Capstone - Word Prediction

Suresh Subramaniam

March 7, 2018

Next Word Prediction - Models and Observations

The goal of the project is to build a word prediction model using text mining and NLP.

The Data

The data provided is in the form of 3 test files, available at https://d396qusza40orc.cloudfront.net/dsscapstone/dataset/Coursera-SwiftKey.zip (https://d396qusza40orc.cloudfront.net/dsscapstone/dataset/Coursera-SwiftKey.zip)

It consists of 3 files as mentioned below

- en_US.blogs.txt (Size: 205mb, # Words: 37865888, #Characters: 163325412)
- en_US.news.txt (Size: 200mb, # Words: 2665742, #Characters: 12502954)
- en_US.twitter.txt (Size: 163mb, #Words: 30578933 #Characters: 125769474)

These are obviously large datasets. Hence a 10% random sample was extracted to train the model and a small sample was used to test the model.

Some exploratory data analysis was conducted on this data to derive some basic understanding and the report is available at http://rpubs.com/joresh/363337 (http://rpubs.com/joresh/363337)

The Approach

1. The model consists of n-grams derived from the text files above and various methods to derive the next predicted word from them.
2. 10% of the text from each of the 3 text files above was extracted using random sampling without replacement. The samples from all three files were combined into one text file. The function for sampling a text file is below

```
################################
#Function for sampling a text file
################################
sampleText <- function(textfile, samplePercentage) {
    textfile <- readLines(con=textfile, skipNul=TRUE)
    samples <- sample(textfile[-1], samplePercentage*length(textfile)/100, replace=FALSE)
    sampleText <- paste(samples, collapse = ' ')
    return (sampleText)
}
```

If a text file name and the sample percentage are supplied as arguments, the function returns a sample of the defined size.

3. Data cleaning processes were performed on this text. The punctuations were removed, extra white spaces were stripped, numbers were removed and all characters were converted to lower case.

```
#####################################
#Function for creating a tidy data frame of the text file
#####################################

createDataframe <- function(textfile) {
    library(tm)
    library(NLP)
    library(tidytext)

    txt <- tolower(textfile)
    txt <- removeNumbers(txt)
    txt <- stripWhitespace(txt)
    txt <- removePunctuation(txt)
    txt <- tidy(txt)
    names(txt) <- 'text'
    return (txt)
}
```

If a text file is supplied to the function, it cleans it as described above and return a data frame object.

4. Five orders of n-grams from 1-gram to 5-gram were created and all of them were appended to form one data frame of n-grams. The total number of rows in this data frame was about 29 million.

```
#Function to create ngrams

createNgrams <- function(df, n=5) {

    library(tidytext)
    library(dplyr)
    library(data.table)
    allNgrams <- data.frame('Var1'= integer(), 'Freq'=integer(), 'ngrams'=integer())
    for ( i in 1:n) {
        #Create the ngram
        ngram <- unnest_tokens(df, words, text, token='ngrams', n=i)
        #Convert to frequency table and sort on Frequency
        ngramFreq <- as.data.frame(table(ngram$words))
        ngramFreq <- arrange(ngramFreq, desc(Freq))
        ngramFreq$ngram <- i
        allNgrams <- rbind(allNgrams, ngramFreq)
    }

    #Split into string and last word
    allNgrams$Var1 <- as.character(allNgrams$Var1)
    allNgrams$main <- lapply(allNgrams$Var1,function(x) {paste(head(strsplit(x,split=" ")[[1]],-
1), collapse= ' ')})
    allNgrams$last <- lapply(allNgrams$Var1,function(x) {paste(tail(strsplit(x,split=" ")[[1]],1
), collapse= ' ')})
    allNgrams <- allNgrams[,c('main', 'last', 'Freq', 'ngram')]
    names(allNgrams) <- c('main', 'nextword', 'freq', 'ngram')
    #Convert to data table
    allNgrams <- as.data.table(allNgrams)
    allNgrams <- arrange(allNgrams, desc(ngram), desc(freq))
    return (allNgrams)
}
```

The function accepts the data frame object created by tidy() and the number of ngrams to create. The default is 5. The ngram is then split into two strings - the last word and the rest of the words. The output is one consolidated datatable with the following columns, sorted on the descending order of ngram and frequency within the ngram.

