Text Data

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
- St. vs St
- DeWitt vs De Witt

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
- To mine the speeches, we want to create a word vector for each speech, which tracks the 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”

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/2004:10:47:58 -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/2004:10:47:58 -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 such as in a csv file,
- nor is it placed consistently in the same columns in the file.

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 2004 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
```

Email

- Body is separated from the header by a single blank line.
- Attachment is included in the body of the message.
- To figure out what part of the body is the message and what part is an attachment mail programs use an Internet standard called MIME, Multipurpose Internet Mail Extensions.

- *Content-Type:* has value *multipart* when attachments are present

Content-Type: MULTIPART/Mixed;
BOUNDARY="**_===669732====calmail-me.berkeley.edu===_**”

- *Boundary* key provides a special character string to mark the beginning and end of the message parts.
- The last component of the message is followed by a line containing the boundary string with two hyphens at the front and end of the string:

--_===669732====calmail-me.berkeley.edu===_--

What characteristics can you derive from the email?

- Sent in the early morning:

Numeric 00 – 24 hour received

- Has an Re: in the subject line

Logical: TRUE if Re: in subject line

- Funny words like v!@gra

Logical: TRUE if punctuation in the middle of word

- Lots of YELLING IN THE EMAIL

Numeric: proportion of characters that are capitals

One example

A small problem

- County names in census file have no “.” after St, e.g. “St John”
- County names in geographic file do have the “.”, e.g. “St. John”
- Let’s find a way to update the county names in the census file to add the period after “St”

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 one character pieces

String manipulation functions

- `paste(x, y, z, …, sep = " ", collapse = NULL)` – paste together character strings separated by one blank
- `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”
>test = cNames[3]
```

One possible solution

```
> substring(test, 1, 2)
[1] "St"
> substring(test, 1, 2)  ==  "St"
[1] TRUE
> newName =
paste("St.",
      substring(test, 3, nchar(test)),
      sep ="")
```

Do you see any problems with this?

Second possible solution

```
> substring(test, 1, 3)
[1] "St "
> substring(test, 1, 3)  ==  "St "
[1] TRUE
> newName =
paste("St. ",
      substring(test, 4, nchar(test)),
      sep ="")
```

Do you see any problems with this?

Practice with

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

The Web log

169.237.46.168 - - [26/Jan/2004:10:47:58 -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 :

Return to St vs St.

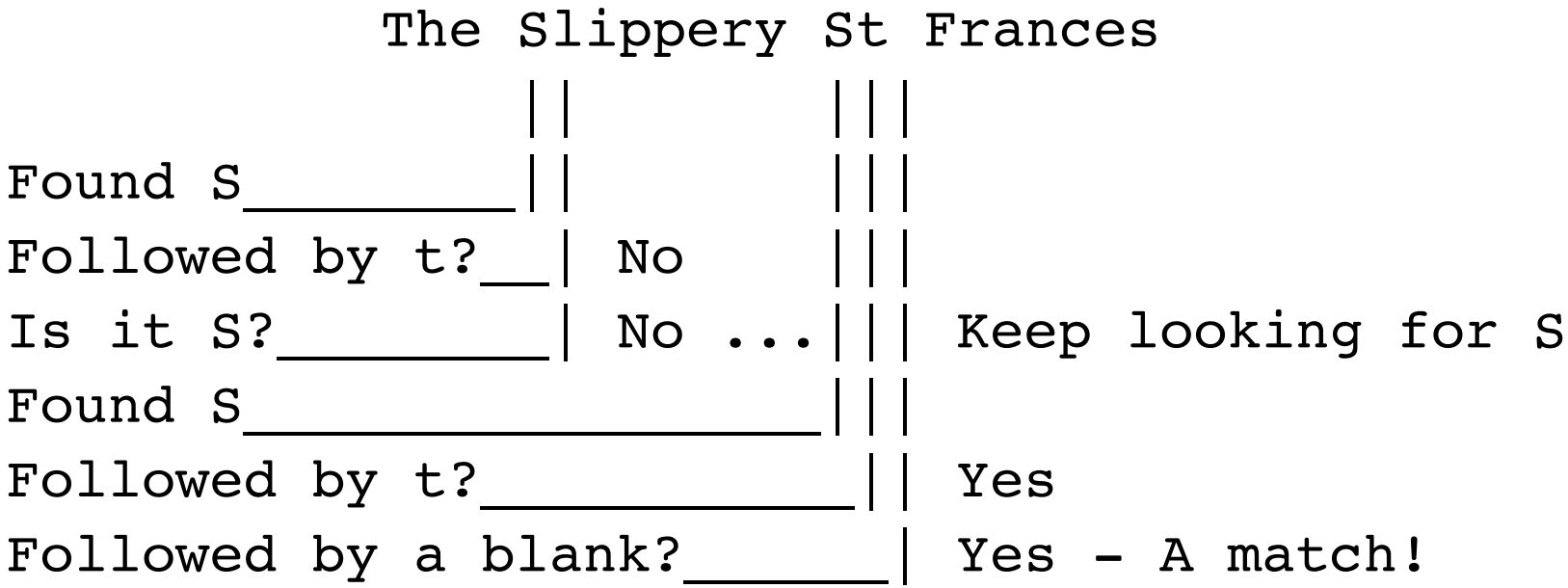
Another idea

```
> string = "The Slippery St Frances"
> chars = unlist(strsplit(string, ""))
> chars
 [1] "T" "h" "e" " " "S" "l" "i" "p" "p" "e" "r"
[12] "y" " " "S" "t" " " "F" "r" "a" "n" "c" "e"  "s"
> possible = which(chars == "S")
> possible
[1]  5 14
> substring(string, possible, possible + 2)
[1] "Sli" "St "
> substring(string, possible, possible + 2) == "St "
[1] FALSE  TRUE
```

What are we doing here?

