Lecture 4: Counting at Scale
Modeling Social Data, Spring 2017
Columbia University

Drew Johnston

February 10th, 2017

Lecture 4 can be broadly separated into two parts. The first is an expansion on methods for data preparation and cleaning, which we covered last week. The second expanded on our discussion of counting, talking about how to work on a substantially larger scale than before, using Hadoop and MapReduce.

1 Data Preparation Part 2

1.1 Groupby and Vectorization

We discussed groupby again, which is a function in R that we'll be using quite extensively. As a refresher, it allows you to break up a dataframe into groups, with each containing all the observations that share certain entries. For instance, let's say that you had a dataframe containing information about various people, with columns for gender, eye color, etc. Grouping the data by one of these columns breaks up the data according to the values in that column, so that you can consider the groups separately. For instance, grouping by eye color would create groups for each option–say, a group containing all of the blue-eyed people, another for the brown-eyed, etc. These could then easily be compared separately and concisely. Some important observations:

- Groupby is powerfully combined with the pipe command. For instance,

  ```
  iris %>% group_by(Species) %>% summarize(count=n(),mean_length=mean(Sepal.Length))
  ```

  will compute the number of observations for each species and the mean sepal length within each species and display a summary table, as we saw in lecture.
- The majority of important R verbs (like filter) will respect the groups within groupby and process accordingly.
- The dataframe outputted by a groupby operation is modified, containing information about the categories. This makes it simple to perform grouped operations, but can make many operations give unexpected answers or perform poorly when this is not the functionality you are looking for. Be careful about when you are using the grouped and ungrouped tables!

1.2 Pointers and Common Mistakes

We then talked about some of the basics of R, and how to write effective and functional programs in it. Here are a few tips I thought were worth reemphasizing:

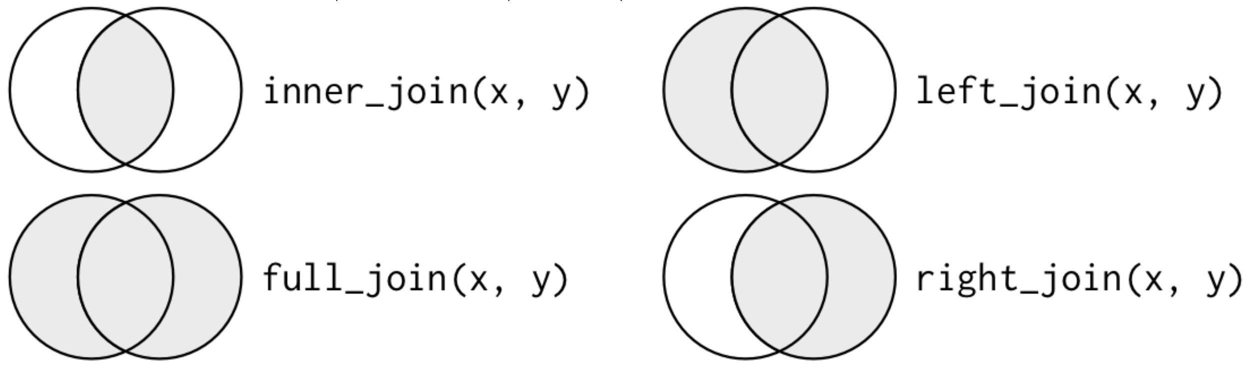
- In data processing, it's often handy to pass the results of one function into another in relatively complex ways. In R, you can do this in the standard way allowed by most programming languages, just nesting the functions within one another. This works fine, but can result in unreadable code if your pipeline is quite long. There is also the %>% function, which acts similarly to a Linux pipe. It normally just pipes the output of the function to its left to the first argument of the function on its right, but you can control the ordering. df %>% f() evaluates to f(df), df %>% f(x, .) evaluates to f(x, df).

- Variable scoping can be a major source of confusion and unexpected behavior. A common error is creating local variables with the same name as columns in a dataframe, which can cause lots of frustration. Avoid this if at all possible!
- Be sure to vectorize your functions if you're writing ones that ought to be applied to columns or rows of a dataframe! Functions designed to process a single item will be mighty confused when passed an array of such items. There's a built in vectorize function in R that you can pass your functions to in order to simplify this process.
- If else statements can often be unsightly in R, especially when applied to vectors. There's an ifelse function that can be of help to you, functioning like a ternary operator. An example to which you can pass a vector of integers, which will provide a vector of strings as a result:

  ```
  ifelse(a%%2==1,"odd","even")
  ```