- main : rest of the words
- nextword : last word in the ngram
- freq : the number of times the ngram occurs in the text
- ngram : the order of the ngram

This output data table was further pruned to remove unnecessary rows.

```
##########################################################
##Function for pruning the ngrams dataset
##########################################################
pruneNgrams <- function(ngramsDf)  {  #the ngrams as created by createNgrams() function and then
 combined
    library(data.table)
    #convert df to data table and order by main and freq
    sampleDT <- as.data.table(ngramsDf)
    sampleDT$main <- as.character(sampleDT$main)
    sampleDT$nextword < as.character(sampleDT$nextword)
    sampleDT <- setorder(sampleDT, main, -freq)

    # extract only first row with the highest frequency from each unique 'main' strings
    setkey(sampleDT, main)
    sampleDTpruned <- sampleDT[J(unique(main)), mult = "first"]
    #remove unecessary columns to reduce file size and set keys
    ngrams <- sampleDTpruned[,c('main', 'nextword', 'ngram')]
    setkey(ngrams, main, ngram)
    ngrams$nextword <- as.character(ngrams$nextword)
    return(ngrams)
}
```

For each duplicate n-gram string, only the row with the maximum frequency was retained. All the others were deleted. The reasoning was that, when we are choosing a row to get the next word, we would anyway choose the row with the maxmimum probabilty, i.e. the maximum frequency. Therefore, all the other rows for the same n-gram string are redundant. This brough down the number of rows to around 19 million.

After this, the rows with frequency of 1 was removed. This brought down the number of rows to around 600K. These are sparse entries and removing them could affect the accuracy sligtly but the reduction in number of rows is enormous. We are sacrificing accuracy for performaance gain. After this step, the column ‘freq’ was deleted to reduce the file size even further so that the loading time of the file is reduced even further. This improves user experience.

5. The first version of the function to predict the next word was not using the stupid backoff model. The function finds all rows in the data frame for which the column ‘main’ is equal to the string supplied or a part of the string, and then selects the next word from the rows that have the highest frequency (except the unigram). Two versions of the function was created - one using a dataframe and one using data table. The data table version of the function is below.

```
########################################
###Function to predict word using all finds of the string using data table
########################################


predictNextWord2DT <- function(df, string) {
  library(dplyr)
  library(data.table)
  #df <- as.data.table(df)

  #create empty df
  shortlistDf <- df[NULL,]

  #Pad spaces in string to make it 4 words
  split <- strsplit(string, ' ')[[1]]
  len <- length(split)
  if ( len > 4) string <- split[ (len-3) : len]
  if (len == 3) string <- paste('xxxxx', string)
  if (len == 2) string <- paste('xxxxx', 'xxxxx', string)
  if (len == 1) string <- paste('xxxxx', 'xxxxx', 'xxxxx', string)

  #Split string to 1,2,3,4 word strings
  s4 <- paste(string, collapse= ' ')
  print(paste("s4 :", s4))
  s3 <- paste(strsplit(s4, ' ')[[1]][-1], collapse= ' ')
  print(paste("s3 :", s3))
  s2 <- paste(strsplit(s3, ' ')[[1]][-1], collapse= ' ')
  print(paste("s2 :", s2))
  s1 <- paste(strsplit(s2, ' ')[[1]][-1], collapse= ' ')
  print(paste("s1 :", s1))

  #Search for strings in ngrams and add to df
  dfsub <- df[main == s4 & ngram == 5,]
  if (nrow(dfsub) != 0) shortlistDf <- rbind(shortlistDf, head(dfsub,1))
  dfsub <- df[main == s3 & ngram==4,]
  if (nrow(dfsub) != 0) shortlistDf <- rbind(shortlistDf, head(dfsub,1))
  dfsub <- df[main == s2 & ngram==3,]
  if (nrow(dfsub) != 0) shortlistDf <- rbind(shortlistDf, head(dfsub,1))
  dfsub <- df[main == s1 & ngram==2,]
  if (nrow(dfsub) != 0) shortlistDf <- rbind(shortlistDf, head(dfsub,1))
  dfsub <- df[ngram==1,]
  shortlistDf <- rbind(shortlistDf, head(dfsub,1))

  shortlistDf <- arrange(shortlistDf, desc(probability))
  if (nrow(shortlistDf) == 0) return('No find!')
  if (nrow(shortlistDf) == 1) return(unlist(head(shortlistDf,1)[,'nextword']))
  if (nrow(shortlistDf) > 1) return(unlist(shortlistDf[2,'nextword']))

}
```

