Capstone Project: Milestone Report

Rohit Benny Abraham

8/25/2020

Overview

Lacking a full-size keyboard on our mobile device has made text entry on touch screen devices a tedious job. Automated text prediction aims to solve this by using entered text to predict the next word.

This report does an exploratory analysis of a large body of text to design a text prediction system that could be run efficiently on a mobile device.

RStudio and the R package tm for text mining were used to perform the analysis. The final prediction model will consist of a transition matrix (which represents a Markov process) that can be easily imported into a Shiny app.

Data Acquistion

Predictive text modeling using NLP follows generally the same approach to data as any other prediction problem. The data is obtained, cleaned and explored, before moving to the predictive modeling stage using a training, validation and test set, and finally text prediction itself.

The Coursera dataset analyzed for this report is downloaded from this link There seem to be three different categories of text ("blogs", "news" and "twitter") for four different languages (German, English, finnish and Russian). For this analysis I used only the text files in American english language.

Data Preprocessing

Loading dependencies

Importing the data

Basic outline of the corpus used:

```
##                    Length Class             Mode
## en_US.blogs.txt    2      PlainTextDocument list
## en_US.news.txt     2      PlainTextDocument list
## en_US.twitter.txt 2      PlainTextDocument list
```

Calculating the memory attributes of the files in corpus

Summary of the files in corpus:

```
## Warning in stri_count_regex(string, pattern, opts_regex = opts(pattern)):
## argument is not an atomic vector; coercing

## Warning in stri_count_regex(string, pattern, opts_regex = opts(pattern)):
## argument is not an atomic vector; coercing

## Warning in stri_count_regex(string, pattern, opts_regex = opts(pattern)):
## argument is not an atomic vector; coercing
```

	File.Size.Mb.	No..of.Lines	No..of.words
blogs	200.4242	899287	37334441
news	196.2775	77258	2643972
twitter	159.3641	2360147	30373792
total	556.0658	3336692	70352205

- It can be seen that when loading all 3 text files into corpus, the object occupies 375.7 Mb memory and in total we got 3.336692^{6} lines of text to handle.
- To reduce the computational cost we will go with sampling i.e. ranodmly selecting the lines needed to be inlcuded to get an accurate assumaption to results that would be obtained using all the data.

Sampling

Created a separate sub-sample dataset by reading in a random subset of the original data and writing it out to a separate file (here I use rbinom function to "flip a biased coin" to decide which line to read in with a sampling rate of 10%)

```
## Warning in readLines(paste(getwd(), "/en_US/en_US.news.txt", sep = "")):
## incomplete final line found on 'D:/Coursera/Capstone Project/final/en_US/
## en_US.news.txt'
```

Text Mining

Creating a corpus of the sample data

```
corpus <- VCorpus(DirSource(paste(getwd(), "/sample", sep = "")),
                     readerControl = list(reader = readPlain,
                                          language = "en_US",
                                          load = TRUE))
```

