Text Data

Four Examples

1. Election Study

- Geographic Data – longitude and latitude of the *county* center
- Population Data from the census for each *county*
- Election results from 2008 for each *county* (scraped from a Website)

Want to match/merge the information from these three different sources

What issues arise in matching?

```
"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
```

```
"St. John the Baptist Parish","43,044","52.6","44.8",...
"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", ...
```

```
DeWitt  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
```

What problems need resolving to match counties across sources?

- Capitalization: qui vs Qui
- County/Parish missing in one file
- Periods: St. vs St
- & vs and: Lewis and Clark vs Lewis & Clark
- Blanks: DeWitt vs De Witt

2. Text mining
State of Union Addresses

- How long are the speeches?
- How do the distributions of certain words change over time?
- Which presidents have given “similar” speeches?

***

State of the Union Address
George Washington
December 8, 1790

Fellow-Citizens of the Senate and House of Representatives:

In meeting you again I feel much satisfaction in being able to repeat my congratulations on the favorable prospects which continue to distinguish our public affairs. The abundant fruits of another year have blessed our country with plenty and with the means of a flourishing commerce.

Text mining State of Union Addresses

- All speeches in one large plain text file
- Each speech starts with "***" on a line followed by 3 lines of information about who gave the speech and when
- One way to mine the speeches: create a *word vector* for each speech, which holds counts of how many times a particular word was said in each speech.
- Words such as nation, national, nations should collapse to the same "word"

3. Web behavior

Every time you visit a Web site, information is recorded about the visit:

- the page visited,
- date and time of visit
- browser used
- operating system
- IP address

Two lines of the Web log

169.237.46.168 - - [26/Jan/2014:10:47:58 -0800]
"GET /stat141/Winter04 HTTP/1.1" 301 328
"http://anson.ucdavis.edu/courses/"
"Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR 1.1.4322)”

169.237.46.168 - - [26/Jan/2014:10:47:58 -0800]
"GET /stat141/Winter04/ HTTP/1.1" 200 2585
"http://anson.ucdavis.edu/courses/"
"Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR 1.1.4322)"

- The information in the log has a lot of structure, for example the date always appears in square brackets.
- However, the information is not consistently separated by the same characters, as in a csv file,
- nor is it placed consistently in the same columns in the file, as in a fwf file.

4. Spam filtering: Anatomy of email message

- Three parts:
  - header,
  - body,
  - attachments (optional).
- Like regular mail, the header is the envelope and the body is the letter.
- Plain text

Header:

- date, sender, and subject
- message id,
- who are the carbon-copy recipients,
- return path.

- SYNTAX – *KEY:VALUE*

Example header

```
Date: Mon, 2 Feb 2015 22:16:19 -0800 (PST)
From: nolan@stat.Berkeley.EDU
X-X-Sender: nolan@kestrel.Berkeley.EDU
To: Txxxx Uxxx <txxxx@uclink.berkeley.edu>
Subject: Re: prof: did you receive my hw?
In-Reply-To: <web-569552@calmail-st.berkeley.edu>
Message-ID: <Pine.SOL.4.50.0402022216120.2296-100000@kestrel.Berkeley.EDU>
References: <web-569552@calmail-st.berkeley.edu>
MIME-Version: 1.0
Content-Type: TEXT/PLAIN; charset=US-ASCII
Status: O
X-Status:
X-Keywords:
X-UID: 9079
```

What features can you derive from the email that might be helpful in predicting spam?

- Sent in the early morning:
- Has an Re: in the subject line
- Funny words like "rep1!c@ted" and "v!@gra”
- Lots of YELLING IN THE EMAIL

Four Examples

- Election data – *clean text* for analysis
- Web log – *locate and extract variables* based on patterns in the text
- State of the Union address – Speech represented by a *vector of word counts* (text mining)
- Spam – *derive features* from text for analysis and use in prediction problem

Election Data

Simple String Manipulation Functions can help us clean our data

Recall:

```
"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
```

```
"St. John the Baptist Parish","43,044","52.6","44.8",...
"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", ...
```

```
DeWitt  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
```

5 String manipulation functions

- `substring(text, first, last)` – extract a portion of a character sting from text, beginning at `first`, ending at `last`
- `nchar(text)` – return the number of characters in a string
- `strsplit(x, split)` – split the string into pieces using `split` to divide it

  `strsplit(x, "")` – splits into single characters

5 String manipulation functions

- `paste(x, y, z, …, sep = " ", collapse = NULL)` – paste together character strings separated by one blank (value of `sep`)
- `tolower(x)` `toupper(x)` - convert upper-case characters to lower-case, or vice versa. Non-alphabetic characters are left unchanged

Test Data

```
> cNames
```

[1] "Dewitt County"

[2] "Lac qui Parle County"

[3] "St. John the Baptist Parish"

[4] "Stone County”

[5] "Lewis & Clark County"

Practice with

`tolower()`, `paste()`, `substring()`, `nchar()`, `strsplit()`

Web log data

With simple string manipulation we can extract the day, month, and year

The Web log

169.237.46.168 - - [26/Jan/2014:10:47:58 -0800]

"GET /stat141/Winter04 HTTP/1.1" 301 328

"http://anson.ucdavis.edu/courses/"

"Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR 1.1.4322)”

- How to extract the day of month, month, and year from the log entry?

