MTH 3270 Notes 9

19 Text as Data (19)

- Fields such as *natural language processing* and *computational linguistics* work with **text** documents to extract meaning using computers.
- We'll look at tools for working with **text**, including **regular expressions**.

19.1 Tools for Working With Text

- R has several built-in functions for working with ***character strings*** (**text**):

```
tolower(),
toupper()      # Change the case of the letters in a character string
grep(),
grepl()        # Search a character vector for a specified character
               # pattern, and return the vector indices (or a logical
               # vector) indicating the matches
sub(), gsub()  # Replace the first instance (or all instances) of one
               # character pattern by another
nchar()        # Returns the number of characters in a character string
paste()        # Concatenate (combine) character strings
strsplit()     # Splits a character string into substrings according
               # to a character pattern for splitting
substr()       # Returns the substring at a specified character posi-
               # tion within a character string.  Can also be used to
               # replace the substring.
regexpr(),
gregexpr()     # Returns the character position of the first instance
               # (or all instances) of a specified character pattern
```

19.1.1 Using `tolower()` and `toupper()` to Change the Case of a Character String

- `tolower()` and `toupper()` take an argument `x`, a character string (represented as a one-element `"character"` vector) or entire (multi-element) `"character"` vector, and convert capital letters to lower case or vice versa. For example:

```
tolower(x = "The rain in Spain stays mainly in the plain")

## [1] "the rain in spain stays mainly in the plain"

toupper(x = "The rain in Spain stays mainly in the plain")

## [1] "THE RAIN IN SPAIN STAYS MAINLY IN THE PLAIN"
```

19.1.2 Using `grep()` and `grepl()` to Search for a Character Pattern

- `grep()` takes arguments `pattern`, a character pattern, and `x`, a `"character"` *vector*, and returns the **indices** of the elements of `x` that contain the pattern.

`grepl()` takes the same arguments, but returns a `"logical"` **vector** indicating the elements that contain the pattern.

- For example, consider the `"character"` vector:

```
pets <- c("dog", "cat", "gerbil", "hamster", "parakeet", "goldfish", "iguana")
```

To determine which elements of `pets` contain the pattern `"er"`, type:

```
grep(pattern = "er", x = pets)

## [1] 3 4
```

This indicates that the 3rd and 4th elements, namely `"gerbil"` and `"hamster"`, contain `"er"`.

Now watch what happens when we use `grepl()`:

```
grepl(pattern = "er", x = pets)

## [1] FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE
```

`grepl()` returns a `"logical"` vector whose elements are `TRUE` or `FALSE` depending on whether the corresponding element of `x` contains the `pattern`.

- Setting the optional argument `value = TRUE` in `grep()` gets the actual elements of `x` that contain the `pattern`:

```
grep(pattern = "er", x = pets, value = TRUE)

## [1] "gerbil"  "hamster"
```

19.1.3 Using `sub()` and `gsub()` to Substitute One Character Pattern for Another

- `sub()` takes arguments `pattern`, a character pattern, `replacement`, another character pattern, and `x`, a character string (or entire `"character"` vector), and replaces the *first instance* of the `pattern` in `x` by the `replacement`.

`gsub()` takes the same arguments, but replaces *all instances* of the `pattern` by the `replacement`.

- For example, consider the tongue twister in which `"pack"` was incorrectly typed (twice) instead of `"pick"`:

```
twister <- "Peter Piper packed a peck of packled peppers"
```

```
sub(pattern = "pack", replacement = "pick", x = twister)

## [1] "Peter Piper picked a peck of packled peppers"
```

Note that `sub()` only replaced the *first instance* of `"pack"` by `"pick"`.

To replace *all instances*, use `gsub()`:

```
gsub(pattern = "pack", replacement = "pick", x = twister)

## [1] "Peter Piper picked a peck of pickled peppers"
```

19.1.4 Using `nchar()` to Count Characters

- `nchar()` counts the number of characters in a character string. For example, `"Mississippi"` has 11 characters:

```
nchar("Mississippi")

## [1] 11
```

- `nchar()` is *vectorized*. For example (using the `"character"` vector `pets` from above):

```
nchar(pets)

## [1] 3 3 6 7 8 8 6
```

19.1.5 Using `paste()` to Combine Character Strings

- `paste()` combines two (or more) character strings together into one character string. An optional argument, `sep`, is used to specify the character separator to use when pasting the strings together. Its default value is `" "`, which separates the terms by a blank space.
- Here's an example:

```
paste("I", "love", "R")

## [1] "I love R"
```

- If the arguments passed to `paste()` are vectors, they're combined term-by-term to give a character vector result.

```
paste(c("A", "B", "C"), 1:3, sep = "")

## [1] "A1" "B2" "C3"
```

Note the use of `sep = ""` to indicate pasting with no separation.

Vector arguments are recycled as needed. For example, below, the one-element vector `"A"` gets recycled:

```
paste("A", 1:6, sep = "")

## [1] "A1" "A2" "A3" "A4" "A5" "A6"
```

19.1.6 Using `substr()` to Extract or Replace Character Substrings

- `substr()` takes arguments `x`, a character string (or entire `"character"` vector), and `start` and `stop`, two character positions, and returns the substring of `x` from `start` to `stop`. For example:

```
MM <- "Mickey Mouse"
```

```
substr(MM, start = 3, stop = 6)

## [1] "ckey"
```

This says that the four characters occupying the 3rd through 6th positions of `"Mickey Mouse"` are `"ckey"`.