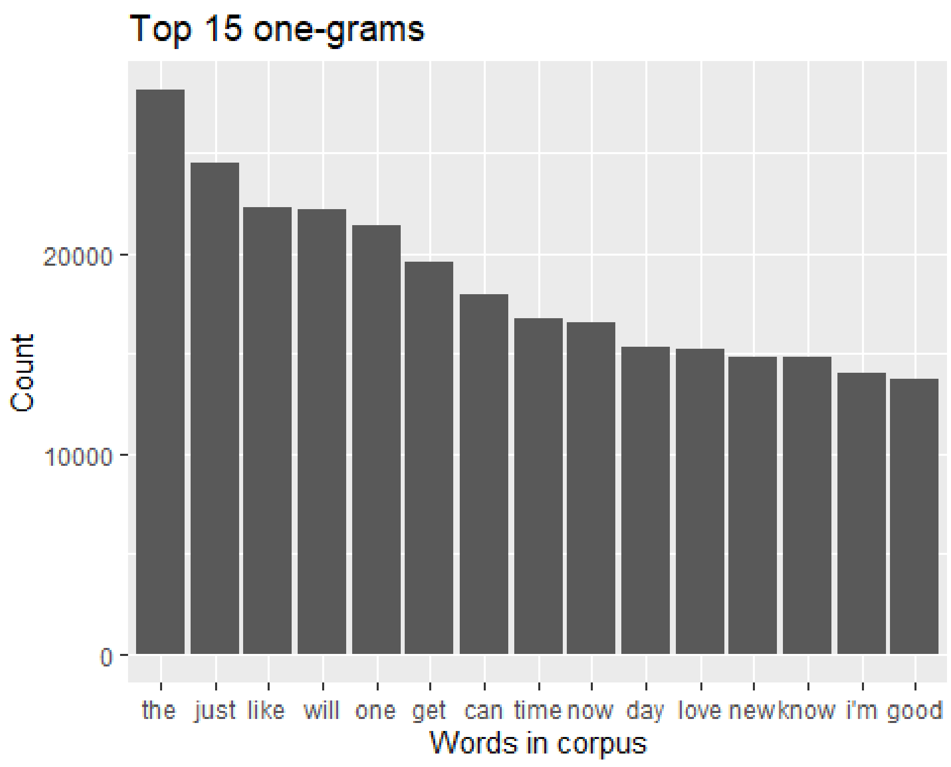
Corpus cleaning

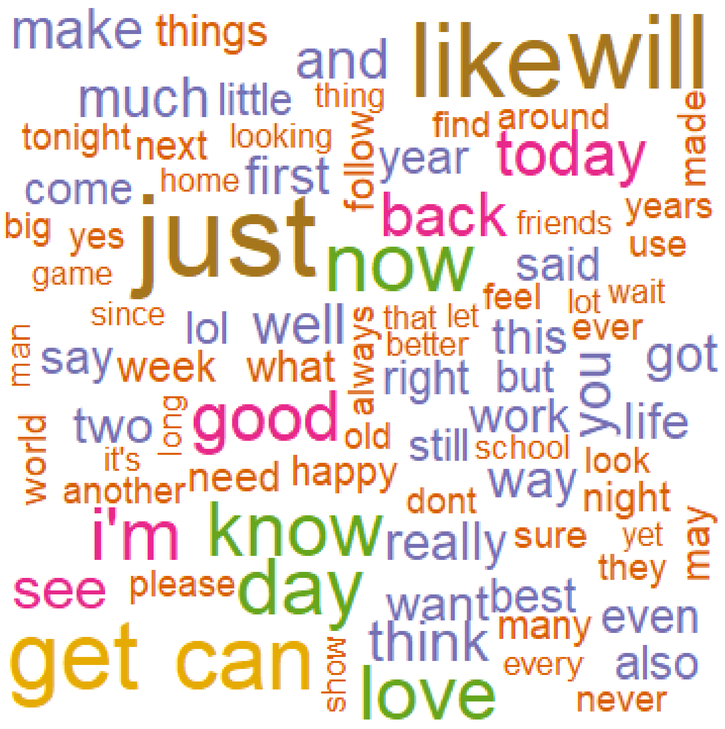
Stemming is not performed in our analysis and the stop words are treated for now to visualize and analyze the gram models, so that the frequency of the stop words does not skew the observations. In general, if we were working with feature extraction (for machine learning), we would do that, but given that stop words are relevant for predicting text typing in natural languages, we will keep them in our modeling.

```
replacePunctuation <- content_transformer(function(x) {
        gsub("[^[:alnum:][:space:]'`]", " ", x)
})

clean_corpus <- function(corpus) {
        corpus <- tm_map(corpus, stripWhitespace)
        corpus <- tm_map(corpus, replacePunctuation)
        corpus <- tm_map(corpus, removeNumbers)
        corpus <- tm_map(corpus, removeWords, stopwords("en"))
        corpus <- tm_map(corpus, content_transformer(tolower))
        corpus
}

corpus <- clean_corpus(corpus)

profanityList <- read.csv("profanity_words.txt", header = FALSE,
                          stringsAsFactors = FALSE)
profanityWords <- profanityList$V1
corpus <- tm_map(corpus, removeWords, profanityWords)
```

