Lecture 2: Data Manipulation

Intro to Data Science for Public Policy, Spring 2016

by Jeff Chen & Dan Hammer, Georgetown University McCourt School of Public Policy

Contents

To most, the following looks like part of a speech:

> Now, one place to start is serious financial reform. Look, I am not interested in punishing banks. I'm interested in protecting our economy. A strong, healthy financial market makes it possible for businesses to access credit and create new jobs. It channels the savings of families into investments that raise incomes. But that can only happen if we guard against the same recklessness that nearly brought down our entire economy. We need to make sure consumers and middle-class families have the information they need to make financial decisions. We can't allow financial institutions, including those that take your deposits, to take risks that threaten the whole economy.

But to a data scientist, it looks like an opportunity to structure a dataset:

```
##            x freq len
## 24 financial    4   9
## 21   economy    3   7
## 23  families    2   8
## 3     access    1   6
## 5      allow    1   5
## 8      banks    1   5
```

All the steps required to convert unstructured information into usable data can be done with a little bit of planning, technical imagination and data manipulation. Every little detail about the data needs to be considered and meticulously converted into a usable form. From a data format perspective, upper cases are not the same as lower cases. Contractions are not the same as terms that are spelled out. Punctuation affect spacing. Carriage returns and new line markers, while not visible in reading mode, are recorded.

Let's take one line from above and dissect the changes that need to be made:

```
## [1] "We can't allow financial institutions, including those that take your deposits, to take risks th
```

We then turn everything into lower case so all letters of the alphabet are read the same.

```
## [1] "we can't allow financial institutions, including those that take your deposits, to take risks th
```

Then, we get rid of punctuation by substituting values with empty quotations ('“”').

```
## [1] "we cant allow financial institutions including those that take your deposits to take risks that
```

Each space between each word can be used as a 'delimiter'. What that means is we can break apart words into values in a vector or list.

```
##  [1] "we"           "cant"         "allow"        "financial"
##  [5] "institutions" "including"    "those"        "that"
##  [9] "take"         "your"         "deposits"     "to"
## [13] "take"         "risks"        "that"         "threaten"
## [17] "the"          "whole"        "economy"
```

There are words in there that don't add much value as they are commonplace and filler. In text processing, these words are known as *stopwords*. They're not perfect, but in each domain, data scientists typically build a customized list. Below are the remaining words that did not match the stopword list.

```
## [1] "cant"         "allow"        "financial"    "institutions"
## [5] "including"    "deposits"     "risks"        "threaten"
## [9] "economy"
```

From that data, we can aggregate the data into a form that is meaningful to answer a research question. Since the words have been standardized, the data can now be merged and compared with other polished data.

```
##              x freq
## 1        allow    1
## 2         cant    1
## 3     deposits    1
## 4      economy    1
## 5    financial    1
## 6    including    1
## 7 institutions    1
## 8        risks    1
## 9     threaten    1
```

In more complex analyses, a document-term matrix can be developed, which essentially a matrix of word counts for each unique word and document combination.

In this section, we'll cover basics of data manipulation and cleansing. We'll start off by addressing data manipulation as a process. Then, we will learn about processing at the data element or data value level before moving onto processing steps for data frames and matrices. The lesson will then wrap up with some rudimentary processing of State of the Union speeches into easy to analyze formats.

Data manipulation as a process

Manipulating values

What do the following three data points have in common?

```
## [1] "Captain's Log, Stardate 1513.8. Star maps reveal no indication of habitable planets nearby. Orig
## [2] " 1,20"
## [3] "datafile_2010.json"
```

Each value contains data locked in a string.

- The first value is a full sentence that can benefit from *feature extract* or creation of new variables such as the date (*stardate* = 1513.8), the type of entry (*type* = "captain's log"), habital planets status (*habital* = FALSE), or idle time (*idle_time* = 18).
- The second value is a numerical value, but is coded as a string due to the leading white space before the "1" and the comma in place of a period.
- The third value is a file name that contains possibly the year that the datafile contains.

These scenarios are quite common. Whether a numerical value is read into a programming language as a string or values need to be extracted, more often than not, the type of value manipulation is string-based. Thus, in this section, we'll learn a few tricks using *regular expressions* or RegEx as well as rely on string operations such as `regexpr()`, `grep()` and `substr()`.