What features of the entry are useful?

- Date is between [ ]
- Day, month, year are separated by /
- Year is separated from time by :

How do we use these to extract the information that we want?

```
strsplit(wl1, split = " ")
[[1]]
 [1] "169.237.46.168"
 [2] "-"
 [3] "-"
 [4] "[26/Jan/2014:10:47:58"
 [5] "-0800]"
 [6] "\"GET”
…
```

Notice a list is returned

Will the date always be in the 4th position?

```
xtractDate = strsplit(wl1, split = " ")[[1]][4]

xtractDate1 = substr(xtractDate, start = 2,
                     stop = nchar(xtractDate))
xtractDate1
[1] "26/Jan/2014:10:47:58"
```

Can we instead use
`nchar(xtractDate) – 9`
for `stop` value

A.Yes
B.No

```
xtractDate2 = strsplit(xtractDate1, split = "/")

day = xtractDate2[[1]][1]
mon = xtractDate2[[1]][2]
yr = strsplit(xtractDate2[[1]][3], ":")[[1]][1]
```

Better Way

```
v = strsplit(wl1,
                "\\[|/|:")[[1]][2:4]
```

WHAT IS THIS????

```
v
[1] "26"   "Jan"  "2014"
```

"\\[|/|:"

- Read | as “or”
- Split on and one of these 3: [OR / OR :
- The [is a special character that has a special meaning so \\[means split on the literal left bracket; this is not as the special character

"\\[|/|:“ is a Regular Expression

These are very powerful and can simplify our work with strings

Regular Expressions

Recall our County Names

```
cNames
[1] "Dewitt County"
[2] "Lac qui Parle County"
[3] "St. John the Baptist Parish"
[4] "Stone County"
[5] "Lewis & Clark County"
```

Tasks

- Change & to and
- Remove County and Parish
- Remove .
- Convert all names to lower case

sub(pattern, replacement, x)

```
> sub("&", "and", cNames)
[1] "Dewitt County”
[2] "Lac qui Parle County"
[3] "St. John the Baptist Parish"
[4] "Stone County"
[5] "Lewis and Clark County"
```

If we want to change all &s, not just the first we use `gsub`

The g stands for global substitution

Eliminate "County"

```
> sub(" County", "", cNames)
[1] "Dewitt"
[2] "Lac qui Parle"
[3] "St. John the Baptist Parish"
[4] "Stone"
[5] "Lewis & Clark"
```

Subtlety to Pattern Matching

How does the pattern matching work? Pattern: Cat

- Look at each character one at a time starting at the left.
- When you find a "C", then look at the next character and check to see if it is "a".
- If it is, then look at the next character and check to see if it is "t". If it is, then we have a match!
- At any point if a literal does not match, back up and continue the search for "C" at the character immediately following the first character matched (i.e., the first "C")

```
sub("Parish","", cNames[3])
```

Parish	1	2	3	4	5	6	7	8	9		1	2	3	4	5	6	7	8	9		1	2	3	4	5	6
	S	t		J	o	h	n		t	h	e		B	a	p	t	i	s	t		P	a	r	i	s	h
Find P	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✔					

Search proceeds one literal at a time.

Begin with a search for “P”

```
sub("Parish","", cNames[3])
```

Parish	1	2	3	4	5	6	7	8	9		1	2	3	4	5	6	7	8	9		1	2	3	4	5	6
	S	t		J	o	h	n		t	h	e		B	a	p	t	i	s	t		P	a	r	i	s	h
Find P	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✔					
Followed by a																						✔				
Followed by r																										

When find the first literal, look for the second immediately following it.

```
sub("Parish","", cNames[3])
```

Parish	1	2	3	4	5	6	7	8	9		1	2	3	4	5	6	7	8	9		1	2	3	4	5	6
	S	t		J	o	h	n		t	h	e		B	a	p	t	i	s	t		P	a	r	i	s	h
Find P	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✔					
Followed by a																						✔				
Followed by r																							✔			
Followed by i																								✔		
Followed by s																									✔	
Followed by h																										✔
Match start 21 length 6																										

Continue matching the 3rd,4th, 5th, and 6th literals, one after another.