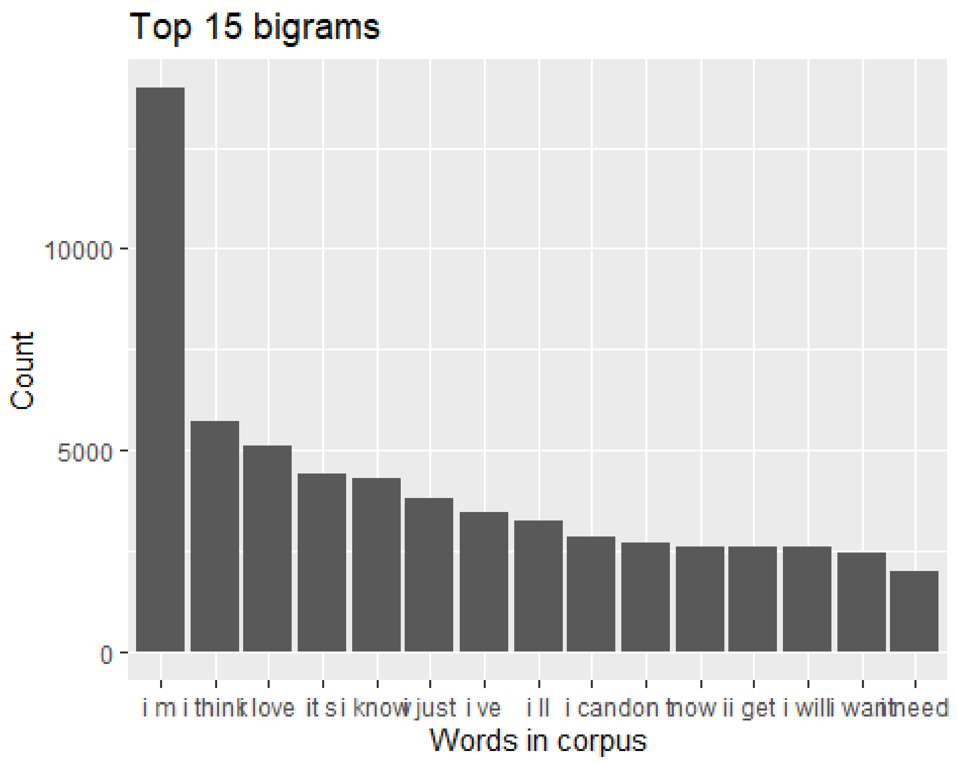
Exploratory data analysis

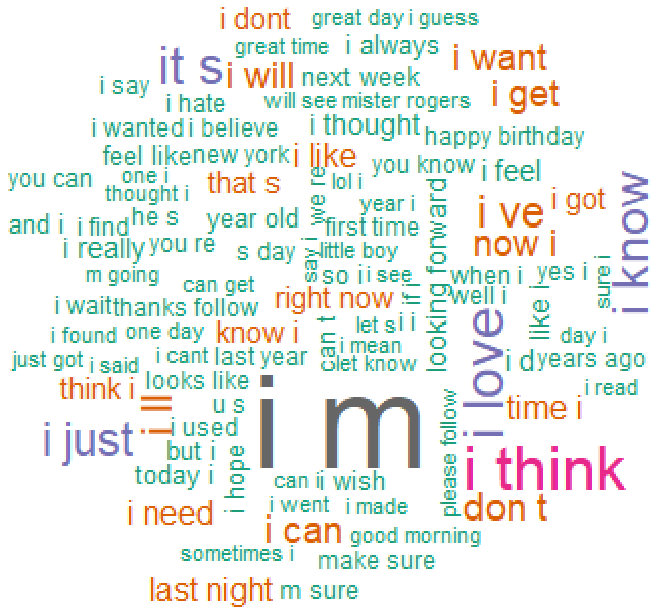
Creating and visulaizing uni-grams



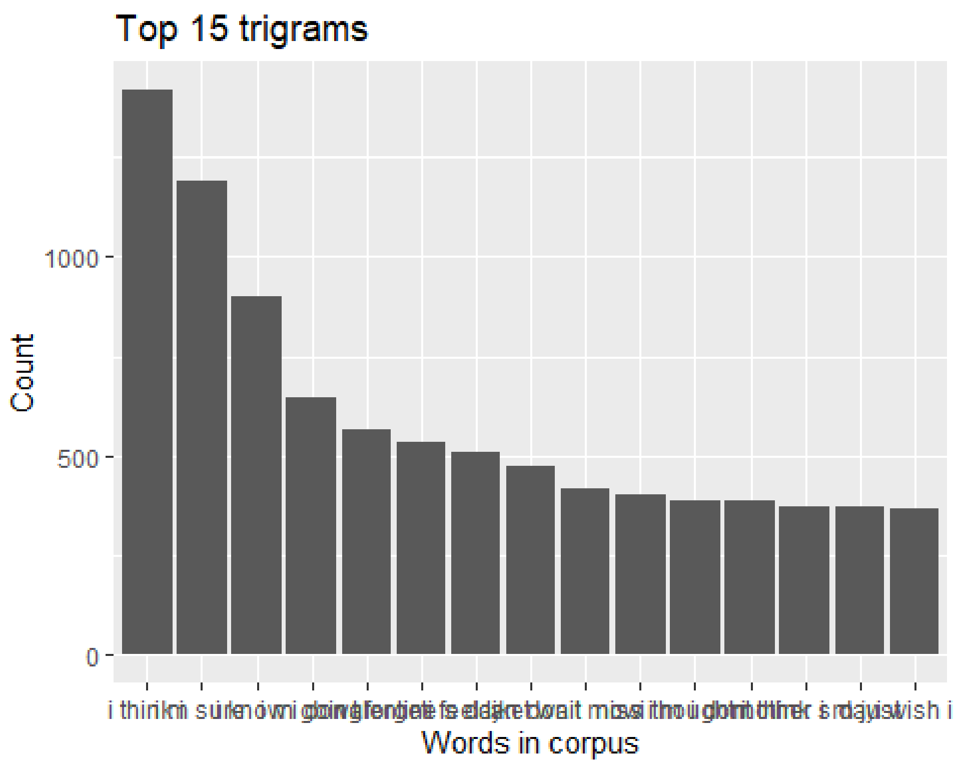


Creating and visulaizing bi-grams





Creating and visulaizing tri-grams



i think i
i know i
i m sure
i feel like
valentine s day
can t wait
i dont think
don t miss
let us know
i m looking
pu bef th
little boy big you know i
my year old
s cold balls
that s awesome
i m glad
i i m
u know clap
florence italy rentals
i feel good
i ve got
i ve never
i always try
next week i
it s hot
m sure i
d love tell
i ll post
toast mom i
i got the
even though
right now i
can i get
say i m
i d love
time i m
i wish i
it s friday
hi hi hi
i m really
i just saw
i m one
store called regarding
yes i m
next time i
i m pretty
i can meet
i m good
i m free
it s awesome
when i think
i ll get

Creating 4-grams

Computed 4-grams for the as we will be needing this size of n-grams for our prediction model

```
##                                               word frequency
## little boy big sword little boy big sword       336
## i love toast mom           i love toast mom       284
## i m sure i                       i m sure i       237
## i m pretty sure           i m pretty sure       236
## i m glad i                       i m glad i       235
```

Word coverage

Word coverage is determined by the count of most frequent words required to cover a given percentage of words present in the entire data-set.

