Capstone: The Prediction Algorithm

Javier Angoy

Feb 10th, 2018

INTRODUCTION

The goal of this final capstone project is to work on understanding and building predictive text models. As a result, we created a Shiny web application that, given a sequence of words, predicts the most likely word that completes the sequence.

Once we have done with the Exploratory Analysis, we are ready to continue to the next step. As our project involves creating a text predictor, we must find a way to handle the texts we have been given to create a corpus that we can use as a base for our app.

THE DATA

Overall Vision

The steps of the procedure will be:
1. Starting with the texts that we have been given, we create a corpus
2. Split the corpus into 3 smaller corpuses for easier and faster processing
3. Create n-grams of 5 to 1 tokens
4. Generate frequency matrices and bind them into 1 unique data.table
5. Create needed columns
6. Create final frequency table with sqlf::sqldf()

This frequency table is what we will use to predict the next word, comparing the preceding words to the existing n-grams on our table.

Reading

Those texts are written in four different languages: en_US (English - United States), fi_FI (Finnish -Finland), de_DE (German - Germany) and ru_RU (Russian - Russia). Consequently, we would have to handle every text separately and - after processing it in a specific manner, as every language has its own grammar and punctuation rules- create a multi language corpus. Mostly due to time constrictions, as well as to processing capacity, a decission was made to focus on english language, as the particularities for the other languages are unknown.

```r
# Reading Files
myDirectory <- paste0(getwd(),'/final/en_US')   ### Modify for each language
listf <- list.files(path = myDirectory, full.names = TRUE)
text_raw <- parallel::mclapply(listf, readr::read_lines)
no_texts <- length(unlist(text_raw))
```

We start reading the “en_US” files, what gives us a raw text of 4269678 documents (lines). To do so the parallel::mclapply() function will help us achieve a 60% faster reading speed, taking advantage of parallel multi-threading processing.

```
# Create Train and Test data sets
set.seed(100)
resample <- sample(no_texts, size = no_texts*0.999,replace=FALSE)
Test <- unlist(text_raw)[-resample]
Train <- unlist(text_raw)[resample]
```

Split Corpus

Next, we split our big corpus into train and test data sets in a 0.999/0.001 proportion. This will allow us later to check the accuracy of our model.

As we said earlier, the main issues will be the memory (and time) limitations, due to the big size of the training text.

If we read all the texts at once, this will produce a corpus of 1.0 GiB. Being this corpus too big to process it, the logical manner is to split it into three equally sized corpuses ($\approx$ 353 MiB) that we can load into memory to work with the quanteda package.

Processing

Next thing is to tidy our text, that is to say, proceed to remove undesired characters and some additional fixes:

- replace dash “-” with space " "
- remove twitter’s “rt”
- remove apostrophe character " ’ "
- remove numbers
- remove non alphabetic characters
- remove non printable characters
- collapse empty spaces to single space

Stopwords were not removed, since they are necessary to produce meaningful expressions. Profane words were also kept: being their frequencies very low, the additional cost of including them is not that big - neither in terms of index size nor in terms of query processing time. The risk of getting one of them as prediccion is thus very low.

We have built a function called myCleanCorpus that preforms all the cleaning tasks at once. We call it three times, one for each of the corpus that we have split our corpus into:

```
# Clean corpuses
corp_clean_1 <- myCleanCorpus(myCorpus_1)
corp_clean_2 <- myCleanCorpus(myCorpus_2)
corp_clean_3 <- myCleanCorpus(myCorpus_3)
rm(myCorpus_1,myCorpus_2,myCorpus_3,text_raw) ##CAUTION !!! Frees up memory
```

Tokenizing

Data from our corpus were split into chunks (“n-grams”) of two to five words (“tokens”).

Creating the Frequency Table

Later on, frequency for each of the ngrams was computed. create_dfm() function makes use of the dfm() and textstat_frequency() functions included on the quanteda package. Those functions, along with the data.table features, such as low memory usage, and ability to do indexed searches, will make things easier to our model. Now we start bindinig the separated dfms.

```r
#Create DFMs - Frequency Matrices
## 1
myDFM1_3 <- create_dfm(tokens_1_3);rm(tokens_1_3)
myDFM1_4 <- create_dfm(tokens_1_4);rm(tokens_1_4)
myDFM1_5 <- create_dfm(tokens_1_5);rm(tokens_1_5)
myFreqTable_1 <- rbind(myDFM1_3,myDFM1_4,myDFM1_5)
rm(myDFM1_3,myDFM1_4,myDFM1_5)

## 2
myDFM2_3 <- create_dfm(tokens_2_3);rm(tokens_2_3)
myDFM2_4 <- create_dfm(tokens_2_4);rm(tokens_2_4)
myDFM2_5 <- create_dfm(tokens_2_5);rm(tokens_2_5)
myFreqTable_2 <- rbind(myDFM2_3,myDFM2_4,myDFM2_5)
rm(myDFM2_3,myDFM2_4,myDFM2_5)

## 3
myDFM3_3 <- create_dfm(tokens_3_3);rm(tokens_3_3)
myDFM3_4 <- create_dfm(tokens_3_4);rm(tokens_3_4)
myDFM3_5 <- create_dfm(tokens_3_5);rm(tokens_3_5)
myFreqTable_3 <- rbind(myDFM3_3,myDFM3_4,myDFM3_5)
rm(myDFM3_3,myDFM3_4,myDFM3_5)
```

Our term-frequency matrices contain many terms with low frequency counts and few terms with high frequency counts. We can save memory if we keep terms with highest frequency, as rare items take up memory while adding little value to our language model. So we set a threshold of 5 counts, and only n-grams with a frequency equal to 5 or higher were kept by pruneDT() function.

