to `0.005` using `which`. I'll save these to a vector called `keepers.v`

```
keepers.v<-which(freq.means.v >=.005)
```

and since there are not going to be too many of them, I'll inspect the entire vector in the console.

```
keepers.v
##    a  and   as   at   be  but  for  had   he  her
##    3   83  116  122  171  285  806  940  970  993
##  him  his    i   in   is   it  not   of   on    s
## 1008 1010 1050 1072 1110 1113 1424 1441 1448 1765
##  she that  the   to  was with  you
## 1849 2102 2103 2149 2282 2357 2397
```

[3] Note that in this chapter the function has *not* been written to multiply the relative frequency values by `100`.

From this vector of values that met the condition, you can grab the names of the word types using `names(keepers.v)`. You could then use those names to identify the subset of columns in the `authorship.df` object that you want to retain for analysis:

```
smaller.df<-authorship.df[, names(keepers.v)]
```

Unfortunately, the line of code above does not include the metadata columns about the authors and texts that you were so careful to preserve and organize. While you *could* just `cbind` those meta columns back in to the new `smaller.df` like this:

```
smaller.df<-cbind(author.v, metacols, smaller.df)
```

a simpler solution would be to identify the columns you want right from the start but combining the names from the `keeper.v` vector with the column names of the first three columns in the main `authorship.df` data frame.

```
smaller.df<-authorship.df[, c(names(authorship.df)[1:3],
                              names(keepers.v))]
```

12.9 Performing the Classification with SVM

With all of the data preparation done, you are finally ready to perform the classification analysis and see if you can figure out who wrote that anonymous book! Begin by identifying the rows in the new data frame belonging to the anonymous author.

```
anon.v<-which(smaller.df$author.v == "anonymous")
```

Next identify the data that will be used to train the model by telling `R` to take only the rows of `smaller.df` that do *not* include those identified in the `anon` vector. This negation is done using the “-” operator before the object name. It has the effect of saying “all the rows except for these” or “less these.” For this classification, you do not want to include the first three columns where the metadata is stored, so use `4:ncol(smaller.df)` to grab only the 4th through last columns.

```
train <- smaller.df[-anon.v,4:ncol(smaller.df)]
```

Now identify a *class* column that the classifier will use to organize the data. That is, you need to give the classifier a vector of values for the classes that are already known. In this case, the true author names are stored in the column headed *author.v*.

```
class.f <- smaller.df[-anon.v,"author.v"]
```

This new vector is of a special type that `R` calls a factor. [4] With the classes identified, you just need to pick a classifier and run the classification. [5] To keep things

[4] Factors are very similar to vectors except that in addition to storing the vector data, in this case a set of character strings referring to authors, the factor also stores *levels*. Factors provide an efficient way of storing repetitive character data because the unique character values are actually only stored once and the data itself is stored as a vector of integers that refer back to the single character strings.

[5] There are many good classification algorithms that can be used for authorship attribution testing, and in 2010 Daniela Witten and I published a bench-marking study of five well known algorithms.

simple and to avoid having to load a lot of complex classification packages, we'll use a comparatively familiar algorithm, `SVM` or *Support Vector Machines* which is part of the `e1071` package.

If you need help installing a package, see *chapter 10 section 4: Installing* `R` *Packages* for instructions. Otherwise, load the library:

```
library(e1071)

## Loading required package: class
```

You can now generate a model using the `svm` classifier function and the data contained in the `train` and `class.f` objects:

```
model.svm <- svm(train, class.f)
```

Once the model is generated, you can examine the details using the `summary` function
To test the accuracy of the model, use the `predict` function with the `model.svm` and the training data in the `train` object.

```
pred.svm <- predict(model.svm, train)
```

The `pred.svm` object will now contain a vector of text labels and the machine's guesses. If you examine the contents of `pred.svm`

```
as.data.frame(pred.svm)
```

you'll see that the text sample labeled *Carleton5_1* was incorrectly assigned to *Quigley* and that *McHenry1_5* was incorrectly assigned to *Carleton*. For the most part, however, you'll see that the model has done very well. To see a summary in the form of a confusion matrix you can use `table`:

```
table(pred.svm, class.f)
```

When you look at the *Carleton* column in the results, you'll see, that `139` of the *Carleton* samples were assigned correctly to *Carleton* and only one was incorrectly assigned to *Quigley*. You'll see that all `20` of the *Donovan* samples were correctly assigned to *Donovan*, and so on. This model has performed very well in terms of accurately classifying the known authors.

Based on this validation of the model's accuracy in classifying the known authors, you can classify the anonymous text with a good deal of confidence. First isolate the test data:

```
testdata <- smaller.df[anon.v,4:ncol(smaller.df)]
```

and then send the test data to the model for prediction. View the results using `as.data.frame`.

See Jockers, Matthew L. and Daniela M. Witten. "A Comparative Study of Machine Learning Methods for Authorship Attribution." *Literary and Linguistic Computing*, 25.2, (2010): 215-224; doi: 10.1093/llc/fqq001. We concluded that the Nearest Shrunken Centroids was especially good, but frankly, the others we tested also performed quite well. Interested readers should also look at the work of Jan Rybinci and Maciej Eder found at the Computational Stylistics Group website:https://sites.google.com/site/computationalstylistics/

```
final.result<-predict (model.svm, testdata)
as.data.frame(final.result)
##              final.result
## anonymous_1          Kyne
## anonymous_10         Kyne
## anonymous_2          Kyne
## anonymous_3          Kyne
## anonymous_4          Kyne
## anonymous_5          Kyne
## anonymous_6          Kyne
## anonymous_7          Kyne
## anonymous_8          Kyne
## anonymous_9          Kyne
```

The results of this `svm` classification confirm what was observed in the clustering test in chapter eleven; Kyne has been identified as the most likely author of every single segment of the anonymous book!

Practice

12.1. Now that you know the author and have seen how the classifier correctly guesses the author of each of the ten samples, increase the number of features the model uses by decreasing the feature `mean` used to determine the number of features retained in the `keepers.v` object. In the example in this chapter, `27` high-frequency features were retained using a mean relative frequency threshold of `.005`. Decrease this number in order to observe how the attributions change with the addition of context sensitive words. When using the `681` highest frequency word features, for example, the classifier gets every single attribution wrong. Why?

12.2. For the example in this chapter, I used a corpus wide mean relative frequency threshold to select the high-frequency features to keep for the analysis. In practice exercise 12.1 you saw how increasing the number of features can lead to incorrect results because author style gets lost in context. Another winnowing method involves choosing features based on the restriction that every selected feature must appear at least once in the work of every author. Write code that will implement such winnowing in order to generate a new set of values for the `keepers.v` object. Here is some sample code for you to consider. Notice that feature `f1` is not found in either of the samples from author `C` and feature `f5` is not found in any of the samples from author `A`. Features `f1` and `f5` should, therefore, be removed from the analysis.

```
authors<-c("A","A","B","B","C", "C")
f1<-c(0, 1, 2, 3, 0,0)
f2<-c(0, 1, 2, 3, 0,1)
f3<-c(3, 2, 1, 2, 1,1)
f4<-c(3, 2, 1, 2, 1,1)
f5<-c(0, 0, 1, 2, 1,1)
author.df<-data.frame(authors, f1,f2,f3,f4, f5)
author.df # Show the original data frame
##   authors f1 f2 f3 f4 f5
## 1       A  0  0  3  3  0
## 2       A  1  1  2  2  0
## 3       B  2  2  1  1  1
## 4       B  3  3  2  2  2
## 5       C  0  0  1  1  1
## 6       C  0  1  1  1  1
author.sums<-aggregate(author.df[, 2:ncol(author.df)],
  list(author.df[,1]), sum)
reduced.author.sums<-author.sums[,
  colSums(author.sums==0) == 0]
keepers.v<-colnames(
  reduced.author.sums)[2:ncol(reduced.author.sums)]
smaller.df<-author.df[, c("authors", keepers.v)]
smaller.df # show the new data frame
##   authors f2 f3 f4
## 1       A  0  3  3
## 2       A  1  2  2
## 3       B  2  1  1
## 4       B  3  2  2
## 5       C  0  1  1
## 6       C  1  1  1
```

Chapter 13
Topic Modeling

Abstract
This chapter introduces topic modeling using the `mallet` package, part of speech tagging with `openNLP` and topic-based word cloud visualization using the `wordcloud`.[1]

13.1 Introduction

I think it is safe to say that topic modeling is one of the hottest trends in digital humanities research today. I was introduced to topic modeling sometime in 2004 or 2005. At the time I was naively working on some code for a tool that I calling "the canonizer." Basically, my canonizer was built to do a kind of "comparative corcording." I needed software that would compare word frequency lists and collocates across different metadata facets such as year, author gender, nationality and so on. With this code I hoped to isolate thematic elements in each text by identifying frequently collocated word clusters. I'd then use these feature clusters to compare texts and explore the question of "canonicity" and whether there were marked differences between books in the traditional canon and those that have been historically marginalized.

Sometime during my work on the "canonizer," I bumped into LSA Latent Semantic Analysis. I jumped into that rabbit hole and spent a good deal of time tying to apply LSA to my canonizer problem. LSA seemed complicated enough, but then in 2006 I read about the work of David Newman and his group at UCI.[2] It was then that I got formally introduced to and seduced by Topic Modeling.

[1] In this chapter I assume that readers are already familiar with the basic idea behind topic modeling. Readers who are not familiar may consult Appendix B for a general overview and some suggestions for further reading.

[2] Newman, Smyth, Steyvers (2006). "Scalable Parallel Topic Models." *Journal of Intelligence Community Research and Development* and Newman and Block (2006). "Probabilistic Topic Decomposition of an Eighteenth Century Newspaper." In *JASIST*, March 2006.

Topic modeling offered a way of sifting through the noise in my canonizer application; it was an approach that would better isolate the gold nuggets and better sift out all the sand. If the method worked, it would offer a way of tracking collocates on a grand scale. I began a correspondence with David and eventually invited him to Stanford to give a lecture on topic modeling.

In advance of the meeting, I gave David a small corpus of books by Jane Austen, Dickens, and Melville. I asked him to run the models and give us a tour of the results. For me and a few others in the room, it was dazzling experiment. David described, for example, a cluster of words the machine had identified that seemed to be about "sentiment." It was a cluster of words related to expressions of feeling and emotion. Not only was it present in this corpus, but using the output of the model, David could track and visualize its "presence" across the various books in the corpus.

Almost immediately, some in the room saw that there was something similar in the charts David was showing for *Emma*, *Mansfield Park*, and *Northanger Abbey*, something that was different from the patterns of "sentiment" use seen in the other three Austen novels. Of course, the question on all our minds, whether we agreed that there was a pattern on not, was whether this new sort of quantitative data could be usefully interpreted. Did these thematic trends mean something and, more importantly, if they did mean something was it something new?

Since that meeting, I've spent a good deal of time trying to disambiguate the complexities of topic modeling while applying the method to in ways that I believe generate new knowledge.[3] In the later chapters of *Macroanalysis*, I use topic modeling as a way to study thematic trends in a corpus of 19th century fiction, 3,346 books in total, most all of them are novels. Over a period of 4 years, I modeled and remodeled this data, mostly using David's Mimno's implementation of LDA in `MALLET`. Along the way, I learned a few tricks about how to derive what I consider to be satisfactory themes. In this chapter, I show you a few of these tricks.

13.2 R and Topic Modeling

At the time of this writing there are three topic modeling packages for `R`. These include `topicmodels` from Bettina Grün and Kurt Hornik, `lda` by Johnathan Chang and `mallet` by David Mimno.[4] Though the `mallet` package for `R` is a relative newcomer, the `Java` package upon which it is based is not. I have chosen to

[3] Readers seeking a user-friendly introduction to how topic modeling actually works, should consult Appendix B.

[4] The `topicmodels` package provides an implementation of (or interface to) the C code developed by LDA pioneer David Blei. See Blei, David M., Ng, Andrew Y., and Jordan, Michael I. "Latent Dirichlet Allocation." *Journal of Machine Learning Research*, 3 (2003) 993-1022.

Johnathan Chang is a researcher at Facebook who has worked with Blei and with whom he has co-authored several papers including the influential topic modeling paper: J Chang, S Gerrish, C Wang, JL Boyd-Graber, DM Blei. "Reading tea leaves: How humans interpret topic models." *Advances in neural information processing systems*, 2009 http://machinelearning.wustl.edu/mlpapers/paper_files/NIPS2009_0125.pdf.

use the `mallet` package here because Mimno's implementation of topic modeling in the `MALLET JAVA` package is the *de facto* tool used by literary researchers.[5] In short, `MALLET` is the most familiar topic modeling package in the humanities, and it makes the most sense to work with it here.[6]

13.3 Text Segmentation and Preparation

Topic modeling treats each document as a *bag of words* in which word order is disregarded. Since the topic model works by identifying words that tend to co-occur, the bigger the bag, the more words that will tend to be found together in the same bag. If novels, such as those we will analyze here, tended to be constrained to only a very small number of topics or themes, then treating each entire novel as one bag might be fruitful. In reality, though, novels tend to have some themes that run throughout and others that appear at specific points and then disappear. In order to capture these transient themes, it is useful to divide novels and other large documents into *chunks* or *segments* and then run the model over those segments instead of over the entire text. [7] You must, therefore, begin by pre-processing the novels in the corpus into segments.

Unlike the previous chapter where you segmented texts based on percentage (i.e. each book was chunked into 10 equal sized portions), here you will write a function that allows for chunking based on a set number of words. That is, you will be able to set a specific chunk size, such as `1000` words, and then divide each text into some number of `1000` word segments.

Begin by loading the `XML` package, referencing the corpus directory, and then generating a vector of file names. This should be familiar from previous chapters.

```
library(XML)
inputDir<-"data/XMLAuthorCorpus"
files.v<-dir(path=inputDir, pattern=".*xml")
```

Now create and set a variable called `chunk.size`.

David Mimno, a professor at Cornell, is the developer and maintainer of the Java implementation of LDA in the popular *MAchine Learning for LanguagE Toolkit* (*MALLET*) developed at the University of Massachusetts under the direction of Andrew McCallum: McCallum, Andrew Kachites. "MALLET: A Machine Learning for Language Toolkit." 2002. See http://mallet.cs.umass.edu.

[5] Mimno released (to `CRAN`) his `R` "wrapper" for the `MALLET` Topic modeling package on August 9, 2013.

[6] I have used all three of these packages to good effect and prior to the release of the `mallet` package I taught workshops using both `topicmodels` and `lda`. Each one has its advantages and disadvantages in terms of ease of use, but functionally they are all comparable.

