Final Report

Brian Perron

October 10, 2014

Overview

This report is submitted in partial fulfillement of the Data Science Specialization Capstone. The focus of the capstone project involves applying data science to the area of Natural Language Processing. As part of the capstone, students were expected to 1) Develop a predictive text model that is tested on a real data set; 2) Create a reproducible R markdown document describing your model building process; and 3) Build a Shiny or Yhat application to demonstrate the use of the product.

This report the final predictive text model that I developed for the course. The predictive model is built on an n-gram probability model and implemented as a Shiny app. The predictive text model is designed to predict a word based on either one or two preceding words. The predictive model is then extended by generating an entire sentence, using a loop structure that takes predicted words, and then making them predictors of subsequent words to form a sentence.

One key learning point of this project is knowing how the underlying data can result in fundamentally different predictions. To help reveal this point, I have taken the same n-gram prediction model and trained it using four unique data sources. The Shiny app allows the user to see how similar words or combination of words result in much different predictions, which is apparent when generating complete sentences.

This report is divided into two parts. The first part focuses on the acquisition and cleaning of the data. The second part describes the n-gram probability model on which text predictions are based. The third part of the report describes the Shiny app that is used to present the model and the features of the underlying data.

Data Preparation

This study relied on text data from a corpus called NC Corpora from www.corpora.heliohost.org. The specific data set I selected from this data source was news reports. I selected this data set, as opposed to merging it with the Twitter and blog files, to help improve predictions. More specifically, news reports seem to have more consistency in their language style – e.g., formal tone, whereas the communication in tweets and blogs are much more informal. And, tweets often contain many abbreviations that make both cleaning and prediction very difficult. For this project, I relied on 30,000 unique news articles randomly selected from the original corpus. The following code snippet shows my strategy for random selection:

```r
newsFULL <- readLines("~/Git/capstoneCoursera/en_US/en_US.news.txt")
a <- sample(c(1:length(tweetsFULL)), 20000, replace = FALSE)
news <- newsFULL[a]
```

To facilitate the cleaning of the data files, I created a simple function to serve as a wrapper for a number of procedures. My rules for cleaning and pre-processing followed the general procedures in Natural Language Processing research. Provided below are some snippets of code to serve as an example of how I prepared the data. The full set of code is available on GitHub. Given the limitations on computer memory and speed, it was necessary that I reduced the data file substantially. It should be noted that I did not exclude any profanity from the predictions. I believed profanity is part of the 'natural language,' even though it may not be appropriate across different occasions. However, in the extensive testing of the model, I was not able to predict a single profane word.

```
dataCleaner.f <-function(x){
text <- x

#Sample cleaning procedure.  Full cleaning procedure available on GitHub repository
cleaner.f <- function(x, print=FALSE){
     x <- lapply(x, function(row) iconv(row, "latin1", "ASCII", sub=""))
     x <- tolower(x);
     x <- sub("^\\s+", "", x)
     x <- removeNumbers(x); x <- removePunctuation(x)
     x <- sub("\\<rt\\>", "", x); x <- stripWhitespace(x)
     return(unlist(x))}

# Here I am attempting to break paragraphs into sentences, but I am losing the fullstop.
text <- unlist(strsplit(text, "\\."))

# This code here eliminates the white space at the very beginning and end of the sentence.
text <- str_trim(text)

#I need to add periods to the end of each sentence. Strategy - write a function
#that counts the number of characters in each line.  Then, add a period to the nchar+1 position.
text <- paste(text, ".", sep="")
x <<- text  # This returns the cleaned file to be accessed in the global environment.
}
```

Four separate data files were created to hold uni-grams, bi-grams, tri-grams, and 4-grams. I chose to pre-process these datafiles as opposed to having them constructed by the app. This decision was made to increase the speed of the app.

N-gram Probability Model

A set of prediction functions were informed by the common n-gram probability models in the field of Natural Language processing. This involved computing probabilities for each n-gram based on the chain rule. The chain rule in its general form is:

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})}$$

The search algorithm accepts up to three words as input from the user, and then seraches for the most probable word based on this algorithm.

