Text Processing in R

Overview

This tutorial goes over some basic concepts and commands for text processing in R. R is not the only way to process text, nor is it really the best way. Python is the de-facto programming language for processing text, with a lot of builtin functionality that makes it easy to use, and pretty fast, as well as a number of very mature and full featured packages such as NLTK and textblob. Basic shell scripting can also be many orders of magnitude faster for processing extremely large text corpora -- for a classic reference see Unix for Poets. Yet there are good reasons to want to use R for text processing, namely that we can do it, and that we can fit it in with the rest of our analyses. I primarily make use of the `stringr` package for the following tutorial, so you will want to install it:

```
install.packages("stringr", dependencies = TRUE)
library(stringr)
```

I have also had success linking a number of text processing libraries written in other languages up to R (although covering how to do this is beyond the scope of this tutorial). Here are links to my two favorite libraries:

- The Stanford CoreNLP libraries do a whole bunch of awesome things including tokenization and part-of-speech tagging. They are much faster than the implementation in the `OpenNLP` R package.
- MALLET does a whole bunch of useful statistical analysis of text, including an extremely fast implementation of LDA. You can check out examples here, but download it from the first link above.

Regular Expressions

Regular expressions are a way of specifying rules that describe a class of strings (for example -- every word that starts with the letter "a") that are more succinct and general than simply specifying a dictionary and checking against every possible value that meets some rule. You can start by checking out this link to an overview of regular expressions, and then take a look at this primer on using regular expressions in R. What is important to understand is that they can be far more powerful than simple string matching.

If you want to get started using regular expressions, you can check out many of the tutorials posted above, but I have also found it very helpful to just start trying out examples and seeing how they work. One simple way to do this si to use an online app with a graphical interface that

Text Processing in R

which are quite common and are used by some R packages. Whichever program you choose, I would suggest just messing around and reading random articles on the internet for a few hours before you get started using Regular Expressions in R. I also tend to use one of these programs to prototype any complex RegEx I want to use in production code.

Example Commands

Lets start with an example string.

```
my_string <- "Example STRING, with numbers (12, 15 and also 10.2)?!"
```

First we can lowercase the entire string -- often a good starting place. This will prevent any future string matching from treating "Table" and "table" as distinct words, for example, just because one came at the beginning of a sentence:

```
lower_string <- tolower(my_string)
```

We can also take a second string and paste it on the end of the first string:

```
second_string <- "Wow, two sentences."
my_string <- paste(my_string,second_string,sep = " ")
```

Now we might want to split out string up into a number of strings, we can do this by using the `str_split()` function, available as part of the stringr R package. The following line will split the combined string from above on exclamation points:

```
my_string_vector <- str_split(my_string, "!")[[1]]
```

Notice that the splitting character gets deleted, but we are now left with essentially two sentences, each stored as a separate string. Note that a list object is returned, so to access the actual vector containing the split strings, we need to use the list operator and get the first entry.

Now, lets imagine we are interested in sentences that contain questions marks. We can search for the string in the resulting `my_string_vector` that contains a ? by using the grep() command.

```
grep("\\?",my_string_vector)
```

Text Processing in R

to escape it with a "\". However, due to the way that strings get passed in to the underlying function, we actually need a second "\" to ensure that one of them is present when the input is provided to c. You will get the hang of this with practice, but amy want to check out this list of special characters that need to be escaped to make them literal. We may also want to check if any individual string in `my_string_vector` contains a question mark. This can be very useful for conditional statements -- for example, if we are processing lines of a webpage, we may want to handle lines with header tags `<h1>` differently than those without header tags, so using a conditional statement with the logical grep, `grepl()`, may be very useful to us. Lets look at an example:

```
grepl("\\?",my_string_vector[1])
```

There are two other very useful functions that I use quite frequently. The first replaces all instances of some characters with another character. We can do this with the `str_replace_all()` function, which is detailed below:

```
str_replace_all(my_string, "e","___")
```

Another thing I do all the time is extract all numbers from a string using the `str_extract_all()` function:

```
str_extract_all(my_string,"[0-9]+")
```

note here that we used out first real regex -- `[0-9]+` which translates to "match any substring that is one or more contiguous numbers". These are just a few of the many more complicated commands available to process text in R. I have also only shown you some very simple Regular Expressions. There is so much to learn in this domain that it can feel overwhelming when you are starting out so I would suggest starting by using these tools and then Googling to expand your abilities as you need to deal with more complicated chunks of text or text processing tasks.

