Milestone Report: Text Prediction App - Data Exploration and Design Strategy

Rohit Padebettu

12/25/2016

Description

As part of the Final Capstone Project for the Data Science Course provided by Johns Hopkins University, we are required to build a Shiny data product, which would predict the next word given a word, phrase or sentence. The text corpora for building the product was provided and we are required to use Natural Language Processing techniques and packages in R to build a model which would be light on memory, highly efficient, responsive and accurate.

In this report, we begin by downloading the text corpora provided by SwiftKey. We then load the files into R session to begin our analysis. We initially obtain summary statistics and later proceed to process the text data using NLP packages in R. We clean the data, tokenize, generate ngrams and create dictionaries of document term frequency. These dictionaries would later be used with our prediction model to predict the next words. In this report, we also use the dictionaries to explore the data graphically. We examine the frequency distribution of the terms in the various n-gram sets. We also plot the top 10 most frequently occuring words and phrases.

In the final part of the report, we summarize our understanding from exploring the data and come up with likely strategies to build the predictive model.

Getting and Loading the Data

We begin by loading the required libraries which would be used in our analysis

```
suppressPackageStartupMessages(library(data.table))
suppressPackageStartupMessages(library(stringi))
suppressPackageStartupMessages(library(ngram))
suppressPackageStartupMessages(library(knitr))
suppressPackageStartupMessages(library(readr))
suppressPackageStartupMessages(library(ggplot2))
```

The text data to be used in building the predictive model, was provided by SwiftKey on the Coursera Capstone project site. The file is in form of a zip file with three text documents in each languages:

- Blogs
- News
- Twitter

We download the data using the code below to read it into R.

```
## Downloading File
url<-"https://d396qusza40orc.cloudfront.net/dsscapstone/dataset/Coursera-SwiftKey.zip"
download.file(url,"./Data/SwiftKey.zip",method = "curl")
unzip("./Data/SwiftKey.zip",exdir = "./Data/")

## Referencing files
cname<-"./Data/final/en_US/"
blogsfile<-paste0(cname,"en_US.blogs.txt")
newsfile<-paste0(cname,"en_US.news.txt")
```

```r
twitfile<-paste0(cname,"en_US.twitter.txt")

Sys.setlocale("LC_CTYPE", "en_US.UTF-8")

## Reading Files
blogs_file<-readLines(blogsfile,skipNul = T)
news_file<-readLines(newsfile,skipNul = T)
twit_file<-readLines(twitfile,skipNul = T)
```

Descriptive Summary of the data

In this step we proceed to generate some descriptive statistics to get an idea about the file size, number of lines in each file and the number of words in each file.

```r
Desc<-data.frame(
    file_name = c("en_US.blogs.txt",
                  "en_US.news.txt",
                  "en_US.twitter.txt"),
    file_size_mb = c(file.size(blogsfile)/1024^2,
                     file.size(newsfile)/1024^2,
                     file.size(twitfile)/1024^2),
    file_chars = c(length(blogs_file),
                   length(news_file),
                   length(twit_file)),
    wordcount = c(wordcount(blogs_file," "),
                  wordcount(news_file," "),
                  wordcount(twit_file," "))
)
```

file_name	file_size_mb	file_chars	wordcount
en_US.blogs.txt	200.4242	899288	37334131
en_US.news.txt	196.2775	1010242	34372530
en_US.twitter.txt	159.3641	2360148	30373583

We observe that each of these documents is several MegaBytes in size and have millions of words in them.

Analysis of the Data

From the summary statistics above it is very clear that analysing the documents in full would overwhelm the memory of an individual machine in addition to taking many hours of computing time. Keeping in mind that the final application needs to run on a hosted shiny server with **1GB** memory limitation, we would need to design our analysis and application accordingly.

We sample about **20%** of the data from each of the three files and begin our analysis on this set. Given the sheet volume of the corpora, although this might be sacrificing some accuracy, we would nevertheless get some useful insights from exploring even this sample.