Let’s Try Another String

`sub("Parish","", cNames[2])`

Parish	1	2	3	4	5	6	7	8	9		1	2	3	4	5	6	7	8	9	
	L	a	c		q	u	i		P	a	r	l	e		C	o	u	n	t	y
Find P	✗	✗	✗	✗	✗	✗	✗	✗	✔											
Followed by a										✔										
Followed by r											✔									
Followed by i												✗								

Search proceeds one literal at a time.

Begin with a search for “P”.

When find “P”, continue with match for the 2nd literal and so on.

```
sub("Parish","", cNames[2])
```

Parish	1	2	3	4	5	6	7	8	9		1	2	3	4	5	6	7	8	9	
	L	a	c		q	u	i		P	a	r	l	e		C	o	u	n	t	y
Find P	✗	✗	✗	✗	✗	✗	✗	✗	✔											
Followed by a										✔										
Followed by r											✔									
Followed by i												✗								
Back up & Resume										⊙										

When we get a mismatch,
back up and resume the search for “P”
at the literal immediately following
our earlier first match.

sub("Parish","", cNames[2])

Parish	1	2	3	4	5	6	7	8	9		1	2	3	4	5	6	7	8	9	
	L	a	c		q	u	i		P	a	r	l	e		C	o	u	n	t	y
Find P	✗	✗	✗	✗	✗	✗	✗	✗	✔											
Followed by a										✔										
Followed by r											✔									
Followed by i												✗								
Back up & Resume										⊙										
Find P										✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
No Match																				

The search for the “P” beginning at position 10 doesn’t find a match.

Regular Expressions

- *Regular expressions* give us a powerful way of matching patterns in text data
- Importantly, we do this all *programatically* rather than by hand, so that we can easily reproduce our work if needed.

Syntax:

- ***Literal*** *characters* are matched only by the character itself.
- *A* ***character class*** is matched by *any* single member of the specified class. For example, **[A-Z]** is matched by any capital letter.
- ***Modifiers*** operate on literal characters, character classes, or combinations of the two. For example, ^ is an anchor that indicates the literal must appear at the beginning of the string

Equivalent Characters

Equivalent Characters

- Suppose we want to be flexible with capitalization

```
sub("Parish","", cNames[2],
    ignore.case = TRUE)
```

- What if we don't want to ignore case in the entire string, but only in "Parish"?

```
sub("[pP]arish","", cNames[2])
```

Web log example

```
169.237.46.168 - - [26/Jan/2014:10:47:58 -
0800] "GET /stat141/Winter04 HTTP/1.1" 301
328 "http://anson.ucdavis.edu/courses/"
"Mozilla/4.0 (compatible; MSIE 6.0;
Windows NT 5.0; .NET CLR 1.1.4322)"
```

Extract the day, month and year

String Split on [or / or :

```
strsplit(wl1, "[[/:]“)
```

```
[[1]]
[1] "169.237.46.168 - - "
[2] "26"
[3] "Jan"
[4] "2014"
[5] "10"
[6] "47"
```

"[[/:]“

Says that 3 characters are equivalent – left bracket [

Forward slash /

And colon :

Equivalent Characters

- We can enumerate any collection of characters within `[ ]`
- The character `"-"` when used within the character class pattern identifies a range. Examples: `[0-9]`, `[A-Za-z]`
- If we put a caret (`^`) as the first character , this indicates that the equivalent characters are the **complement** of the enumerated characters. Example: `[^0-9]`

Equivalent Characters

- If we want to include the character "**-**" in the set of characters to match, put it at the beginning of the character set to avoid confusion. Example: **[-+][0-9]**

- Note that here we have created a pattern from a *sequence* of two sub-patterns.

Named Equivalence Classes

`[:alpha:]`	All alphabetic
`[:digit:]`	Digits 0123456789
`[:alnum:]`	All alphabetic and numeric
`[:lower:]`	Lower case alphabetic
`[:upper:]`	Upper case alphabetic
`[:punct:]`	Punctuation characters
`[:blank:]`	Blank characters, i.e. space or tab

These can be used in conjunction with other characters, for example **`[[:digit:]_]`**

Meta Characters

Let’s search for a subject line that begins with Re:

Meta characters

^ As the first character in the pattern, anchor for the beginning of the string/line
e.g. `^[lg]ame` matches "lame" and "game" but not the last four characters in "flame"

As the first character in `[]`, *exclude* these
e.g. `[^[:alnum:]]` matches any single character that's not a letter or number

$ End of string/line anchor
e.g. `^[^[:lower:]]+$` (What does it match?)