Cleaning Text

One of the most common things we might want to do is read in and clean a raw input text file. To do this, we will want to make use of two functions, the first of these will clean and individual string, removing any characters that are not letters, lowercasing everything, and getting rid of additional spaces between words before tokenizing the resulting text and returning a vector of individual words:

Text Processing in R

```
    #' Remove everything that is not a number or letter (may want to keep more
    #' stuff in your actual analyses).
    temp <- stringr::str_replace_all(temp,"[^a-zA-Z\\s]", " ")
    # Shrink down to just one white space
    temp <- stringr::str_replace_all(temp,"[\\s]+", " ")
    # Split it
    temp <- stringr::str_split(temp, " ")[[1]]
    # Get rid of trailing "" if necessary
    indexes <- which(temp == "")
    if(length(indexes) > 0){
      temp <- temp[-indexes]
    }
    return(temp)
}
```

Lets give this function a try by running the bit of code about in the console (thus defining the function), and then cleaning and tokenizing the following sentence:

```
sentence <- "The term 'data science' (originally used interchangeably with 'datalogy') has existed f
clean_sentence <- Clean_String(sentence)
> print(clean_sentence)
 [1] "the"              "term"             "data"
 [4] "science"          "originally"       "used"
 [7] "interchangeably"  "with"             "datalogy"
[10] "has"              "existed"          "for"
[13] "over"             "thirty"           "years"
[16] "and"              "was"              "used"
[19] "initially"        "as"               "a"
[22] "substitute"       "for"              "computer"
[25] "science"          "by"               "peter"
[28] "naur"             "in"
```

As we can see, all of the special characters have been removed and we are left with a well-behaved vector of individual words. Now we will want to scale this up to working on an entire input document, to do so, we will want to loop over the input lines of that document and in addition to returning the cleaned text itself, we may also want to return some useful metadata like the total number of tokens, or the set of unique tokens. We can do so using the following function:

```
#' function to clean text
Clean_Text_Block <- function(text){
    if(length(text) <= 1){
        # Check to see if there is any text at all with another conditional
        if(length(text) == 0){
            cat("There was no text in this document! \n")
```

Text Processing in R

```
            clean_text <- Clean_String(text)
            num_tok <- length(clean_text)
            num_uniq <- length(unique(clean_text))
            to_return <- list(num_tokens = num_tok, unique_tokens = num_uniq, text = clean_text)
        }
    }else{
        # Get rid of blank lines
        indexes <- which(text == "")
        if(length(indexes) > 0){
            text <- text[-indexes]
        }
        # Loop through the lines in the text and use the append() function to
        clean_text <- Clean_String(text[1])
        for(i in 2:length(text)){
            # add them to a vector
            clean_text <- append(clean_text,Clean_String(text[i]))
        }
        # Calculate the number of tokens and unique tokens and return them in a
        # named list object.
        num_tok <- length(clean_text)
        num_uniq <- length(unique(clean_text))
        to_return <- list(num_tokens = num_tok, unique_tokens = num_uniq, text = clean_text)
    }
    return(to_return)
}
```

Now lets give it a try. You can download the plain text of a speech given by Barak Obama on February 24, 2009 to a joint session of Congress from the University of Virginia Miller Center Presidential Speech Archive by **clicking the link here**. Once you have save this file, you will want to set your working directory in R to the folder where you saved it and then read it in to R using the following lines of code:

```
con <- file("Obama_Speech_2-24-09.txt", "r", blocking = FALSE)
text <- readLines(con)
close(con)
```

You can now run it through the `Clean_Text_Block()` function and then take a look at the output:

```
clean_speech <- Clean_Text_Block(text)
> str(clean_speech)
List of 3
 $ num_tokens   : int 6146
 $ unique_tokens: int 1460
 $ text         : chr [1:6146] "madam" "speaker" "mr" "vice" ...
```

We can see that there are a total of 6146 words in the document, with 1460 of them being unique. You are now past one of the biggest hurdles in text analysis, getting your data into R

Text Processing in R

Generating A Document-Term Matrix

One of the things we will want to do most often for social science analyses of text data is generate a document-term matrix. This is actually a relatively challenging programming task, and it is also usually very computationally intensive, so I will be using functions written in C++ in order to accomplish this task. Before going any further, I suggest you check out my tutorial Using C++ and R code Together with Rcpp to get the basics of C++ programming under your belt. You may also need to follow some of the steps at the beginning of this tutorial before you will even be able to install the Rcpp package and get it working, especially if you are using Windows or a certain versions of Mac OS X. Before you go any further, you will want to make sure you have the following packages installed:

```
install.packages("Rcpp",dependencies = T)
install.packages("RcppArmadillo",dependencies = T)
install.packages("BH",dependencies = T)
```

