Capstone

Andrew J Fox

2023-04-29

Download Data, Packages, and misc.

```
## Download data
if (!file.exists("data")){
  dir.create("data")
}
fileURL <- "https://d396qusza40orc.cloudfront.net/dsscapstone/dataset/Coursera-SwiftKey.zip"
filePATH <- "./data/rawdata.zip"
download.file(fileURL, destfile=filePATH, method="curl")
unzip(filePATH, exdir="./data")
```

Install necessary packages.

```
#install.packages("RWeka")
#install.packages("dplyr")
#install.packages("ggplot2")
#install.packages("stringr")
#install.packages("markovchain")
```

Load packages.

```
library(RWeka)
library(dplyr)
library(ggplot2)
library(stringr)
library(markovchain)
```

Random sampling is done in this project so the seed is set to 423.

```
## Set seed
set.seed(423)
```

Getting and Cleaning the Data

The data is filed into directories based on language (de, en, fi, ru), each of which contains .txt files containing many lines of text extracted from different three sources (blogs, news, twitter).

```
## Create data directory path
datadir <- "./data/final/en_US"

## List files in data dir
filenames <- list.files(datadir)
filenames
```

```
## [1] "en_US.blogs.txt"   "en_US.news.txt"    "en_US.twitter.txt"
```

```
## Rename files for easier access
filePaths <- paste(datadir, "/", filenames, sep="")

## Remove unwanted objects
rm(filenames, datadir)
```

```
## Read in all lines of each file
blogs <- readLines(filePaths[1])
news <- readLines(filePaths[2])
twitter <- readLines(filePaths[3])
```

```
## Initialize data frame
filesinfo <- data.frame()

## Create file names
filenames <- c("Blogs", "News", "Twitter")

## Get Lengths
blogLen <- length(blogs)
newsLen <- length(news)
twitterLen <- length(twitter)
line_lengths <- c(blogLen, newsLen, twitterLen)

## Get name, number of lines, and size for each file
for (file in seq(filePaths)){
  name <- filenames[file] #get name
  size <- signif(file.info(filePaths[file])$size/1024/1024, 4) #get size in mb, 1 decimal
  filesinfo <- rbind(filesinfo, c(name, line_lengths[file], size)) #rbind
}

## Rename columns
colnames(filesinfo) <- c("File", "Lines", "Size (mb)")

## Display to system
filesinfo
```

```
##      File   Lines Size (mb)
## 1   Blogs  899288     200.4
## 2    News 1010242     196.3
## 3 Twitter 2360148     159.4
```

```
## Remove unwanted objects
rm(blogLen, newsLen, twitterLen, line_lengths, filenames, size, name, file)
```

Because all the lines of each file will not be needed for this project, I sample a small percent of each file. I then create a corpus of lines from each file which will be used as the overall dataset. Because uniform, clean text is important for proper analysis of text data, I define a cleaning function.

`sampleFile()` samples a file down to a percent of that file.

```
#' Samples a file down to p% of that file.
#' @param file character file of lines to sample from
#' @param p percent of file to sample
#' @param replace replace lines during sampling?
sampleFile <- function(file, p, replace=F){
  stopifnot(p < 100) #percent to sample must be <100
  numlines <- floor(length(file)*(p/100)) #get num of lines to read in
  lines <- sample(file, numlines, replace=replace) #sample lines; returns lines
  lines
}
```

`createCorpus()` creates a corpus of lines from the all the files.

```
#' Creates a corpus of lines from a list of files
#' @param files files to create corpus from
#' @param p percent of each file that should be sampled
createCorpus <- function(files, p){
  stopifnot(p < 100)#percent to sample must be <100
  corpus <- c() #intilaize corpus
  for (i in seq(files)){
    file <- sampleFile(files[[i]], p) #sample p% of file
    corpus <- c(corpus, file) #append sampled file to corpus
  }
  corpus #return finalized corpus
}
```