- Blank spaces are considered characters, so the 7th position of `"Mickey Mouse"` is occupied by `" "`, not `"M"`.
- `substr()` is a *replacement function*, so it can also be used to replace a substring. For example:

```
substr(MM, start = 3, stop = 6) <- "nnie"
```

```
MM

## [1] "Minnie Mouse"
```

19.1.7 Using `strsplit()` to Split Character Strings

- `strsplit()` does the opposite of `paste()`. It takes arguments `x`, a character string (or entire `"character"` vector), and `split`, a character pattern, and splits the character string into substrings according to matches of `split`. It returns a *list* of `"character"` vectors, one for each element of `x`.
- For example, to split the tongue twister

```
twister <- "Peter Piper picked a peck of pickled peppers"
```

into separate words (so `split = " "`), type:

```
strsplit(twister, split = " ")

## [[1]]
## [1] "Peter"   "Piper"   "picked"  "a"       "peck"    "of"      "pickled"
## [8] "peppers"
```

Note that in this case, because `twister` is a single character string (i.e. a one-element `"character"` vector), `strsplit()` returns a one-element *list*, with that element being a `"character"` vector.

19.1.8 Using `regexpr()` and `gregexpr()` to Search for Character Patterns

- `regexpr()` takes arguments `pattern`, a character pattern, and `text`, a character string (or entire `"character"` vector), and searches the `text` for the `pattern`, returning the *starting character position* of the *first match* (or -1 if there's none).

  If `text` is a `"character"` vector, `regexpr()` returns a vector of the same length indicating the position of the first match in each element of `text`.

  `gregexpr()` takes the same arguments, but returns a vector containing the *starting positions* of *all matches* of the `pattern` in the `text`.

  If `text` is a `"character"` vector, `gregexpr()` returns a *list* of the same length, each element of which is a vector indicating the *starting positions* of *all matches* of the `pattern` in the corresponding element of `text`.
- For example:

```
twister <- "Peter Piper picked a peck of pickled peppers"
```

```
regexpr(pattern = "pick", text = twister)

## [1] 13
## attr(,"match.length")
## [1] 4
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

This indicates that the *first match* of the pattern `"pick"` begins at the 13th character of `twister`.

The *attribute* `match.length` gives the length of the `pattern`. The `useBytes` *attribute* indicates whether matching was done byte-by-byte, as opposed to character-by-character. See the help page.

Note that `regexpr()` only located the *first instance* of `"pick"`. To find *all instances*, use `gregexpr()`:

```
gregexpr(pattern = "pick", text = twister)

## [[1]]
## [1] 13 30
## attr(,"match.length")
## [1] 4 4
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

This says that `"pick"` appears twice in `twister`, once starting at the 13th character position, and a second time starting at the 30th position.

Section 19.1 Exercises

Exercise 1 `paste()` combines two (or more) character strings together into one character string. Try the following (taken from the `paste()` help page) and report the results:

a)
```
paste("Today's date is", date())
```

b)
```
paste("X", 1:5, sep = "")
```

Exercise 2 This exercise concerns `paste()` and it's opposite, `strsplit()`.

a) `paste()` *combines* character strings. Consider the following three character strings.

```
first <- "Louis"
middle <- "Daniel"
last <- "Armstrong"
```

Write a command involving `paste()` that returns the *single* character string

```
## [1] "Louis Daniel Armstrong"
```

Report your **R command**.

b) `paste()` is *vectorized*. Consider the following three `"character"` vectors:

```
first <- c("Louis", "John", "Miles", "Ella")
middle <- c("Daniel", "William", "Dewey", "Jane")
last <- c("Armstrong", "Coltrane", "Davis", "Fitzgerald")
```

Write a command involving `paste()` that returns the *single* character string

```
## [1] "Louis Daniel Armstrong" "John William Coltrane"
## [3] "Miles Dewey Davis"      "Ella Jane Fitzgerald"
```

Report your **R command**.

c) `strsplit()` does the opposite of `paste()` – it *splits* a character string. Consider the following character string.

```
full <- "Sarah Lois Vaughan"
```

Write a command involving `strsplit()` that returns the *three* character strings

```
## [[1]]
## [1] "Sarah"   "Lois"    "Vaughan"
```

Report your **R command**.

d) `strsplit()` is *vectorized*. Consider the following `"character"` vector:

```
full <- c("Sarah Lois Vaughan", "Thelonious Sphere Monk", "Chet Henry Baker",
          "Wynton Learson Marsalis")
```

Write a command involving `strsplit()` that returns the *three* vectors of character strings

```
## [[1]]
## [1] "Sarah"   "Lois"    "Vaughan"
##
## [[2]]
## [1] "Thelonious" "Sphere"     "Monk"
##
## [[3]]
## [1] "Chet"  "Henry" "Baker"
##
## [[4]]
## [1] "Wynton"   "Learson"  "Marsalis"
```

Report your **R command**.

Exercise 3 This exercise concerns `regexpr()` and `gregexpr()`.