Processing and Cleaning the data

As a standard procedure in Natural Language Processing, we begin by cleaning the sampled data and loading it into a *R processible* data structure. Specifically, we do the following steps

- Remove Spacing
- Remove Punctuation
- Remove Numbers
- Convert all words to Lower Case

We use the convenient R package `ngram package` to help us do this.

```
text<-read_file(textfile)

text_clean<-preprocess(x = text,
                       case = "lower",
                       remove.punct = TRUE,
                       remove.numbers = TRUE,
                       fix.spacing = TRUE)
```

Creating N-Grams

The `ngram package` also provides us a very quick and efficient method `ngram()` to help us generate and store n-grams, their frequencies and probabilities in data structures accessed by reference.

For our purpose here, we write a custom convenience function `getNgram` which helps us iteratively generate, combine and store `unigrams`,`bigrams`,`trigrams` and `quadgrams` in a `data.table` structure.

```
Dict<-getNgram(text_clean,ngrams)

getNgram<- function(text,n){
    ngram_data_table = data.table(NULL)
    for (i in 1:n){
        ngram_table<-ngram(text,n = i,sep = " ")
        ngram_data_table_temp<-get.phrasetable(ngram_table)
        ngram_data_table_temp$term<-trimws(ngram_data_table_temp$ngrams,"r")
        ngram_data_table_temp$ngrams<-NULL
        ngram_data_table<-data.table(rbind(ngram_data_table,ngram_data_table_temp))
    }
    ngram_data_table[,ngram:=wordcount(term," "),by=term]
    setnames(ngram_data_table,"freq","term_freq")

    return(ngram_data_table)
}
```

Creating Individual Dictionaries

We also wrote another custom convenience function `CreateDictionary` which helps us automate the following tasks:

- Reading a given text file
- Pre-process the file
- Generate N-Gram data tables
- Optionally prune the dicitonary to limit the size
- Save the dictionary as RDS file to the disk

We use this `CreateDictionary` function to read and generate dictionaries from each of the 3 sample files: blogs, news and twitter. Since this is a time consuming task, we preferred to do it once, so that the dictionaries are later persistent and accessible for further analysis.

```
blog_dict<-CreateDictionary(DictName = "US_Blogs_Dict_s",
                             textfile = blogsfile_sample,
                             ngrams = 4,
                             saveRDS = T,
                             prune = F
)

news_dict<-CreateDictionary(DictName = "US_News_Dict_s",
                             textfile = newsfile_sample,
                             ngrams = 4,
                             saveRDS = T,
                             prune = F
)

twit_dict<-CreateDictionary(DictName = "US_Twit_Dict_s",
                             textfile = twitfile_sample,
                             ngrams = 4,
                             saveRDS = T,
                             prune = F
)
```