`clean()` cleans a character file.

```
#' Cleans characters down to only words, all lowercase
#' @param char character to clean
clean <- function(char){
  char <- tolower(char) #lowercase everything
  char <- gsub("\\d+th", " ", char) #remove digit references, like 9th
  char <- gsub("\\d+st", " ", char) #remove digit references, like 1st
  char <- gsub("\\d+", " ", char) #remove digits
  char <- gsub("[\\...]", " ", char) #remove elipses
  char <- gsub("[[:punct:]]", "", char) #remove punctuation
  char <- gsub("\\s+", " ", char) #replace extra spaces with single space
  char <- trimws(char) #trim leading and trailing white space
  char
}
```

Creating corpus

I create a corpus of lines from only 3% of the blogs, news, and twitter files. This corpus can be thought of as the entire dataset, which will be split into training, testing, and validation sets when building the model.

```
## Creates a corpus that has 3% of lines from blogs, news, and twitter files
files <- list(blogs, news, twitter)
corpus <- createCorpus(files, 3)
paste("The corpus has", length(corpus), "lines and is",
```

```
      format(object.size(corpus), units="Mb"), "large.")
```

```
## [1] "The corpus has 128089 lines and is 25.1 Mb large."
```

```
## Remove unwanted objects
rm(files)
```

Cleaning corpus

I redefine the corpus to a cleaned version of it. A "cleaned" line is displayed.

```
## Clean corpus
corpus <- clean(corpus)
corpus[1]
```

```
## [1] "ill get my coat"
```

```
#write.csv(corpus, file="corpus.txt")
```

Train, Test, and Validation Sets

I split the corpus into a 60% train set, 20% test set, and a 20% validation set.

```
## Split sample into training, test, and validation sets
splits <- sample(c("train", "test", "valid"),
                 size=length(corpus), prob=c(0.6,0.2,0.2), replace = TRUE)

## Select for train, test, and validation sets
trainset <- corpus[splits=="train"]
testset <- corpus[splits=="test"]
validset <- corpus[splits=="valid"]

paste("The train set has", length(trainset), "lines, compared to the",
      length(corpus), "lines in the corpus.")
```

```
## [1] "The train set has 76704 lines, compared to the 128089 lines in the corpus."
```

The training set, `trainset`, will be used going forward for model building.

Exploratory Data Analysis

Getting ngrams

An n-gram is a sequence of words of length n. Getting n-grams from the data and calculating their frequencies will be crucial to building the model. The RWeka package is used to extract n-grams; The function for doing so is rather complicated, so I simplify it here.

`ngram()` extracts ngrams from data.

```
## Simpler function for creating n-grams
#' Returns list of n-grams from a preprocessed character
#' @param char character to create n-grams from
#' @param n number of grams
#' @details requires 'RWeka' package
ngram <- function(char, n){
  NGramTokenizer(char, Weka_control(min=n,max=n))
}
```

The model will only use 1-, 2-, and 3- grams, extracted from the training set here.

```
## Extract 1-, 2-, 3- grams
unigrams <- ngram(trainset, 1)

bigrams <- ngram(trainset, 2)

trigrams <- ngram(trainset, 3)

## Display
list(paste("There are", length(unigrams), "unigrams,", format(object.size(unigrams), units="Mb")),
  paste("There are", length(bigrams), "bigrams,", format(object.size(bigrams), units="Mb")),
  paste("There are", length(trigrams), "trigrams,", format(object.size(trigrams), units="Mb")))
```

```
## [[1]]
## [1] "There are 1791630 unigrams, 18.1 Mb"
##
## [[2]]
## [1] "There are 1714956 bigrams, 57.1 Mb"
##
## [[3]]
## [1] "There are 1638729 trigrams, 103.2 Mb"
```

N-gram Tables that present the frequency (and probability) of each n-gram are required for building the model. These tables can be trimmed remarkably by removing n-grams that do not occur enough to be useful.

N-gam Tables are built, sorted, and presented here.

```
## Create ngram tables
unitbl <- arrange(data.frame(table(unigrams)), desc(Freq))
bitbl <- arrange(data.frame(table(bigrams)), desc(Freq))
tritbl <- arrange(data.frame(table(trigrams)), desc(Freq))

## Add probabilty column
unitbl <- mutate(unitbl, Prob=round(Freq/sum(Freq), digits=4))
bitbl <- mutate(bitbl, Prob=round(Freq/sum(Freq), digits=4))
tritbl <- mutate(tritbl, Prob=round(Freq/sum(Freq), digits=4))

## Remove ngrams that occur less than 3 times
```

```
unitbl <- filter(unitbl, Freq > 3)
bitbl <- filter(bitbl, Freq > 3)
tritbl <- filter(tritbl, Freq > 3)

## Show top 5 ngrams
head(unitbl, 5); head(bitbl, 5); head(tritbl, 5)
```

```
##   unigrams  Freq   Prob
## 1      the 84789 0.0473
## 2       to 49507 0.0276
## 3      and 43405 0.0242
## 4        a 43047 0.0240
## 5       of 35627 0.0199
```