Here's a quote from J.K. Rowling's book *Harry Potter and the Chamber of Secrets*:

```
quote <- "Aunt Petunia was horse-faced and bony; Dudley was blond, pink, and
porky. Harry, on the other hand, was small and skinny, with brilliant green
eyes and jet-black hair that was always untidy. He wore round glasses, and on
his forehead was a thin, lightning-shaped scar."
```

After creating the character string `quote` above, inspect it for embedded *newline* characters `"\n"`, and if there are any, remove them, for example by typing:

```
quote <- gsub(pattern = "\n", replacement = " ", x = quote)
```

Then remove periods(`.`), commas (`,`), and semicolons (`;`) using the *regular expression* `[\\.,;]` (Section 19.2 gives an explanation):

```
quote <- gsub(pattern = "[\\.,;]", replacement = "", x = quote)

quote
```

a) `gregexpr()` searches a character string for a pattern. Search `quote` for *hyphens* (`-`) by running the following command.

```
gregexpr(pattern = "-", text = quote)
```

What do the *three* values returned by `gregexpr()`, **23**, **156**, and **256**, represent?

b) `regexpr()` is *another* way to search a character string for a pattern. Run the following command.

```
regexpr(pattern = "-", text = quote)
```

What does the *one* value returned by `gregexpr()`, **23**, represent?

Exercise 4 Here's a famous quote from the 1948 film The Treasure of the Sierra Madre:

```
badges <- "Badges? We ain't got no badges. We don't need no badges. I don't have to
           show you any stinking badges!"
```

Create the `badges` character string. It will be a one-element vector whose one element is the entire quote. If you copy and paste into R, make sure it's all on one line so that you don't end up with a newline character, `\n`, included in the quote.

a) Use `tolower()` to convert the quote to all lower case, overwriting the previous version of `badges`. You should now have this:

```
badges

## [1] "badges? we ain't got no badges. we don't need no badges. i don't have to show you any stin
```

Report your **R command**.

b) We want to "clean up" the quote a bit by removing the punctuation marks (periods, question mark, and exclamation marks).

Use `gsub()`, with `pattern = "!"` and `replacement = ""`, to remove the exclamation mark from `badges`, overwriting the previous version of `badges`. Report your R command.

c) Removing the question mark and periods is a bit tricky. We need to specify `pattern = "\\?"` and `pattern = "\\."` in the call to `gsub()` (with and `replacement = ""`). Using `pattern = "?"` and `pattern = "."` **won't work**. (Section 19.2 gives an explanation.)

Remove the question mark and periods, overwriting the previous version of `badges`. You should now have this:

```
badges

## [1] "badges we ain't got no badges we don't need no badges i don't have to show you any stinkin
```

Report your **R command(s)**.

d) Now use `strsplit()`, with `split = " "`, to split the `badges` quote into individual words, overwriting the previous version of `badges`. You should end up with this:

```
badges

## [[1]]
##  [1] "badges"   "we"       "ain't"    "got"      "no"       "badges"
##  [7] "we"       "don't"    "need"     "no"       "badges"   "i"
## [13] "don't"    "have"     "to"       "show"     "you"      "any"
## [19] "stinking" "badges"
```

Report your **R command**.

e) Note that `strsplit()` returned a *list* with one element, which is a `"character"` vector of words from the quote. Extract the vector from the list, for example by typing:

```
badges <- badges[[1]]    # You could also type unlist(badges)
```

You should now have this `"character"` vector:

```
badges
```

```
##  [1] "badges"   "we"       "ain't"    "got"      "no"       "badges"
##  [7] "we"       "don't"    "need"     "no"       "badges"   "i"
## [13] "don't"    "have"     "to"       "show"     "you"      "any"
## [19] "stinking" "badges"
```

Now use `grep()`, with `pattern = "badges"`, to find the instances of the word `"badges"` in the quote. Then do the same thing, but using `grepl()`. Report your **two R commands**.

f) Now use `nchar()` to count the number of letters (characters actually) in each word. Report your **R command**.

Exercise 5 Here's a quote from MLK's "I have a dream" speech:

```
quote <- "I have a dream that one day this nation will rise up and live out
the true meaning of its creed, 'We hold these truths to be self-evident,
that all men are created equal.' I have a dream that one day on the
red hills of Georgia, sons of former slaves and the sons of former
slave owners will be able to sit down together at the table of
brotherhood. I have a dream that one day even the state of Mississippi,
a state sweltering with the heat of injustice, sweltering with the
heat of oppression, will be transformed into an oasis of freedom and
justice. I have a dream that my four little children will one day live
in a nation where they will not be judged by the color of their skin
but by the content of their character."
```

After creating the character string `quote` above, inspect it for embedded *newline* characters `"\n"`, and if there are any, remove them, for example by typing:

```
quote <- gsub(pattern = "\n", replacement = " ", x = quote)
```

Then remove periods(`.`), commas (`,`), and quotation marks (`'`) using the *regular expression* `[\\.,']` (Section 19.2 gives an explanation):

```
quote <- gsub(pattern = "[\\.,']", replacement = "", x = quote)

quote
```