Creating a Combined Dictionary

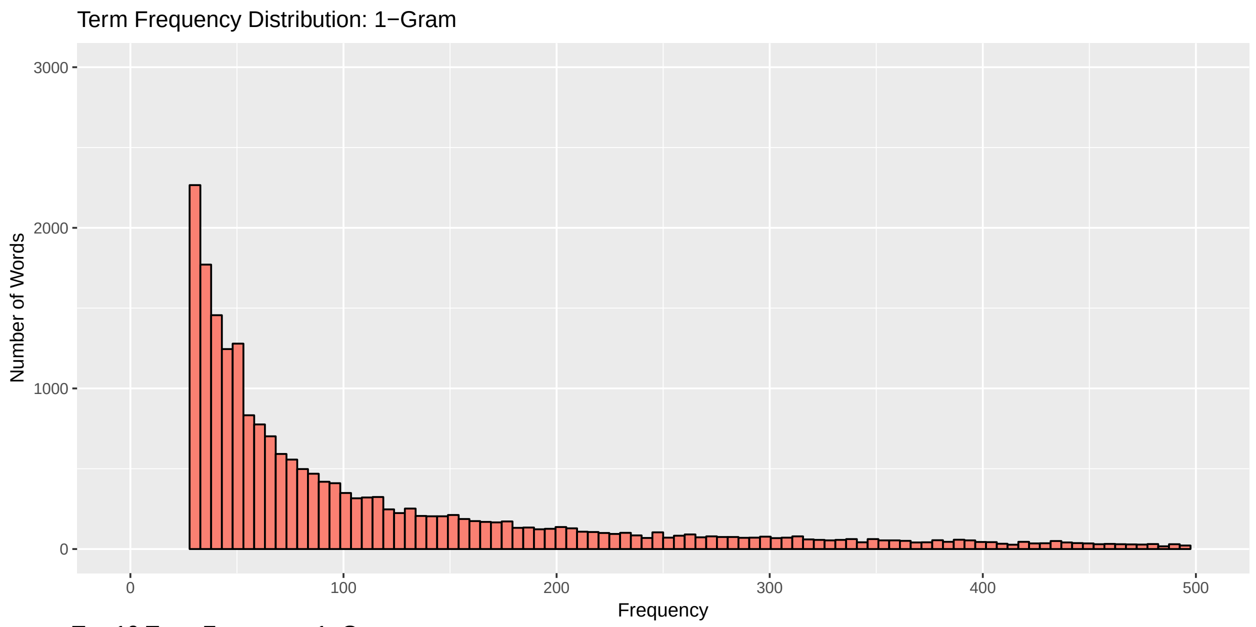
Once we have the dictionaries from the individual sources saved to the disk, we can read them into R `data.table`, using the `readr::read_rds()` function. In this step, prior to beginning further detailed exploration, we combine these dictionaries into a `combo` dictionary.

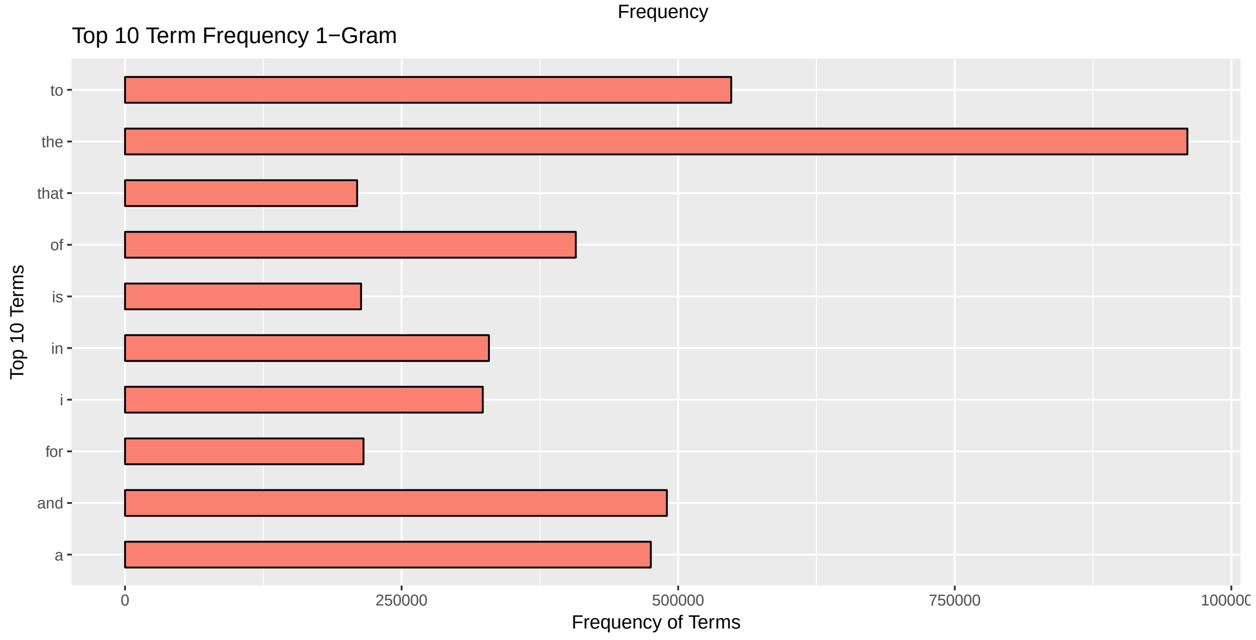
```
## Reading Appropriate Dictionary into memory
blog="US_Blogs_Dict_s"
news="US_News_Dict_s"
twit="US_Twit_Dict_s"

b<-read_rds(paste0("./Dictionaries/",blog,".RDS"))
n<-read_rds(paste0("./Dictionaries/",news,".RDS"))
t<-read_rds(paste0("./Dictionaries/",twit,".RDS"))

## Combining the dictionaries and grouping
combo<-rbind(b,n,t)
setkey(combo,term)
combo<-combo[,.(term_freq=sum(term_freq),ngram=mean(ngram)),term][order(-ngram,-term_freq)]
```

