Working with Text Data and Basic Text Mining in R

Lecture 6

Kevin McAlister

University of Michigan

April 14, 2017

Processing Text Data in R

Text Data

- We've spent the last couple of weeks discussing scraping data from the Web.
- Working with numeric data is fairly straightforward. Convert the scraped data to a data frame and do everything you would normally do with the new data object.
- Text data is becoming more and more common in the social sciences.
- Working with text data can be painful. What we can do with text data is heavily dependent on language, encoding, and many other variables.
- What if we have a set of texts that we want to analyze? Which documents are similar to one another? Do the documents contain happy or sad tones? Is there a latent variable that dictates the content of documents in our text set?
- These are all questions that we can answer using Text Analysis tools in R.

Text Data

- Today, we're going to work through a simple example of text analysis.
- Let's examine the 69 top selling artists of all time and attempt to group them together based on the intros of their respective Wikipedia pages. (I stopped at 69 because Earth, Wind, and Fire had some weird punctuation and I didn't feel like dealing with that...)
- In order to analyze these texts and find similarities, we'll use the following workflow:
  1. Pull top selling artists of all time using simple web scrape
  2. Using artist names, pull intros for each artist/band
  3. Process intro into consistent format
  4. Push into new folder
  5. Read folder in as corpus object and make term-document matrix
  6. Analyze!

Step 1: Getting Top Selling Artists

- Wikipedia has a page with the top selling artists of all time.
- This link is at the top of the provided R script.
- Look at this page. How is the data shown on the page?
- The fact that this data is contained in a table makes our life much, much easier.
- `bb <- read_html(link)`
- `tabs <- html_table(bb)`
- This function extracts HTML tables.
- There are 6 tables that contain artist names, so we extract the first 6 and concatenate.

Steps 2 and 3: Get the Text and Process

- We're going to combine the next two steps.
- Now that we know the artist names, we can use the Wikipedia API to pull the article intros for each artist.
- For each artist, we start by using `gsub()` to change spaces to %20.
- We then paste the artist's name to the end of the API call.
- Using `fromJSON()`, we can get the info from the API call.
- A list is returned with nested elements. We're after the last element of the second part of the list.

Steps 2 and 3: Get the Text and Process

- This ends up being tricky to extract. Each article has a unique ID that is part of the returned JSON that is not readily apparent to the user.
- To get around this, we use a lazy evaluation trick:

```
text.json <- eval(parse(text=paste("art.json[[2]]$pages$‘",
                names(art.json[[2]]$pages),"‘$extract",sep="")))
```

- `eval()` evaluates an object. If `a <- 1`, then `eval(a)` returns `1`.
- `parse()` converts a character string to an R global env object.
- This process let's us evaluate an object whose name we build as a character string.

Steps 2 and 3: Get the Text and Process

- Now that we have the text, we want to process it into a standard format. In general, this requires removing any HTML tags and programmatic spaces, putting everything in lower case, removing punctuation, and removing extra spaces.
- We can use `gsub()` for most of this. We also utilize HW's `str_replace_all()`, which is a cleaned up version of gsub.
- In the R script, there's a function called `Clean_String()`. This is a function I found online that I've just been adding lines to over time. It does most of the needed cleaning for text files. It takes in a character string and returns a vector with each element and individual word.
- Given the vector of words, we paste them back together for ease of writing and return this new character string to a list element.
- Your processing may require different gsub calls. Use this function as a base and move from that.

Steps 2 and 3: Get the Text and Process

- Text analysis in R is very English-centric.
- If you're working with another language, then the words are going to look weird when processed.
- One way we can prevent this is to change the encoding.
- Encoding is the way in which your computer is able to understand characters. In general, English is encoded as UTF-8. We may want to change this for other languages.
- We can do this in R using `iconv(x,from,to)`. `x` is a character string. From is the current encoding. To is the desired encoding.
- For example, `iconv(x, "UTF-8", "LATIN1")` will change the encoding of `x` from UTF-8 to LATIN1, which is a general encoding for languages in Latin America.
- The documentation on iconv will include all the possibilities.

Step 4: Write Everything to a Corpus Folder

- We now have a list that includes the text from 69 articles.
- The easiest way to work with these articles is to write them out to a new folder where the name of each article is the name of the artist.
- `write(x,fp)` is the most general write function in R. It literally writes whatever is in x to a file. With text, this is what we're after.
- For each article, start by building the file name. We start by taking the first artist name and processing it down to something convenient.
- Then we create the file path where we want to send this article. Use paste.
- Start with the path to the directory that you made for this set of articles. Then paste the new name to the end of the file path.
- Use write to write the article to the designated file path!

Step 5: Get to TDM

- The text processing is really the most difficult part of this process. Now that we have everything in a consistent format, our life gets a lot easier.
- The package tm is a real contribution to the world. It makes text analysis super easy in R.
- tm utilizes the folder that we just made and loads it into R as a collection of documents, i.e. a *Corpus*.
- Once we have associated a set of documents, we can use tm to find the term-document matrix. The TDM for a Corpus is a matrix with documents on the columns and words on the rows. Each element is the number of times a word appeared in a document.
- When using TDMs, we're assuming that documents are nothing more than a *bag of words*. The order in which they appear is unimportant. A word is a word is a word, nothing more.
- Using the TDM, we can look for similarities between documents using math magic.