[7] There appears to be no conventional wisdom regarding ideal text-segmentation parameters. David Mimno reports in email correspondence that he frequently chunks texts down to the level of individual paragraphs. Until new research provides an algorithmic alternative, trial and experimentation augmented by domain expertise appear to be the best guides in setting segmentation parameters.

```
chunk.size<-1000 # number of words per chunk
```

Now you need to write a function to handle the chunking. You wrote a similar function in chapter 12 and you can reuse some of that code here. Find your `getTEIWordSegmentTableList` function from chapter 12 and copy and paste it with the new name: `makeFlexTextChunks`. The modified function you will write now will be designed to do percentage-based chunking by default (as in the original from chapter 12) but also able to accommodate word-count-based segmentation. Unlike the prior function, this one will not be returning lists of tables. For topic modeling, we do not need to pre-calculate the word frequencies. The new `makeFlexTextChunks` function will take three arguments: an `XML` document object, a chunk size value (expressed as either a percentage or as a number of words), and a third argument called `percentage` that will be set to `TRUE` by default but can be reset to `FALSE` at run time in order to allow for word-count-based chunking. Here is the initial framework for the function:

```
makeFlexTextChunks<-function(doc.object, chunk.size=10, percentage=TRUE){
  paras<-getNodeSet(doc.object,
                    "/d:TEI/d:text/d:body//d:p",
                    c(d = "http://www.tei-c.org/ns/1.0"))
  words<-paste(sapply(paras,xmlValue), collapse=" ")
  words.lower<-tolower(words)
  . . . # some new code here
}
```

As long as we are improving the way this function handles chunking, we will go ahead and make a small change to the regular expression you have been using for tokenizing the words in the file. In the older version of the function, you used the `\\W` regular expression to break the files into words based on *word boundaries*. With this regular expression, all alphanumeric characters (alphabetic letters and digits) are retained, and all of the punctuation is stripped out. One of the byproducts of this stripping is that contractions and possessives, that use an apostrophe, get split into two tokens: "Can't" becomes "Can" and "t," and "Bob's" becomes "Bob" and "s."[8]

There are a variety of ways that one might deal with this apostrophe situation and there are many regular expression recipes for doing fairly precise tokenization, but here things will be kept simple and straight-forward. First you will use `R's gsub` function to find all punctuation marks *excepting the apostrophe* and replace them with a blank spaces. Then you will tokenize the file using `strsplit` the regular expression `s+` that will break the text string apart based on places where it finds one or more (indicated by the +) blank space characters.[9]

```
words.lower<-gsub("[^[:alnum:][:space:]']", " ", words.lower)
words.l<-strsplit(words.lower, "\\s+")
```

[8] Another problem involves hyphens. Hyphens can appear at the end of lines as a printing convention but also in compound adjectives. We'll not deal with that trickier problem here.

[9] The regular expression used here may appear complicated compared to the simple `W` that has been used thus far. In this case, the expression simply says: "replace anything, except for an apostrophe, that is not an alphanumeric character with a blank space. " ")." `[:alnum:]` matches any alphabetic or numeric character and `[:space:]` matches any blank space. `'` then matches any apostrophes. The ˆ at the beginning of the expression serves as a negation operator, in essence indicating that the engine should match on anything that is *not* a character, digit, space, or apostrophe: i.e. match all other characters!

Recall from prior chapters that `strsplit` returns a list that you do not need. The next line in the function removes the list. The line after that simply creates a vector of word positions that you will need for figuring our where to chunk the text.

```
word.v<-unlist(words.list)
x <- seq_along(word.v)
```

With the words in a simple vector, the function must now either chunk by percentage (as in the default) or chunk by words. To handle this forking path, you can use an `if/else` conditional expression that examines the value of the `percentage` argument to make a decision. If `percentage` is `TRUE` (the default) the function will execute the code exactly as it did in the original function from Chapter 12. If `percentage` is `FALSE`, then the function will execute new code for doing word-count-based segmentation. At this point the function looks just as it did when you wrote it for chapter 12, except that you have improved the tokenization and added the internal `if/else` conditional. Here it is:

```
makeFlexTextChunks<-function(doc.object, chunk.size=1000, percentage=TRUE){
  paras<-getNodeSet(doc.object,
                    "/d:TEI/d:text/d:body//d:p",
                    c(d = "http://www.tei-c.org/ns/1.0"))
  words<-paste(sapply(paras,xmlValue), collapse=" ")
  words.lower<-tolower(words)
  words.lower<-gsub("[^[:alnum:][:space:]']", " ", words.lower)
  words.l<-strsplit(words.lower, "\\s+")
  word.v<-unlist(words.l)
  x <- seq_along(word.v)
  if(percentage){
    max.length<-length(word.v)/chunk.size
    chunks.l <- split(word.v, ceiling(x/max.length))
  } else {

  }
  chunks.l<-lapply(chunks.l, paste, collapse=" ")
  chunks.df<-do.call(rbind, chunks.l)
}
```

Unlike the percentage-based method where you divide the length of the word vector (`word.v`) by the `chunk.size` variable, here you want to split the word vector into chunks *equal in length* to the value of the `chunk.size` variable. The word-count-based segmentation code is, therefore, very similar to the code in the percentage method, but it does create one added complication. When chunking a file using percentages, each chunk is almost exactly the same size. When you split a text into `500` or `1000` word chunks, however, the last chunk will typically be something smaller than the chunk size you have set. Thus, you need a way of dealing with these remainder chunks.

A simple way to deal with this situation is to add the remainder chunk onto the second to last chunk, but you might not always want to do this. Say, for example, that the `chunk.size` variable is set to `1000` words, and the last chunk ends up being `950` words long. Would you really want to add those `950` words to the previous chunk? The answer is, of course, a subjective one, but a chunk of `950` words is probably close enough to `1000` to warrant full "chunk" status; it should remain a chunk of its own. But what if the last chunk were just `500` words, or `100` words; those samples are getting fairly small. Since you must pick a cutoff value, it is convenient to set a condition such that the last chunk must be at least *one-half* the number of words as the value inside the `chunk.size` variable. You can code this exception easily using the `length` function and some simple division wrapped up inside another `if` conditional, like this:

```
if(length(chunks.l[[length(chunks.l)]]) <= chunk.size/2){
  chunks.l[[length(chunks.l)-1]]<-c(chunks.l[[length(chunks.l)-1]],
                                    chunks.l[[length(chunks.l)]])
  chunks.l[[length(chunks.l)]]<-NULL
}
```

This conditional expression begins by getting the length of the word vector held in the last item of the `chunks.l` list and then checks to see if it is less than or equal to (<=) one-half of `chunk.size`. If the condition is met (i.e. TRUE), then the words in the last chunk

```
chunks.l[[length(chunks.l)]]
```

are added to the words in the second-to-last chunk

```
chunks.l[[length(chunks.l)-1]]
```

and the last chunk is then removed by setting it to `NULL`

```
chunks.l[[length(chunks.l)]]<-NULL
```

Once this is done, the word vectors in each item of the `chunks.l` list can be reduced back into strings using `lapply` to apply the `paste` function to each list item (`chunks.l<-lapply(chunks.l, paste, collapse=" ")`). The resulting list can then be morphed into a data frame object using `do.call` and `rbind` (`chunks.df<-do.call(rbind, chunks.l))`) exactly as you did in Chapter 12. The entire function now looks like this:

```
makeFlexTextChunks<-function(doc.object, chunk.size=1000, percentage=TRUE){
  paras<-getNodeSet(doc.object,
                    "/d:TEI/d:text/d:body//d:p",
                    c(d = "http://www.tei-c.org/ns/1.0"))
  words<-paste(sapply(paras,xmlValue), collapse=" ")
  words.lower<-tolower(words)
  words.lower<-gsub("[^[:alnum:][:space:]']", " ", words.lower)
  words.l<-strsplit(words.lower, "\\s+")
  word.v<-unlist(words.l)
  x <- seq_along(word.v)
  if(percentage){
    max.length<-length(word.v)/chunk.size
    chunks.l <- split(word.v, ceiling(x/max.length))
  } else {
    chunks.l <- split(word.v, ceiling(x/chunk.size))
    #deal with small chunks at the end
    if(length(chunks.l[[length(chunks.l)]]) <=
         length(chunks.l[[length(chunks.l)]])/2){
      chunks.l[[length(chunks.l)-1]] <-
        c(chunks.l[[length(chunks.l)-1]],
          chunks.l[[length(chunks.l)]])
      chunks.l[[length(chunks.l)]]<-NULL
    }
  }
  chunks.l<-lapply(chunks.l, paste, collapse=" ")
  chunks.df<-do.call(rbind, chunks.l)
  return(chunks.df)
}
```

With the function built, all need is a simple loop that will send all of the files in the `files.v` variable to this function and bind the results into a master data frame that you can call `topic.df`. Naturally, however, there is one complicating factor.

In addition to keeping track of the file names, you also need to keep track of the segment numbers. Eventually you want to be able to move from the topic model back to the original texts and the text segments. So you need to retain this metadata in some form or another.

You did something similar in Chapter 12 with the custom `my.mapply` function. Things are easier here. You can capture the original file names from the `files.v` and massage them a bit using `gsub` to remove the file extensions:[10]

```
textname<-gsub("\\..*","", files.v[i])
```

With the unique file names in hand, you can label the segments numerically by generating a sequence of number from `1` to the total number of chunks, which in this case is the same as the number of rows (found using the `nrow` function) in the `chunks.df` variable: i.e. `1:nrow(chunk.df)`. All of this data can then be bound, *column-wise*, into a new matrix object called `segments.m` which will then be *row-bound* to a matrix object (`topic.m`) that you instantiate prior to entering the loop, as follows:

```
topic.m<-NULL
for(i in 1:length(files.v)){
  doc.object<-xmlTreeParse(file.path(inputDir, files.v[i]),
                           useInternalNodes=TRUE)
  chunk.df<-makeFlexTextChunks(doc.object, chunk.size,
                           percentage=FALSE)
  textname<-gsub("\\..*","", files.v[i])
  segments.m<-cbind(paste(textname,
                          segment=1:nrow(chunk.df), sep="_"), chunk.df)
  topic.m<-rbind(topic.m, segments.m)
}
```

The result is a matrix object (`topic.m`) with two columns: the first column contains the unique file-segment identifiers and the second column contains the strings of text from each segment. To prepare this matrix for ingestion into the `mallet` topic modeling package, you must now convert it to a data frame and rename the column headers:

```
documents<-as.data.frame(topic.m, stringsAsFactors=F)
colnames(documents)<-c("id", "text")
```

Mission accomplished; now let's do some topic modeling!

13.4 The R mallet Package

The first and most important thing to know about the `mallet` package is that it is not a complete wrapper for the entire `MALLET` toolkit. As the documentation for the package notes: "Mallet has many functions, this wrapper focuses on the topic modeling sub-package written by David Mimno." So, do not look to this `R` wrapper if you want to access any of `MALLET's` other functions, such as document classification or hidden markov models for sequence tagging. This package is strictly for topic modeling. `mallet` is installed like any other package in `R` (see 10.4 for instructions). Once installed it is invoked using the `library(mallet)` expression.

[10] You did this in Chapter 12 as well.

```
   library(mallet)

## Loading required package: rJava
```

You will notice that `mallet` relies on the `rjava` package, which basically allows `R` to create `Java` objects and call `Java` methods. Recall that `MALLET` is written in `Java`, not `R`.

13.5 Simple Topic Modeling with a Standard Stop List

In the stylistic analysis that was covered in prior chapters high-frequency words were retained and used as markers of individual authorial style. In topic modeling you will typically want to remove or *stop-out* high frequency words such as *the, of, and, a, an, etc* because these words carry little weight in terms of thematic or topical value. If you do not remove these common function words, the topic model will generate topics (weighted word clusters) that are less about shared semantic sense, that is, less about topics or themes, and more about syntactical conventions. Here is an example showing the top seven words from 20 "topics" that I derived from the exercise corpus without using a stop list:

```
   0  0.04674 the to a of and don in
   1 0.06203 the to and of a bryce in
   2 0.1085 of the and in a is to
   3 0.14428 the of to and they their in
   4 0.29792 the and of a in to was
   5 0.16571 the a to of and was i
   6 0.45213 you i to it a and that
   7 0.24112 the of and a in by with
   8 0.35832 i you to my and is me
   9 0.55731 the of to and in that which
   10 0.59945 a to of he was his had
   11 0.13994 the to you an a that it
   12 0.03846 hycy ye bryan an o a the
   13 0.10359 the and a to of in susan
   14 0.50041 the of and a in was which
   15 0.07111 and the a  of in with
   16 0.18016 the a and of his sir is
   17 0.39585 her she and the to was a
   18 0.12091  and i a the in my
   19 0.58441 he the his and was to him
```

As you can see, these are semantically meaningless.

In the data directory of the exercise corpus, I have included a stop list (*stoplist.csv*) of `606` high frequency words. In your own work you may want to add to or cut this list to suit your research objectives, but this list will be sufficient for our purposes here.

The first step in generating a topic model with the `mallet` package is to invoke the `mallet.import` function. This function takes five arguments:

1. `id.array`
2. `text.array`
3. `stoplist.file`
4. `preserve.case`

5. `token.regexp`

The first argument (`id.array`) is an array, or vector, of document ids. You have this information stored in the first column of the `documents` data frame object that you created above. The second argument (`text.array`) is a vector of text strings, and you have this data in the second column of the `documents` data frame object. Next is the stop list file which will be referenced using a relative path to its location on your computer; in this case the path will be data/stoplist.csv. The next argument, `preserve.case`, is irrelevant in this example because you have already elected to lowercase all of the words as part of the `makeFlexTextChunks` function. Had you not already done this, then `mallet` would allow you to choose to do so or not at this point.[11] The final argument, `token.regexp` allows you to define a specific regular expression for tokenizing the text strings. The default expression is one that keeps any sequence of *one or more unicode characters*. Because you have gone to a lot of trouble to retain apostrophes, you'll need to give `mallet.import` a new value to replace the default `token.regexp` argument. To keep those apostrophes, the default expression (`[\\p{L}]+`) should be replaced with `[\\p{L}']+`. The complete expression, with the slightly modified regular expression, is as follows:

```
mallet.instances <- mallet.import(documents$id,
                                  documents$text,
                                  "data/stoplist.csv",
                                  FALSE,
                                  token.regexp="[\\p{L}']+")
```

The `mallet.instances` object created here is a `Java` object that is called a *Mallet instance list*. This in not an `R` object and must be accessed using other `Java` methods. The `mallet` package provides other functions as a gateway or bridge to those methods.