Data Exploration

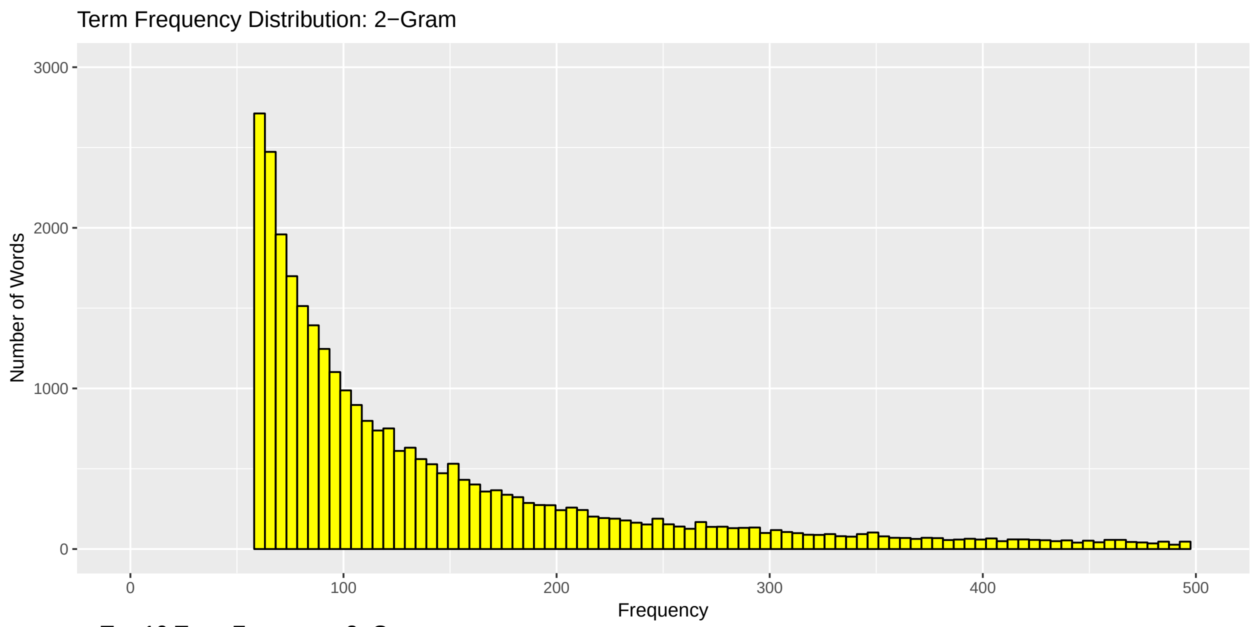
With the combo dictionary, we proceed to further explore the data by plotting some features and characteristics of the individual N-Gram sets. Specifically, we want to see how the frequency of terms is distributed within each n-gram set. We also would like to see the most common words and phrases and their frequencies within each n-gram set