Meta characters that control *how many times* something is repeated

? Preceding element *zero or one* time
e.g. ba? matches "b" or "ba"

\+ Preceding element *one or more* times
e.g. ba+ matches "ba", "baa", "baaa", and so on, but not "b"

* Preceding element *zero or more* times
e.g. ba* matches "b", "ba", "baa", and so on.

`.` Any single character
e.g. `.*` matches any character, any number of times
(like * as a UNIX wildcard)

`[ ]` Character class
e.g. `[a-cx-z]` matches “a”, “b”, “c”, “x”, “y”, or “z”

`-` Range within a character class

`|` Alternation, i.e. one subpattern or another
e.g. `abc|vwxyz` matches “abc” and “vwxyz”

`()` Identify a subpattern
e.g. `ab(c|x)yz` matches “abcyz” and “abxyz”

`\\<`	Beginning of a word
`\\>`	End of a word
`{n}`	Preceding item n times
`{n,}`	Preceding item n or more times
`{n,m}`	Preceding item between n and m times (inclusive)

Pattern: ^[^[:lower:]]+$

```
[1] ""       "HELP!" "Hi"    "123"
```

A. 1 2

B. 2 3

C. 2 4

D. 1 2 3

E. 1 2 4

The position of a character in a pattern determines whether it is treated as a meta character.

Examples: `[-+*/]`, `[1-9]*`

When you want to refer to one of these symbols literally, you need to precede it with a backslash (\). However, this already has a special meaning in R's character strings -- it's used to indicate control characters like newline (\n).

So, to refer to these symbols in R's regular expressions, you need to precede them with *two* backslashes.

The characters for which you need to do this are:

`. ^ $ + ? ( ) [ ] { } | \`

Eliminate . from county names

```
> gsub(".", "", cNames)
[1] "" "" "" "" ""
```

Above was not what we wanted!

"." stands for any character

```
> gsub("\\.", "", cNames)
[1] "Dewitt County"
[2] "Lac qui Parle County"
[3] "St John the Baptist Parish"
[4] "Stone County"
[5] "Lewis & Clark County"
```

Functions that use Regular Expressions

`gsub(pattern, replacement, x)`

Look the regular expression in `pattern` in `x` and replace the matching characters with `replacement` (all occurrences)

`sub()` works the same way but only replaces the first occurrence.

Functions that use Regular Expressions

`grep(pattern, x)`

Look for the regular expression in `pattern` in the character string(s) in `x`. It returns the *indices* of the elements in `x` for which there was a match.

`grepl()` returns a logical vector with TRUE for the elements for which there was a match in `x`.

Functions that use Regular Expressions

`regexpr(pattern, text)`

Look for `pattern` in `text`

Returns an integer vector giving the starting position of the first match or *-1* if there is none. The return value has an attribute "match.length", that gives the length of the matched text (or *-1* for no match).

Functions that use Regular Expressions

`gregexpr(pattern,text)`

Same as `regexpr`, but it returns the locations of all occurrences of the pattern in each element of text. The return value is a list.

Practice: Indicate which strings contain a match to the pattern

	“hi mabc”	“abc”	“ abcd”	“abccd”	“abcabcdx”	“cab”	“abd”	“cad”
abc								
^abc								
abc.d								
abc+d								
abc?d								
abc$								
abc.*d								
abc?								
a[b?d]								

Practice: Indicate which strings contain a match to the pattern

	“hi mabc”	“abc”	“ abcd”	“abccd”	“abcabcdx”	“cab”	“abd”	“cad”
abc	✔	✔	✔	✔	✔			
^abc		✔	✔	✔	✔			
abc.d				✔				
abc+d			✔	✔				
abc?d			✔				✔	
abc$	✔	✔						
abc.*d			✔	✔	✔			
abc?	✔	✔	✔	✔	✔	✔	✔	
a[b?d]	✔	✔	✔	✔	✔	✔	✔	✔

Greedy Matching

- Be careful with patterns matching too much.
- The matching is greedy in that it matches as much as possible
- For example: the regular expression `abc.*d` matches the entire string, abcdzfgdsad.

Why?

Because any character any number of times between abc and d is considered a match, and any character includes d

Greedy Matching

Pattern: `abc.*d`

String: abcdzfgdsad

Matches: abcdzfgdsad

Alternative Pattern: `abc[^d]*d`

Matches: abcd

Summary

- The syntax for regular expressions is *extremely* concise
- It can be overwhelming if you try to read it like you would regular text.
- Always break it down into these three components: literals, character classes, modifiers
- Build your regular expression up a bit at a time