The next step is to create a *topic model trainer object*, which, for the moment, can be thought of as a kind of place holder object that you will fill with data in the next few steps. Notice that it is at this stage that the number of topics that the model will contain is set.[12] For the sake of this tutorial, I am setting the number of topics equal to the number of novels in the corpus. The reasons for this choice are purely pedagogical and will make more sense as we work through the rest of this chapter.

```
## Create a topic trainer object.
topic.model <- MalletLDA(num.topics=43)
```

Because the `mallet` package is simply providing a bridge to the `Java` application, this might feel a bit obtuse, and it can be a bit disconcerting when you are unable to employ `R` functions, such as `class` and `str`, to explore the makeup of these objects. If you try, you'll see references to the `rJava` package

[11] `mallet's` default is to convert to lowercase.

[12] How to set the number of topics is a matter of significant discussion in the topic modeling literature, and there is no obvious way of knowing in advance exactly where this number should be set. In the documentation for the `MALLET` program, Mimno writes: "The best number depends on what you are looking for in the model. The default (`10`) will provide a broad overview of the contents of the corpus. The number of topics should depend to some degree on the size of the collection, but `200` to `400` will produce reasonably fine-grained results." Readers interested in more nuanced solutions, may wish to consult Chapter 8 of Jockers, Matthew L. *Macroanalysis: Digital Methods and Literary History*. University of Illinois Press, 2013, or visit http://www.matthewjockers.net/2013/04/12/secret-recipe-for-topic-modeling-themes/ for my "Secret" Recipe for Topic Modeling Themes.

```
class(topic.model)
## [1] "jobjRef"
## attr(,"package")
## [1] "rJava"
```

In this case, `jobjRef` is a *reference* (or pointer) to a `Java` object that has been created and masked behind the scenes. Unless you are willing to dig into the actual source, that is, leave the world of `R` and go study the `MALLET` Java application, then you will have to accept a bit of obscurity.[13]

With the trainer object (`topic.model`) instantiated, you must now fill it with the textual data. For this you will call the `loadDocuments` method with the `mallet.instances` object that was created a moment ago as an argument.

```
topic.model$loadDocuments(mallet.instances)
```

When invoked, some initial processing of the documents occurs as `mallet` prepares the data for modeling. Among other things, `mallet` will output to the `R` console some information about the number of token found in the entire corpus after stop word removal (*total tokens*) and about the length of the longest individual document after stop word removal (*max tokens*). At this point, if you wish to access a list of the entire vocabulary of the corpus, you can invoke the `getVocabulary` method to return a character vector containing all the words:

```
vocabulary <- topic.model$getVocabulary()
```

You can then inspect this character vector using typical `R` functions:

```
class(vocabulary)
## [1] "character"
length(vocabulary)
## [1] 55444
head(vocabulary)
## [1] "summer"   "topsail"  "schooner" "slipped"
## [5] "cove"     "trinidad"
vocabulary[1:50]
##  [1] "summer"      "topsail"     "schooner"
##  [4] "slipped"     "cove"        "trinidad"
##  [7] "head"        "dropped"     "anchor"
## [10] "edge"        "kelp"        "fields"
## [13] "fifteen"     "minutes"     "small"
## [16] "boat"        "deposited"   "beach"
## [19] "man"         "armed"       "long"
## [22] "squirrel"    "rifle"       "axe"
## [25] "carrying"    "food"        "clothing"
## [28] "brown"       "canvas"      "pack"
## [31] "watched"     "return"      "weigh"
## [34] "stand"       "sea"         "northwest"
## [37] "trades"      "disappeared" "ken"
## [40] "swung"       "broad"       "powerful"
## [43] "back"        "strode"      "resolutely"
## [46] "timber"      "mouth"       "river"
## [49] "john"        "cardigan"
# etc. . .
```

[13] The `MALLET` program is not terribly difficult to run outside of `R` and there are now many good tutorials available online. A few of these are specifically written with humanities applications of topic modeling in mind. Perhaps the best place to start is with Shawn Graham, Scott Weingart, and Ian Milligan's online tutorial titled "Getting Started with Topic Modeling and MALLET." See http://programminghistorian.org/lessons/topic-modeling-and-mallet

At this point, you can also access some basic information about the frequency of words in the corpus and in the various documents of the corpus using the `R mallet` method (function) `mallet.word.freqs`.

```
word.freqs <- mallet.word.freqs(topic.model)
```

Calling this function will return a data frame containing a row for each unique word type in the corpus. The data frame will have three columns:

1. `words`
2. `term.freq`
3. `doc.freq`

The word types are in the `words` column; `term.freq` provides a count of the total number of tokens of that given word type in the corpus; and, finally, `doc.freq` provides a count of the total number of documents that contain that word at least once. You can look at the first few rows in the data frame using `R's head` function:

```
head(word.freqs)
##       words term.freq doc.freq
## 1    summer        62       49
## 2  topsail         1        1
## 3 schooner        16       10
## 4  slipped       130      120
## 5     cove         7        7
## 6 trinidad        17        9
```

Invoking `head` reveals that the word type *summer* occurs `63` times in the corpus in `52` different documents. *topsail*, on the other hand occurs just once, in one document.[14]

With the documents all pre-processed, you are now ready to run the actual training process. Before that, however, you have the opportunity to *tweak* the *optimization hyperparameters*! Though this step is not required (if you skip it the default values of `200` burn-in iterations and `50` iterations between optimization will be implemented), it is worth knowing that you can control the *optimization interval* and the *burn-in* using the following expression:[15].

```
topic.model$setAlphaOptimization(40, 80)
```

Because hyperparameter optimization is on by default, you can skip this step and go directly to the training of the model. The key argument that must now be set is the number of iterations to use in training. This argument determines the number of sampling iterations. In theory, as you increase the number of iterations the quality of the model will improve, but model quality is a rather subjective measure based on human evaluation of the resulting topic word clusters. In my own tests, I have observed that as one increases the number of iterations, topic quality increases only

[14] Do not forget that prior to modeling you have chunked each novel from the example corpus into `1000` word segments.

[15] The ramifications of resetting these values is beyond the scope of this chapter, but interested readers may wish to consult Hanna Wallach, David Mimno and Andrew McCallum. “Rethinking LDA: Why Priors Matter.” In proceedings of *Advances in Neural Information Processing Systems (NIPS)*, Vancouver, BC, Canada, 2009

to a certain point and then levels off. That is, after you reach a certain number of iterations, the composition and quality of the resulting topics does not change much.[16] For now, set the number of iterations to 400. In your own work, you may wish to experiment with different values and examine how topic composition changes with different values.

When you run this command, a great deal of output (which I have not shown here) will be sent to your R console. Every 50 iterations, for example, R will spit out a set of the seven top words in each topic. Here is a small snippet of that output:

```
. . .
8 0.06017 school city editor story job news paper
9 0.21164 mrs mother woman heart room life eyes
10 0.32898 sir mr good replied man make friend
11 0.02035 cloth mo vo gilt irish post tale
12 0.10557 love heart father jane charles thou papa
13 0.02116 don parker farrel pablo mike miguel kay
14 0.01771 bryce cardigan shirley colonel pennington timber sequoia
. . .
```

R will also provide probabilistic information about how likely the data are given the model at as it exists at a specific moment in the process. This figure is represented as a log-likelihood and appears as `INFO: <190> LL/token: -9.3141` in the output. Although the meaning of the log likelihood number is beyond the scope of this book, numbers closer to zero generally indicate better fitting models. [17]

13.6 Unpacking the Model

With the model now run, you can inspect the results and begin to see what is revealed about the corpus in terms of its thematic content. Begin by exploring the composition and coherence of the 43 topics you instructed mallet to identify. For extracting this information from the model, mallet provides two functions that return R objects: mallet.topic.words and mallet.top.words. Use the first of these to generate a matrix in which each row is a topic and each column a unique word type in the corpus. Once run, you can determine the size of the resulting matrix using dim:

```
topic.words.m <- mallet.topic.words(topic.model,
                                     smoothed=TRUE,
                                     normalized=TRUE)
dim(topic.words.m)
## [1]    43 55444
```

The values that appear in the cells of this matrix vary depending upon how you set the normalized and smoothed arguments. In this example I have set both normalized and smoothed to TRUE. When normalization is set to TRUE the values in each topic (row) are converted to percentages that sum to one. This can be easily checked with the rowSums function:

[16] My anecdotal experience seems consistent with more scientific studies, and interested readers may wish to consult Griffiths, T. L., & Steyvers, M. (2004). "Finding scientific topics." *Proceedings of the National Academy of Science*, 101, 5228-5235.

[17] David Mimno's "Topic Modeling Bibliography" provides a comprehensive list of resources for those wishing to go beyond this text. See http://www.cs.princeton.edu/ mimno/topics.html.

```
rowSums(topic.words.m)
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [24] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

When set to FALSE, the value in any given cell will be an integer representing the count of the occurrences of that word type that were assigned to a particular topic (row) during processing.

If you wanted to explore this matrix further, you could use bracketed sub-setting to access the values, for example:

```
topic.words.m[1:3, 1:3]
##           [,1]      [,2]      [,3]
## [1,] 2.233e-03 9.060e-07 9.060e-07
## [2,] 1.285e-07 1.285e-07 1.285e-07
## [3,] 2.979e-07 2.979e-07 9.203e-05
```

These results are not terribly informative because there is no column header to show which word types are associated with each column of values. You can, however, retrieve that information from the model and then add the column headers yourself using the colnames function.

```
vocabulary <- topic.model$getVocabulary()
colnames(topic.words.m)<-vocabulary
topic.words.m[1:3, 1:3]
##         summer   topsail  schooner
## [1,] 2.233e-03 9.060e-07 9.060e-07
## [2,] 1.285e-07 1.285e-07 1.285e-07
## [3,] 2.979e-07 2.979e-07 9.203e-05
```

Having set the column values, you can compare the relative weight of specific word types (as a percentage of each topic). In this example, I use R's c function to create a vector of key words and then use that vector as a way to select named columns from the matrix:

```
keywords<-c("california", "ireland")
topic.words.m[, keywords]
##       california   ireland
##   [1,]  9.774e-04 9.060e-07
##   [2,]  1.285e-07 1.285e-07
##   [3,]  2.979e-07 2.979e-07
##   [4,]  1.491e-03 2.236e-03
##   [5,]  5.326e-07 5.326e-07
##   [6,]  8.753e-07 8.753e-07
##   [7,]  3.969e-07 3.969e-07
##   [8,]  8.617e-07 8.617e-07
##   [9,]  1.672e-06 1.672e-06
##  [10,]  5.435e-04 1.007e-06
##  [11,]  2.982e-07 2.982e-07
##  [12,]  3.559e-07 7.950e-04
##  [13,]  1.502e-03 7.502e-07
##  [14,]  1.019e-06 1.019e-06
##  [15,]  3.688e-07 5.399e-04
##  [16,]  3.943e-07 8.200e-04
##  [17,]  6.408e-07 6.408e-07
##  [18,]  6.266e-07 6.266e-07
##  [19,]  2.012e-04 2.008e-07
##  [20,]  6.205e-07 6.205e-07
##  [21,]  1.659e-07 1.659e-07
##  [22,]  7.223e-07 7.223e-07
##  [23,]  3.916e-07 3.916e-07
##  [24,]  8.162e-07 8.162e-07
##  [25,]  5.598e-07 5.598e-07
##  [26,]  1.054e-06 1.054e-06
## [27,]  7.892e-07 7.892e-07
## [28,]  3.196e-07 5.143e-03
## [29,]  1.484e-07 1.484e-07
## [30,]  6.114e-07 6.114e-07
## [31,]  8.858e-07 8.858e-07
## [32,]  9.880e-03 1.031e-02
## [33,]  7.266e-07 7.266e-07
## [34,]  5.216e-03 3.962e-07
## [35,]  2.576e-07 2.576e-07
## [36,]  3.065e-07 3.065e-07
## [37,]  7.470e-07 7.470e-07
## [38,]  2.360e-07 2.360e-07
## [39,]  1.475e-07 1.475e-07
## [40,]  8.914e-07 7.557e-04
## [41,]  7.502e-07 7.502e-07
## [42,]  1.013e-06 1.013e-06
## [43,]  1.661e-07 2.574e-05
```

You can calculate which of the topic rows has the highest concentration of these key terms using `R's` `rowSums` and `max` functions inside a call to `which`. Save that row number in a new variable called `imp.row`.[18]

```
imp.row<-which(rowSums(topic.words.m[, keywords]) ==
                 max(rowSums(topic.words.m[, keywords])))
```

Examining these results shows that the topic in row `32` has the highest incidence of these keywords.[19] While exploring the `topic.words.m` object in this manner can be fruitful, we are usually less interested in specific words and more interested in examining the *top* or most heavily weighted words in each topic.

For this ranked sorting of topic words, `mallet` offers another function: `mallet.top.words`. This function takes three arguments:

1. `topic.model`
2. `word.weights`
3. `num.top.words`

The first of these is the model itself, the second is a row from the matrix of word weights that you have already created and stored in the `topic.words.m` object, and finally a third argument stipulating a user-defined number of "top words" to display. Assuming you wish to see the top `10` words from topic `32` (the row number you saved in the `imp.row` variable), you would enter:

```
mallet.top.words(topic.model, topic.words.m[imp.row,], 10)
##                  words  weights
## irish            irish 0.021701
## san                san 0.011778
## ireland        ireland 0.010312
## california  california 0.009880
## city              city 0.008974
```

[18] Note: if you are copying and executing this code as your read along, your row values and weights are likely to be different because the topic model employs a process that begins with a random distribution of words across topics. Though the topics you generate from this corpus will be generally similar, they may not be exactly the same as those that appear in this text.

[19] It must be noted here that in the `MALLET` Java program, topics are indexed starting at zero. Java, like many programming languages begins indexing with `0`. `R`, however, begins with `1`. Were we to run this same topic modeling exercise in the Java application, the topics would be labeled with the numbers `0 - 42`. In `R` they are `1 - 43`.

```
## family         family 0.007723
## francisco   francisco 0.007162
## county         county 0.006817
## race             race 0.006817
## native         native 0.006774
```

The most heavily weighted word in this topic is the word *irish*. Were you to assign a label to this topic, you might, after examining all these top words, choose *Irish California* or *Irish-American West* as a general descriptor. [20]

13.7 Topic Visualization

Looking only at the top ten words in a topic can be a bit misleading. Bear in mind, that each topic in this model consists of values for 55,443 word types! Generally you will want to examine more than just the top ten words when making a decision about how to label/interpret the topical or thematic essence of a topic. It can, therefore, be useful to visualize a larger number of the top words in the topic using a word cloud visualization. Thanks to Ian Fellows, `R` has a package for generating word cloud images from exactly the type of data returned by the `mallet.top.words` function.

Begin by installing and then loading the `wordcloud` package:[21]

```
library(wordcloud)

## Loading required package: Rcpp
## Loading required package: RColorBrewer
```

Now employ the `mallet.top.words` function again to grab `100` of the top words and their associated weights from the model. Instead of simply printing the results to the `R` console, save the output into a new variable called *topic.top.words*.