Regular Expressions

The simplest form of string manipulation is string or pattern find and replace. In the example below, white spaces before and after each value are problematic in addition to the `$` and the `,`.

```
price_data <- c(" $1,23", " $2,5 ", "$5,21")
print(price_data)
```

```
## [1] " $1,23" " $2,5 " "$5,21"
```

Enter `gsub()`, a simple way to find and replace matched strings. The syntax is as follows:

```
gsub("[pattern-to-be-searched]","[replacement-pattern]", [data-goes-here])
```

In addition we can use a simple method to trim white spaces:

```
trimws([data-goes-here])
```

To replace the commas and dollar signs, the following commands can be used. Notice that the period and dollar sign are preceded by two backslashes – known as *escape values*. In R and in many programming languages, searching for characters rely on *special characters* to conduct complex processes. As it turns out `.` is short hand for 'match any single character' whereas `$` denotes 'match the end of a string'. These are the foundations of *regular expressions*.

```
#Replace comma with period
  price_data <- gsub(",","\\.",price_data)
  print(price_data)
```

```
## [1] " $1.23" " $2.5 " "$5.21"
```

```
#Replace dollar sign with blank
  price_data <- gsub("\\$","",price_data)
  print(price_data)
```

```
## [1] " 1.23" " 2.5 " "5.21"
```

```
#Trim white space
  price_data <- trimws(price_data)
  print(price_data)
```

```
## [1] "1.23" "2.5"  "5.21"
```

```
#Convert into numeric
  price_data <- as.numeric(price_data)
  print(price_data)
```

```
## [1] 1.23 2.50 5.21
```

We can think about regular expressions in terms of the R-specific functions that support string manipulation, escape sequences, position matching, and character classes. Regular expressions are patterns that describe a pattern in a string or part of a string (substring).

Functions

There six metods that will make string manipulation easier. They include:

- `grep()`: Returns either the index position of a matched string or the string containing the matched portion.
- `gsub()`: Searches and replaces patterns in strings.
- `regexpr()`: Returns the character position of a pattern in a string.

- `strsplit()`: Splits strings into a list of values based on a delimiter
- `substr()`: Extract substring from a string based on string positions.
- `regmatches()`: Extract substring using information from `regexpr()`

Let's apply these methods to the following:

```
laws <- c("Dodd-Frank Act", "Glass-Steagall Act", "Hatch Act", "Sarbanes-Oxley Act")
```

Using `grep()`, we'll return the index positions of laws that are named after two people, which is often denoted by `-`.

```
grep("-", laws)
```

```
## [1] 1 2 4
```

`regexpr()` is similar to `grep()` except that it is used to find the position of a match string within each string. `grep()` returns the position within a vector or list. If we run the same search for `-`, `regexpr()` returns two sets of attributes. The first indicates the position of the first character that is matched in each string (e.g. 5 indicates that the `-` value is the fifth character is the first string) whereas the second set of attributes indicates whether there is a match. Positive values indicate matches and a value of -1 indicates no match.

```
  regexpr("-", laws)
```

```
## [1]  5  6 -1  9
## attr(,"match.length")
## [1]  1  1 -1  1
## attr(,"useBytes")
## [1] TRUE
```

By default, `grep()` returns index positions, but can be changed to return matched values by adding in the argument `value = TRUE`.

```
grep("-", laws, value = TRUE)
```

```
## [1] "Dodd-Frank Act"     "Glass-Steagall Act" "Sarbanes-Oxley Act"
```

To keep only the name of the laws without the redundancy of the word "Act", we can use `gsub()`.

```
  laws <- gsub(" Act","", laws)
  print(laws)
```

```
## [1] "Dodd-Frank"     "Glass-Steagall" "Hatch"          "Sarbanes-Oxley"
```

Using `substr()`, substrings or parts of strings can be extracted based on their position. To get the first two letters of each law, we can do the following:

```
substr(laws,1,2)
```

```
## [1] "Do" "Gl" "Ha" "Sa"
```