1.3 Joins

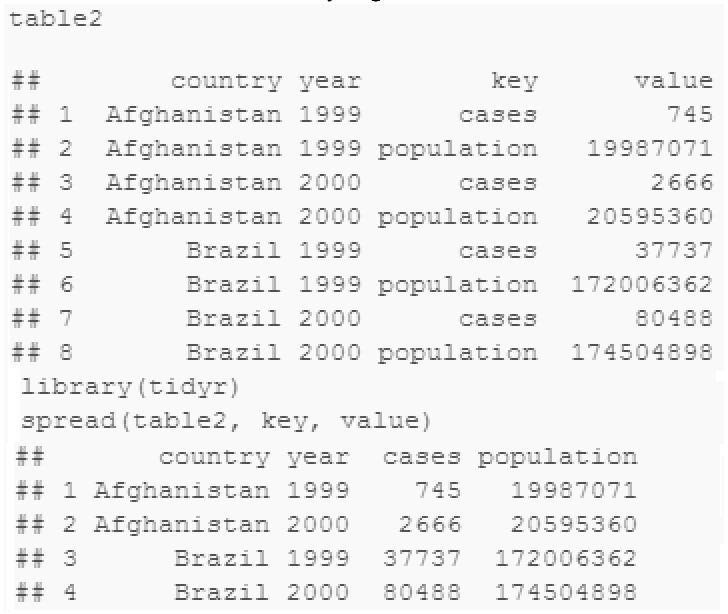
R also includes functionality to allow for SQL-style joins between tables. This is quite important in the real world, as it allows you to combine data from a number of sources into one dataframe. There are several methods for joining data, depending on how you want data that appears in one data set but not the other to be treated. Here's a handy diagram from /r4ds.had.co.nz/diagrams/join-venn.png that I found helpful:



- There's a "by" argument that allows you to specify what columns in each dataframe you would like to join by comparing. For instance, $inner_join(df1, df2, by = c('firstname' =' namefirst'))$ will perform an inner join on df1 and df2 by comparing column "firstname" in df1 to column "namefirst" in df2. R will try to guess how to perform the join if you don't include this argument, but it's best to keep it for the sake of clarity.
- R also has an anti-join command which can be used to see all of the values that are dropped when performing a join such as an inner join that does not preserve all of the information. This can be useful to ensure that you're not accidentally dropping lots of your information!
- In this class, it's considered bad style to use right join. For the sake of clarity and consistency, it's preferred that you switch the order of the arguments and do a left join instead.

1.4 Spread and Gather

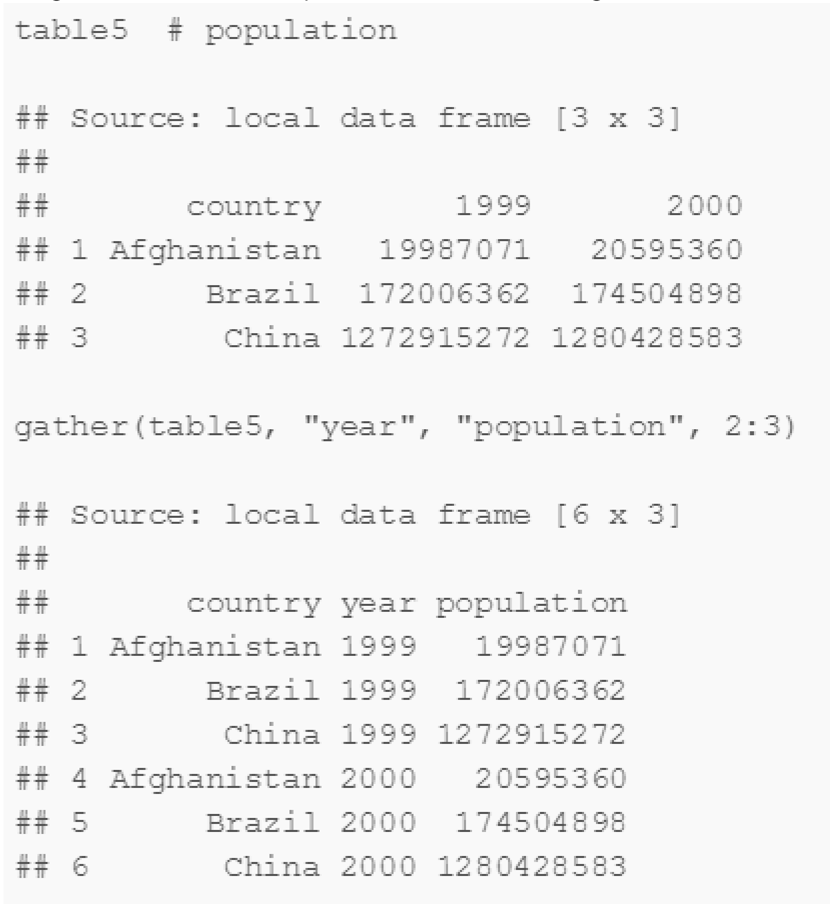
Spread and gather are two functions introduced by tidyr, which allow for you to "widen" or "lengthen" data by splitting up categorical values in rows into columns or by doing the reverse. This is very useful in plotting results. I found a neat and helpful example of spread on the internet (http://garrettgman.github.io/tidying/), which I'll reproduce here. It shows how you can turn the categories in a column full of categorical keys and a data column into two neatly organized columns.



```
table2

##        country year        key      value
## 1  Afghanistan 1999      cases        745
## 2  Afghanistan 1999 population   19987071
## 3  Afghanistan 2000      cases       2666
## 4  Afghanistan 2000 population   20595360
## 5       Brazil 1999      cases      37737
## 6       Brazil 1999 population  172006362
## 7       Brazil 2000      cases      80488
## 8       Brazil 2000 population  174504898
library(tidyr)
spread(table2, key, value)
##       country year  cases population
## 1 Afghanistan 1999    745   19987071
## 2 Afghanistan 2000   2666   20595360
## 3      Brazil 1999  37737  172006362
## 4      Brazil 2000  80488  174504898
```