```
topic.top.words<-mallet.top.words(topic.model, topic.words.m[imp.row,], 100)
```

You can now call the `wordcloud` function providing a vector of words and a vector of word weights from the `topic.top.words` object as the first two arguments. To these I have added three more arguments that control the aesthetic look of the final word cloud.[22]

```
wordcloud(topic.top.words$words,
          topic.top.words$weights,
          c(4,.8), rot.per=0, random.order=F)
```

[20] For those who may not have intuited as much, the corpus of texts used in this book is composed of novels written entirely by Irish and Irish-American authors.

[21] see chapter 10 for package installation instructions.

[22] To see how to control the look of the visualization, just consult the help documentation for the `wordcloud` function using `?wordcloud`

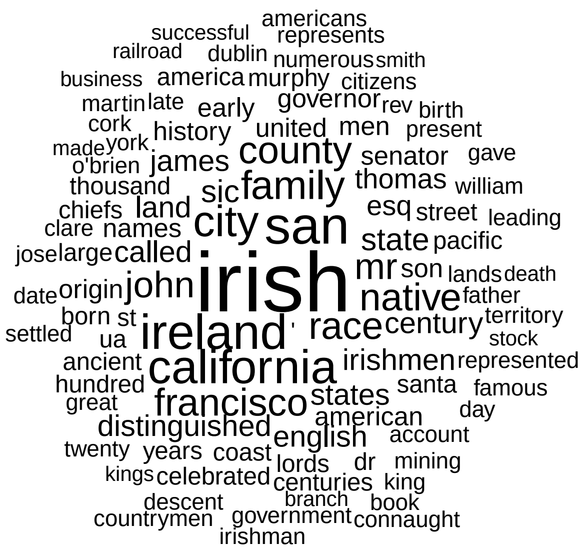


Fig. 13.1 Word Cloud of Topic 32

13.8 Topic Coherence and Topic Probability

Because I am familiar with this corpus, I knew that choosing the words *california* and *irish* would prove useful in identifying a topic that deals with the Irish presence in California and San Francisco, a topic found prominently in several books in this corpus. Often, however, you will be dealing with larger corpora and you will have to inspect the makeup of each topic in order to determine if the topics are *coherent*. You need to inspect them to see if they are *topical* or *thematic* in nature. If you complete exercise 13.1 right now, you will be able to examine 43 different word clouds. During that inspection, you will inevitably notice a high number of character names in many of the topics. Depending on your research goals, this presence of character names could be a real big problem.

Let's assume that you are hoping to track thematic change throughout a corpus. If that is the case, then the presence of character names is going to skew your results rather dramatically. There is a topic, for example, where the words *nell*, *tim* and *sheila* are prominent. Without even doing the calculations, I can tell you that this topic is going to be dominant in one book in the corpus (Josephine Donovan's novel *Black Soil*). This same topic will be comparatively absent from the other novels. I can also predict, based on my knowledge of the corpus, that there will be another topic featuring *gerald* from a book titled *Gerald Ffrench's Friends* and still another topic with *john* and *big* and *flurry*. These will be dominant in the book *Kansas Irish* where *Big Flurry* is the nickname of the main character, *Florence Driscoll*.

As noted previously, I intentionally picked 43 topics in order to highlight this problem (as you will recall, there are 43 books in the corpus). Even if I had not rigged the system, we would have had a way of exploring the extent to which certain topics were more probable or *present* in certain documents. `mallet` provides a function (`mallet.doc.topics`) for inspecting the probability of each topic appearing in each document. Or in more simple terms, `mallet` provides a function for assessing the *proportion* of a document that is *about* each topic.

```
doc.topics.m <- mallet.doc.topics(topic.model, smoothed=T, normalized=T)
```

Calling this function returns a matrix object in which each column is a topic and each row is a document from the corpus. The values in the cells of the matrix are the corresponding probabilities of a given topic (column) in a given document (row).

When the `normalized` argument is set to `TRUE` (as it is here) then the values in each row will sum to one. In other words, summing the 43 topic probability measurements for each document will return `1`. This makes it easy to think about the values as percentages or proportions of the document. In topic modeling, we assume that documents are composed of topics in different proportions. In truth, though, that is a bit of an oversimplification because this is a closed system and the model is only able assign proportions for the 43 topics in this particular configuration. So, what we are really assuming is that documents are composed of *these* 43 topics in differing proportions. It would not be entirely fair to say that the book *Gerald Ffrench's Friends* is 36% about *topic 8* (even though .36 is the mean proportion of this topic across all the segments from this book). A slightly better way to express this proportion, might be to explicitly say that: "of the 43 topics in this particular model that could be assigned to *Gerald Ffrench's Friends topic 8* is assigned with the highest probability, a probability of .36.

Let us now write some code to explore the proportions of each topic in each document and see if there are documents in the corpus that are dominated by specific topics. If you discover that a particular topic is more or less unique to one particular text, then you might have grounds to suspect a problem. Of course it is perfectly reasonable to imagine a situation in which there is one outlier book in the corpus, perhaps one book about vampires in a corpus of books about faeries. Here, however, you will find that there is a problem and that it is most certainly associated with character names.

Recall that every book in this corpus was split into segments before modeling. You now want to look at the books as a whole again and calculate the `mean` topical values across the segments as a way of assessing the general saturation of topics in books. You begin with the doc.topics.m object, a matrix of dimension 3523, 43. You know that these 3523 rows correspond to the segments from all of the novels, and you still have a data frame object called documents instantiated in the workspace where these document's *ids* are stored. You can put those values into a new vector called file.ids.v.

```
file.ids.v<-documents[,1]
head(file.ids.v)
## [1] "anonymous_1" "anonymous_2" "anonymous_3"
## [4] "anonymous_4" "anonymous_5" "anonymous_6"
```

Now you must massage the names in this vector so that the chunk identifier is split off into a separate vector from the main file name. You can use the strsplit function to break these character strings on the *underscore* character and return a list object. You can then use lappy and do.call, as you have done before, to convert these values into a two column matrix.

```
file.id.l<-strsplit(file.ids.v, "_")
file.chunk.id.l<-lapply(file.id.l, rbind)
file.chunk.id.m<-do.call(rbind, file.chunk.id.l)
head(file.chunk.id.m)
##      [,1]        [,2]
## [1,] "anonymous" "1"
## [2,] "anonymous" "2"
## [3,] "anonymous" "3"
## [4,] "anonymous" "4"
## [5,] "anonymous" "5"
## [6,] "anonymous" "6"
```

The first column provides a way of identifying which rows in the doc.topics.m object correspond to which text files. With that information, you can then use R's aggregate function to calculate the topical mean for each topic in each document. First save a copy of doc.topics.m as a data frame because you will need an object that allows both character data and numerical values.

```
doc.topics.df<-as.data.frame(doc.topics.m)
```

Now use cbind to bind the character data values in the first column of file.chunk.id.m to the topical values in doc.topics.df

```
doc.topics.df<-cbind(file.chunk.id.m[,1], doc.topics.df)
```

R's aggregate function can then be used to calculate the mean across the segments of each document.

```
doc.topic.means.df<-aggregate(doc.topics.df[, 2:ncol(doc.topics.df)],
                              list(doc.topics.df[,1]), mean)
```

The aggregate function returns a new data frame of 43 rows by 44 columns. There is now one row for each text, a column (the first) with the file name, or what the aggregate function titles a *group* and then 43 more columns, one for each of the topics you modeled. With this data in one place, you have several options for how to assess the mean values. Since you only have 43 documents in the corpus, you can visualize the document means using a simple bar plot.

```
barplot(doc.topic.means.df[, "V6"], names.arg=c(1:43))
```

A couple of things to note here: first, notice that the aggregate function renamed the columns by prefixing each with a V. Thus when you call barplot here, you use bracketed sub-setting to send data for all the rows and only the column labeled

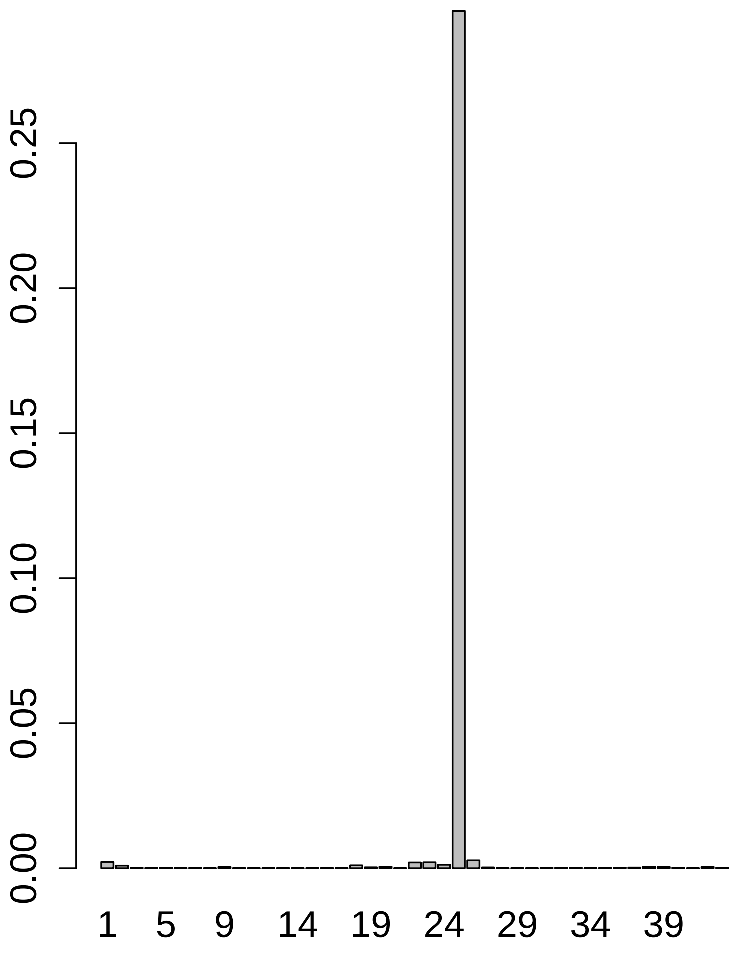


Fig. 13.2 Scatter plot of Topic Means in 43 documents

V6. [23] More important, notice the one outlier document, document 25. This is the document in row 25 of the data frame. You can get its file name by retrieving the value held in the group.1 column of row 25, like this :

```
filename<-as.character(doc.topic.means.df[25, "Group.1"])
```

```
filename
## [1] "Kyne1"
```

If you are following along, then you should be asking why I added the as.character function. Try it for yourself without the call to as.character and you will discover that the column called Group.1 is of the class factor.[24] So while the expression will return the same answer, it will also return information about the factor levels that you do not need.

The barplot reveals that there is one topic dominating the file called *Kyne1*, and, not surprisingly, the word cloud (13.3) for that topic indicates that it is clearly a character-driven topic. The top words in this topic include the names *parker*, *farrel*, *don*, *pablo*, *mike*, *miguel*, *kay*, and *panchito*. A simple way to deal with the character name problem is to add these names to the stop list. You will have the opportunity to do this and to compare your results in exercise 13.2. A more complicated way of dealing with these and related problems of topic coherence involves pre-processing the corpus with a part-of-speech tagger and then culling out words of different classes.

13.9 Pre-Processing with a Part-of-Speech Tagger

In my research for Chapter 8 of *Macroanalysis*, I discovered that by modeling the nouns in my documents, I could generate what I considered to be highly coherent and highly thematic topics.[25] In order to do this kind of modeling, I had to first pre-process the corpus with a part-of-speech (POS) tagger and identify which words were nouns and which were verbs, adjectives, etc. In that work, I used the Stanford Log-linear Part-Of-Speech Tagger outside of the R environment and then post-processed the results in R. Now, however, there is an R package for POS tagging and the entire process can be done in one place. Having said that, POS tagging is a time consuming process, so for this book I have provided another directory (taggedCorpus) containing pre-tagged versions of all of the files. Since you may want to tag your own files, here is the code you would need:

```
library(openNLP)
for(i in 1:length(files.v)){
  doc<-xmlTreeParse(file.path(inputDir, files.v[i]), useInternalNodes=TRUE)
```

[23] Your plots will not look the same since each run of the model is slightly different.

[24] A full discussion of factors is beyond the scope of this chapter, but for simplicity think about factors as a type of variable that can hold some limited set of values. These are often referred to as categorical variables. See also: Chapter 12.9, footnote 3

[25] All 500 of them can be viewed at http://www.matthewjockers.net/macroanalysisbook/macro-themes/

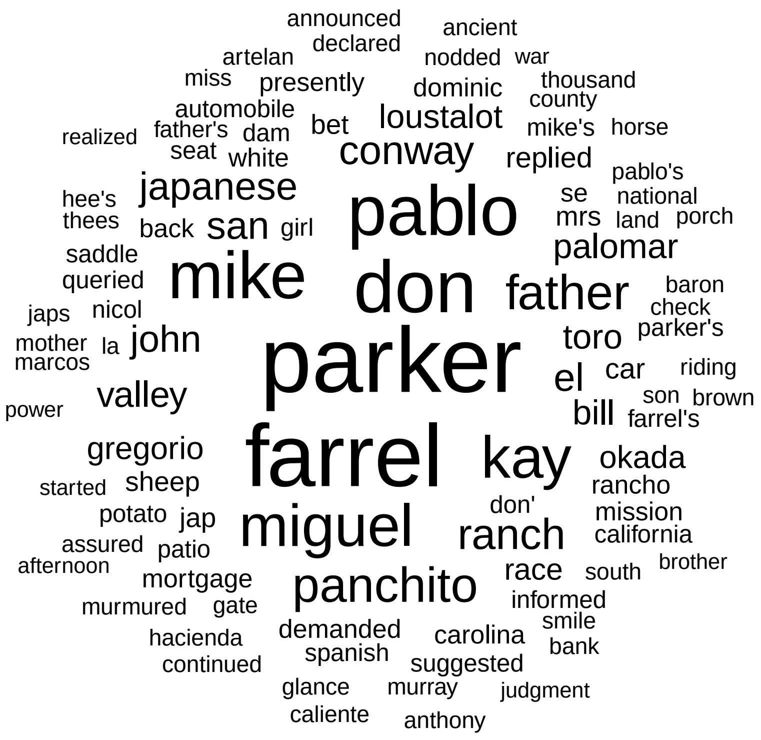


Fig. 13.3 Word Cloud of Main characters in *The Pride of Palomar*

```
    paras<-getNodeSet(doc, "/TEI/text/body//p")
    words<-paste(sapply(paras,xmlValue), collapse=" ")
    tagged_text <- tagPOS(words)
    write(tagged_text, paste("data/taggedCorpus/",files.v[i], ".txt", sep=""))
  }
```

This code will iteratively load each XML file in the corpus, run it through the openNLP POS tagger and then write the result to the directory called taggedCorpus. You do not need to run this code now since I have already tagged the files and put them into the directory for you.

With the files tagged, you can now write a modified version of the text chunking script that you used for the untagged files. In this new script, you will add a few lines of R code to remove all the words that are not nouns prior to segmentation. Begin just as you did with the XML files, but this time reference the taggedCorpus directory. You should decrease the chunk.size to 500 bearing in mind that in this new process you will be segmenting based on nouns only, thus each segment will have 500 nouns.