Or if we would like to obtain the second name in each law, we could use a combination of `regexpr()` and `regmatches()`. Below, we use a more advanced expression (to be covered later in this chapter) to extract text that follows the pattern `"-\\w+"` or hyphen followed by a word.

```
regmatches(laws,regexpr("-\\w+",laws))
```

```
## [1] "-Frank"    "-Steagall" "-Oxley"
```

Lastly, `strsplit()` can be used to create a list of all names that have been used in laws, simply by using the `-` as a separator or delimiter.

```
  strsplit(laws,"-")
```

```
## [[1]]
## [1] "Dodd"  "Frank"
##
## [[2]]
## [1] "Glass"    "Steagall"
##
## [[3]]
## [1] "Hatch"
##
## [[4]]
## [1] "Sarbanes" "Oxley"
```

Syntax

Escaped characters

As touched upon before, there are characters that need to be represented differently in the code in order for one to use them. A backslash makes it possible to pick up on specific characters:

- `\n`: new line
- `\r`: carriage return
- `\'`: single quote when strings are enclosed in single quotes (`'Nay, I can\'t'`)
- `\"`: double quote when strings are enclosed in double quotes

In other cases, double backslashes should be used:

- `\\.`: period. Otherwise, un-escaped periods are used for wildcard matches.
- `\\$`: dollar sign. Otherwise, indicates to match the end of a string.

Position matching

Regex builds in functionality to search for patterns based on position of a substring in a string, such as at the start or end of a string. There are quite a few other position matching patterns, but the following two are the workhorses.

- `$`: End of string
- `^`: Start of string

To demonstrate these patterns, we'll apply `grep()` to three headlines from the BBC.

```
  headlines <- c("May to deliver speech on Brexit", "Pound falls with May's comments", "May: Brexit pla
  print(headlines)
```

```
## [1] "May to deliver speech on Brexit"
## [2] "Pound falls with May's comments"
## [3] "May: Brexit plans to be laid out in new year"
```

```
#Beginning
  grep("^May", headlines, value = TRUE)
```

```
## [1] "May to deliver speech on Brexit"
## [2] "May: Brexit plans to be laid out in new year"
```

```
#End
  grep("Brexit$", headlines, value = TRUE)
```

```
## [1] "May to deliver speech on Brexit"
```

Character Classes

Character classes are particularly helpful with matching values belonging to specific classes of characters. In R, these are heavily relied upon. Below are a few of the most used:

- `[:punct:]`: Punctuation
- `[:alpha:]`: Alphabetic characters
- `[:digit:]`: Numerical values. Can be interchangeably used with `\d` or [0-9]
- `[:alnum:]`: Alphanumeric characters
- `[:space:]`: Spaces such as tabs, carriage returns, etc.
- `[:graph:]`: Human readable characters
- `\\w`: Word characters
- `\\W`: Not word characters
- `\\s`: Space
- `\\S`: Not space

Using the first paragraph from a National Bureau of Economic Research on recessions, what if we replaced certain values based on their character class?

```
para <- "The Business Cycle Dating Committee of the National Bureau of Economic Research met by conferen
  print(para)
```

```
## [1] "The Business Cycle Dating Committee of the National Bureau of Economic Research met by conferenc
```

```
#Remove any punctuation
  gsub("[[:punct:]]", "",para)
```

```
## [1] "The Business Cycle Dating Committee of the National Bureau of Economic Research met by conferenc
```

```
#Mad libs for numbers
  gsub("[[:digit:]]", "__",para)
```

```
## [1] "The Business Cycle Dating Committee of the National Bureau of Economic Research met by conferenc
```

Quantifiers

Sometimes, matching is done on length of patterns. The following allow for setting specific or loose parameters on pattern lengths.