a) Now use `strsplit()`, with `split = " "`, to split the `quote` into individual words, overwriting the previous version of `quote`. You should end up with this:

```
quote

## [[1]]
##   [1] "I"            "have"         "a"            "dream"
##   [5] "that"         "one"          "day"          "this"
##   [9] "nation"       "will"         "rise"         "up"
##  [13] "and"          "live"         "out"          "the"
##  [17] "true"         "meaning"      "of"           "its"
##  [21] "creed"        "We"           "hold"         "these"
##  [25] "truths"       "to"           "be"           "self-evident"
##  [29] "that"         "all"          "men"          "are"
##  [33] "created"      "equal"        "I"            "have"
##  [37] "a"            "dream"        "that"         "one"
##  [41] "day"          "on"           "the"          "red"
##  [45] "hills"        "of"           "Georgia"      "sons"
##  [49] "of"           "former"       "slaves"       "and"
##  [53] "the"          "sons"         "of"           "former"
##  [57] "slave"        "owners"       "will"         "be"
##  [61] "able"         "to"           "sit"          "down"
##  [65] "together"     "at"           "the"          "table"
##  [69] "of"           "brotherhood"  "I"            "have"
##  [73] "a"            "dream"        "that"         "one"
##  [77] "day"          "even"         "the"          "state"
##  [81] "of"           "Mississippi"  "a"            "state"
##  [85] "sweltering"   "with"         "the"          "heat"
##  [89] "of"           "injustice"    "sweltering"   "with"
##  [93] "the"          "heat"         "of"           "oppression"
##  [97] "will"         "be"           "transformed"  "into"
## [101] "an"           "oasis"        "of"           "freedom"
## [105] "and"          "justice"      "I"            "have"
## [109] "a"            "dream"        "that"         "my"
## [113] "four"         "little"       "children"     "will"
## [117] "one"          "day"          "live"         "in"
## [121] "a"            "nation"       "where"        "they"
## [125] "will"         "not"          "be"           "judged"
## [129] "by"           "the"          "color"        "of"
## [133] "their"        "skin"         "but"          "by"
## [137] "the"          "content"      "of"           "their"
## [141] "character"
```

Report your **R command**.

b) Note that `strsplit()` returned a *list* with one element, which is a `"character"` vector of words from the quote. Extract the vector from the list, for example by typing:

```
quote <- quote[[1]]    # You could also type unlist(quote)
```

You should now have this 141-element (use `length(quote)` `"character"` vector:

```
head(quote)

## [1] "I"     "have"  "a"     "dream" "that"  "one"
```

Now use `nchar()` to obtain a vector, `my.nchars`, say, containing **character counts** for in the words of `quote`. Report your **R command(s)**.

c) Use `mean()` to determine the **mean** number of characters of the words in `quote`. Report the **value** of the **mean**.

d) Make a histogram of the numbers of characters of the words in `quote`:

```
ggplot(data = data.frame(n = my.nchars)) +
  geom_histogram(mapping = aes(x = n), fill = "blue", binwidth = 1)
```

Describe the **shape** of the histogram (**right skewed**, **left skewed**, or **symmetrical** and **bell-shaped**).

19.2 Regular Expressions

- ***Regular expressions*** are sequences of characters used to search for and replace character patterns in text.

  Consider again the tongue twister:

```
twister <- "Peter Piper picked a peck of pickled peppers"
```

  The character sequence `"pick"` is a **regular expression** that could be searched for in `twister`.

- The fundamental building blocks of **regular expressions** are single characters, including *letters* and *digits*, that *match themselves*. These are called ***literal characters***.

  For example, each of the letters `"p"`, `"i"`, `"c"`, and `"k"` is a **literal character**.

- Literal characters can be combined with some **"wildcard" characters** called ***metacharacters*** that, unless preceded by a backslash, have special meaning.

  For example, a period `"."` is a **metacharacter** that *matches any character*.

  Thus both `"pick"` and `"p.ck"` are **regular expressions**, but the first one matches only the exact pattern `"pick"`, whereas the second one matches `"pick"`, `"pack"`, `"peck"`, etc.

  Recall that `gregexpr()` returns the starting character positions of all matches of a `pattern`. Thus, whereas specifying `pattern = "pick"` only identifies *two* instances, specifying `pattern = "p.ck"` identifies *three* (the two `"pick"`s plus the `"peck"`):

```
gregexpr(pattern = "pick", text = twister)

## [[1]]
## [1] 13 30
## attr(,"match.length")
## [1] 4 4
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE

gregexpr(pattern = "p.ck", text = twister)

## [[1]]
## [1] 13 22 30
## attr(,"match.length")
## [1] 4 4 4
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

- In addition to **literal characters** and **metacharacters**, **regular expressions** can also contain:
  - ***Character sets*** using square brackets `[ ]`.
  - ***Character alternatives*** using the `|` operator wrapped in parentheses `( )`.
  - ***Anchors*** using the `^` operator to **anchor** a pattern to the beginning of a piece of text, and `$` to **anchor** it to the end.
- Some of R's **metacharacters** and **character set**, **character alternative**, and **anchor** operators are below.