```
inputDir<-"data/taggedCorpus"
files.v<-dir(path=inputDir, pattern=".*xml")
chunk.size<-500
```

Before you get into the main script, you'll need to write a couple of new functions for handling the POS tagged text. If you open one of the files in the taggedCorpus directory, you will see that each word token is followed by a *forward slash* and then an abbreviated part-of-speech marker.[26] Here, for example, is the first line of the the file titled "anonymous.xml.txt":

In/IN the/DT summer/NN of/IN 1850/CD a/DT topsail/NN schooner/NN slipped/VBD into/IN the/DT cove/NN under/IN Trinidad/NNP Head/NNP and/CC dropped/VBD anchor/NN at/IN the/DT edge/NN of/IN the/DT kelp-fields./NN.

So begin by writing a function to split the tagged file into a vector in which each value is a single *word/POS* pair:

```
splitText <- function(text) {
  unlist(strsplit(text," "))
}
```

The elements of this function should be familiar to you: it uses strsplit and unlist and returns the result.[27]

You must now write another function that will walk through this vector and pick out only those values that contain certain *target* part-of-speech markers. Name this function selectTaggedWords and set it up to take two arguments: a vector of *word/POS* pairs and a target.tag to search for in the vector.

```
selectTaggedWords <- function(tagged.words, target.tag) {
  tagged.words[grep(target.tag, tagged.words)]
}
```

[26] The openNLP tagger implements the Penn Treebank tag set. See http://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html

[27] In previous functions you have written you have explicitly called the return function to send the results of the function back to the main script. This is not always necessary as R's default behavior is to return the last object generated by the function. This function is simple enough that I have chosen to leave off an explicit call to return.

In this function I have embedded a call to `grep` inside the brackets of the tagged word vector. When `grep` finds a match for the `target.tag` inside the `tagged.words` vector it returns `TRUE`. The output from `grep` will be a vector of `TRUE/FALSE` values that then gets used to mark the location of the matching strings. In this way, `grep` is similar to `which`, returning only those positions that meet a certain condition.

You now need a function to strip off the POS markers. You already know that each word token is followed by a forward slash and then a POS marker, so you can identify this pattern using a regular expression. In order to write the right `regex`, however, you also need to know a bit about the variety of POS tags that are used by the tagger. As it happens, these tags are always composed of two or three capital letters. So you can write a regular expression to find a pattern that begins with a forward slash and is followed by two or three capital letters. Such an expression looks like this: `/[A-Z]{2,3}`. You can use `R's sub` function (a kin of `grep`) to remove matches found for this pattern by replacing (*subbing*) them with nothing. The whole function, therefore, is as simple as this:

```
removeTags <- function(word.pos) {
  sub("/[A-Z]{2,3}", "", word.pos)
}
```

You can now wrap calls to all of these functions inside a larger function that will be almost identical to the `makeFlexTextChunks` function written at the beginning of this chapter.

```
makeFlexTextChunksFromTagged<-function(tagged.text,
                                   chunk.size=500, percentage=TRUE){
  tagged.words <- splitText(tagged.text)
  tagged.words.keep <- c(selectTaggedWords(tagged.words,"/NN$"))
  words <- removeTags(tagged.words.keep)
  words.lower<-tolower(words)
  word.v<-gsub("[^[:alnum:][:space:]']", "", words.lower)
  x <- seq_along(word.v)
  if(percentage){
    max.length<-length(word.v)/chunk.size
    chunks.l <- split(word.v, ceiling(x/max.length))
  } else {
    chunks.l <- split(word.v, ceiling(x/chunk.size))
    if(length(chunks.l[[length(chunks.l)]]) <=
         length(chunks.l[[length(chunks.l)]])/2){
      chunks.l[[length(chunks.l)-1]]<-
        c(chunks.l[[length(chunks.l)-1]], chunks.l[[length(chunks.l)]])
      chunks.l[[length(chunks.l)]]<-NULL
    }
  }
  chunks.l<-lapply(chunks.l, paste, collapse=" ")
  chunks.df<-do.call(rbind, chunks.l)
  return(chunks.df)
}
```

Save this new function into your *corpusFunction.R* file as `makeFlexTextChunksFromTagged` along with the others you have just written and then load them into your active workspace using a call to `source`:

```
source("code/corpusFunction.R")
```

With all of the new functions loaded, you run a loop that is almost identical to the one you ran earlier in the chapter.

```
topic.m<-NULL
for(i in 1:length(files.v)){
  tagged.text<-scan(file.path(inputDir, files.v[i]),
                    what="character", sep="\n")
  chunk.df<-makeFlexTextChunksFromTagged(tagged.text,
      chunk.size, percentage=FALSE)
  textname<-gsub("\\..*","", files.v[i])
  segments.m<-cbind(paste(textname, segment=1:nrow(chunk.df), sep="_"),
      chunk.df)
  topic.m<-rbind(topic.m, segments.m)
}
```

Finally, with the chunk matrix now prepared, you can run the topic modeling code again on the refined data:

```
documents<-as.data.frame(topic.m, stringsAsFactors=F)
colnames(documents)<-c("id", "text")
library(mallet)
mallet.instances <- mallet.import(documents$id,
                                  documents$text,
                                  "data/stoplist.csv",
                                  FALSE,
                                  token.regexp="[\\p{L}']+")
topic.model <- MalletLDA(num.topics=43)
topic.model$loadDocuments(mallet.instances)
vocabulary <- topic.model$getVocabulary()
word.freqs <- mallet.word.freqs(topic.model)
topic.model$train(400)
topic.words.m <- mallet.topic.words(topic.model,
                                    smoothed=TRUE,
                                    normalized=TRUE)
colnames(topic.words.m)<-vocabulary
```

Based on what you have learned here and what you will learn by completing the exercises, you should now be able to inspect these new topics and visualize them as word clouds. When you do so, you will see that they have improved considerably.[28]

Practice

13.1. Write a script that uses a `for` loop to iterate over all of the topics data in order to produce a word cloud for each.

13.2. In the data directory you will find a fairly comprehensive stop list of common names and high frequency words: `stoplist-exp.csv`. Replace the reference to `stoplist.csv` in the example code of this chapter with `stoplist-exp.csv` and generate a new model with a new set of topics and document proportions. Plot the means as you did in this chapter, and then assess the extent to which these new topics are distributed across the corpus. Make word clouds for each topic and consider how the new ones compare to those that included character names.

[28] In the `topicClouds` sub-directory of the data directory, you will find two .pdf file showing 43 word clouds each. The file titled "fromUnTagged.pdf" contains the clouds produced without part-of-speech based pre-processing, "fromTagged.pdf" includes the clouds produced only using words tagged as nouns.

Appendix A

Appendix A: Variable Scope Example

This is an example of scope within functions. First create a variable outside of a function.

```
my.var.to.process<-10
```

Now create a function that uses the same name for an argument as an existing variable.

```
my.func<-function(my.var.to.process){
  # overwrite the value in my.var.to.process
  # with a new value that adds ten
  my.var.to.process<-my.var.to.process + 10 # add ten
  # return the new value
  return(my.var.to.process)
}
```

The value returned by calling `my.func` is `20`.

```
my.func(my.var.to.process)
## [1] 20
```

But the value in the original variable is still `10` even though the same name was used inside the function.

```
my.var.to.process
## [1] 10
```

Appendix B
Appendix B: The LDA Buffet

A version of what follows was originally posted to http://www.matthewjockers.net/macroanalysisbook/lda/ on August 12, 2012.

. . . imagine a quaint town, somewhere in New England perhaps. The town is a writer's retreat, a place they come in the summer months to seek inspiration. Melville is there, Hemingway, Joyce, and Jane Austen just fresh from across the pond. In this mythical town there is spot popular among the inhabitants; it is a little place called the "LDA Buffet." Sooner or later all the writers go there to find themes for their novels. . .

One afternoon Herman Melville bumps into Jane Austen at the bocce ball court, and they get to talking.

"You know," says Austen, "I have not written a thing in weeks."

"Arrrrgh," Melville replies, "me neither."

So hand in hand they stroll down Gibbs Lane to the LDA Buffet. Now, down at the LDA Buffet no one gets fat. The buffet only serves light (leit?) motifs, themes, topics, and tropes (seasonal). Melville hands a plate to Austen, grabs another for himself, and they begin walking down the buffet line. Austen is finicky; she spoons a dainty helping of words out of the bucket marked "dancing." A slightly larger spoonful of words, she takes from the "gossip" bucket and then a good ladle's worth of "courtship."

Melville makes a bee line for the "whaling" trough, and after piling on an Ahab-sized handful of whaling words, he takes a smaller spoonful of "seafaring" and then just a smidgen of "cetological jargon."

The two companions find a table where they sit and begin putting all the words from their plates into sentences, paragraphs, and chapters.

At one point, Austen interrupts this business: "Oh Herman, you must try a bit of this courtship."

He takes a couple of words but is not really fond of the topic. Then Austen, to her credit, asks permission before reaching across the table and sticking her fork in Melville's pile of seafaring words, "just a taste," she says. This work goes on for a little while; they order a few drinks and after a few hours, voila! Moby Dick and Persuasion are written . . .

[Now, dear reader, our story thus far provides an approximation of the first assumption made in LDA. We assume that documents are constructed out of some finite set of available topics. It is in the next part that things become a little complicated, but fear not, for you shall sample themes both grand and beautiful.]

. . . Filled with a sense of deep satisfaction, the two begin walking back to the lodging house. Along the way, they bump into a blurry-eyed Hemingway, who is just then stumbling out of the Rising Sun Saloon.

Having taken on a bit too much cargo, Hemingway stops on the sidewalk in front of the two literati. Holding out a shaky pointer finger, and then feigning an English accent, Hemingway says: “Stand and Deliver!”

To this, Austen replies, “Oh come now, Mr. Hemingway, must we do this every season?”

More gentlemanly then, Hemingway replies, “My dear Jane, isn’t it pretty to think so. Now if you could please be so kind as to tell me what’s in the offing down at the LDA Buffet.”

Austen turns to Melville and the two writers frown at each other. Hemingway was recently banned from the LDA Buffet. Then Austen turns toward Hemingway and holds up six fingers, the sixth in front of her now pursed lips.

“Six topics!” Hemingway says with surprise, “but what are today’s themes?”

“Now wouldn’t you like to know that you old sot.” Says Melville.

The thousand injuries of Melville, Hemingway had borne as best he could, but when Melville ventured upon insult he vowed revenge. Grabbing their recently completed manuscripts, Hemingway turned and ran toward the South. Just before disappearing down an alleyway, he calls back to the dumbfounded writers: “All my life I’ve looked at words as though I were seeing them for the first time. . . tonight I will do so again! . . . ”

[Hemingway has thus overcome the first challenge of topic modeling. He has a corpus and a set number of topics to extract from it. In reality determining the number of topics to extract from a corpus is a bit trickier. If only we could ask the authors, as Hemingway has done here, things would be so much easier.]

. . . Armed with the manuscripts and the knowledge that there were six topics on the buffet, Hemingway goes to work.

After making backup copies of the manuscripts, he then pours all the words from the originals into a giant Italian-leather attache. He shakes the bag vigorously and then begins dividing its contents into six smaller ceramic bowls, one for each topic. When each of the six bowls is full, Hemingway gets a first glimpse of the topics that the authors might have found at the LDA Buffet. Regrettably, these topics are not very good at all; in fact, they are terrible, a jumble of random unrelated words . . .

[And now for the magic that is Gibbs Sampling.]

. . . Hemingway knows that the two manuscripts were written based on some mixture of topics available at the LDA Buffet. So to improve on this random assignment of words to topic bowls, he goes through the copied manuscripts that he kept as back ups. One at a time, he picks a manuscript and pulls out a word. He examines the word in the context of the other words that are distributed throughout each of the six bowls and in the context of the manuscript from which it was taken. The first word he selects is “heaven,” and at this word he pauses, and asks himself two questions:

“How much of ‘Topic A,’ as it is presently represented in bowl A, is present in the current document?” “Which topic, of all of the topics, has the most ‘heaven’ in it?” . . .

[Here again dear reader, you must take with me a small leap of faith and engage in a bit of further make believe. There are some occult statistics here accessible only to the initiated. Nevertheless, the assumptions of Hemingway and of the topic model are not so far-fetched or hard to understand. A writer goes to his or her imaginary buffet of themes and pulls them out in different proportions. The writer then blends these themes together into a work of art. That we might now be able to discover the original themes by reading the book is not at all amazing. In fact we do it all the time–every time we say that such and such a book is about "whaling" or "courtship." The manner in which the computer (or dear Hemingway) does this is perhaps less elegant and involves a good degree of mathematical magic. Like all magic tricks, however, the explanation for the surprise at the end is actually quite simple: in this case our magician simply repeats the process 10 billion times! NOTE: The real magician behind this LDA story is David Mimno. I sent David a draft, and along with other constructive feedback, he supplied this beautiful line about computational magic.]

. . . As Hemingway examines each word in its turn, he decides based on the calculated probabilities whether that word would be more appropriately moved into one of the other topic bowls. So, if he were examining the word "whale" at a particular moment, he would assume that all of the words in the six bowls except for "whale" were correctly distributed. He'd now consider the words in each of those bowls and in the original manuscripts, and he would choose to move a certain number of occurrences of "whale" to one bowl or another.

Fortunately, Hemingway has by now bumped into James Joyce who arrives bearing a cup of coffee on which a spoon and napkin lay crossed. Joyce, no stranger to bags-of-words, asks with compassion: "Is this going to be a long night."

"Yes," Hemingway said, "yes it will, yes."

Hemingway must now run through this whole process over and over again many times. Ultimately, his topic bowls reach a steady state where words are no longer needing to be being reassigned to other bowls; the words have found their proper context.

After pausing for a well-deserved smoke, Hemingway dumps out the contents of the first bowl and finds that it contains the following words:

"whale sea men ship whales penfon air side life bounty night oil natives shark seas beard sailors hands harpoon mast top feet arms teeth length voyage eye heart leviathan islanders flask soul ships fishery sailor sharks company. . . "

He peers into another bowl that looks more like this:

"marriage happiness daughter union fortune heart wife consent affection wishes life attachment lover family promise choice proposal hopes duty alliance affections feelings engagement conduct sacrifice passion parents bride misery reason fate letter mind resolution rank suit event object time wealth ceremony opposition age refusal result determination proposals. . ."

After consulting the contents of each bowl, Hemingway immediately knows what topics were on the menu at the LDA Buffet. And, not only this, Hemingway knows exactly what Melville and Austen selected from the Buffet and in what quantities. He discovers that Moby Dick is composed of 40 percent whaling, 18 percent seafaring and 2 percent gossip (from that little taste he got from Jane) and so on . . .

[Thus ends the fable.]

For the rest of the (LDA) story, see David Mimno's *Topic Modeling Bibliography* at http://www.cs.princeton.edu/ mimno/topics.html.

Appendix C
Appendix C: Code Repository

C.1 Chapter 3 Start Up Code