- `{n}`: match pattern n times
- `{n, m}`: match pattern at least n-times, but not more than m times.
- `{n, }`: match at least n times
- `*`: Wildcard, or match at least 0 times
- `+`: Match at least once
- `?`: Match at most once

```
dates <- c("In the year 2010", "In the year 2000", "In the year 50000", "Inauguration Day is  Jan 20, 20

#Match 0 thrice
  grep("0{3}", dates, value = TRUE)
```

```
## [1] "In the year 2000"  "In the year 50000"
```

```
#Match Jan 20, 2017
  grep("[[:alpha:]]{3}\\s+[[:digit:]]{2}\\,\\s[[:digit:]]{4}", dates, value = TRUE)
```

```
## [1] "Inauguration Day is  Jan 20, 2017"
```

```r
#or

grep("\\w+\\s+\\d+\\,\\s\\d{4}", dates, value = TRUE)
```

```
## [1] "Inauguration Day is  Jan 20, 2017"
```

Exercises 2.1

Personally identifiable information or PII is often a barrier to sharing information For the following financial record, anonymize records by removing age and name using `gsub()` and `regmatches()` to extract the amount of money John owes the bank.

```r
statement <- "John (SSN: 012-34-5678) owes $1004 to the bank located at 49-29 Pewter Street."
```

Data Frame Operations

Moving onto the more macro-scale, rows of the data table are analytical units, like countries or voters. Columns are attributes of the units, like GDP or gender. An example of a data table is voter composition by county, with information on population, ethnicity, and economic characteristics. The county is the analytical unit and the metric of population is an attribute. This example data represents a basis data set. Analysis of the data set requires *views* onto the table – pulling out subsets for analysis.

Suppose we wanted to rank all Michigan counties by share of minority voters. Or suppose we wanted to aggregate the counties to find the U.S. state with the highest per capita income. These *views* on the data table require operations to pivot, merge, aggregate, and sort the raw data.

We will review the most powerful functions for data manipulation in the lecture: `sort`, `reshape`, `collapse`, and `merge`. A mastery of these basic operations can yield just about any derived data set from a structured table in `R`.

Indexing and Sorting

Consider a data table with two individuals, indexed by the variable `id`, over two time periods, indexed by the variable `t`. Each individual has a different record for each time period, with observations on income (`income`) and voter preference (`vote`) on a 1-10 scale where higher numbers indicate more progressive voting. Consider the data table stored in variable `X`:

```r
(X <- data.frame(id=c(1,1,2,2), t=c(1,2,1,2), income=c(50,55,101,123), vote=c(8,7,4,3)))
```

```
  id t income vote
1  1 1     50    8
2  1 2     55    7
3  2 1    101    4
4  2 2    123    3
```

Extract second record

```r
X[2, ] # extract second row
```

```
  id t income vote
2  1 2     55    7
```

Extract income for all records

```
X[, 3] # extract third column
```

```
[1]  50  55 101 123
```

```
X[, "income"] # extract column with "income" label
```

```
[1]  50  55 101 123
```

```
X[["income"]] # list syntax to extract column from data frame
```

```
[1]  50  55 101 123
```

```
X$income # compact version of data column manipulation
```

```
[1]  50  55 101 123
```

Extract multiple records

```
print(1:3) # index range
```

```
[1] 1 2 3
```

```
X[1:3, ] # apply the index range to extract rows
```

```
  id t income vote
1  1 1     50    8
2  1 2     55    7
3  2 1    101    4
```

```
X[c(1, 3), ] # specific indices
```

```
  id t income vote
1  1 1     50    8
3  2 1    101    4
```

Extract multiple columns

```
X[ , 3:4] # extract multiple columns
```

```
  income vote
1     50    8
2     55    7
3    101    4
4    123    3
```

```
X[ , c("income", "vote")] # multiple column labels
```

```
  income vote
1     50    8
2     55    7
3    101    4
4    123    3
```

Extract records by attribute value

```
(idx <- X[["income"]] > 50) # indices of records with income greater than 50
```

```
[1] FALSE  TRUE  TRUE  TRUE
```

```
X[idx, ] # select the indices
```

```
  id t income vote
2  1 2     55    7
3  2 1    101    4
4  2 2    123    3
```

Reorder records

```
X[c(2, 4, 3, 1), ] # random ordering, note row labels
```

```
  id t income vote
2  1 2     55    7
4  2 2    123    3
3  2 1    101    4
1  1 1     50    8
```

```
X[c(4, 3, 2, 1), ] # backwards ordering (by force)
```

```
  id t income vote
4  2 2    123    3
3  2 1    101    4
2  1 2     55    7
1  1 1     50    8
```