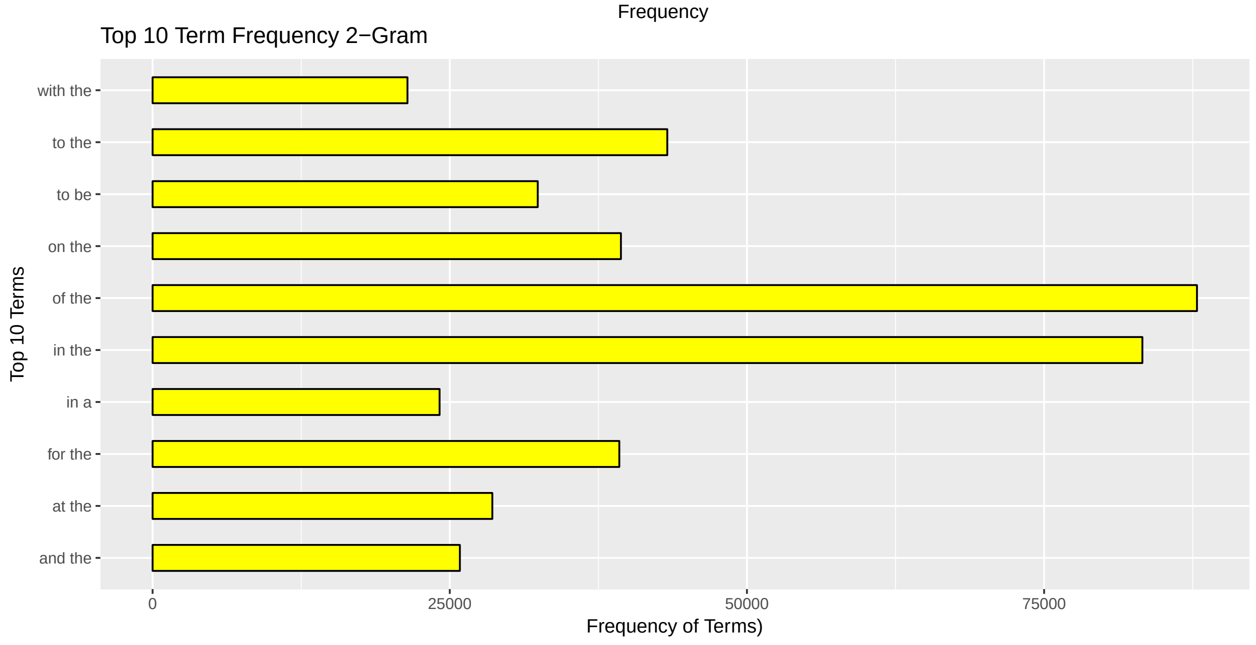
Unigrams - Single words



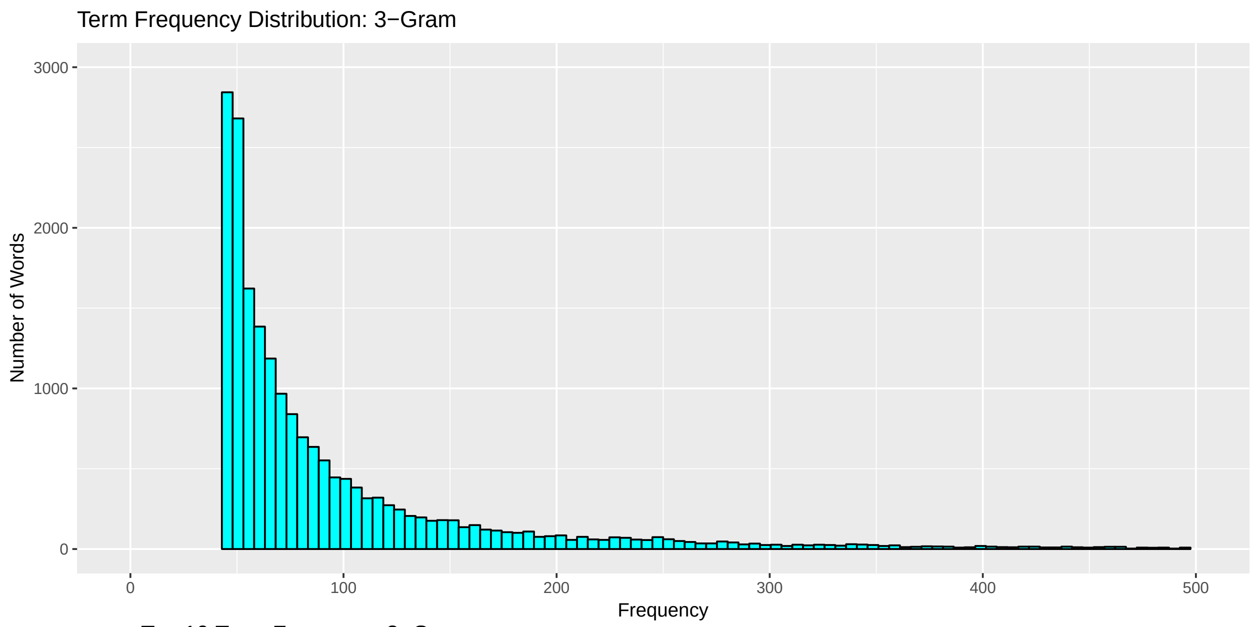


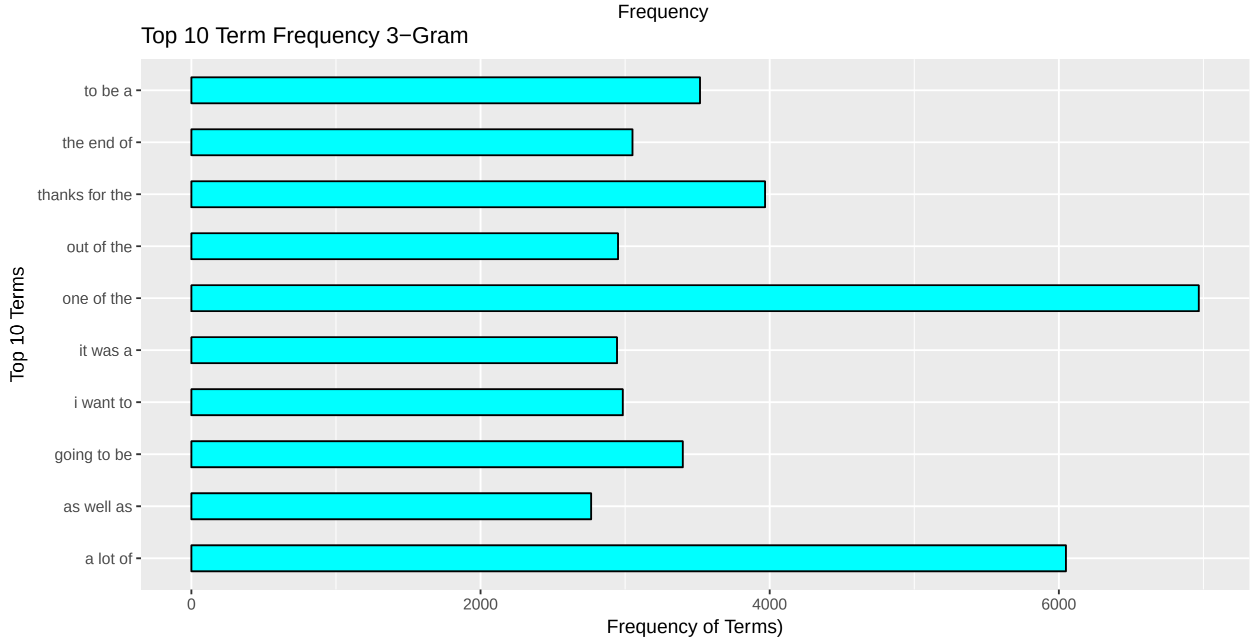
Bigrams - Two word phrases



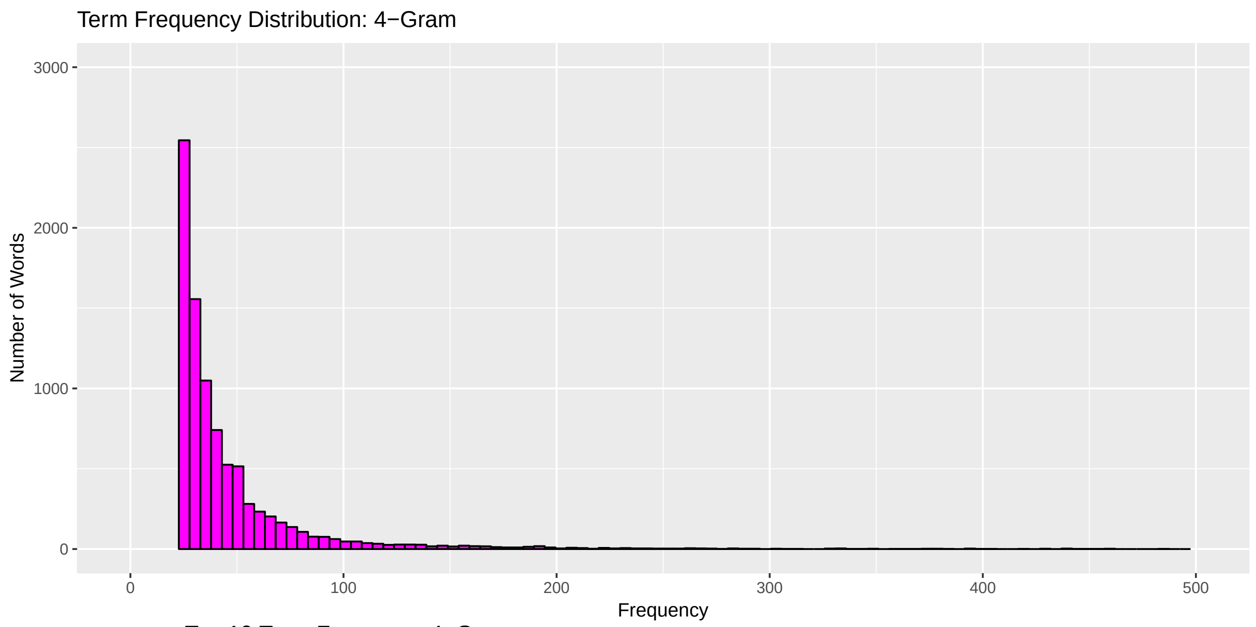


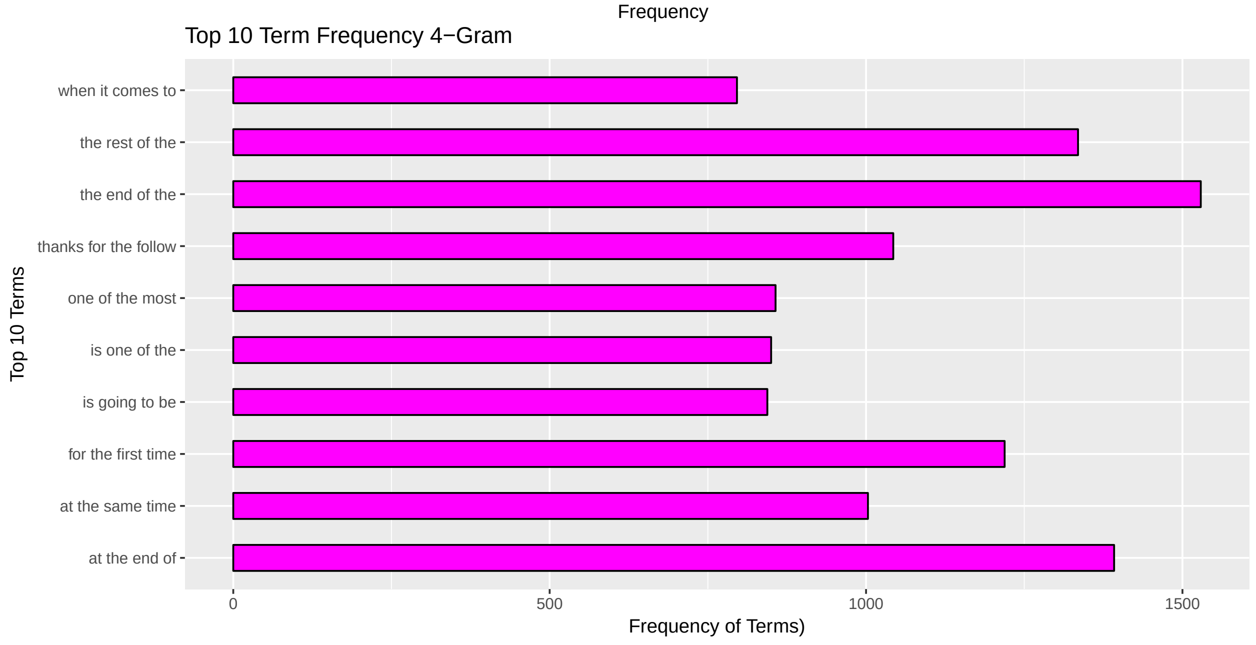
Trigrams - Three word phrases





Quadgrams - Four word phrases





Data Insights

From the charts above and from exploring the individual data tables, we can summarize that:

- A small majority of words and phrases have very high frequencies in the data set.
- A large majority of words and phrases across the n-grams occur very infrequently.
- A majority of words in the trigram and quadgram data sets have frequencies less than 2