```
text.v<-scan("data/plaintext/melville.txt", what="character", sep="\n")
start.v<-which(text.v == "CHAPTER 1. Loomings.")
end.v<-which(text.v == "orphan.")
start.metadata.v<-text.v[1:start.v -1]
end.metadata.v<-text.v[(end.v+1):length(text.v)]
metadata.v<-c(start.metadata.v, end.metadata.v)
novel.lines.v<- text.v[start.v:end.v]
novel.v<-paste(novel.lines.v, collapse=" ")
novel.lower.v<-tolower(novel.v)
moby.words.l<-strsplit(novel.lower.v, "\\W")
moby.word.v<-unlist(moby.words.l)
not.blanks.v <- which(moby.word.v!="")
moby.word.v<- moby.word.v[not.blanks.v]
whale.hits.v<-length(moby.word.v[which(moby.word.v=="whale")])
total.words.v<-length(moby.word.v)
moby.freqs.t<-table(moby.word.v)
sorted.moby.freqs.t<-sort(moby.freqs.t , decreasing=T)
```

C.2 Chapter 4 Start Up Code

```
text.v<-scan("data/plaintext/melville.txt", what="character", sep="\n")
start.v<-which(text.v == "CHAPTER 1. Loomings.")
end.v<-which(text.v == "orphan.")
novel.lines.v<- text.v[start.v:end.v]
novel.v<-paste(novel.lines.v, collapse=" ")
novel.lower.v<-tolower(novel.v)
moby.words.l<-strsplit(novel.lower.v, "\\W")
moby.word.v<-unlist(moby.words.l)
not.blanks.v <- which(moby.word.v!="")
moby.word.v<- moby.word.v[not.blanks.v]
```

C.3 Chapter 5 Start Up Code

```
# Chapter 4 Start up code
text.v<-scan("data/plaintext/melville.txt", what="character", sep="\n")
start.v<-which(text.v == "CHAPTER 1. Loomings.")
end.v<-which(text.v == "orphan.")
novel.lines.v<- text.v[start.v:end.v]
novel.lines.v<-unlist(novel.lines.v)
chap.positions.v<-grep("^CHAPTER \\d", novel.lines.v)
last.position.v<- length(novel.lines.v)
chap.positions.v <- c(chap.positions.v , last.position.v)
chapter.freqs.l<-list()
chapter.raws.l<-list()
for(i in 1:length(chap.positions.v)){
  if(i != length(chap.positions.v)){
    chapter.title<-novel.lines.v[chap.positions.v[i]]
    start<-chap.positions.v[i]+1
    end<-chap.positions.v[i+1]-1
    chapter.lines.v<-novel.lines.v[start:end]
    chapter.words.v<-tolower(paste(chapter.lines.v, collapse=" "))
    chapter.words.l<-strsplit(chapter.words.v, "\\W")
    chapter.word.v<-unlist(chapter.words.l)
    chapter.word.v<-chapter.word.v[which(chapter.word.v!="")]
    chapter.freqs.t<-table(chapter.word.v)
    chapter.raws.l[[chapter.title]]<- chapter.freqs.t
    chapter.freqs.t.rel<-100*(chapter.freqs.t/sum(chapter.freqs.t))
    chapter.freqs.l[[chapter.title]]<-chapter.freqs.t.rel
  }
}
whale.l<-lapply(chapter.freqs.l, '[', 'whale')
whales.m<-do.call(rbind, whale.l)
ahab.l<-lapply(chapter.freqs.l, '[', 'ahab')
ahabs.m<-do.call(rbind, ahab.l)
whales.v<-as.vector(whales.m[,1])
ahabs.v<-as.vector(ahabs.m[,1])
whales.ahabs.m<-cbind(whales.v, ahabs.v)
colnames(whales.ahabs.m)<-c("whale", "ahab")
```

C.4 Chapter 6 Start Up Code

```
# Chapter 6 Start up code
text.v<-scan("data/plaintext/melville.txt", what="character", sep="\n")
start.v<-which(text.v == "CHAPTER 1. Loomings.")
end.v<-which(text.v == "orphan.")
novel.lines.v<- text.v[start.v:end.v]
novel.lines.v<-unlist(novel.lines.v)
chap.positions.v<-grep("^CHAPTER \\d", novel.lines.v)
last.position.v<- length(novel.lines.v)
chap.positions.v <- c(chap.positions.v , last.position.v)
chapter.freqs.l<-list()
chapter.raws.l<-list()
for(i in 1:length(chap.positions.v)){
  if(i != length(chap.positions.v)){
    chapter.title<-novel.lines.v[chap.positions.v[i]]
    start<-chap.positions.v[i]+1
    end<-chap.positions.v[i+1]-1
    chapter.lines.v<-novel.lines.v[start:end]
    chapter.words.v<-tolower(paste(chapter.lines.v, collapse=" "))
    chapter.words.l<-strsplit(chapter.words.v, "\\W")
    chapter.word.v<-unlist(chapter.words.l)
    chapter.word.v<-chapter.word.v[which(chapter.word.v!="")]
    chapter.freqs.t<-table(chapter.word.v)
    chapter.raws.l[[chapter.title]]<- chapter.freqs.t
    chapter.freqs.t.rel<-100*(chapter.freqs.t/sum(chapter.freqs.t))
    chapter.freqs.l[[chapter.title]]<-chapter.freqs.t.rel
  }
}
```

C.5 Chapter 7 Start Up Code

```
# Chapter 7 Start up code
text.v<-scan("data/plaintext/melville.txt", what="character", sep="\n")
start.v<-which(text.v == "CHAPTER 1. Loomings.")
end.v<-which(text.v == "orphan.")
novel.lines.v<- text.v[start.v:end.v]
novel.lines.v<-unlist(novel.lines.v)
chap.positions.v<-grep("^CHAPTER \\d", novel.lines.v)
last.position.v<- length(novel.lines.v)
chap.positions.v <- c(chap.positions.v , last.position.v)
chapter.freqs.l<-list()
chapter.raws.l<-list()
for(i in 1:length(chap.positions.v)){
  if(i != length(chap.positions.v)){
    chapter.title<-novel.lines.v[chap.positions.v[i]]
    start<-chap.positions.v[i]+1
    end<-chap.positions.v[i+1]-1
    chapter.lines.v<-novel.lines.v[start:end]
    chapter.words.v<-tolower(paste(chapter.lines.v, collapse=" "))
    chapter.words.l<-strsplit(chapter.words.v, "\\W")
    chapter.word.v<-unlist(chapter.words.l)
    chapter.word.v<-chapter.word.v[which(chapter.word.v!="")]
    chapter.freqs.t<-table(chapter.word.v)
    chapter.raws.l[[chapter.title]]<- chapter.freqs.t
    chapter.freqs.t.rel<-100*(chapter.freqs.t/sum(chapter.freqs.t))
    chapter.freqs.l[[chapter.title]]<-chapter.freqs.t.rel
  }
}
chapter.lengths.m<-do.call(rbind, lapply(chapter.raws.l,sum))
```

Appendix D
Appendix D: R Resources

NOTE: This page is still a work in progress.

D.1 Books

D.2 Packages and R Code Resources

D.3 References and Tutorials (Online)

Practice Exercise Solutions

Practice Exercises for Chapter 1

1.1

```
10+5
## [1] 15
10-5
## [1] 5
```

1.2

```
10*1576
## [1] 15760
```

1.3

```
15760/10
## [1] 1576
```

1.4

```
10+pi
## [1] 13.14
10/pi
## [1] 3.183
```

1.5

```
10^2
## [1] 100
```

1.6

```
x <- 10
x - 3
## [1] 7
```

1.7

```
x <- 10
x - 3 + 10 / 2
## [1] 12
```

1.8

```
x <- 10
```

1.9

```
sqrt(12)
## [1] 3.464
abs(-23)
## [1] 23
round(pi)
## [1] 3
```

1.10

```
1:10
##  [1]  1  2  3  4  5  6  7  8  9 10
12:37
##  [1] 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
## [16] 27 28 29 30 31 32 33 34 35 36 37
```

Practice Exercises for Chapter 2

2.1 The top ten most frequent words are found in the 1st through 10st position in the sorted vector. To see them, just enter:

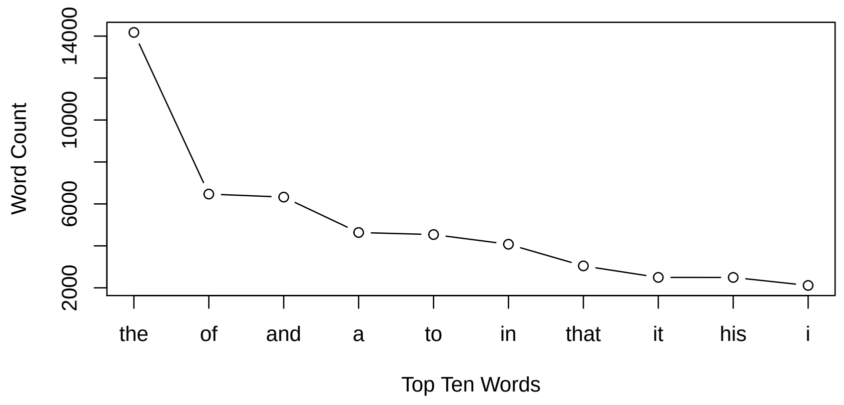
```
sorted.moby.freqs.t[1:10]
## moby.word.v
##   the    of   and     a    to    in  that    it
## 14175  6469  6325  4636  4539  4077  3045  2497
##   his     i
##  2495  2114
```

Visualizing the results is really as simple as using

```
plot(sorted.moby.freqs.t[1:10])
```

But adding a few more arguments to the `plot` function gives you a more informative graph.

```
plot(sorted.moby.freqs.t[1:10], type="b",
     xlab="Top Ten Words", ylab="Word Count",xaxt = "n")
axis(1,1:10, labels=names(sorted.moby.freqs.t[1:10]))
```



Practice Exercises for Chapter 3

3.1 First load Moby Dick

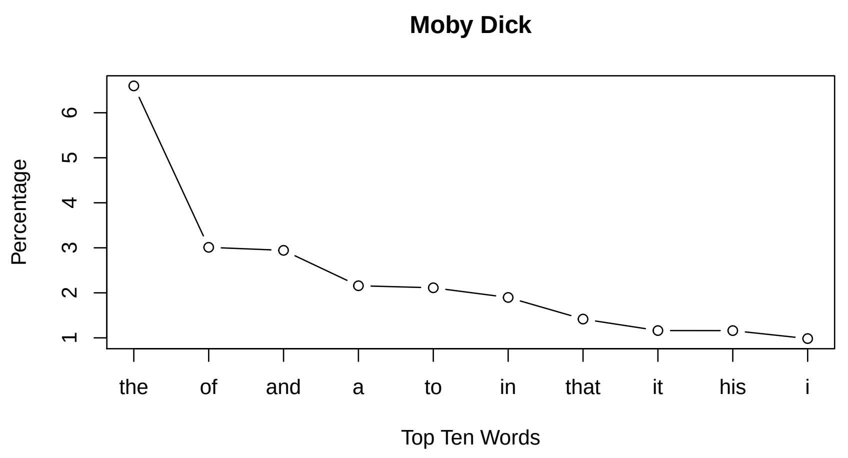
```
text.v<-scan("data/plaintext/melville.txt",
             what="character", sep="\n")
start.v<-which(text.v == "CHAPTER 1. Loomings.")
end.v<-which(text.v == "orphan.")
```

Now remove the boilerplate

```
start.metadata.v<-text.v[1:start.v -1]
end.metadata.v<-text.v[(end.v+1):length(text.v)]
metadata.v<-c(start.metadata.v, end.metadata.v)
novel.lines.v<- text.v[start.v:end.v]
novel.v<-paste(novel.lines.v, collapse=" ")
novel.lower.v<-tolower(novel.v)
moby.words.l<-strsplit(novel.lower.v, "\\W")
moby.word.v<-unlist(moby.words.l)
not.blanks.v <- which(moby.word.v!="")
moby.word.v<- moby.word.v[not.blanks.v]
moby.freqs.t<-table(moby.word.v)
sorted.moby.freqs.t<-sort(moby.freqs.t , decreasing=T)
```

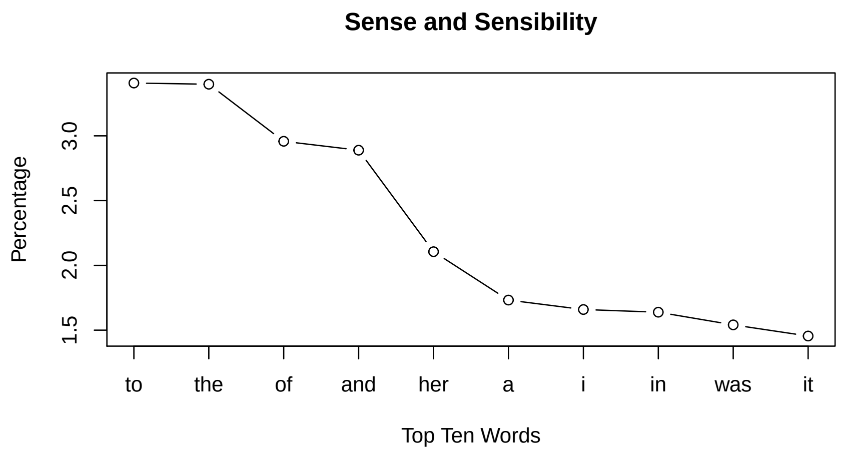
This next line was not in Exercise 2

```
sorted.moby.rel.freqs.t<-100*(sorted.moby.freqs.t/sum(sorted.moby.freqs.t))
plot(sorted.moby.rel.freqs.t[1:10],
     main="Moby Dick",
     type="b",
     xlab="Top Ten Words",
     ylab="Percentage",
     xaxt = "n")
axis(1,1:10, labels=names(sorted.moby.rel.freqs.t[1:10]))
```



Now do something similar by loading *Sense and Sensibility*. Since we'll only need the final variables (`sorted.moby.rel.freqs.t` and `sorted.sense.rel.freqs.t`), we can reuse many of the same variable names.