- Look at each character
- Check to see if it is “S”
- If it is, then look at the next character(s)
- This is the idea behind regular expressions

The regular expression “St ” is made up of three literal characters. The **regular expression matching engine** does something very similar to what we just did.



Luckily, we don't actually need to write our own functions for replacement. The R functions `gsub()` and `sub()` look for a pattern and replace it within a string with some other text.

The "g" in `gsub()` refers to global. It changes all the matches, whereas `sub()` only replaces the first match (in each element – both `gsub()` and `sub` are vectorized).

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

```
> strings = c("a test", "and one and one is two",
+               "one two three")

> gsub("one", "1", strings)
[1] "a test"  "and 1 and 1 is two"  "1 two three"

> sub("one", "1", strings)
[1] "a test"  "and 1 and one is two"  "1 two three"
```

Regular Expressions to the Rescue

Regular Expressions

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

With regular expressions, we can

- extract pieces of text – e.g., find all links in an HTML document
- create variables from information found in text
- clean and transform text into a uniform format, resolving inconsistencies in format between files
- mine text by treating documents directly as data
- “scrape” the web for data

- A *regular expression* (aka regex or regexp) is a pattern that describes a set of strings.
- This set may be finite or infinite, depending on the particular regexp. We say the regexp "matches" each element of that set.
- For example, the regexp

```
grey|gray
```

matches both grey and gray, whereas

```
^A.*
```

matches any string starting with capital A.

- The idea is similar to wildcards in UNIX, but with many more possibilities.

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

Warning

- The syntax for regexps 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

How to find fake words?
rep1!c@ted

- What makes this different from a regular word?
- Numbers and punctuation surrounded by letters
- Concepts of “numbers”, “punctuation”, and “regular letters” get at the idea of *equivalent characters* or *character classes*.

Equivalent Characters

- We can enumerate any collection of characters within `[ ]`. Example: `[Tt]his`
- 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:]_]`**

Return to `rep1!c@ted`

- What will this match?

```
[[:alpha:]][[:digit:][:punct:]][[:alpha:]]
```

- Can you foresee any problems with it?

Functions that use Regular Expressions

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

- `regexpr(pattern, 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).
- `gregexpr(pattern,text)` Returns the locations of all occurrences of the pattern in each element of text. The return is a list.

```
> subjectLines
[1] "Re: 90 days" "Fancy rep1!c@ted watches" "It's me"
> grep("[[:alpha:]][[:digit:][:punct:]][[:alpha:]]",
subjectLines)
[1] 2 3
```

We can either remove the apostrophe first:

```
> newString = gsub("'", "", subjectLines)
> grep("[[:alpha:]][[:digit:][:punct:]][[:alpha:]]",
newString)
[1] 2
```

Or we can specify the particular punctuation marks we're looking for:

```
> grep("[[:alpha:]][[:digit:]!@#$%^&*():;?,.]
[[:alpha:]]", subjectLines)
[1] 2
```

`gregexpr()` shows exactly where the pattern was found:

```
> newString
[1] "Re: 90 days" "Fancy rep1!c@ted watches" "Its me"
> gregexpr("[[:alpha:]][[:digit:][:punct:]]
[[:alpha:]]", newString)
[[1]]
[1] -1
attr(,"match.length")
[1] -1
[[2]]
[1] 12
attr(,"match.length")
[1] 3
[[3]]
[1] -1
attr(,"match.length")
[1] -1
```

No match

Starting at 12,
match of length 3

No match

Did we miss anything??

We didn't find `p1!c` because it consists of four characters: a letter, a digit, a punctuation mark, and another letter.

To search for the more general pattern of *any* number of digits or punctuation marks between letters, we use

```
[[:alpha:]][[:digit:][:punct:]]+[[:alpha:]]
```

The plus sign indicates that members from the second character class (digits and punctuation) may appear *one or more* times.

The plus sign is an example of a *meta character*.

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.
Note the difference compared to the UNIX wildcard!

`.` 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|xyz matches “abc” and “xyz”

`( )` 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)

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:

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

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]								

More Practice: Write a regular expression that matches

1. only the words “cat”, “at”, and “t”

2. The words “cat”, “caat”, “caaat”, and so on

3. “dog”, “Dog”, “dOg”, “doG”, “DOg”, etc. (i.e., the word dog in any combination of upper and lower case) anywhere in the string

4. Any number, with or without a decimal point

Greedy Matching

- Be careful with patterns matching too much.
- The matching is greedy in that it matches as much as possible
- For example: when trying to remove HTML tags from a document, the regular expression

`<.*>` will match too much but the regular expression `<[^>]*>` will be just right. Why?