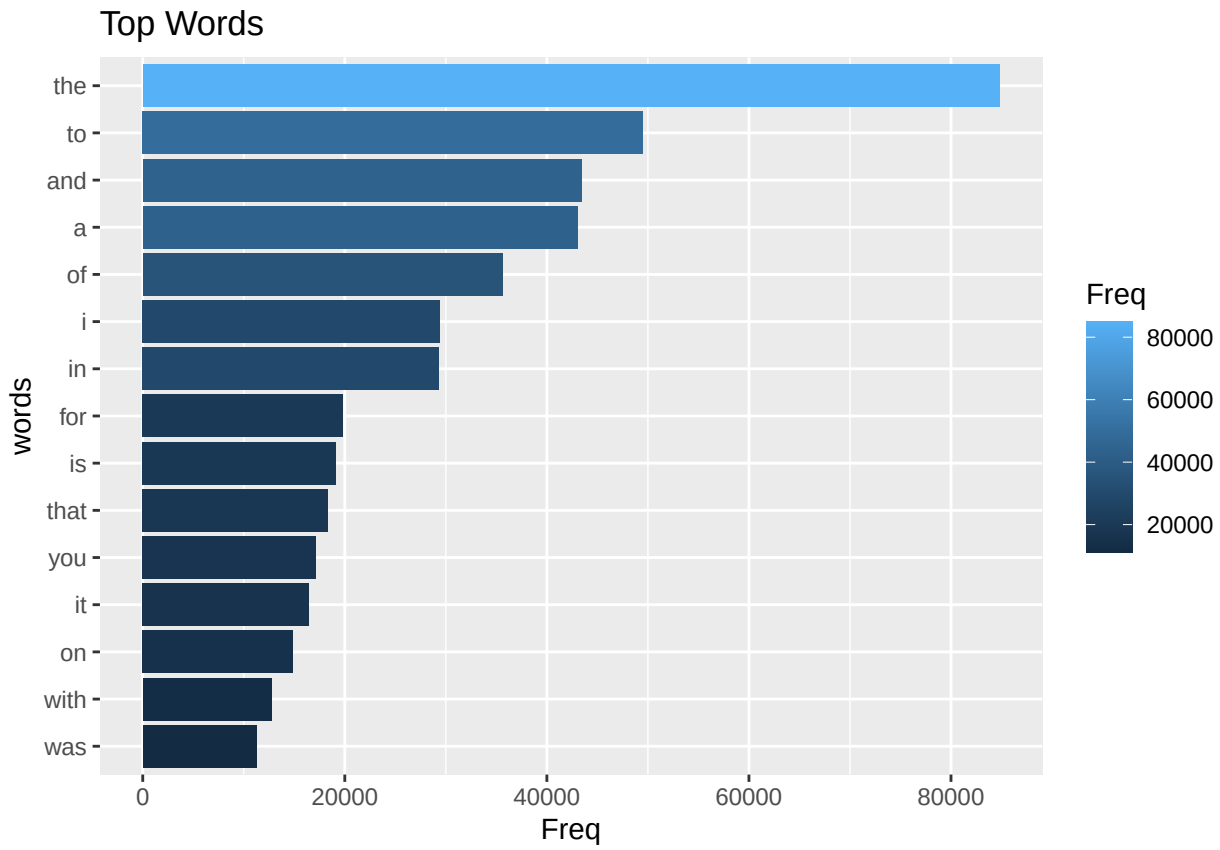
```
##   bigrams Freq   Prob
## 1  of the 7493 0.0044
## 2  in the 7446 0.0043
## 3  to the 3850 0.0022
## 4  on the 3635 0.0021
## 5 for the 3529 0.0021
```

```
##         trigrams Freq  Prob
## 1     one of the  587 4e-04
## 2       a lot of  546 3e-04
## 3 thanks for the  402 2e-04
## 4        to be a  330 2e-04
## 5    going to be  326 2e-04
```