Step 5: Get to TDM

- tm has lots and lots of functions. One nice supplementary package is slam. slam allows us to do math with the TDMs.
- Let's start by telling R where the corpus is located.
- `Corpus(DirSource(dir.path))` tells R that the Corpus is in the specified directory.
- This loads all of the text data into R. In general, text data is not all that space consuming. However, as with anything, as the number of documents gets very large, we can run into problems with this approach. This is a discussion for another day.
- Once the Corpus is specified, we can start processing the text data using tm.

Step 5: Get to TDM

- The first thing that we want to do is remove meaningless words like "and", "the", "or", etc. These words are going to occur frequently across all documents and keep similar articles from popping in our analysis.
- To remove stopwords, we can use `tm_map(my.corpus, removeWords, stopwords("english"))`. If the document language is different, we can remove that language's stopwords using this function.
- Next, we want to stem the words. This means that "run", "runner", "running", etc. are equivalent. In this case, I don't think this is going to matter all that much, so we'll do it here. However, I strongly recommend running any analyses you do both with and without the stemming.
- We can stem using `tm_map(my.corpus, stemDocument, language = "english")`. Once again, we can stem other languages.

Step 5: Get to TDM

- Finally, we can convert the Corpus to a Term Document Matrix.
- This is really easy using tm: `TermDocumentMatrix(my.corpus)`.
- To look at the TDM, we use `inspect(tdm)`.
- Note that this contains raw frequencies. This is the most basic way to summarize documents.
- Given that all of these articles are about music, there are going to be tons of shared words related to music. These words will have a similar effect on the analysis that stopwords would; knowing that the word "record" appears 50 times probably won't give us much information.
- One way to help with this problem is to use frequency weighting. The most common weighting in TfIdf weighting.
- TfIdf weighting is essentially frequency of word in document $\times$ rarity of word in corpus.
- TfIdf heavily weights high information words. In this case, "Rap", "R&B", "Hall", etc. are going to weight heavily.
- In R, we can weight a TDM by TdIdf using `weightTfIdf(tdm, normalize = TRUE)`.

Step 6: Analyze

- Time for some math magic!
- Our analysis is going to consist of finding pairs of artists that are similar.
- Given two artists, consider the vectors formed by their respective columns in the TDM (a,b).
- If the two vectors are really close to one another, then we can reasonably assume that the two artists are similar.
- How can we tell if two vectors are similar mathematically? What metric can we use to determine this?
- One way to think about this is the angle created by the two vectors. A small angle (0) means two vectors are highly similar. A large angle (180) means two vectors are highly dissimilar.

Step 6: Analyze

- Recall the Euclidian dot product formula:

$$a \cdot b = \|a\|\|b\|cos(\theta)$$

- With some rearrangement, we get:

$$cos(\theta) = \frac{a \cdot b}{\|a\|\|b\|}$$

- In sum notation:

$$cos(\theta) = \frac{\sum a_i b_i}{\sqrt{\sum a_i^2}\sqrt{\sum b_i^2}}$$

- When the angle is small, $cos(\theta)$ is close to 1. When the angle is large, $cos(\theta)$ is close to -1.
- This measure is called *cosine similarity*.
- In general, we expect that two documents won't have a negative similarity. English is pretty structured, so there is going to be some similarity.

Step 6: Analyze

- Given a TDM, T, with n documents, the numerator is $T'T$.
- Let C be the $1 \times n$ matrix of column sum of squares. Then the denominator is $C'C$.
- In R, we can calculate this using some functions from `library(slam)`. See the related section in the R code.
- Now we have the similarity matrix. Each element gives the pairwise similarity between the row and column artist. An article compared to itself has a cosine similarity of 1.
- However, this is a big matrix. It's hard to see what artists are similar.
- To deal with this, we want to melt the matrix.

Step 6: Analyze

- Given a matrix, the melted matrix has a row for each element of the matrix. Each element is indexed by its row name and column name. In this case, we'll get a matrix with each pairwise artist comparison and its similarity.
- The matrix melt is included in HW's `library(reshape2)`.
- We can melt the matrix using `melt(x)`.
- After we melt our similarity matrix, we remove all rows with similarity equal to 1. Then we sort by similarity. The top rows include the most similar artists.
- Do the most similar artists make sense? Now look at the least similar artists. Does this make sense?

Concluding Remarks

- This analysis method is simple and finds broad similarities. But, it leaves room for improvement.
- We can do better. One way is to pull full articles. More information means better matches.
- We can also use better methods of analysis. Latent Dirichlet Analysis is really good for finding similarities using latent variable methods.
- I've posted a pretty cool talk I heard about LDA on Canvas.
- Text analysis is a hot area in data analysis right now. It's something that is not a fad, so it's something we should all be familiar with.
- R provides many tools for making text analysis easy. However, it does lack when we move to more complicated ideas of documents. For example, if we want to move from single words to phrases, R is currently unable to handle the analysis.
- Python shines with its text analysis libraries. If this is something important to you, I'd recommend learning some Python.