Ordering and sorting

```
order(X$vote)
```

```
[1] 4 3 2 1
```

```
X[order(X$vote), ] # order records by vote
```

```
  id t income vote
4  2 2    123    3
3  2 1    101    4
2  1 2     55    7
1  1 1     50    8
```

```
X[order(-X$vote), ] # order records by vote, decreasing
```

```
  id t income vote
1  1 1     50    8
2  1 2     55    7
3  2 1    101    4
4  2 2    123    3
```

```
X[order(X$vote, decreasing=TRUE), ] # order records by vote, decreasing
```

```
  id t income vote
1  1 1     50    8
2  1 2     55    7
3  2 1    101    4
4  2 2    123    3
```

Order by multiple columns

```
X[order(X$t, X$vote), ] # order records, first by time period and then by vote
```

```
  id t income vote
3  2 1    101    4
1  1 1     50    8
4  2 2    123    3
2  1 2     55    7
```

```
X[order(X$t, -X$vote), ] # order records, first by time period and then dec. by vote
```

```
  id t income vote
1  1 1     50    8
3  2 1    101    4
2  1 2     55    7
4  2 2    123    3
```

Exercises 2.2

1. Extract records with income greater than 120 in the second time period
2. Extract records with vote greater than 5 and income less than 51
3. Briefly explain the difference between assigning a variable with surrounding parentheses versus no parentheses. How does this behavior relate to the concept of a side effect?

Reshape

There are two basic *shapes* of data, **wide** and **long**. The data table in the previous section was **long**, where each row represents an individual in a separate time period. Row 2 in the original table `X`, for example, was the first individual in the second time period:

```
X[2, c("id", "t")]
```

```
  id t
2  1 2
```

We will find that the **wide** format is also useful. Here, the wide format is every attribute for each individual over time. The analytical unit is the individual, not the individual in each, separate time period. An attribute is no longer just *income* but rather *income in time period*.

```
(wide <- reshape(X, idvar="id", timevar="t", direction="wide"))
```

```
  id income.1 vote.1 income.2 vote.2
1  1       50      8       55      7
3  2      101      4      123      3
```

There are now just two rows, where there were previously four. Returning to the original **long** format is straightforward; but we aren't left with the exact same data table. There are artifacts of the change-in-shape for both the column and row names. Sort of like data manipulation breadcrumbs. This will be cleaned up in the next subsection.

```
(long <- reshape(wide, idvar="id", timevar="t", direction="long"))
```

```
    id t income.1 vote.1
1.1  1 1       50      8
2.1  2 1      101      4
1.2  1 2       55      7
2.2  2 2      123      3
```

Note that there are other frameworks that may be more useful, depending on your application. An especially useful framework to "melt" and "cast" data is the `reshape` package.

Rename row and column headers

The newly assigned column or row names may not match the meaning in the data. It is good practice to maintain the column headers at each stage of analysis, even if you don't immediately use the intermediate data table. Otherwise editing code gets confusing, quickly. The column names are stored in an attribute of the data frame:

```
names(long)
```

```
[1] "id"       "t"        "income.1" "vote.1"
```

Renaming column headers using the built-in, base functions in `R` looks confusing. In words, the following code identifies the positions in `names(long)` where the values are `income.1` and `vote.1`. At the specified positions, the values is reassigned with new values `income` and `vote`, respectively.

```
names(long)[names(long) == "income.1"] <- "income"
names(long)[names(long) == "vote.1"] <- "vote"
print(long)
```

```
    id t income vote
1.1  1 1     50    8
2.1  2 1    101    4
1.2  1 2     55    7
2.2  2 2    123    3
```

This relatively simple example offers an overture to non-core `R` functions. The standard `R` functions are extended through external packages. For example, the `rename()` function is provided through the `plyr` package and abstracts away the list indexing idiosyncracies of renaming columns. It packages the previous reassignments into more readable code:

```
library(plyr)
long <- reshape(wide, idvar="id", timevar="t", direction="long")
rename(long, c("income.1"="income", "vote.1"="vote"))
```

```
    id t income vote
1.1  1 1     50    8
2.1  2 1    101    4
1.2  1 2     55    7
2.2  2 2    123    3
```

Collapse

Reshape and collapse are often used in conjunction in order to calculate summary statistics by group. The built-in `R` functions are reasonably effective; but there are frameworks that are much more powerful. Consider the `reshape` package for more complex data manipulation. To round out the basic concept, consider the aggregate function to collapse functions by group.