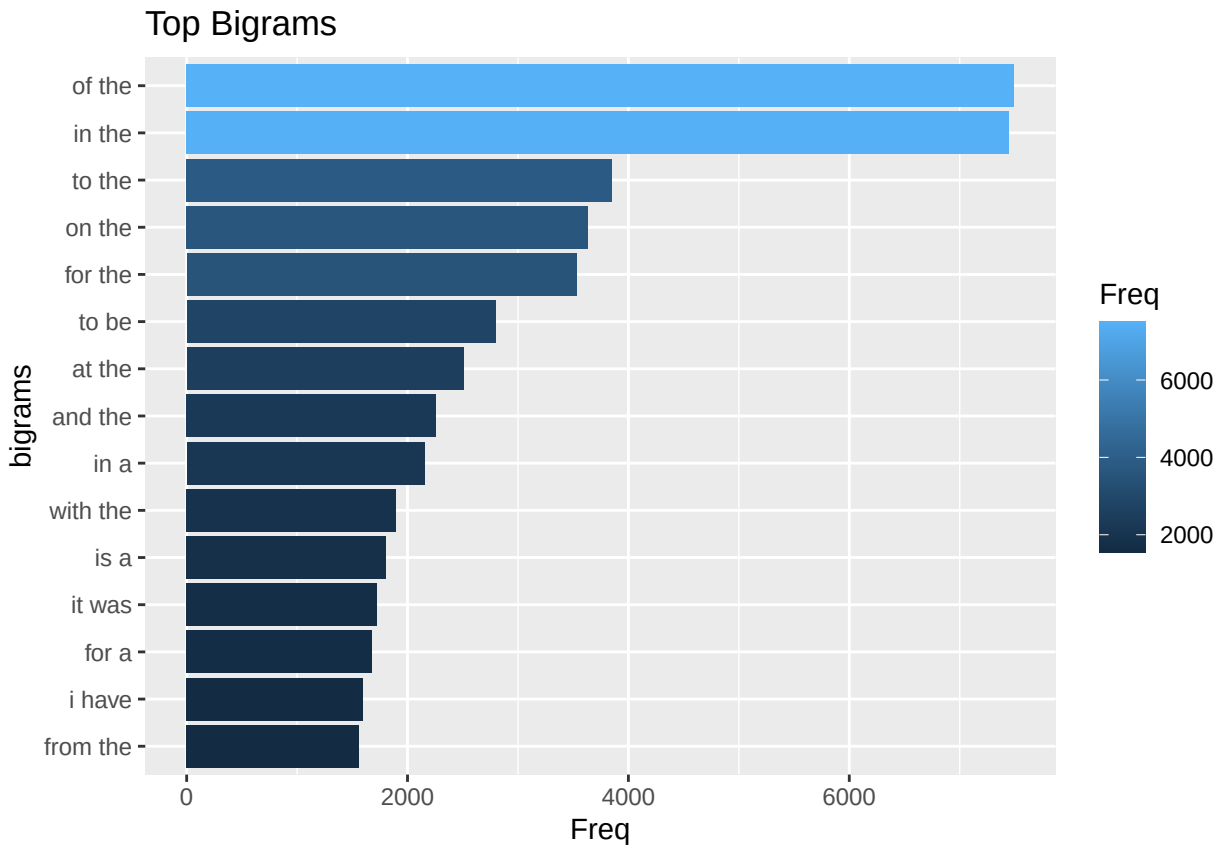
```
## Write
# write.csv(unitbl, file="unitbl.csv")
# write.csv(bitbl, file="bitbl.csv")
# write.csv(tritbl, file="tritbl.csv")
```

Here I present graphs of n-gram frequencies.