Backoff Strategy

To improve the performance of the model, particularly with unseen n-grams, the algorithm relies on a back-off method. More specifically, the algorithm first searches for the most probably 4-gram based on the three words. If the algorithm doesn't find a match for the 4-gram, it "backs-off" to search for a tri-gram. The back-off involves using the last two words of the sequence (as opposed to all three) and searching for the most probable tri-gram. If a match is found, the tri-gram is returned. If not, then only the last word of the user's input is used to search for the most probably bigram. Finally, if a bi-gram isn't found, the algorithm simply returns the most frequently occuring unigram. Provided below shows the algorithm.

```r
newsBi.f <- function(x){
  lower.x <- tolower(x)
  a <- grep(paste0(lower.x, '$'), newsBigram.df[,"token.1"])
  bigram.a <- newsBigram.df[a,] # Subset data
  bigram.a.ordered <- bigram.a[with(bigram.a, order(-biCount)),]
  b <- as.character(bigram.a$token.2[1]) #select bigram
  top.unigrams <- newsUnigram.df[1:5, 2]
  ifelse(is.na(b), return(sample(top.unigrams, 1)), return(b))
}

newsTri.f <- function (x){
  lower.x <- tolower(x)
  c <- colsplit(lower.x, " ", c("t1", "t2"))
  a <- grep(paste0(lower.x, '$'), newsTrigram.df[,"token.1.2"])
  trigram.a <- newsTrigram.df[a,] # Subset data
  trigram.a.ordered <- trigram.a[with(trigram.a, order(-triCount)),]
  ifelse(length(trigram.a.ordered$triCount) == 0, return(newsBi.f(c$t2)),
         return(as.character(trigram.a.ordered$token.3[1])))
}

newsQuad.f <- function (x){
  lower.x <- tolower(x)
  c <- colsplit(lower.x, " ", c("t1", "t2"))
  a <- grep(paste0(lower.x, '$'), newsQuadgram.df[,"token.1.2.3"])
  quadgram.a <- newsQuadgram.df[a,] # Subset data
  quadgram.a.ordered <- quadgram.a[with(quadgram.a, order(-quadCount)),]
  ifelse(length(quadgram.a.ordered$quadCount) == 0,
         return(newsTri.f(c$t2)), return(as.character(quadgram.a.ordered$token.4[1])))
}
```

From the aforementioned code, it was easy to obtain a predicted word. The following code takes the predicted word and builds a full sentence. t operates in the same fashion as the word prediction, allowing the user to enter up to three words and predicting the next word. However, the algorithm takes the predicted word and then appends it to the original sequence of words that were input by the user. Then, the algorithm takes the last three words to predict the next word. This continues until the algorithm predicts a period. Please note that this model is much slower than the single word prediction!

```r
newsSb <- function(x){

  w3 <- newsQuad.f(x)

  #Make a sentence by taking the predicted word and paste it on the end of the sentence
  sentence <- paste(x, w3, sep = " ")
  sentence.end <- length(grep("\\.", sentence))

  while(sentence.end == 0)
  {
    #Split the sentence into tokens so the last set of bigrams can be used as the
    #input for predicting a new   word
    tokenize <- unlist(strsplit(sentence, " "))

    #Grab the last two words and save them in a vector
    a <- tail(tokenize, 2)
    #Paste the words together with a space
    b <- paste(a[1], a[2], sep = " ")

    #Take the last two words and use that as an input
    c <- newsQuad.f(b)

    #Paste the results to the end of the sentence
    sentence <- paste(sentence, c, sep = " ")
    sentence.end <- length(grep("\\.", sentence))
  }

  #sentence
  cap.first.letter <- toupper(unlist(strsplit(sentence, ''))[1])
  sentence.no.first.letter <- paste0(strsplit(sentence, '')[[1]][-1], collapse='')
  formatted.sentence <- paste(cap.first.letter, sentence.no.first.letter, sep = '')
  formatted.sentence
}
```

Shiny App

The final phase of this process involved developing a web application to demonstrate the prediction algorithm. I used the Shiny App system provided by RStudio. The process of building the app was straight forward, involving the construction of a user-interface `ui.R` and server commands `server.R`. I created one additional file `global.R` to help organize some code that has to be run only upon launch. For the user interface, I have three separate tabs. The first tab provides a basic overview of the model, providing the user with some basic background on its functionality. The second tab demonstrates a word prediction, and the third tab demonstrates the word builder. The app can be accessed at: https://beperron.shinyapps.io/nGram/

Strengths, Limitations, and Future Directions

This was a very challenging assignment, given that I am still new to R programming. Thus, I find it a success that I was able to build a working model and demonstrate it online, even though my underlying code may not conform to best practices. Also, I have incorporated a basic back-off strategy for this model. But, I recognize a limitation is that I did not effectively incorporate a smoothing or discounting procedure that is recommended by a variety of sources. One part of my model is not working exactly as I would like – that is, I am not able to figure out how to suppress the initial loading of the data, which results in an initial return that was not called by the user. However, after the initial load, the app functions as expected. Improving my code and gaining a better understanding of langauge modeling are my next steps. I am also especially interested in building Shiny Apps for my work-related activities.