The BH package is not essential for sourcing the function below, but it is a good idea to have installed for use with future C++ functions. Now, lets take a look at a C++ function that will help us generate a document term matrix:

```
#include <RcppArmadillo.h>
//[[Rcpp::depends(RcppArmadillo)]]
using namespace Rcpp;

// [[Rcpp::export]]
arma::mat Generate_Document_Word_Matrix(int number_of_docs,
                                        int number_of_unique_words,
                                        std::vector<std::string> unique_words,
                                        List Document_Words,
                                        arma::vec Document_Lengths
                                        ){
    arma::mat document_word_matrix = arma::zeros(number_of_docs,number_of_unique_words);

    for(int n = 0; n < number_of_docs; ++n){
        Rcpp::Rcout << "Current Document: " << n << std::endl;
        std::vector<std::string> current = Document_Words[n];
        int length = Document_Lengths[n];
        for(int i = 0; i < length; ++i){
            int already = 0;
            int counter = 0;
            while(already == 0){
                if(counter == number_of_unique_words ){
                    already = 1;
                }else{
                    if(unique_words[counter] == current[i]){
```

Text Processing in R

```
                    counter +=1;
                }
            }
        }
    }
    return document_word_matrix;
}
```

You can download the source file for the c++ code you see above by **clicking the link here**. Once you have saved the file somewhere where you can access it (the example code below assumes it is in your working directory), you can now source the code which will give you access to an R function that has C++ code under the hood.

```
Rcpp::sourceCpp('Generate_Document_Word_Matrix.cpp')
```

You can now, use the function, lets try it out on a toy example. The first thing we will want to do is get a second document so we can make a document term matrix that is more than just one document long. You can download another Obama Speech (this time his 2010 state of the union) by **clicking the link here**. We can now read in an tokenize this piece of text as follows:

```
# Read in the file
con <- file("Obama_Speech_1-27-10.txt", "r", blocking = FALSE)
text2 <- readLines(con)
close(con)

# Clean and tokenize the text
clean_speech2 <- Clean_Text_Block(text2)
```

Now we are ready to set things up and use our document word matrix generator function:

```
#' Create a list containing a vector of tokens in each document for each
#' document. These can be extracted from the cleaned text objects as follows.
doc_list <- list(clean_speech$text,clean_speech2$text)

#' Create a vector of document lengths (in tokens)
doc_lengths <- c(clean_speech$num_tokens,clean_speech2$num_tokens)

#' Generate a vector containing the unique tokens across all documents.
unique_words <- unique(c(clean_speech$text,clean_speech2$text))

#' The number of unique tokens across all documents
n_unique_words <- length(unique_words)

#' The number of documents we are dealing with.
ndoc <- 2
```

Text Processing in R

```
                        number_of_docs = ndoc,
                        number_of_unique_words = n_unique_words,
                        unique_words = unique_words,
                        Document_Words = doc_list,
                        Document_Lengths = doc_lengths
                        )

#' Make sure to add column names to you Doc-Term matrix, then take a look!
colnames(Doc_Term_Matrix) <- unique_words
```

Once we have generated this matrix, we can use it for all sorts of analyses from statistical topic models like LDA (using the `topicmodels` package in R) to just including the counts of them in a regression model.

Using Existing Libraries for Text Processing

This tutorial has primarily focussed on illustrating some very basic tools and under the hood functionality necessary to generate a document term matrix. However, there are easier ways to do this. One of the most full function packages for doing text processing (including in multiple languages) in R is the Quanteda package. If we want to use the package, we will first have to install it:

```
install.packages("quanteda",dependencies = T)
```

Now lets say we want to work with the same two speeches from the previous example. We can generate a document term matrix using the following snippet of code:

```
#' Create a vector with one string per document:
#' document. These can be extracted from the cleaned text objects as follows.
docs <- c(paste0(text,collapse = " "),paste0(text2,collapse = " "))

#' load the package and generate the document-term matrix
require(quanteda)
doc_term_matrix <- dfm(docs, stem = FALSE)

#' find the additional terms captured by quanteda
missing <- doc_term_matrix@Dimnames$features %in% colnames(Doc_Term_Matrix)

#' We can see that our document term matrix now includes terms with - and ' included.
doc_term_matrix@Dimnames$features[which(missing == 0)]
```

This is certainly easier and more efficient than writing the code yourself. In general using Quanteda to generate document-term matrices (along with a lot of other functionality for the package is that it automatically stores text data as sparse matrix objects, which tends to be enormously more space efficient than using dense matrices. However, at this time, Quanteda does require that the user be able to load all documents they wish to process into a document-term matrix into their R session at the same time. This is often not a big deal, but for some very large scale text processing applications, it may be advantageous to work out-of-memory, processing data in chunks.

If you are interested in working with the Stanford CoreNLP and MALLET libraries from R , I have a (beta) R package that wraps these libraries, along with providing a number of utility and document comparison functions. This package is meant to serve as a complement to the Quanteda package, and may be a good option if the user is interested in heavy NLP applications in R, or working with an obscenely large corpus that they cannot fit in memory. The package is available on GitHub here: https://github.com/matthewjdenny/SpeedReader.

Thank you for checking out this tutorial, and please shoot me an email if you are interested in any new inclusions. If you are interested in trying to run the R code from this tutorial you can download the .R file here.