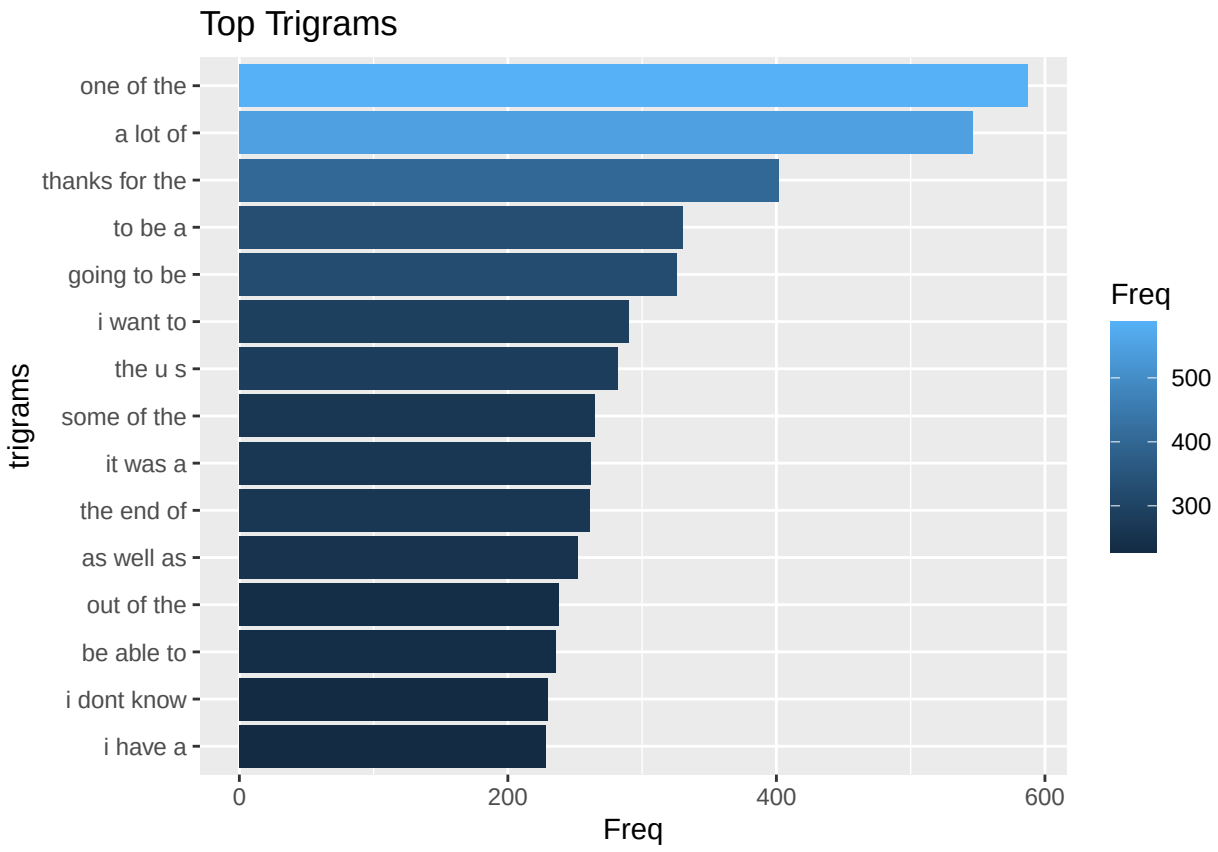
```
## Plot n-gram frequencies
# Unigrams
ggplot(data=unitbl[1:15, ], aes(x=reorder(unigrams, Freq), y=Freq)) +
  geom_bar(stat="identity", width=0.9, aes(fill=Freq)) + coord_flip() +
  labs(x="words") + ggtitle("Top Words")
```




```
# Bigrams
ggplot(data=bitbl[1:15, ], aes(x=reorder(bigrams, Freq), y=Freq)) +
  geom_bar(stat="identity", width=0.9, aes(fill=Freq)) + coord_flip() +
  labs(x="bigrams") + ggtitle("Top Bigrams")
```




```
# Trigrams
ggplot(data=tritbl[1:15, ], aes(x=reorder(trigrams, Freq), y=Freq)) +
  geom_bar(stat="identity", width=0.9, aes(fill=Freq)) + coord_flip() +
  labs(x="trigrams") + ggtitle("Top Trigrams")
```


Top Trigrams
one of the
a lot of
thanks for the
to be a
going to be
i want to
the u s
some of the
it was a
the end of
as well as
out of the
be able to
i dont know
i have a
trigrams
0
200
400
600
Freq
Freq
500
400
300

Modeling

Because trigrams are the highest Ngram used, the model will only be able to suggest the next word based on trigrams that match the input text's previous two words, a pre-bigram. If the pre-bigram is not found in the trigram table, or it's probability is not sufficient enough for a safe recommendation, the model "backs off" to suggest words based on bigrams that match only the previous word, the pre-unigram. This backing off is implemented in the function below.