6. The second version of the function to predict the next word was using the stupid backoff model. The function starts with the highest order ngram and searches for the string in column ‘main’ with number of words = ngram order - 1. If it find the string, the search stops and the next word is returned. If not, the search

proceeds to the next lower order ngram, and so on. If the string or its parts are not found in any of the ngrams, the unigram with the highest frequency is returned. Two versions of the function was created - one for using a dataframe and one for using data table.

```
########################################
##Function using backoff and data table
################################################
predictNextWord <- function(df, string) {

  library(dplyr)
  library(data.table)
  library(tm)
  #The input df is the pruned and reduced data table with columns main, nextword and ngram
  #the key should be set as main + ngram for the data table
  string <- tolower(string)
  split <- strsplit(string, ' ')[[1]]
  len <- length(split)
  if ( len > 4) string <- split[ (len-3) : len]
  if (len == 3) string <- paste('xxxxx', string)
  if (len == 2) string <- paste('xxxxx', 'xxxxx', string)
  if (len == 1) string <- paste('xxxxx', 'xxxxx', 'xxxxx', string)


  s4 <- paste(string, collapse= ' ')
  s3 <- paste(strsplit(s4, ' ')[[1]][-1], collapse= ' ')
  s2 <- paste(strsplit(s3, ' ')[[1]][-1], collapse= ' ')
  s1 <- paste(strsplit(s2, ' ')[[1]][-1], collapse= ' ')

  dfsub <- df[J(s4,5)]
  if (!is.na(dfsub$nextword)) {
    result <- dfsub
  } else {
    dfsub <- df[J(s3,4)]
    if (!is.na(dfsub$nextword)) {
      result <-dfsub
    } else {
      dfsub <- df[J(s2,3)]
      if (!is.na(dfsub$nextword)) {
        result <- dfsub
      } else {
        dfsub <- df[J(s1,2)]
        if (!is.na(dfsub$nextword)) {
          result <- dfsub
        } else {
          dfsub <- df[ngram==1,]
          result = head(dfsub,1)
        }
      }
    }
  }

  return(unlist(head(result,1)[,'nextword']))
}
```

The function takes the ngram created in the previous function and the string to be analyzed. The string is first searched for in the 5-gram rows. If found, the 'nextword' is returned. If not, the last 3 words are searched in the 4 gram rows, and so on. If the reduced string is not found in any ngram, the most frequent single word is returned.

Accuracy Tests

Various accuracy tests were performed using the backoff function. The function to test the accuracy is below

```
library(tm)
library(tidytext)
library(dplyr)

##Prepare accuracy test data
#Read test file
test <- paste(readLines('testBlogs.txt'), collapse=' ')

#Clean data
test <- tolower(test)
test <- removeNumbers(test)
test <- stripWhitespace(test)
test <- removePunctuation(test)
test <- tidy(test)
names(test) <- 'text'

#Create the ngram  - only 5 gram
ngram <- unnest_tokens(test, words, text, token='ngrams', n=5)
#Convert to frequency table to get unique list of ngram strings
ngramFreq <- as.data.frame(table(ngram$words))

#Split into string and last word, add 2 more columns
ngramFreq$Var1 <- as.character(ngramFreq$Var1)
ngramFreq$main <- lapply(ngramFreq$Var1,function(x) {paste(head(strsplit(x,split=" ")[[1]],-1),
collapse= ' ')})
ngramFreq$last <- lapply(ngramFreq$Var1,function(x) {paste(tail(strsplit(x,split=" ")[[1]],1), c
ollapse= ' ')})
ngramFreq <- ngramFreq[,c('main', 'last', 'Freq')]
names(ngramFreq) <- c('main', 'nextword', 'freq')
ngramFreq$predicted <- ' '
ngramFreq$score <- 0

#Testing the accuracy
for (i in 1:nrow(ngramFreq)) {
    str <- unlist(ngramFreq[i,'main'])
    predicted <- predictNextWord(ngramFreq, str)
    ngramFreq[i,'predicted'] <- predicted
    ngramFreq[i,'score'] <- ifelse(ngramFreq[i,'nextword'] == predicted, 1, 0)
}

#Accuracy calculation
table(ngramFreq$score)
```