```
.          # Used to match any character (except a newline character
           # "\n")
|          # Used to match either of alternative characters (e.g.
           # "(a|b)c" matches "ac" and "bc")
[ ]        # Used to match any of several characters (e.g. "[ab]"
           # matches both "a" and "b")
^          # Used to anchor a pattern toe the beginning of a piece of
           # text, e.g. "^ab" matches "ab" in "absolute" but not "ab"
           # in "fabulous" nor in "prefab"
$          # Used to anchor a pattern toe the end of a piece of
           # text, e.g. "ab$" matches "ab" in "prefab" but not "ab"
           # in "fabulous" nor in "absolute"
[^ ]       # Used to negate one or more characters (e.g. "[^ab]"
           # matches any character except "a" and "b")
\s         # Used to match (a single) white space (use "\\s")
[0-9]      # Used to match any digit (use "[0-9]")
[A-Z]      # Used to match any upper case letter (use "[A-Z]")
[a-z]      # Used to match any lower case letter (use "[a-z]")
[:alpha:]  # Used to match any alphabetic character (use "[[:alpha:]]")
[:digit:]  # Used to match any single digit (use "[[:digit:]]")
[:blank:]  # Blank characters (space and tab) (use "[[:blank:]]"),
[:space:]  # Space characters (including not only space and tab, but
           # also newline and some others) (use "[[:space:]]")
[:punct:]  # Used to match any punctuation symbol (e.g. ! " # $ % & '
           # ( ) *, + , - . / : ; < = > ? @ [  ] ^ _ `  |  ~) (use
           # "[[:punct:]]")
*          # Repetition quantifier: The preceding character or sub-
           # pattern appears 0 or more times (e.g. "(ab)*" matches any
           # single character as well as the patterns "ab",  "abab",
           # "ababab", etc.)
+          # Repetition quantifier: The preceding character or sub-
           # pattern appears 1 or more times (e.g. "(ab)+" matches the
           # patterns "ab", "abab", "ababab", etc.)
?          # Preceding character or subpattern appears 0 or 1 time
           # (e.g. "(ab)?" matches any single character as well as the
           # pattern "ab")
{n}        # Preceding character or subpattern appears exactly n times
           # (e.g. "b{3}" matches the pattern "bbb")
{m,n}      # Preceding character or subpattern appears between m and n
           # times, inclusive (e.g. "b{2,4}" matches the patterns "bb",
           # "bbb", and "bbbb").  Note that there's no space after the
           # comma.
```

For the full list, type:

```
? regex
```

Any of these can be used in the `pattern` passed to `grep()`, `grepl()`, `regexpr()`, `gregexpr()`, `sub()`, `gsub()`, and `strsplit()`.

- For example, consider (again) the vector:

```
pets <- c("dog", "cat", "gerbil", "hamster", "parakeet", "goldfish", "iguana")
```

To use `grep()` to search for any `pet` that includes the letter `"g"` or the letter `"h"`, type:

```
grep(pattern = "[gh]", x = pets, value = TRUE)

## [1] "dog"      "gerbil"   "hamster"  "goldfish" "iguana"
```

- As another example using:

```
twister

## [1] "Peter Piper picked a peck of pickled peppers"
```

to search for blank spaces, type:

```
gregexpr(pattern = "\\s", text = twister)

## [[1]]
## [1]  6 12 19 21 26 29 37
## attr(,"match.length")
## [1] 1 1 1 1 1 1 1
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

- To search our text for a **character** that's *also* a **metacharacter** (e.g. to search for a period "."), which is a **metacharacter**), we need to ***escape*** the **metacharacter status** using the **backslash operator**:

```
\        # Used to escape a metacharacter (e.g. "\." matches a period)
```

- For example, to search for the period (i.e. the symbol ".") in:

```
my.date <- "Jan. 27, 2014"
```

it **doesn't work** to type:

```
regexpr(pattern = ".", text = my.date)
```

Instead, we have to type:

```
regexpr(pattern = "\\.", text = my.date)

## [1] 4
## attr(,"match.length")
## [1] 1
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

The reason why we had to type **two** backslashes, i.e. `"\\."`, is that it turns out that the **escape character**, `"\"`, *is itself* a **metacharacter** that needs to be **escaped**.

- Note that setting `fixed = TRUE` in `grep()`, `regexpr()`, `gsub()`, etc. applies *literal* pattern matching rather than treating the contents of `pattern` as metacharacters.

Section 19.2 Exercises

Exercise 6 Here's a quote from MLK's "I have a dream" speech:

```
quote <- "I have a dream that one day this nation will rise up and live out
the true meaning of its creed, 'We hold these truths to be self-evident,
that all men are created equal.' I have a dream that one day on the
red hills of Georgia, sons of former slaves and the sons of former
slave owners will be able to sit down together at the table of
brotherhood. I have a dream that one day even the state of Mississippi,
a state sweltering with the heat of injustice, sweltering with the
heat of oppression, will be transformed into an oasis of freedom and
justice. I have a dream that my four little children will one day live
in a nation where they will not be judged by the color of their skin
but by the content of their character."
```

After creating the character string `quote` above, inspect it for embedded *newline* characters `"\n"`, and if there are any, remove them, for example by typing:

```
quote <- gsub(pattern = "\n", replacement = " ", x = quote)
```

Then remove periods(`.`), commas (`,`), and quotation marks (`'`) using the *regular expression* `[\\.,']`:

```
quote <- gsub(pattern = "[\\.,']", replacement = "", x = quote)

quote
```

a) `gregexpr()` takes arguments `pattern`, a character pattern (or *regular expression*), and `text`, a `"character"` string, and returns the starting character positions of *all* matches of `pattern`.

Use `gregexpr()` to determine the starting character position(s) of the word "freedom" in the quote. Report your **R command(s)**.

b) `strsplit()` splits the elements of a `"character"` string into substrings according to matches of a character pattern (or *regular expression*) specified via the argument `split`, and returns a *list* with one element, a `"character"` vector of the substrings.

Run the following commands, and report the results:

```
quote.list <- strsplit(x = quote, split = " ")

quote.vec <- unlist(quote.list)   # Could also use quote.vec <- quote.list[[1]]

quote.vec
```

c) `grep()` takes arguments `pattern`, a character pattern (or *regular expression*), and `x`, a `"character"` vector, and returns the indices of the elements of `x` that contain the `pattern`.

Explain why the returned values of the following two commands differ:

```
grep(pattern = "the", x = quote.vec)

