9: Classifying

Ben Schmidt

3/19/2015

A brief digression on code organization

You remember that we've been cutting and pasting in large chunks of code periodically–to read in the SOTUs, and so forth.

Probably you've been frustrated by this. But it's not the only one! Just as we can group commonly used lines of code into a function and apply them one at a time, we can also group commonly used *functions* together into a single file. You'll find it on the course repository called "commonFunctions.R".

We've already seen functions to read in data like `read.csv` and `read.table`. The `source` function reads in a file location, but **as code**. It provides an easy way to share and test code without having to cut and paste continually.

RStudio even provides an extremely useful checkbox in the upper left-hand corner of every `.R` file called "source on save." That means that every time you save the file, the code will get run, instantly applying any changes to functions you've made.

This can be an extremely useful way to organize your work. But two notes.

1. You must save into a file with the suffix `.R`. We've been working mostly in RMarkdown files, because they let you type out text and explanations. But they can't be directly imported.
2. As a rule, the functions you save in a file you'll be `source`ing repeatedly should not actually **do** anything–they should instead **give functions that will later do things as needed**. So for instance, if you have a complicated set of rules for reading in and transforming a file, **you shouldn't read in the file inside the functions file**; instead you should write a function that will do the reading when you ask it to. It's OK to have a few things that aren't functions: the name of a directory where you store data, say, or a bunch of calls to load libraries. As a rule, if your `source` command takes more than 4 seconds, you should do things differently.

```
source("Problem_Sets/commonFunctions.R")
```

You can use those frames to build up some of your own work. But I've also exposed a function in there to pull directly from one of my bookworm database, which stores word counts and metadata for easy access. Some of you are already building them–if any others want to, let me know. It's easy enough.

For this to work, you'll need to install two new packages:

```
install.packages("RCurl")
install.packages("RJSONIO")
```

If that doesn't work, you can keep doing the same some of work on the existing files.

```
presidentWords = bookworm(database="SOTU",groups=list("year","unigram"),search_limits=list("year"=list(
presidentNames = bookworm(database="SOTU",groups=list("year","name","party"),counttype="TotalWords") %>%
  as.tbl() %>% mutate(speech_id=paste(name,year))
```

By merging these together, we can start to look at the interesting questions of comparison and classification that Underwood and others for this week have been writing about.

First, we'll merge in the metadata, and create an *id-field* for each speech (here it will be the president and the year).

```
allPresidents = presidentWords %>% inner_join(presidentNames)
```

Then we'll use the trick from last week with `spread` to make a term document matrix consisting of words that appear more than 1000 times.

```
tdMatrix = allPresidents %>% group_by(unigram) %>% filter(nchar(unigram)>3,sum(WordCount)>1000) %>% ungr
```

Comparison

I've given you a function `dunning.log` in that file that compares the most likely words between two sets.

```
group1 = allPresidents %>% filter(party=="Republican") %>% group_by(unigram) %>% summarize(WordCount = s
group2 = allPresidents %>% filter(party=="Democratic") %>% group_by(unigram) %>% summarize(WordCount = s

a = dunning.log(group1,group2)

a %>% ungroup %>% arrange(-dunning) %>% head(30)
```

How could we display this?

Classification

The simplest classification is called "naive Bayes." It's what Mosteller and Wallace used.

```
fedWords = bookworm(database="federalist",groups=list("author","title","unigram"),counttype = "WordCount

authorProbs = fedWords %>%
  group_by(author) %>%
  mutate(authorWords = sum(WordCount)) %>%
  group_by(author,unigram) %>%
  summarize(probability=sum(WordCount)/authorWords[1])
```

Bayesian Classifiers in dplyr

That new matrix gives us the chance of each author using any individual word. We just need to merge that in, and we can gather the overall probability for any author using the *set of* words that he or she does.

The chance of two one in a hundred events happening back to back is 1/100 * (1/100), or 1 in ten thousand. Multiplying numbers lots of times is hard, but this is another case where logarithms are helpful. Maybe you remember from high school math that `exp(log(a) + log(b)) = a*b`. If not, don't worry about it! What is means, is that we can calculate probabilities through addition of log probabilities.

```
probs = authorProbs %>% select(authorCandidate = author,unigram,probability)

chances = fedWords %>% inner_join(probs)

classification = chances %>% group_by(title,authorCandidate) %>% summarize(logProbability = sum(log(prob
```

Underwood uses logit regression. Here's some code to get you started.

```
guessParty = function (tdMatrix,partyName) {

  modelable = tdMatrix %>% mutate(thisparty=party==partyName) %>% select(-party,-speech_id)
  model = glm(thisparty ~ ., data=modelable,family="binomial")
  predict(model,newdata=modelable)
  predictions = predict(model,newdata=modelable)
  tdMatrix[[partyName]] = predictions
}

a = guessParty()

ggplot(tdMatrix %>% inner_join(presidentNames,by=c("speech_id"))) + geom_point(aes(x=party.x,y=predicti
```