The function loads a test text file, and converts into a unique list of 5-grams. Each 5-gram is split into 4 + 1 words. The 4 word string is fed into the next word predictor one by one and the predicted word is stored in the data frame. The 5th word in the string is compared with the predicted word. If they are the same, the score is updated to 1, else it is set to 0. After all the strings are fed into the accuracy tester, the number of 1s are counted and it is divided by the number of records to get the percentage correct.

The following tests were performed.

1. The ngram data table was created as explained above with 10% of 'en_US.news.txt' and the test file was also taken from the same file. When the accuracy test was run the prediction accuracy was 95.7%. This is expected because the ngrams and test cases are taken from the same file. This also proves that the prediction algorithm works.
2. Ran the accuracy test for a part of the blogs data (2240 records) using the ngrams list from 'en_US.news.txt' (1 to 5 ingrams of 8181655 records). Accuracy was 11.13%, i.e. 11.13% records were correct.
3. Ran the accuracy test for a part of the twitter data (2870 records) using the ngrams list from 'en_US.news.txt' (1 to 5 ingrams of 8181655 records). Accuracy was 8.5%, i.e. 8.5% records were correct.
4. 10% was sampled from each of the three text files and the ngram data table was created. It had 29m rows. The test was conducted using this data table and a test file drawn from the blogs file. The first word accuracy is 12.7%.
5. The second word accuracy is 16.16%
6. The third word accuracy is 19.4%
7. The accuracy when using the 5-gram was 20%.

Inferences

1. The maximum average accuracy I could achive using the given dataset was 20%.
2. The accuracy is heavily dependent on the source of the ngram. When the ngram was from 'en_US.news.txt' and the test sample was from 'en_US.blogs.txt', the accuracy was low. This could be because the language styles of these texts are different. Blogs are more informal and less severely edited for language and grammar than News texts. When the text is from News and the sample is from Twitter, it was the worst. Twitter is replete with slang and informal language and the constraint of 140 characters forces people to use contractions and abbreviations. The News text, on the other hand is formal.
3. Accuracy can be improved by creating context specific ngrams. For example, ngrams created from medical terminology and texts will be most accurate when predicting medical text.
4. The test above show that the larger the ngram from which the next word is predicted, the better the accuracy. This is because a larger ngram has more context and gives a more accurate representation of how the word is used. But larger the order of the ngram, the higher the number of unique ngrams created from a piece of text. If larger ngrams are used, it could impact performance. For example, the counts of the various ngrams in the combined samples is below.

ngram order	1	2	3	4	5
count	227020	2905759	7000809	9202546	9845881

- The last word in the supplied string was searched for in the data frame and the next word in the 2-gram was reported as the next word. The average accuracy (the number of times the correct word was found) of this approach was 12.7%.
- The last 2 words in the supplied string was searched for in the data frame and the next word in the 3-gram was reported as the next word. The average accuracy (the number of times the correct word was found) of this approach was 16.16%.

Performance improvement

Performance is one of the greatest concerns in this application because of the large volume of data to process and number of searches that need to be performed. Some of the strategies implemented to improve performance are below Two versions of the functions were used - using backoff algorithm and not using backoff. The functions were timed using data frame, the data frame converted to a data table and the optimized data table. The dataframes were converted to data tables and setkey() was used to set the index keys and increase search speeds. The data tables were optimized by keeping the rows which has the highest frequencies for each combination of ngrams and string, and deleting the rest of the rows. The number of rows was reduced from 29 million to 19 million. The execution time in seconds was estimated by using the system.time() function.