`backoff()` returns the last n words from an input character of words.

```
#' Returns the last n words of a text string
#' @param text text to backoff
#' @param n number of words at end of text to return
#' @details requires 'stringr' package
backoff <- function(text, n){
  if (trimws(text) == ""){
    stop("Input words")
  }
  if(n <= 0){
    stop("n must be > 0")
  }
  if(n > length(str_split_1(text, " "))){
    stop("n cannnot be longer than length of text")
  }

  words <- character()
  while (n > 0){
    word <- word(text, -n)
    words <- paste(words, word)
    n <- n-1
  }
  trimws(words)
}
```

The model must be able to suggest a word even if no matching pre-Ngrams are found and no suggestion can be drawn from the Ngram table. In such a case, it is easy and sufficient enough to suggest the most common words that appear in the corpus.

Model 1

One approach are markov chains. Markov chains predict the condition of the next state based only on the condition of the previous state by storing the probability of each state in a probability transition matrix.

`predict1()` predicts words to follow an Ngram from Ngram transition matrix.

```
#' Helper function for model2, predicts words based on markoc chain transition matrix
#' @param ngram ngram to predict on
#' @param tranmat transition matrix
#' @param numpreds number of predictions
predict1 <- function(ngram, tranmat, numpreds){
  if (! ngram %in% names(tranmat)){
    return("No prediction. Make sure to use respective 'tranmat' for 'ngram'")
  }
  prob <- tranmat[ngram, ]
  #prob <- prob[prob > 0]
  prob <- sort(prob, decreasing=T)
  c <- c()
```

```
  for (i in 1:numpreds){
    c <- c(c, word(names(prob[i]), -1))
  }
  c <- c[!is.na(c)]
  return(c)
}
```

`model1()` is the prediction function which uses markov chains to predict words following an Ngram.

```
#' Predicts words using markov chains
#' @param ngram ngram to predict for
#' @param uni_bi_tri markov chain estimate/transition matrix for respective Nrams
#' @param numpreds number of predicitons to return
model1 <- function(ngram, uni, bi, tri, numpreds){
  len <- length(str_split_1(ngram, " "))

  if (len == 1){
    if (!ngram %in% names(uni$estimate)){
      return(NULL)
    }
    else{
      words <- predict1(ngram, uni$estimate, numpreds)
    }
  }

  if (len == 2){
    if (!ngram %in% names(bi$estimate)){
      ngram <- backoff(ngram, 1) #get last word
      if (!ngram %in% names(uni$estimate)){
        return(NULL)
      }
      else{
        words <- predict1(ngram, uni$estimate, numpreds)
      }
    }
    else{
      words <- predict1(ngram, bi$estimate, numpreds)
    }
  }

  if (len >= 3){
    ngram <- backoff(ngram, 3) #get last 3 words
    if (!ngram %in% names(tri$estimate)){
      ngram <- backoff(ngram, 2) #get last 2 words
      if (!ngram %in% names(bi$estimate)){
        ngram <- backoff(ngram, 1) #get last word
        if (!ngram %in% names(uni$estimate)){
          return(NULL)
        }
        else{
          words <- predict1(ngram, uni$estimate, numpreds)
        }
      }
      else{
```

```
        words <- predict1(ngram, bi$estimate, numpreds)
      }
    }
    else{
      words <- predict1(ngram, tri$estimate, numpreds)
    }
  }
  return(words)
}
```

Below I build markov chains for the Ngrams. 'MC' in variable names indicates Markov Chain. I can only use the first 5,000 lines of the Ngrams to save on memory.

```
## Build markov chains of unigrams, bigrams, and trigrams
uniMC <- markovchainFit(unigrams[1:5000])
biMC <- markovchainFit(bigrams[1:5000])
triMC <- markovchainFit(trigrams[1:5000])
```

Testing Model 1

Testing the output of a model involves taking random Ngrams and their actual following word from a test set and using the Ngram as input in the model. Then the actual following word of the input Ngram can be compared to the output of the model. Repeating this step numerous times and assessing the number of accurate predictions would provide an accuracy of the model.

