Texts, Functions, and Probabilities

Ben Schmidt

February 15, 2015

Texts

Research using texts is perhaps the most defining feature of digital humanities work. We're going to start by looking at one of the most canonical sets of texts out there: the State of the Union Addresses.

We're also going to dig a little deeper into two important aspects of R we've ignored so far: **functions** (in the sense that you can write your own) and **probability**.

Reading Text into R

You can read in a text the same way you would a table. But it's easier to simply read it in as a character vector. (Note: you can find a process very similar to this at greater length in the Jockers text analysis book online.)

```
text = scan("data/SOTUS/2015.txt",sep="\n",what="raw")
```

If you type in "text," you'll see that we get the full 2015 State of the Union address.

As structured here, the text is divided into *paragraphs.* For most of this class, we're going to be interested instead in working with *words.*

Tokenization.

Let's pause and note a perplexing fact: there's no good definition of a word. In the field of corpus linguistics, the term is ignored in favor of the idea of a "token" or "type." Where a word is more abstract, a "token" is a concrete term used in actual language, and a "type" is the particular string we're interested in.

Breaking a piece of text into words is thus called "tokenization." There are many ways to do it–coming up with creative tokenization methods will be helpful in the algorithms portion of this class. But the simplest is to simply to remove anything that isn't a letter. Using regular expression syntax, the R function `strsplit` lets us do just this: split a string into pieces. We'll use the regular expression `[^A-Za-z]` to say "split on anything that isn't a letter between A and Z." Note, for example, that this makes the word "Don't" into two words.

```
text %>%
  strsplit("[^A-Za-z]")
```

You'll notice that now each paragraph is broken off in a strange way. We're now working with data structures other than a data.frame. This can be useful: but

```
words = text %>%
  strsplit("[^A-Za-z]") %>%
  unlist
```

Finally, you'll see that this is a character. We've been working with, instead, `data.frame` objects in `dplyr`. This individual word, though, can be turned into a column in a data.frame by using the function to create it.

```
SOTU = data.frame(word=words)
```

What can we do with such a column put into a data.frame?

First off, you should be able to see that the good old combination of `group_by`, `summarize`, and `n()` allow us to create a count of words in the document.

This is perhaps the time to tell you that there is a shortcut in `dplyr` to do all of those at once: the `count` function.

```
wordcounts = SOTU %>% group_by(word) %>% summarize(count=n()) %>% arrange(-count)
wordcounts
```

Using ggplot, we can plot the most frequent words.

```
wordcounts = wordcounts %>% mutate(rank = rank(-count))  %>% filter(count>2,word!="")

ggplot(wordcounts) + aes(x=rank,y=count,label=word) + geom_text()
```

As always, you should experiment with multiple scales. Putting logarithmic scales on both axes reveals something interesting:

```
ggplot(wordcounts) + aes(x=rank,y=count,label=word) + geom_text() + scale_x_continuous(trans="log") + s
```

The logarithm of rank decreases linearily with the logarithm of count.

This is "Zipf's law:" the phenomenon means that the most common word is twice as common as the second most common word, three times as common as the third most common word, four times as common as the fourth most common word, and so forth.

It is named after the linguist George Zipf, who first found the phenomenon while laboriously counting occurrences of individual words in Joyce's *Ulysses* in 1935.

This is a core textual phenomenon, and one you must constantly keep in mind: common words are very common indeed, and logarithmic scales are more often appropriate for plotting than linear ones. This pattern results from many dynamic systems where the "rich get richer," which characterizes all sorts of systems in the humanities. Consider, for one last time, our city population data:

```
cities = read.csv("data/CESTACityData.csv")
head(cities)
nowadays = cities %>% select(CityST,X2010) %>% arrange(-X2010) %>% mutate(rank = rank(-X2010)) %>% filte

ggplot(nowadays) + aes(x=rank,label=CityST,y=X2010) + geom_text() + scale_x_log10() + scale_y_log10()
```

Concordances

This data frame can also build what used to be the effort of entire scholarly careers: A "concordance." We do this by adding a second column to the frame which is not just the first word, but the second. `dplyr` includes a `lag` and `lead` function that let you combine the next element. You specify by how many positions you want a vector to "lag" or "lead" another one.

```
numbers = c(1,2,3,4,5)
lag(numbers,1)
lead(numbers,1)
```

By using `lag` on a character vector, we can neatly align one series of text with the words that follow.

```
SOTU %>% mutate(word2 = lead(word,1)) %>% head
```

Doing this several times gives us snippets of the text we can read *across* as well as down.

```
multiColumn = SOTU %>% mutate(word2 = lead(word,1),word3=lead(word,2),word4=lead(word,3))
multiColumn %>% head
```

Using `filter`, we can see the context for just one particular word.

```
multiColumn %>% filter(word3=="Congress")
```

Functions

That's just one State of the Union. How are we to read them all in?