Consider this function to collapse the data frame by `id` to create a new data frame with the average vote and income over time.

```
(aggdata <- aggregate(X, by=list(X$id), FUN=mean))
```

```
  Group.1 id   t income vote
1       1  1 1.5   52.5  7.5
2       2  2 1.5  112.0  3.5
```

```r
aggdata[c("id", "income", "vote")]
```

```
  id income vote
1  1   52.5  7.5
2  2  112.0  3.5
```

Exercises 2.3

1. Load the `iris` dataset using the `datasets` package.
2. Calculate the average and maximum sepal length for each Iris species. (**Bonus**: Write the commands without error messages.)
3. Use the `aggregate()` function to count the number of observations of sepal width for each species.
4. (*Difficult*) Create a data frame with the 35th observation of sepal width for each species.

Merge

Create two data frames for illustrative purposes, based on the `state` datasets.

```r
library(datasets)
region <- data.frame(name=state.name, region=state.region)
area <- data.frame(name=state.name, area=state.area)
```

Suppose we want to merge the two datasets, so that we have a single, master dataset with the region and area of the states – a fifty-by-three

```r
full.data <- merge(region, area, by="name")
head(full.data)
```

```
        name region   area
1    Alabama  South  51609
2     Alaska   West 589757
3    Arizona   West 113909
4   Arkansas  South  53104
5 California   West 158693
6   Colorado   West 104247
```

Now we can see the value of the composition of these operations. The following sequence identifies the region of the United States with the greatest land area:

```r
df <- aggregate(full.data$area, by=list(full.data$region), FUN=sum)
df <- rename(df, c("Group.1"="region", "x"="total.area"))
df[order(df$total.area, decreasing=TRUE), ]
```

```
         region total.area
4          West    1783960
2         South     899556
3 North Central     765530
1     Northeast     169353
```

Exercises 2.4

1. Use the `state` datasets to identify the state divisions (e.g., New England) with the highest murder rates in 1976.
2. What was the recorded population in 1977?

An Applied Example

Political scientists and journalists often times count the number of times Congress applauds the President when delivering the State of the Union (SOTU) Address as well as analyzes the number of times words are used. While it is not a clear science, applying data manipulation techniques to create an analyzable dataset can certainly be fascinating. To illustrate a real clean up workflow with some data manipulation, we will use the SOTU transcripts from the Obama Administration. An interesting attribute of the transcripts are that they reflect the number of applause breaks as planned for by the speechwriters and policymakers as opposed to the actual number. Using this data, we will answer the following three questions:

1. How many breaks were planned in 2010 vs 2016?
2. What were the top 10 words used in each of those state of the unions?
3. Which words experienced relatively greater use?

The data is available here.

Be sure to download the data and set the working directory first.

```
setwd("[your directory here]")
```

How many planned applause breaks in 2010 versus 2011?

Looking at the data, the speechwriters included queues for applause as denoted as `(Applause.)`. An example can be seen below:

```
speech <- readLines("sotu_2010.txt")
speech <- speech[speech!=""]
speech[12]
```

```
## [1] "It's because of this spirit -\xd0 this great decency and great strength -\xd0 that I have never
```

Using that piece of information, we can write a relatively short set of steps to match the *Applause* pattern. 57 breaks were planned in 2010, which is 11 more than the 46 breaks planned in 2016. While the same code was run two separate times, we will learn in a subsequent chapter how to automate repetitive tasks.

```
#2010
  #read in lines from the text
  speech10 <- readLines("sotu_2010.txt")

  #remove any blank lines
  speech10 <- speech10[speech10!=""]

  #get string position of each Applause (returns positive values if matched)
  ind <- regexpr("Applause", speech10)
  sum(attr(ind,"match.length")>1)
```

```
## [1] 57
```

```
#2016
  speech16 <- readLines("sotu_2016.txt")
  speech16 <- speech16[speech16!=""]
  ind <- regexpr("Applause", speech16)
  sum(attr(ind,"match.length")>1)
```

```
## [1] 46
```