Below I implement functions that uses our `testset` to grab random Ngrams and use them to assess a model's accuracy.

`getRandomWords()` returns a dataframe of input words and the following word based on an Ngram. The input words can be used as input in a model, and the following word can be used as the expected output of the model.

```
#' Returns a table of ngrams, the first 'numgrams' words of the ngram, and the following word of the ng
#' @param lines lines of text to sample from
#' @param numgrams type of ngram (i.e. 2, 3)
#' @param samples number of lines to sample from lines
getRandomWords <- function(lines, numgrams, samples){
  lines <- sample(lines, samples, replace=F)
  df <- data.frame()
  for (line in lines){
    ngrams <- ngram(line, numgrams)
    for (ngram in ngrams){
      c <- ngram
      input <- word(ngram, 1, numgrams-1)
      actWord <- word(ngram, -1)
      df <- rbind(df, c(c, input, actWord))
    }
  }
  colnames(df) <- c("Ngram", "Input", "Next")
  return(df)
}
getRandomWords(testset, 3, 10)[1:5, ]
```

```
##                Ngram         Input   Next
## 1 jarman then spent  jarman then  spent
## 2 then spent barely   then spent barely
```

```
## 3    spent barely a spent barely      a
## 4     barely a year     barely a   year
## 5       a year here       a year   here
```

The following code presents a rough estimate for the how accurate model1 is.

```
model1count <- 0
x <- 100
for (i in seq(x)){
  words <- getRandomWords(testset, 3, x)
  sugs <- model1(words$Input[i], uniMC, biMC, triMC, 15)
  if (words$Next[i] %in% sugs){
    model1count <- model1count + 1
  }
}
system.time(model1("blah blah blah this is", uniMC, biMC, triMC, 5)); model1count / x
```

```
##    user  system elapsed
##   0.007   0.003   0.015
```

```
## [1] 0.13
```

Model 2

Model 1 is not very accurate and cannot handle onobserved Ngrams. Model 2 will deal with this by implementing a loose version of the Katz Backoff model; For observed Ngram, the most probable words are returned. But for unobserved Ngrams, a probability is assigned based on the probability of the n-1Gram. Moreover, if the amount of possible words is too low (and/or the probability for one suggested word is too high), then the same method can be applied to suggest more likely words. For example, the bigram "when this" only appears once in the Trigram Table, and it's followed by the word "happens." That is a fine suggestion, but we can suggest more and maybe even better words if we only consider the unigram "this," which has many more matching bigrams.

`matchingBigrams()` and `matchingTrigrams()` take an input text and returns a text-matching Ngram table ("this" as in input will return a table of bigrams whose first word is "this"). The returned table also adjusts the probabilities of the Ngrams such that they are relative to only the matched Ngrams.

```
#' Returns the bigram table where bigrams first word matches text.
#' The probababilities of the bigrams are adjusted to represent those of matches.
#' @param text character to match
#' @param bitbl table of bigrams
#' @param adjProb adjust bigram probabilites to represent that of only matched bigrams?
#' @details requires 'stringr' package
matchingBigrams <- function(text, bitbl, adjProb=T){
  text <- word(text, -1) #redfine text to previous word
  newbitbl <- bitbl[word(bitbl$bigrams, 1) == text, ]
  if (nrow(newbitbl) == 0){
    return(NULL)
  }

  if(adjProb==T){
    probs <- c()
    newFreqSum <- sum(newbitbl$Freq)
    for (freq in newbitbl$Freq){
      probs <- c(probs, freq / newFreqSum)
    }
    newbitbl$Prob <- probs
```

```
    return(newbitbl)
  }
  return(newbitbl)
}

#' Returns the trigram table where trigrams first two words matches text
#' @param text character to match
#' @param bitbl table of trigrams
#' @param adjProb adjust trigram probabilites to represent that of only matched trigrams?
#' @details requires 'stringr' package
matchingTrigrams <- function(text, tritbl, adjProb=T){
  if (length(str_split_1(text, " ")) <= 1){
    stop("'text' must be at least two words long to match it to a trigram.")
  }

  text <- word(text, -(2:1)) #redfine text to previous 2 words
  newtritbl <- tritbl[word(tritbl$trigrams, 1) == text[1] &
                        word(tritbl$trigrams, 2) == text[2], ]
  if (nrow(newtritbl) == 0){
    return(NULL)
  }

  if(adjProb==T){
    probs <- c()
    newFreqSum <- sum(newtritbl$Freq)
    for (freq in newtritbl$Freq){
      probs <- c(probs, freq / newFreqSum)
    }
    newtritbl$Prob <- probs
    return(newtritbl)
  }
  return(newtritbl)
}
```

```
matchingTrigrams("when this", tritbl); matchingBigrams("this", bitbl)[1:5, ]
```

```
##                trigrams Freq Prob
## 27186 when this happens    4    1
```

```
##          bigrams Freq       Prob
## 34       this is 1089 0.14975248
## 174    this year  387 0.05321782
## 268    this week  291 0.04001650
## 445 this morning  203 0.02791529
## 459 this weekend  200 0.02750275
```

```
matchingBigrams("bruh", bitbl)
```

```
## NULL
```

```
matchingTrigrams("when this", tritbl)
```

```
##                trigrams Freq Prob
## 27186 when this happens    4    1
```