Text Prediction Strategy

Our next word prediction strategy will be based of N-grams in the dictionaries we created. We hope to follow the Markov chains and Backoff Algorithms to predict the next word given a word or phrase.

Backoff Algorithms

- Given a set of n words, we will come up with probably matches in the `(n+1)gram` data set
- Depending on the frequencies of occurence of each of those `(n+1)grams`, we will determine the best probable match
- The `(n+1)'th word` of the best match will be our prediction
- In case we can't find any matches in the `(n+1)gram` data set, we can **backoff** the first word from the given phrase to make it into a `(n-1)gram word` and look for matches in the `n-gram` data set.
- If the phrase/word entered is totally new, we can recursively follow the above procedure until we are down to one word and then implement the backoff algo at the character level.

Influence of Data Insights on Design

- The large number of records in the individual `n-gram` data sets, leads us to believe that task of searching for matches in a given `n-gram` dataset is going to be computationally intensive and time consuming. We therefore have to come up with clever ways to narrow our search path and or prune the data set.
- The sheer size of the `combo` dictionary, even using just a sample of the data provided, is probably too huge for a hosted shiny app to process. We therefore have to consider pruning the dictionary by removing the low frequency terms in all the data sets. Although this might negatively impact the accuracy of predictions, it has the added benefit of narrowing our search space, thus speeding up the search.
- We can also consider storing only the top few matches for each `(n-1)gram` phrase in the corresponding `n-gram` data set.
- We can also consider removing or appropriately weighting the frequencies of the most commonly occuring stop words to avoid us from predicting these stop words most of the time.
- We can also consider annotating the text with contextual information or cluster the n-grams to allow better contextual predictions