##  [1]  16  24  43  53  65  67  70  79  87  93 124 130 133 137 140

which(quote.vec == "the")

## [1]  16  43  53  67  79  87  93 130 137
```

d) If we specify `value = TRUE` in `grep()`, it returns the *actual elements* of `x` (not their indices) that contain the `pattern`.

Explain in words what each of the following commands does:

```
grep(pattern = "ing$", x = quote.vec, value = TRUE)
```

```
grep(pattern = "^th", x = quote.vec, value = TRUE)
```

Exercise 7 Here are three data sets:

County Center

County	State	Lat	Long
De Witt County	IL	40169623	-88904690
Lac qui Parle County	MN	45000955	-96175301
Lewis and Clark County	MT	47113693	-112377040
St John the Baptist Parish	LA	30118238	-90501892

Census

County	X1	X2	X3	X4
De Witt County	16	798	97.8	0.5
Lac qui Parle County	8	067	98.8	0.2
Lewis and Clark County	55	716	95.2	0.2
St. John the Baptist Parish	43	044	52.6	44.8

← Note the period in "St."

Election Results

County	Res1	Res2	Res3	Res4	Res5	Res6	Res7
De Witt	23	23	4	920	2	836	0
Lac Qui Parle	31	31	2	093	2	390	36
Lewis & Clark	54	54	16	432	12	655	386
St. John the Baptist	35	35	9	039	10	305	74

← Note "County" and "Parish" missing in County names, upper case "Q" and "&" in names, and period in "St."

If we wanted to use `left_join()` to **combine** these data sets **by County**, we'd need to **correct** the **discrepancies** first.

Here are the three `"character"` vectors:

```
County1 <- c("De Witt County", "Lac qui Parle County", "Lewis and Clark County",
             "St John the Baptist Parish")

County2 <- c("De Witt County", "Lac qui Parle County", "Lewis and Clark County",
             "St. John the Baptist Parish")

County3 <- c("De Witt", "Lac Qui Parle", "Lewis & Clark", "St. John the Baptist")
```

a) Find and eliminate the word **"County"** from the `County1` and `County2` vectors. Consider using `sub()` or `gsub()`. Report your **R command(s)**.

b) Now find and eliminate the word **"Parish"** from the `County1` and `County2` vectors. Use `sub()` or `gsub()`. Report your **R command(s)**.

c) The words **"County"** and **"Parish"** both have *six* letters. So another way to eliminate those words from `County1` and `County2` would be to use `nchar()` to find the length of each character string and then `substr()` to keep all but the last 6 characters (or 7?).

Verify that the following commands are another way to accomplish the tasks of parts *a* and *b*:

```
substr(x = County1, start = 1, stop = nchar(County1) - 7)

substr(x = County2, start = 1, stop = nchar(County2) - 7)
```

d) Another way to eliminate **"County"** and **"Parish"** words from `County1` and `County2` would be to use the "or" operator `|` in a **regular expression**.

Verify that the following commands are another way to accomplish the tasks of parts *a* and *b*:

```
gsub(pattern = " County| Parish", replacement = "", x = County1)

gsub(pattern = " County| Parish", replacement = "", x = County2)
```

e) In part *d*, there was a blank before `County` and `Parish`. Why *wouldn't* the following be appropriate?

```
gsub(pattern = " County|Parish", replacement = "", x = County1)

gsub(pattern = " County|Parish", replacement = "", x = County2)
```

f) Next we want to remove the **period** symbol in **"St."** from `County2` and from `County3`. The regular expression `"."` matches *any* character.

Why *wouldn't* the following be appropriate? Try it.

```
gsub(pattern = ".", replacement = "", x = County2)

gsub(pattern = ".", replacement = "", x = County3)
```

g) Write a command involving a **regular expression** with the **escape character** (\) to remove the **period** symbol in **"St."** from `County2` and from `County3`. Report your **R command(s)**.

h) Setting `fixed = TRUE` in `gsub()` applies *literal* pattern matching rather than treating the contents of `pattern` as metacharacters.

Instead of using a **regular expression** with the **escape character** (\) to remove the **period** symbol in **"St."** from `County1` and from `County2`, as in part *g*, rewrite the command from part *f* using `fixed = TRUE` in `gsub()` to accomplish the task. Report your **R command**.

Exercise 8 Be careful with **regular expressions** `"."` and `"*"`.

`"."` matches *any* character, and `"*"` says the preceding character appears *any* number of times. Thus, `".*"` would match *any* character *any* number of times, and so the *entire string* would be a match.

Below, `gregexpr()` only returns the starting position of `"<p>"`, but *not* the starting position of `"</p>"`:

```
my.string <- "<p> This is a short paragraph.</p>"

gregexpr(pattern = "<.*>", text = my.string)

## [[1]]
## [1] 1
## attr(,"match.length")
## [1] 34
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

What does the **regular expression** `"<.*>"` match in `my.string` above?

19.3 Corpora

- **Text mining** is often performed not just on *one* text document, but on a collection of *many* text documents, called a ***corpus***.

  For example, the collection of *all* presidential *State of the Union Addresses* is a **corpus**.

- A ***corpus*** can be represented in R as a `"character"` *vector*, each element of which is a (long) `"character"` *string* containing one entire text document.