```
text.v<-scan("data/plainText/austen.txt",
             what="character", sep="\n")
start.v<-which(text.v == "CHAPTER 1")
end.v<-which(text.v == "THE END")
novel.lines.v<- text.v[start.v:end.v]
novel.v<-paste(novel.lines.v, collapse=" ")
novel.lower.v<-tolower(novel.v)
sense.words.l<-strsplit(novel.lower.v, "\\W")
sense.word.v<-unlist(sense.words.l)
not.blanks.v <- which(sense.word.v!="")
sense.word.v<- sense.word.v[not.blanks.v]
sense.freqs.t<-table(sense.word.v)
sorted.sense.freqs.t<-sort(sense.freqs.t , decreasing=T)
sorted.sense.rel.freqs.t<-100*
  (sorted.sense.freqs.t/sum(sorted.sense.freqs.t))
plot(sorted.sense.rel.freqs.t[1:10],
     main="Sense and Sensibility", type="b",
     xlab="Top Ten Words", ylab="Percentage",xaxt = "n")
axis(1,1:10, labels=names(sorted.sense.rel.freqs.t[1:10]))
```



3.2

```
unique(c(names(sorted.moby.rel.freqs.t[1:10]),
         names(sorted.sense.rel.freqs.t[1:10])))
##  [1] "the"  "of"   "and"  "a"    "to"   "in"
##  [7] "that" "it"   "his"  "i"    "her"  "was"
```

3.3

```
names(sorted.sense.rel.freqs.t[
  which(names(sorted.sense.rel.freqs.t[1:10])
         %in% names(sorted.moby.rel.freqs.t[1:10]))])
## [1] "to"  "the" "of"  "and" "a"   "i"   "in"
## [8] "it"
```

3.4

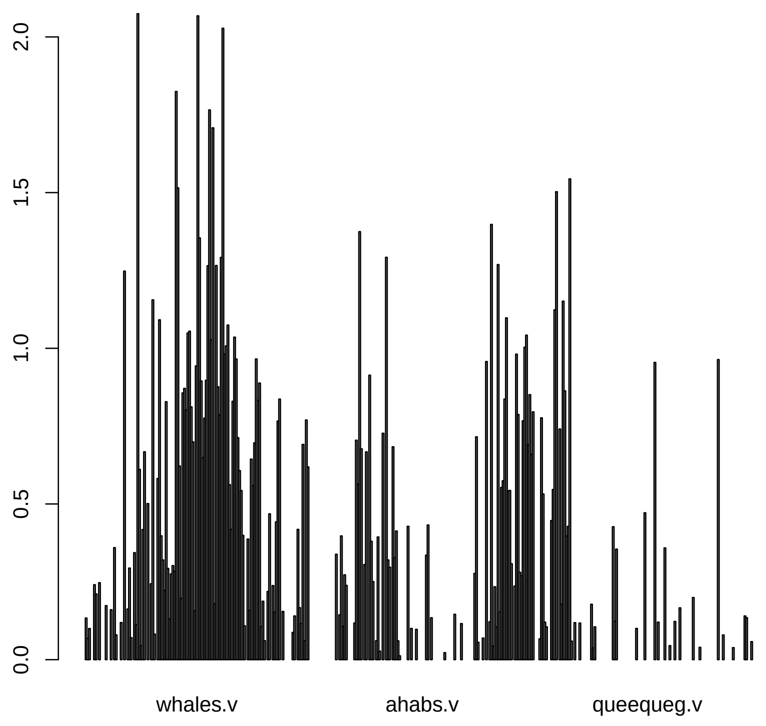
```
presentSense<-which(names(sorted.sense.rel.freqs.t[1:10])
                    %in% names(sorted.moby.rel.freqs.t[1:10]))
names(sorted.sense.rel.freqs.t[1:10])[-presentSense]
```

```
## [1] "her" "was"
presentMoby<-which(names(sorted.moby.rel.freqs.t[1:10])
                   %in% names(sorted.sense.rel.freqs.t[1:10]))
names(sorted.moby.rel.freqs.t[1:10])[-presentMoby]
## [1] "that" "his"
```

Practice Exercises for Chapter 4

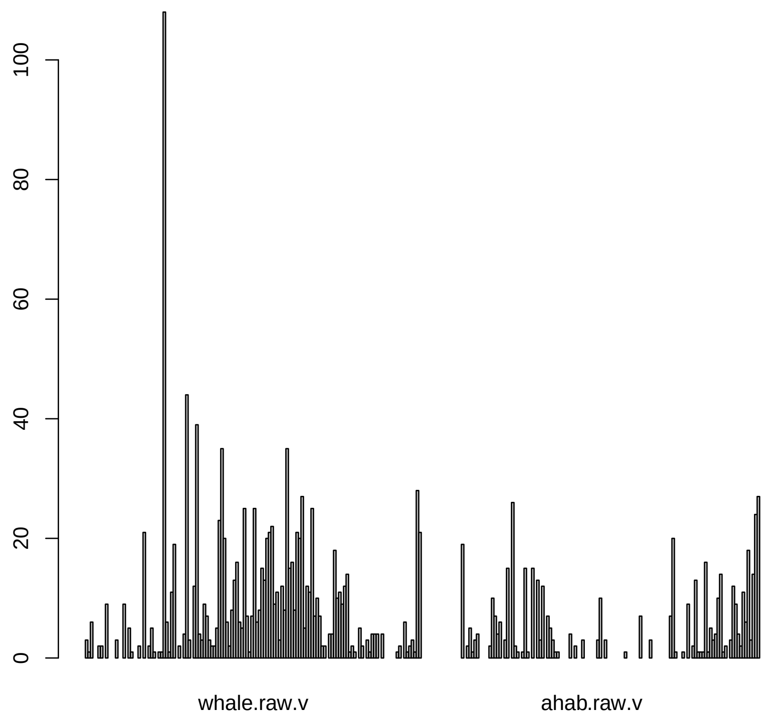
4.1

```
whales.l<-lapply(chapter.freqs.l, '[', 'whale')
whales.m<-do.call(rbind, whales.l)
whales.v<-as.vector(whales.m[,1])

ahabs.l<-lapply(chapter.freqs.l, '[', 'ahab')
ahabs.m<-do.call(rbind, ahabs.l)
ahabs.v<-as.vector(ahabs.m[,1])

queequeg.l<-lapply(chapter.freqs.l, '[', 'queequeg')
queequeg.m<-do.call(rbind, queequeg.l)
queequeg.v<-as.vector(queequeg.m[,1])

whales.ahabs.queequeg.m<-cbind(whales.v, ahabs.v,queequeg.v)
barplot(whales.ahabs.queequeg.m, beside=T, col="grey")
```



4.2

```
whale.raw.l<-lapply(chapter.raws.l, '[', 'whale')
whale.raw.m<-do.call(rbind, whale.raw.l)
whale.raw.v<-as.vector(whale.raw.m[,1])
ahab.raw.l<-lapply(chapter.raws.l, '[', 'ahab')
ahab.raw.m<-do.call(rbind, ahab.raw.l)
ahab.raw.v<-as.vector(ahab.raw.m[,1])
whales.ahabs.raw.m<-cbind(whale.raw.v, ahab.raw.v)
barplot(whales.ahabs.raw.m, beside=T, col="grey")
```



Practice Exercises for Chapter 5

5.1

```
my.l <- lapply(chapter.freqs.l, "[", "my")
my.m<-do.call(rbind, my.l)
my.v<-as.vector(my.m[,1])
i.l <- lapply(chapter.freqs.l, "[", "i")
i.m<-do.call(rbind, i.l)
i.v<-as.vector(i.m[,1])
whales.ahabs.my.i.m<-cbind(whales.v, ahabs.v, my.v, i.v)
whales.ahabs.my.i.m[which(is.na(whales.ahabs.my.i.m))]<-0
cor(whales.ahabs.my.i.m)
##          whales.v  ahabs.v     my.v      i.v
## whales.v   1.0000 -0.24701 -0.26124 -0.28257
## ahabs.v   -0.2470  1.00000  0.09354  0.07104
## my.v      -0.2612  0.09354  1.00000  0.77746
## i.v       -0.2826  0.07104  0.77746  1.00000
```

5.2

```
my.i.m<-cbind(my.v, i.v)
my.i.m[which(is.na(my.i.m))]<-0
my.i.cor.data.df<-as.data.frame(my.i.m)
cor(my.i.cor.data.df$i, my.i.cor.data.df$my)
## [1] 0.7775
i.my.cors.v<-NULL
for(i in 1:10000){
  i.my.cors.v<-c(i.my.cors.v,
                 cor(sample(my.i.cor.data.df$i), my.i.cor.data.df$my))
}
min(i.my.cors.v)
## [1] -0.2826
max(i.my.cors.v)
## [1] 0.3487
range(i.my.cors.v)
## [1] -0.2826  0.3487
mean(i.my.cors.v)
## [1] -0.000627
sd(i.my.cors.v)
## [1] 0.08715
```

Practice Exercises for Chapter 6

6.1

```
ttr.v<-as.vector(ttr.m)
chapter.lengths.m<-do.call(rbind, lapply(chapter.raws.l,sum))
chap.len.v<-as.vector(chapter.lengths.m)
cor(ttr.v, chap.len.v)
## [1] -0.7972
```

A Correlation coefficient of `-0.7972` indicates strong negative correlation. As the length of the chapter increases the TTR scores decrease.

6.2

```
mean.word.use.v<-as.vector(mean.word.use.m)
cor(mean.word.use.v, chap.len.v)
## [1] 0.8924
```

A Correlation coefficient of `0.8924` indicates a strong positive correlation. As the length of a chapter increases, the overall mean word frequency increases as well. More words in the chapter means more repeated words.

6.3

```
cor(ttr.v, chap.len.v)
## [1] -0.7972
my.cors.v<-NULL
for(i in 1:10000){
  my.cors.v<-c(my.cors.v, cor(sample(ttr.v), chap.len.v))
}
min(my.cors.v)
## [1] -0.2882
max(my.cors.v)
## [1] 0.3586
range(my.cors.v)
## [1] -0.2882  0.3586
mean(my.cors.v)
## [1] -0.001157
sd(my.cors.v)
## [1] 0.08666
```

The permutation test reveals that the observed correlation is highly unlikely to be seen by mere chance alone. In `10,000` iterations the highest positive correlation was `0.3586` and the lowest negative `-0.2882`. The `mean` hovered near zero indicating that the observed correlation was far outside the norm expected by chance.

Practice Exercises for Chapter 7

7.1

```
lengths.v.hapax<-cbind(chapter.hapax.v, chapter.lengths.m)
cor(lengths.v.hapax [,1], lengths.v.hapax [,2])
## [1] 0.9677
```

In this case the correlation is extremely strong with an R-value of 97. As the chapters of *Moby Dick* get longer, not only do we observe the same words repeated more often, but we also see an increase in the number of new words being introduced.

7.2

```
ranks<-order(hapax.percentage, decreasing=TRUE)
hapax.percentage[ranks,]
```

A correlation coefficient of `.87` indicates that Jane Austen is less consistent than Melville when it comes to the introduction of new words into her novel even while she increases the length of her chapters. It turns out, in fact, that in terms of vocabulary size and richness, Austen is very consistent. Her working vocabulary in *Sense and Sensibility* contains `6,325` unique word types and from one of her novels to the next she rarely deviates very far a base vocabulary of about `6300` words types. For comparison, recall that Melville's vocabulary in Moby Dick contains `16,872` unique word types spread over `214,889` tokens. Austen uses `6,325` types over `120,766`. Even though Austen's *Sense and Sensibility* is much shorter that *Moby Dick*, Austen has a smaller vocabulary and repeats words much more often. Austen uses each word an average of `19` times whereas Melville uses each word in his vocabulary only about `13` times on average.

Practice Exercises for Chapter 8

8.1

```
context<-5
dog.positions.sense<-which(my.corpus.l[[1]][]=="dog")
dog.positions.moby<-which(my.corpus.l[[2]][]=="dog")

# Answer for Sense and Sensibility
for(i in 1:length(dog.positions.sense)){
  start<-dog.positions.sense[i]-context
  end<-dog.positions.sense[i]+context
  cat(my.corpus.l[[1]][start:end], "\n")
}
## a fellow such a deceitful dog it was only the last

# Answer for Moby Dick
for(i in 1:length(dog.positions.moby)){
  start<-dog.positions.moby[i]-context
  end<-dog.positions.moby[i]+context
  cat(my.corpus.l[[2]][start:end], "\n")
}
## all over like a newfoundland dog just from the water and
## a fellow that in the dog days will mow his two
## was seen swimming like a dog throwing his long arms straight
## filling one at last down dog and kennel starting at the
## not tamely be called a dog sir then be called ten
## t he call me a dog blazes he called me ten
## sacrifice of the sacred white dog was by far the holiest
## life that lives in a dog or a horse indeed in
## the sagacious kindness of the dog the accursed shark alone can
## boats the ungracious and ungrateful dog cried starbuck he mocks and
## intense whisper give way greyhounds dog to it i tell ye
## to the whale that a dog does to the elephant nevertheless
## aries or the ram lecherous dog he begets us then taurus
## is dr bunger bunger you dog laugh out why don t
## to die in pickle you dog you should be preserved to
## round ahab and like a dog strangely snuffing this man s
## lad five feet high hang dog look and cowardly jumped from
## as a sagacious ship s dog will in drawing nigh to
## the compass and then the dog vane and then ascertaining the
```

8.2