We could obviously type all the code in

```
readSOTU = function(file) {
  message(file)
  text = scan(file,sep="\n",what="raw")
  words = text %>%
    strsplit("[^A-Za-z]") %>%
    unlist
  SOTU = data.frame(word=words,year=gsub(".*(\\d{4}).*","\\1",file),stringsAsFactors = FALSE)
  return(SOTU)
}

SOTUS = list.files("data/SOTUS",full.names = TRUE)
```

Now we have a list of State of the Unions and a function to read them in. How do we automate this procedure? There are several ways.

The most traditional way, dating back to the programming languages of the 1950s, would be to write a `for` loop.

```
allSOTUs=data.frame()
for (filename in SOTUS) {
  allSOTUs = rbind(allSOTUs,readSOTU(filename))
}
```

This works, but is relatively slow. Good R programmers *never* write a `for-loop`; instead, they use functions that allow you operate on lists. One of the most basic ones is called "lapply": it takes as an argument a list and a function, and applies the function to each element of the list. We can then collapse the list down to a `data.frame` using the `rbind_all` function.

```r
all = lapply(SOTUS,readSOTU)
allSOTUs = rbind_all(all)
```

Either of these are acceptable: but if you want to remain entirely in the `dplyr` idiom, the way to do this is *create a data frame* and then apply a function to it.

`dplyr` uses the function `do` to describe this. The only tricky thing about `do` is that it requires you to use `dplyr`'s special character, the period, to describe the grouped frame. So to run our `readSOTU` function

```r
all = data.frame(filename=SOTUS,stringsAsFactors=FALSE) %>%
  group_by(filename) %>%
  do(readSOTU(.$filename))
```

Steve Ramsay writes about finding words that are unique to each individual author. We can do that to find words that only appeared in a single state of the union. We'll put in a quick lowercase function so as not to worry about words.

```r
allSOTUs %>%
  mutate(word=tolower(word)) %>%
  distinct(word,year) %>%
  group_by(word) %>% filter(n()==1)

unique = allSOTUs %>%
  mutate(word=tolower(word)) %>%
  distinct(word,year) %>%
  group_by(word) %>% filter(n()==1)

unique %>% filter(year>1861,year<=1865) %>% View
```

Probabilities

```r
allSOTUs %<>% as.tbl %>% filter(word!="")

allSOTUs %<>% mutate(word2=lead(word,1),word3=lead(word,2))

transitions = allSOTUs %>%
  filter(year>1912) %>%
  group_by(word) %>%
  mutate(word1Count=n()) %>%
  group_by(word,word2) %>%
  summarize(chance = n()/word1Count[1])
```

```r
findNextWord = function(current) {
  subset = transitions %>% filter(word==current)
  nextWord = sample(subset$word2,1,prob = subset$chance)
}
```

```r
word = "I"
while(TRUE) {
  message(word)
  word = findNextWord(word)
}
```

How would we make the same function work over longer stretches?

First, we'd just make the transition probabilities apply over stretches of three words. That's the same general principle, but we divide the count of word3 by the bigram count that precedes it.

```
transitions = allSOTUs %>%
  filter(year>1912) %>%
  group_by(word) %>%
  mutate(word1Count=n()) %>%
  group_by(word,word2) %>%
  mutate(chance = n()/word1Count,bigramCount=n()) %>%
  group_by(word,word2,word3) %>%
  summarize(trigramChance = n()/bigramCount[1])
```

Then, we'd make the "find next word" function work over spans of two words instead of one.

```
findNextWord = function(previous,current) {
  subset = transitions %>%
    filter(word==previous,word2==current)
  nextWord = sample(subset$word3,1,prob = subset$trigramChance)
  return(nextWord)
}
```

Finally, we have to remember both the "current" and "previous" word at all times in the loop: and we have to start it with two words instead of one. But with this, we get some wonderfully serendipitous nonsense.

```
previous = "We"
current = "must"
while (TRUE) {
  message(previous)
  nextWord = findNextWord(previous,current)
  previous = current
  current = nextWord
}
```