- We'll create a **corpus** of the presidential *State of the Union Addresses* using the `"tm"` (text mining) package:

```
help(package = "tm")
```

  These functions (from `"tm"`) will be useful for converting a **corpus** from one class of objects to another, which in turn is useful for performing different operations on them:

```
VectorSource()  # Convert a corpus, represented as a "character" vector, to an
                # object of class "VectorSource", for example so it can be
                # passed to VCorpus().
VCorpus()       # Can be used to convert an object of class "VectorSource" to
                # one of class "VCorpus", for example so it can be passed to
                # tm_map().
```

  This function (from `"tm"`) will be useful for applying a text-mining function separately to each text document in a **corpus**:

```
tm_map()    # Apply a transformation function (also called mapping) FUN to
            # each document in a corpus (i.e. to an object of class "VCorpus").
            # Returns a corpus consisting of FUN applied to each document in x.
```

  These text-mining functions (from `"tm"`) can be applied separately to each text document in a **corpus** using `tm_map()`:

```
stripWhitespace()       # Used to remove extra white space from a text document.
removeNumbers()         # Used to remove all numbers from a text document.
removePunctuation()     # Used to remove all punctuation marks from a text
                        # document.
removeWords()           # Used to remove specific words from a text document.
content_transformer()   # Create a function which modifies the content of an
                        # R object (rather than just returning a modified copy
                        # of the object).
stopwords()             # Used to identify stopwords in a given language, that
                        # is, words that are so common they should be filtered
                        # out before a text document is analyzed.  Can be used
                        # with removeWords().
```

- For example, the file **state_of_the_union.txt** contains all **226** presidential *State of the Union Addresses*, demarcated by the pattern ***. We can use `scan()` to read it in:

```
# This creates "character" vector whose elements are words from all speeches.
sotu.wrd.vec <- scan('~/grevstad/mth3270/data_mth3270/state_of_the_union.txt',
                     what = "",
                     blank.lines.skip = TRUE)
```

  Above, specifying `what = ""` indicates the type of data to be read in is `"character"` (text). Specifying `blank.lines.skip = TRUE` is needed because **state_of_the_union.txt** contains many blank lines.

The object `sotu.wrd.vec` is a `"character"` vector, each element of which is a *word* from one of the speeches:

```
length(sotu.wrd.vec)

## [1] 1628505
```

But for the **corpus**, we need a `"character"` vector, each element of which is an entire *speech*. To get this, we'll combine the *words* together into a single `"character"` string (one-element vector), then split it up into separate speeches.

```
# This creates a single "character" string (one-element vector) containing all speeches.
sotu.string <- paste(sotu.wrd.vec, collapse = " ")

length(sotu.string)

## [1] 1
```

To split `sotu.string` into separate speeches (demarcated by the pattern ***), we use `strsplit()`:

```
# This splits sotu.string into separate speeches (demarcated by ***):
sotu.list <- strsplit(sotu.string, split = "\\*\\*\\*")
```

`sotu.list` is a one-element *list* whose (one) element is a `"character"` vector, each element of which is an entire *speech*. Below, we convert the one-element *list* to the `"character"` vector itself:

```
# This converts the one-element sotu.list to a vector:
sotu.spch.vec <- unlist(sotu.list)    # Could also use sotu.spch.vec <- sotu.list[[1]]
```

The first element of `sotu.spch.vec` is an empty `"character"` (due to *** preceding the first speech in **state_of_the_union.txt**), so it needs to be removed:

```
sotu.spch.vec[1]

## [1] ""

# This removes the empty first element:
sotu.spch.vec <- sotu.spch.vec[-1]
```

Now each element of the `"character"` vector is an entire *speech*:

```
length(sotu.spch.vec)

## [1] 226
```

Functions in the `"tm"` package need **corpora** to belong to the `"Corpus"` class of objects, so we'll convert it to that class using a two-step process:

```
library(tm)

# Create a "VectorSource" class object:
sotu.vecsrc <- VectorSource(sotu.spch.vec)

# Convert to a "Corpus" class object:
sotu.corp <- VCorpus(sotu.vecsrc)
```

Before analyzing the speeches, we need to clean them using functions from the `"tm` package:

```
library(dplyr)       # For the pipe operator %>%
```

```
sotu.corp <- sotu.corp %>%
  tm_map(FUN = stripWhitespace) %>%
  tm_map(FUN = removeNumbers) %>%
  tm_map(FUN = removePunctuation) %>%
  tm_map(FUN = content_transformer(tolower)) %>%
  tm_map(FUN = removeWords, stopwords("english"))
```

19.4 Word Clouds

- Now we're ready to begin analyzing the *State of the Union Address* speeches in the `sotu.corp` `corpus`.

  Here's a ***word cloud*** (using the `wordcloud()` function from the `"wordcloud"` package):

```
library(wordcloud)

set.seed(45)   # For reproducing the same word cloud later.

wordcloud(sotu.corp, max.words = 20, scale = c(3.5, 0.3),
          colors = topo.colors(n = 20), random.color = TRUE)
```

  Above, the `scale` argument specifies the sizes of the largest and smallest words in the **word cloud**.

19.5 Document Term Matrices

- We'll use the following functions, from the `"tm"` package, to form and examine a ***document term matrix***, each *row* of which is one of the **226** speeches (**documents**), each *column* of which is one of the **27,591** words (**terms**) contained in the speeches, and whose *elements* are term **frequencies** (counts).

```
DocumentTermMatrix()   # Form a document term matrix from an object of class "VCorpus".
findFreqTerms()        # Find frequent terms in a document-term matrix.
findAssocs()           # Find associations in a document-term matrix.
```

- Here's how to form the ***document term matrix***:

```
dtm <- DocumentTermMatrix(sotu.corp)

dtm
```

```
## <<DocumentTermMatrix (documents: 226, terms: 27591)>>
## Non-/sparse entries: 354346/5881220
## Sparsity           : 94%
## Maximal term length: 27
## Weighting          : term frequency (tf)

dim(dtm)

## [1]   226 27591
```

The `dtm` matrix is sparse – 94% of the entries are 0. This makes sense, since most words do not appear in most speeches.