We are going to answer this question by finding out how the number of words covered in the data-set increases as we add words from the most frequent to the least frequent.

```
wordsCoverage <- data.frame(
        coverage =
round(cumsum(gramdatasorted$frequency)/sum(gramdatasorted$frequency)*100, 2),
        words = 1:nrow(gramdatasorted))

wordsCoverage[1:5, ]
```

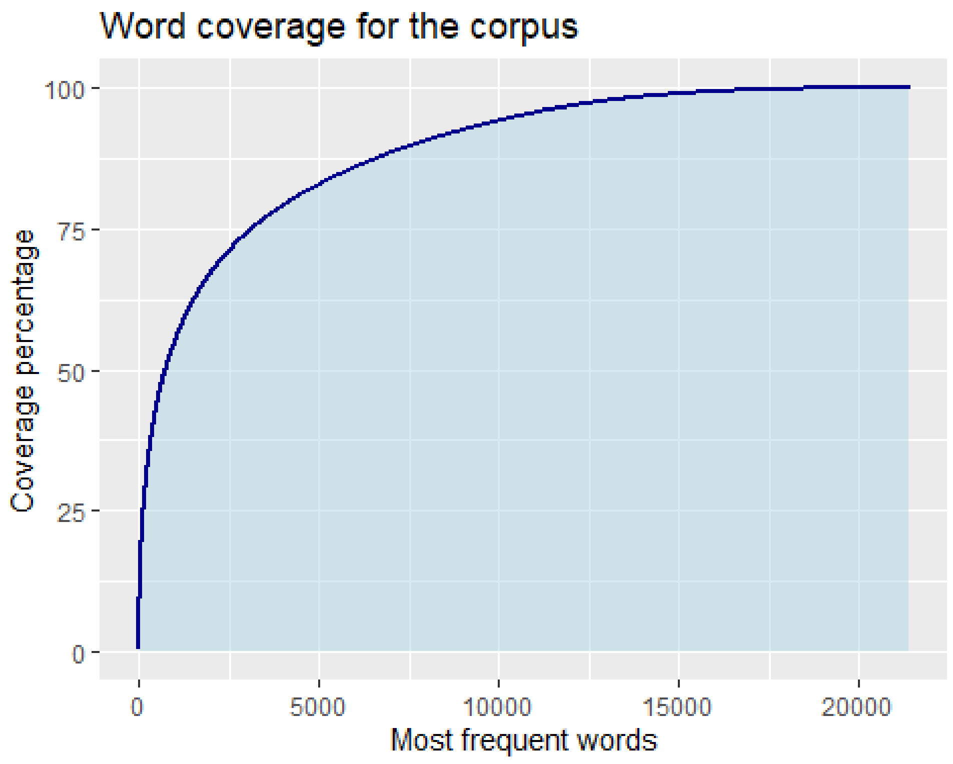
```
##   coverage words
## 1     0.73     1
## 2     1.37     2
## 3     1.95     3
## 4     2.53     4
## 5     3.09     5
```

Now we need to know what's the minimum number of top words added to achieve 50% and 90% coverage:

```
halfCoverage <- min(wordsCoverage[wordsCoverage$coverage > 50, ]$words)
halfCoverage

## [1] 741

ninetyCoverage <- min(wordsCoverage[wordsCoverage$coverage > 90, ]$words)
ninetyCoverage

## [1] 7661
```

According with these computations, we would need 740 words to achieve 50% coverage (i.e. the minimum number of frequent words added that would represent half of the words present in the data) and 7659 words to achieve 90% coverage.

Plot the coverage



Future work

The future work will focus on building some ngram-based statistical language models trained on a much larger sample of the original dataset, prune these models with a validation set and choose the best model based on a testset, with sentences never used in training or validation. When building these models, we'll try to optimize storage and memory utilization keeping in mind the constraints of free Shiny App platform.

For unseen ngrams, some type of smoothing(backoff) will be used to deal with the zero probabilities.

However, this approach is computationally very expensive - the required matrices get very large and it's very likely that for a decent-sized training set my available computer will get overwhelmed. Therefore there is need to explore alternative approaches.