What were the top words in 2010 vs 2016

To do this, we'll need to do some basic cleaning to start (e.g. remove punctuation, remove numbers, remove non-graphical characters like `\r`), parse the words into a vector of words or 'bag of words', and aggregate words into word counts.

```r
#2010
  #Clean up and standardize values
  clean10 <- gsub("[[:punct:]]","",speech10)
  clean10 <- gsub("[[:digit:]]","",clean10)
  clean10 <- gsub("[^[:graph:]]"," ",clean10)

  #convert into bag of words
  bag10 <- strsplit(clean10," ")
  bag10 <- tolower(trimws(unlist(bag10)))

  #Count the number of times a word shows up
  counts10 <- aggregate(bag10, by=list(bag10), FUN=length)
  colnames(counts10) <- c("word","freq")
  counts10$len <- nchar(as.character(counts10$word))
  counts10 <- counts10[counts10$len>2,]
  counts10 <- counts10[order(-counts10$freq),]
  head(counts10, 10)
```

```
##          word freq len
## 1478      the  338   3
## 76        and  237   3
## 1476     that  149   4
## 1030      our  120   3
## 87   applause  116   8
## 603       for   84   3
## 1649     will   61   4
## 1493     this   60   4
## 91        are   57   3
## 695      have   53   4
```

```r
#2016
  clean16 <- gsub("[[:punct:]]","",speech16)
  clean16 <- gsub("[[:digit:]]","",clean16)
  clean16 <- gsub("[^[:graph:]]"," ",clean16)

  bag16 <- strsplit(clean16," ")
  bag16 <- tolower(trimws(unlist(bag16)))

  counts16 <- aggregate(bag16, by=list(bag16), FUN=length)
  colnames(counts16) <- c("word","freq")
  counts16$len <- nchar(as.character(counts16$word))
  counts16 <- counts16[counts16$len>2,]
  counts16 <- counts16[order(-counts16$freq),]
  head(counts16,10)
```

```
##          word freq len
## 1366      the  284   3
## 57        and  193   3
## 1364     that  146   4
## 935       our   94   3
## 70   applause   89   8
## 530       for   59   3
## 1377     this   41   4
## 1521      who   41   3
## 898       not   40   3
## 182       but   39   3
```

Looking at the words above, it feels a bit unsatisfying. To rectify this, we'll scrape a list of stopwords from lextek.com. To do this, we'll use the library `rvest` to take advantage of the webpage's HTML `<pre>` tags and store the stopwords as a vector `stoplist`.

```
library(rvest)
  #Stopwords
  stop1 <- read_html("http://www.lextek.com/manuals/onix/stopwords1.html")

  stopwords <- stop1 %>%
    html_nodes("pre") %>%
    html_text()
  stoplist <- unlist(strsplit(stopwords,"\n"))
  stoplist <- stoplist[stoplist!="" & nchar(stoplist)>1]
  stoplist <- stoplist[4:length(stoplist)]
```

We then can remove the stopwords list to distill the list to more accurate words.

```
  counts10 <- counts10[!(counts10$word %in% stoplist),]
  counts16 <- counts16[!(counts16$word %in% stoplist),]

  head(counts10,10)
```

```
##             word freq len
## 87      applause  116   8
## 1072      people   32   6
## 71     americans   28   9
## 1477       thats   26   5
## 818         jobs   23   4
## 69       america   18   7
## 70      american   18   8
## 192   businesses   18  10
## 1507        time   18   4
## 560     families   17   8
```

```
  head(counts16,10)
```

```
##            word freq len
## 70     applause   89   8
## 53      america   33   7
## 975      people   27   6
## 1547      world   25   5
## 54     american   22   8
## 210      change   18   6
## 411     economy   16   7
## 55    americans   15   9
## 291     country   12   7
## 551      future   12   6
```

At this point, we've arrived at words that are more presidential sounding, but can still be whittled down to the core message. But that can be left for another day.