Finally we re-bind all the matrices into a unique frequency matrix. We have to group (sum) the frequencies by feature.

```r
#Combine into 1 table
myFreqTable_1 <- pruneDT(myFreqTable_1)
myFreqTable_2 <- pruneDT(myFreqTable_2)
myFreqTable_3 <- pruneDT(myFreqTable_3)
myFreqTable <- rbind(myFreqTable_1,myFreqTable_2,myFreqTable_3)
myFreqTable <- myFreqTable[, .(frequency=sum(frequency)),by=feature]
rm(myFreqTable_1,myFreqTable_2,myFreqTable_3)
```

Next thing is to make some arrangements on the frequendy matrix we need for our model to work optimally:
* Create ntok column, which counts the number of tokens (words) for each feature
* Create base column, which removes the last token for multi-token features
* Create prediction column, which is the last token for multi-token features
* Features longer than 5 tokens are discarded (just in case they exist)

```r
#Prepare frequency table
myFreqTable$ntok <- wordcount(myFreqTable$feature)
```

```
myFreqTable$base <- ifelse(myFreqTable$ntok==1,
                           myFreqTable$feature,
                           stringr::word(myFreqTable$feature,start=1,end=-2))
myFreqTable$prediction <- ifelse(myFreqTable$ntok>1,
                                 stringr::word(myFreqTable$feature,-1),"")
myFreqTable <- myFreqTable[ntok <= 5,  ] #Remove n-grams >5 (if existing)
```

Finally, we created calculateScores() function. Using sqldf() on sqldf package allows us to write SQL sentences to compute the most frequently ocurring predictions for each n-gram. The algorithm that computes the score for each feature is explained below (see The Algorithm). The result is a final table that is all that our prediction model will need to do its work.

```
#Creates final frequency data.table
freqTab <- calculateScores()
setorder(freqTab,-score)
setkey(freqTab,bas)
head(freqTab)
```

```
##        feab bas   pred freq toks  mult        score
## 1:    a lot   a    lot 4779    2 0.064 0.0012528150
## 2:    a few   a    few 4273    2 0.064 0.0011201671
## 3: a little   a little 4047    2 0.064 0.0010609212
## 4:   a good   a   good 3970    2 0.064 0.0010407357
## 5:  a great   a  great 3961    2 0.064 0.0010383763
## 6:    a new   a    new 3408    2 0.064 0.0008934073
```

A new key “bas” (base) has been created. This key will speed up the search for the predicted word(s), as explained below where the Prediction Function is explained.

THE ALGORITHM

To choose the predicted next word, we apply an algorithm based on a smoothing method called Stupid Backoff. A score is calculated for each n-gram based on its frequency divided by the frequency of the n-gram that excludes the last word (base).
For n-grams <5 tokens a (generally accepted) multiplier coefficient α= 0.4 scales down its score times the difference with 5.

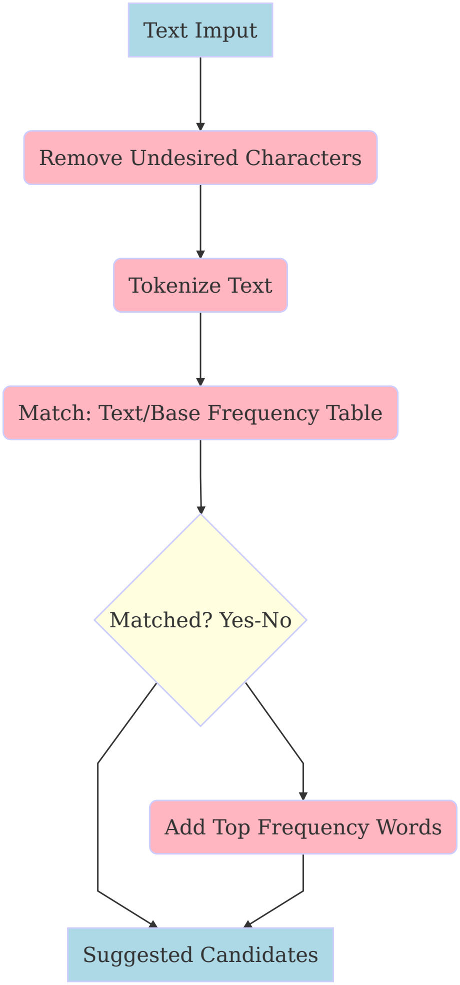
$$S(w_i|w_{i-k+1}^{i-1}) = \begin{cases} \frac{f(w_{i-k+1}^{i})}{f(w_{i-k+1}^{i-1})} & \text{if } f(w_{i-k+1}^{i})>0 \\ \alpha S(w_i|w_{i-k+1}^{i-1}) & \text{otherwise} \end{cases}$$

THE PREDICTION FUNCTION

Finally, a prediction function called matchData() is built. This function performs the following tasks:

1. Starting with a text as imput, the function removes all undesired characters like punctuation signs, numbers, symbols, separators, hyphens, twitter “rt’s”, url “http’s”.
2. The function tokenizes the text, taking only the last four words and discarding the rest.
3. Then a search for matches between the imputed text and the base column of the frequency table is done.
4. The operation is repeated removing words from the sentence, until there is only one (the last word), and keeping the score for the matched results.

5. If no matches are found, words are suggested randomly chosen among the eight top frequent words (the, to, and, a, of, in, for and is).
6. The three results with the highest score are returned, in descending order.


Text Imput
Remove Undesired Characters
Tokenize Text
Match: Text/Base Frequency Table
Matched? Yes-No
Add Top Frequency Words
Suggested Candidates