To further enhance the performance, sparse data was removed and this reduced the rows to around 600k. Further the J() data table function was used to search and this reduced the word prediction time to around 0.01 sec.

Algorithm Type	Data set type	User	System	Elapsed
Backoff	data frame	1.47	0.07	1.53
Non Backoff	data frame	8.57	1.60	10.19
Backoff	data table	0.87	0.11	0.98
Non Backoff	data table	3.31	0.16	3.47
Backoff	data table optimized	0.71	0.19	0.88
Non Backoff	data table optimized	2.58	0.37	2.95

Notes on approach

1. Stop words and profanity should not be removed before creating the n-grams because people use these words when using the language. And it is necessary to suggest these words as next words. If profanity is uncomfortable for the users then we can remove the entire sentence rather than the words themselves. If we only remove the words, the words adjacent to them become the next words and this is not true in reality.
2. Removing full stops, question marks and exclamation points also create the same issue. The next word prediction should not cross these punctuations because the first word in the next sentence cannot be the next word to the last word in the previous sentence. For example, if we have two sentences, viz. ‘This is a cat. And, cats are agile’. If we ignore the full stop, the predicted next word to ‘cat’ is ‘and’, which is not accurate.
3. N-grams upto five-grams can be used to build in accuracy. Higher order ngrams gives better accuracy, but the number of rows increases and ngrams more than 5 words have a high probability of repeating stop words.

Approaches considered but not implemented

1. Ngrams need to be created around punctuations like ‘.’, ‘!’ and ‘?’. The ngrams should be created within the sentences ending with these punctuations. This will increase accuracy and improve performance because only useful ngrams will be created. Example, ‘Zynga, however, as mixed U.S.’ is a string in the news file. It got converted to ‘zynga however as mixed us’ after all the conversions. It is completely misrepresented. The sentence, punctuation and acronyms and context should be used in forming the n-grams to improve the accuracy.

2. Contractions, numbers and emojis are also actually words and they should be properly accounted for. Contractions should be expanded and numbers should be changed to words. Emojis should be considered as a word. They are currently deleted when we remove punctuations.
3. The prediction accuracy can be improved by learning from the user. When predicting words, we can offer multiple choices sorted by probability. When the user chooses the one he/she wants to use, adjust the frequencies and probabilities by giving a higher value to the chosen one. Over a period of time, the frequencies and probability for user preferred words will become higher and will be offered more often to the user thus increasing the accuracy for the user. This method only increases accuracy for the particular user and not for all users.
4. A manual cleanup of ngrams can decrease the volumne and improve performance. For example, ngrams like ‘a a a’ can be removed.

Deploying the Word Prediction App

The next step is to build a Shiny app to predict the next word using the prediction function and ngram data table above, and deploy it to the cloud using shinyapp.io. The UI would accept a string of words and the application would continiously predict the next word as the user types. The Shiny app code is below

```
########################################
#Shiny app for word prediction
########################################

library(shiny)
library(feather)
library(data.table)
#Load the data
ngrams <- as.data.table(read_feather('ngramsSmall.f'))
setkey(ngrams, main, ngram)
#Compile the prediction function
source('predictWordv6.R')

#The UI code
ui <- fluidPage(
  titlePanel("Next Word Prediction App"),

  sidebarLayout(
    sidebarPanel(textInput(inputId='text',
                           label="Enter your text below",
                           value="")),
    mainPanel("Next Word Is...",
              textOutput("nextword"),
              br(),
              br(),
              br(),
              br(),
              br(),
              p("This application predicts the next word when you provide words in the text box
 above.  When you
                enter words in the text box, the application continuously looks for and displays
 the next word as you type.
                "),
              br(),
              p("A short presentation on the application is available at "),
              br(),
              p("The details of the approach, constraints and code used to build the application
 are available at http://rpubs.com/joresh/368348"))
  )
)

# The server
server <- function(input,output) {

  output$nextword <- renderText({
    if (input$text != '' & input$text != ' ') predictNextWord(ngrams, input$text)
  })

}
# Run the app
shinyApp(ui = ui, server = server)
```

This concludes the paper. Hope you enjoyed it!