Gather will do the reverse operation, converting specified columns into a categorical variable. It takes an additional argument, the indexes of the columns, to determine which columns should be converted into categories. Here's an example from the same site of gather in action:



```
table5  # population

## Source: local data frame [3 x 3]
##
##       country       1999       2000
## 1 Afghanistan   19987071   20595360
## 2      Brazil  172006362  174504898
## 3       China 1272915272 1280428583

gather(table5, "year", "population", 2:3)

## Source: local data frame [6 x 3]
##
##       country year population
## 1 Afghanistan 1999   19987071
## 2      Brazil 1999  172006362
## 3       China 1999 1272915272
## 4 Afghanistan 2000   20595360
## 5      Brazil 2000  174504898
## 6       China 2000 1280428583
```

2 Intro to MapReduce and Hadoop

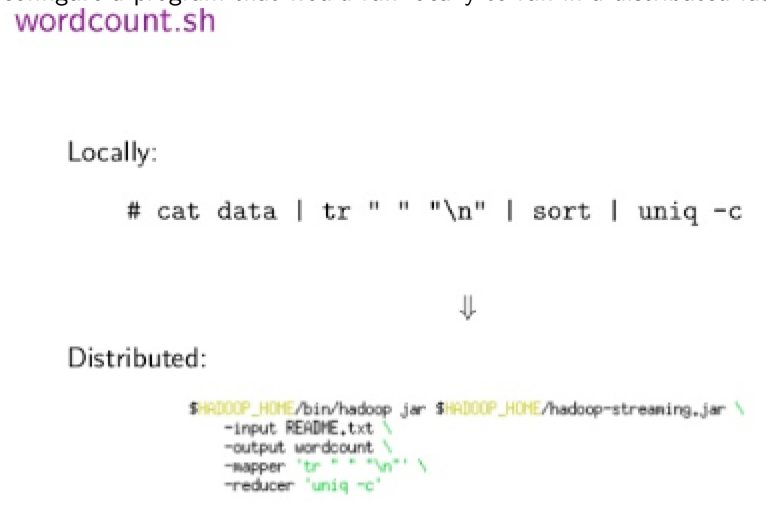
2.1 MapReduce

MapReduce is a distributed solution to computational problems exceeding the processing power of a single machine. It has become extremely prominent within computer and data science in the last decade or so, as it is abstract enough to be applied as a framework to a large variety of parallelizable tasks. Essentially, it asks the programmer to implement a map function, which transforms the data into key-value pairs, as well as a reduce function, which lays out the computations that ought to be performed on each group. There's also a shuffle record that works behind the scenes, to collect all of the records with the same key together for the reduce function. This is advantageous if the processing takes a long time or if the disk access takes a long time, since both can be done in parallel. Some important points:

- Fault tolerance and synchronization are implemented under the hood, so the programmer doesn't need to worry about them when working on a new task.
- MapReduce is designed to scale elegantly and transparently.
- Strongest on tasks that are reading and bandwidth intensive.
- Batch, offline, simple tasks are most frequently implemented in this way.
- Some implementations can struggle when the tasks assigned to each computer take dramatically different amounts of time to finish.
- MapReduce is useful for a great many tasks! But it's not always the right answer.

2.2 Hadoop

Hadoop is one of the most popular implementations of MapReduce principles. It's an open source program, which allows for existing code to be easily incorporated into a parallel structure. In many cases, it can save you from writing an entirely new program, since Hadoop as a streaming feature which allows you to just specify existing functions (like Unix builtins) as mappers and reducers. In the slides, there's an example where it's shown that a word counting program can be implemented in Hadoop without writing any Java code, using two lines of code and the Hadoop Streaming utility. Here's a great example taken from the slides, illustrating how easy it is to configure a program that would run locally to run in a distributed fashion:



wordcount.sh

Locally:

```
# cat data | tr " " "\n" | sort | uniq -c
```

⇓

Distributed:

```
$HADOOP_HOME/bin/hadoop jar $HADOOP_HOME/hadoop-streaming.jar \
    -input README.txt \
    -output wordcount \
    -mapper 'tr " " "\n"' \
    -reducer 'uniq -c'
```

Though using Hadoop and MapReduce concepts in traditional languages such as Python and Java is straightforward, there are also a number of high-level languages that were designed with bulk data processing in mind. Hadoop's team makes one called Pig that implements many common data analysis tasks as MapReduce jobs and is usable both on a single computer and in a larger production environment. We also mentioned Hive, another language for large scale data processing that is similar to SQL.