Here are the words that were used **1,500** or more times:

```
findFreqTerms(dtm, lowfreq = 1500)

##  [1] "act"        "also"       "american"   "can"        "citizens"
##  [6] "congress"   "country"    "every"      "foreign"    "general"
## [11] "government" "great"      "last"       "law"        "made"
## [16] "make"       "many"       "may"        "must"       "nation"
## [21] "national"   "nations"    "new"        "now"        "one"
## [26] "part"       "peace"      "people"     "power"      "present"
## [31] "public"     "service"    "shall"      "state"      "states"
## [36] "system"     "time"       "united"     "upon"       "war"
## [41] "will"       "without"    "work"       "world"      "year"
## [46] "years"
```

By default, `DocumentTermMatrix()` reports term **frequencies**. Alternatively, it will also report ***term frequency-inverse document frequency***, or ***tf-idf*** values, which measure how unique a word is to a particular speech. See the textbook.

Since the `dtm` matrix contains all of the speech-specific **frequencies** for each word, the total frequency for a word across all speeches is a column sum of the `dtm` matrix:

```
dtm %>%
  as.matrix() %>%
  apply(MARGIN = 2, sum) %>%
  sort(decreasing = TRUE) %>%
  head(n = 9)

##       will government     states   congress     united        can
##       9158       6518       6272       4849       4651       4226
##     people       upon       year
##       3820       3744       3533
```

We can determine which words (terms) tend to appear together in the same speeches (documents):

```
findAssocs(dtm, terms = "war", corlimit = 0.7)

## $war
##  recommended expenditures  nazifascist unliquidated  substandard
##         0.73         0.72         0.72         0.72         0.70
##      wartime
##         0.70

findAssocs(dtm, terms = "united", corlimit = 0.7)
```

```
## $united
##      states  government        part         act    december        thus
##        0.92        0.81        0.77        0.76        0.75        0.75
##    congress        july        made   secretary     several    existing
##        0.74        0.74        0.74        0.74        0.74        0.73
##   relations      amount     foreign   interests       state   territory
##        0.73        0.72        0.72        0.72        0.72        0.72
##       early governments         may    republic    required      treaty
##        0.71        0.71        0.71        0.71        0.71        0.71
##         two        upon        view         due      report
##        0.71        0.71        0.71        0.70        0.70
```

The word `"war"` tends to appear in the same speeches as the words `"recommended"`, `"nazifascist"`, ... `"wartime"`.

Not surprisingly, the word `"united"` tends to appear in the same speeches as the word `"states"`, among others.

The values reported by `findAssocs()` are **correlations** between the **frequencies** in the two words' columns of the `dtm` matrix. Two words that have a **high correlation** have a **high tendency** to occur **together** in the **same speeches**.

Section 19.5 Exercises

Exercise 9 The file **state_of_the_union.txt** contains all **226** presidential *State of the Union Addresses*, demarcated by the pattern ***. After saving the file, use `scan()` to read it in and produce a `"character"` vector, each element of which is a word:

```
# This creates "character" vector whose elements are words from all speeches.
sotu.wrd.vec <- scan('~/grevstad/mth3270/data_mth3270/state_of_the_union.txt',
                     what = "",
                     blank.lines.skip = TRUE)
```

Now produce a `"character"` vector, each element of which is a speech:

```
# This creates a single "character" string (one-element vector) contain-
ing all speeches.
sotu.string <- paste(sotu.wrd.vec, collapse = " ")

# This splits sotu.string into separate speeches (demarcated by ***):
sotu.list <- strsplit(sotu.string, split = "\\*\\*\\*")

# This converts the one-element sotu.list to a vector:
sotu.spch.vec <- unlist(sotu.list)    # Could also use sotu.spch.vec <- sotu.list[[1]]

# This removes the empty first element:
sotu.spch.vec <- sotu.spch.vec[-1]
```

Next, convert the `"character"` vector to a `"Corpus"` class object:

```
library(tm)

# Create a "VectorSource" class object:
sotu.vecsrc <- VectorSource(sotu.spch.vec)

# Convert to a "Corpus" class object:
sotu.corp <- VCorpus(sotu.vecsrc)
```

Now clean up the speeches:

```r
# This cleans the speeches:
sotu.corp <- sotu.corp %>%
  tm_map(FUN = stripWhitespace) %>%
  tm_map(FUN = removeNumbers) %>%
  tm_map(FUN = removePunctuation) %>%
  tm_map(FUN = content_transformer(tolower)) %>%
  tm_map(FUN = removeWords, stopwords("english"))
```

Finally, create the **document term matrix**:

```r
dtm <- DocumentTermMatrix(sotu.corp)
```

and use it to answer the following questions.

a) Which words were used **3,000** or more times in the speeches?

b) With which **five** words does the word `"peace"` tend to appear most often in the speeches (i.e. which five words' frequencies are most **correlated** with `"peace"`'s frequencies?). What are the values of their **correlations** with `"peace"`?