```
for(i in 1:length(dog.positions.moby)){
  start<-dog.positions.moby[i]-context
  end<-dog.positions.moby[i]+context
  before<-my.corpus.l[[2]][start:(start+context-1)]
  after<-my.corpus.l[[2]][(start+context+1):end]
  keyword<-my.corpus.l[[2]][start+context]
  cat("----------------------", i, "----------------------", "\n")
  cat(before,"[",keyword, "]", after, "\n")
}
## ---------------------- 1 ----------------------
## all over like a newfoundland [ dog ] just from the water and
## ---------------------- 2 ----------------------
## a fellow that in the [ dog ] days will mow his two
## ---------------------- 3 ----------------------
## was seen swimming like a [ dog ] throwing his long arms straight
## ---------------------- 4 ----------------------
## filling one at last down [ dog ] and kennel starting at the
## ---------------------- 5 ----------------------
## not tamely be called a [ dog ] sir then be called ten
## ---------------------- 6 ----------------------
## t he call me a [ dog ] blazes he called me ten
## ---------------------- 7 ----------------------
## sacrifice of the sacred white [ dog ] was by far the holiest
## ---------------------- 8 ----------------------
## life that lives in a [ dog ] or a horse indeed in
## ---------------------- 9 ----------------------
## the sagacious kindness of the [ dog ] the accursed shark alone can
## ---------------------- 10 ----------------------
## boats the ungracious and ungrateful [ dog ] cried starbuck he mocks and
## ---------------------- 11 ----------------------
## intense whisper give way greyhounds [ dog ] to it i tell ye
## ---------------------- 12 ----------------------
## to the whale that a [ dog ] does to the elephant nevertheless
## ---------------------- 13 ----------------------
## aries or the ram lecherous [ dog ] he begets us then taurus
## ---------------------- 14 ----------------------
## is dr bunger bunger you [ dog ] laugh out why don t
## ---------------------- 15 ----------------------
## to die in pickle you [ dog ] you should be preserved to
## ---------------------- 16 ----------------------
## round ahab and like a [ dog ] strangely snuffing this man s
## ---------------------- 17 ----------------------
## lad five feet high hang [ dog ] look and cowardly jumped from
## ---------------------- 18 ----------------------
## as a sagacious ship s [ dog ] will in drawing nigh to
## ---------------------- 19 ----------------------
## the compass and then the [ dog ] vane and then ascertaining the
```

Practice Exercises for Chapter 9

9.1

```
doitKwicBetter<-function(named.text.word.vector.l){
  show.files(names(named.text.word.vector.list))
  # ask the user for three bits of information
  file.id<- as.numeric(
    readline("Which file would you like to examine? Enter a number: \n"))
  context<- as.numeric(
    readline("How much context do you want to see, Enter a number: \n"))
  keyword<- tolower((readline("Enter a keyword: \n")))
  hits.v<-which(named.text.word.vector.l[[file.id]] == keyword)
  if(length(hits.v)>0){
    result<-NULL
    for(h in 1:length(hits.v)){
      start<-hits.v[h]-context
      if(start < 1){
        start<-1
      }
      end<-hits.v[h]+context
      cat(named.text.word.vector.l[[file.id]][start:end], "\n")
      myrow<-cbind(hits.v[h],
          paste(
            named.text.word.vector.l[[file.id]][start:(hits.v[h]-1)],
                collapse=" "),
          paste(
            named.text.word.vector.l[[file.id]][hits.v[h]],
                collapse=" "),
          paste(
            named.text.word.vector.l[[file.id]][(hits.v[h]+1):end],
          collapse=" "))
      result<-rbind(result,myrow)
    }
  }
  colnames(result)<-c("position", "left", "keyword", "right")
  return(result)
}
```

9.2

```
doitKwicBest<-function(named.text.word.vector.l){
  show.files(names(named.text.word.vector.l))
  # ask the user for three bits of information
  file.id<- as.numeric(
    readline("Which file would you like to examine? Enter a number: \n"))
  context<- as.numeric(
    readline("How much context do you want to see, Enter a number: \n"))
  keyword<- tolower((readline("Enter a keyword: \n")))
  hits.v<-which(named.text.word.vector.l[[file.id]] == keyword)
  if(length(hits.v)>0){
    result<-NULL
    for(h in 1:length(hits.v)){
      start<-hits.v[h]-context
      if(start < 1){
        start<-1
      }
      end<-hits.v[h]+context
      cat("\n-----------------------", h, "-------------------------\n")
      cat(named.text.word.vector.l[[file.id]][start:(hits.v[h]-1)], sep=" ")
      cat(" [", named.text.word.vector.l[[file.id]][hits.v[h]],"] ", sep="")
      cat(named.text.word.vector.l[[file.id]][(hits.v[h]+1):end], sep=" ")
      myrow<-cbind(hits.v[h],
        paste(named.text.word.vector.l[[file.id]][start:(hits.v[h]-1)],
              collapse=" "),
        paste(named.text.word.vector.l[[file.id]][hits.v[h]],
              collapse=" "),
        paste(named.text.word.vector.l[[file.id]][(hits.v[h]+1):end],
              collapse=" "))
      result<-rbind(result,myrow)
    }
    colnames(result)<-c("position", "left", "keyword", "right")
    toprint<-as.numeric((
      readline("Would you like to save this result to a file:
               enter 1=yes or 0=no \n")))
    if(toprint==1){
      write.csv(result,
        paste("results/", keyword,"_In_",
        context, names(named.text.word.vector.l)[file.id], ".csv", sep=""))
    }
  } else {
    cat("YOUR KEYWORD WAS NOT FOUND\n")
  }
}
```

Practice Exercises for Chapter 10

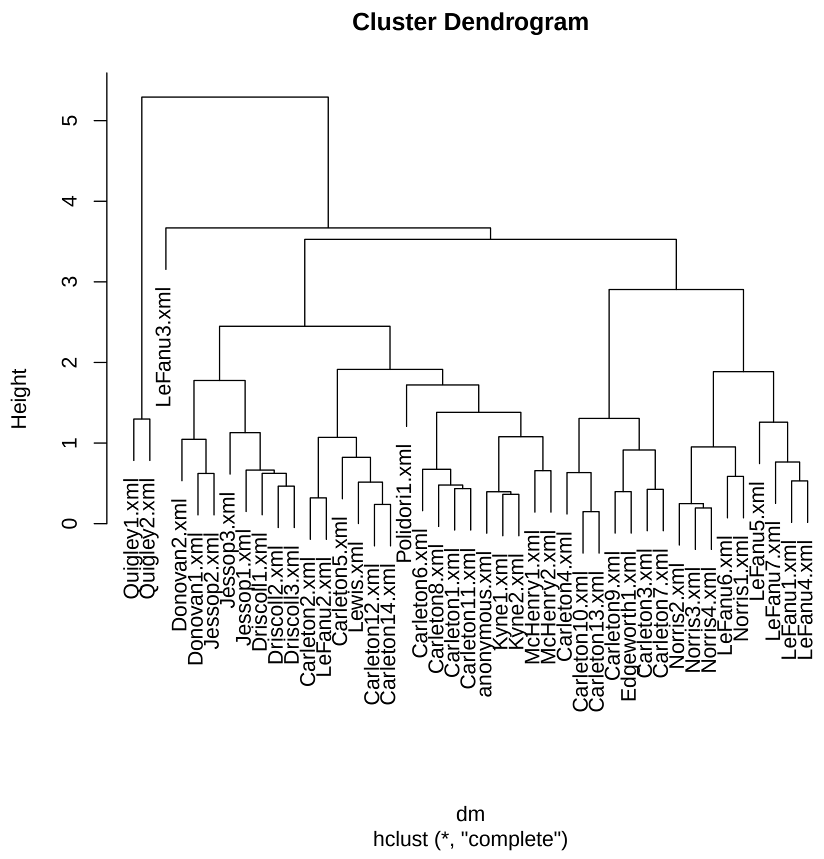
10.1

```
(respStmt<-xpathApply(doc,
                      "/d:TEI//d:fileDesc//d:titleStmt//d:respStmt",
                      namespaces=c(d = "http://www.tei-c.org/ns/1.0"))[[1]])
## <respStmt>
##   <resp>converted into TEI-conformant markup by</resp>
##   <name type="contributor">Eric Lease Morgan</name>
##   <resp>Modified for use as an exercise by</resp>
##   <name type="contributor">Matthew Jockers</name>
## </respStmt>
```

Practice Exercises for Chapter 11

11.1

```
smaller.m <- final.m[,apply(final.m,2,mean)>=2.5]
dim(smaller.m)
## [1] 43  4
dm<-dist(smaller.m)
cluster <- hclust(dm)
cluster$labels<-names(book.freqs.l)
plot(cluster)
```

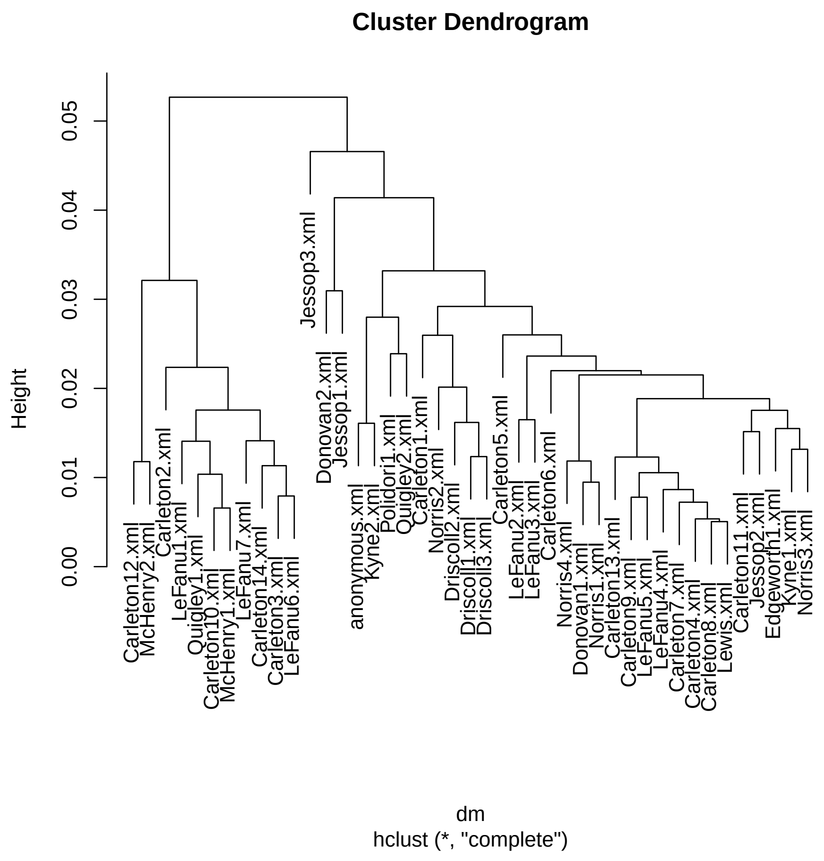


Using a mean threshold of 2.5 reduces the feature set to just 4 words, and the anonymous text is still found closest to Kyne. However, a mean threshold of 2.75 reduces the feature set to just 3 words, and the anonymous text is then found closest to Donovan.

11.2

```
random.m<-final.m[,sample(colnames(final.m), 100)]
dim(random.m)
## [1]  43 100
```

```
dm<-dist(random.m)
cluster <- hclust(dm)
cluster$labels<-names(book.freqs.l)
plot(cluster)
```



Practice Exercises for Chapter 12

12.1

```
keepers.v<-which(freq.means.v >=.00014)
length(keepers.v)
## [1] 681
smaller.df<-authorship.df[, c(names(authorship.df)[1:3],
                              names(keepers.v))]
anon.v<-which(smaller.df$author.v == "anonymous")
train <- smaller.df[-anon.v,4:ncol(smaller.df)]
class.f <- smaller.df[-anon.v,"author.v"]
model.svm <- svm(train, class.f)

## Warning: Variable(s) 'bryce' constant. Cannot scale data.

pred.svm <- predict(model.svm, train)
testdata <- smaller.df[anon.v,4:ncol(smaller.df)]
final.result<-predict (model.svm, testdata)
as.data.frame(final.result)
##              final.result
## anonymous_1      Carleton
## anonymous_10     Carleton
## anonymous_2      Carleton
## anonymous_3      Carleton
## anonymous_4      Carleton
## anonymous_5      Carleton
## anonymous_6      Carleton
## anonymous_7      Carleton
## anonymous_8      Carleton
## anonymous_9      Carleton
```

12.2

```
keepers.v<-which(freq.means.v >=.00014)
smaller.df<-authorship.df[, c(names(authorship.df)[1:3],
  names(keepers.v))]
author.sums<-aggregate(smaller.df[,
  4:ncol(smaller.df)],
  list(smaller.df[,1]), sum)
reduced.author.sums<-author.sums[,
  colSums(author.sums==0) == 0]
keepers.v<-colnames(
  reduced.author.sums)[2:ncol(reduced.author.sums)]
new.smaller.df<-smaller.df[,
  c("author.v","sampletext","samplechunk", keepers.v)]
anon.v<-which(new.smaller.df$author.v == "anonymous")
train <- new.smaller.df[-anon.v,4:ncol(new.smaller.df)]
class.f <- new.smaller.df[-anon.v,"author.v"]
model.svm <- svm(train, class.f)
pred.svm <- predict(model.svm, train)
testdata <- new.smaller.df[anon.v,4:ncol(new.smaller.df)]
final.result<-predict (model.svm, testdata)
as.data.frame(final.result)
##              final.result
## anonymous_1          Kyne
## anonymous_10         Kyne
## anonymous_2          Kyne
## anonymous_3          Kyne
## anonymous_4          Kyne
## anonymous_5          Kyne
## anonymous_6          Kyne
## anonymous_7          Kyne
## anonymous_8          Kyne
## anonymous_9          Kyne
```

Practice Exercises for Chapter 13

13.1

```
for(i in 1:43){
  topic.top.words<-mallet.top.words(topic.model, topic.words.m[i,], 100)
  print(wordcloud(topic.top.words$words,
                  topic.top.words$weights,
                  c(4,.8), rot.per=0,
                  random.order=F))
}
```

13.2

```
mallet.instances <- mallet.import(documents$id,
                                  documents$text,
                                  "data/stoplist-exp.csv",
                                  FALSE,
                                  token.regexp="[\\p{L}']+")
topic.model <- MalletLDA(num.topics=43)
topic.model$loadDocuments(mallet.instances)
vocabulary <- topic.model$getVocabulary()
word.freqs <- mallet.word.freqs(topic.model)
topic.model$train(400)
topic.words.m <- mallet.topic.words(topic.model,
                                    smoothed=TRUE,
                                    normalized=TRUE)
vocabulary <- topic.model$getVocabulary()
colnames(topic.words.m)<-vocabulary
for(i in 1:43){
  topic.top.words<-mallet.top.words(topic.model, topic.words.m[i,], 100)
  print(wordcloud(topic.top.words$words,
                  topic.top.words$weights,
                  c(4,.8), rot.per=0,
                  random.order=F))
}
```