`model2()` used the sorted frequency tables of Ngrams and backs off to n-1grams when given unseen Ngram

inputs.

```
model2 <- function(text, unitbl, bitbl, tritbl, n){
  words <- c()
  len <- length(str_split_1(text, " "))

  if (trimws(text) == ""){
    words <- c(words, as.character(unitbl$unigrams[1:n]))
    return(words)
  }

  if (len == 1){
    matches <- matchingBigrams(text, bitbl)
    if (!is.null(matches)){
      if (nrow(matches) < n){
        words <- c(words, as.character(unitbl$unigrams[1:n]))
      }
    }
    words <- c(words, as.character(word(matches$bigrams[1:n], -1)))
    return(words)
  }

  if (len >= 2){
    text <- backoff(text, 2) #get previous 2 words (preBigram)
    matches <- matchingTrigrams(text, tritbl)
    if(is.null(matches)){ #if no matches, backoff to n-1Gram
      text <- backoff(text, 1) #get previous sord (preUnigram)
      matches <- matchingBigrams(text, bitbl)
      if (is.null(matches)){ #if no matches, return most common words
        words <- c(words, as.character(unitbl$unigrams[1:n]))
      }
      #if not enough words, backoff to unigrams
      if (!is.null(matches)){
        if (nrow(matches) < n){
          words <- c(words, as.character(unitbl$unigrams[1:n]))
          }
      }
      words <- c(words, as.character(word(matches$bigrams[1:n], -1)))
    }
    #if not enough words, backoff to unigrams
    if (!is.null(matches)){
      if (nrow(matches) < n){
        text <- backoff(text, 1)
        matches <- matchingBigrams(text, bitbl)
        words <- c(words, as.character(word(matches$bigrams[1:n], -1)))
      }
    }
    words <- c(words, as.character(word(matches$trigrams[1:n], -1)))
  }
  return(words)
}
```

Testing Model 2

```
model2count <- 0
x <- 100
for (i in seq(x)){
  words <- getRandomWords(testset, 3, x)
  sugs <- model2(words$Input[i], unitbl, bitbl, tritbl, 15)
  if (words$Next[i] %in% sugs){
    model2count <- model2count + 1
  }
}
system.time(model2("this is", unitbl, bitbl, tritbl, 5)); model2count / x
```

```
##    user  system elapsed
##   0.499   0.005   0.507
```

```
## [1] 0.35
```

Model2 predicts the next word exactly with about 42% accuracy. Given that a number of words are suggested at once, this is quite good. It also runs faster than model1 and can handle unseen Ngrams. Model2 will be used in the applicaiton.

Below the validation set is used to assess the model.

```
x <- 100
model2valcount <- 0
for (i in seq(x)){
  words <- getRandomWords(validset, 3, x)
  sugs <- model2(words$Input[i], unitbl, bitbl, tritbl, 15)
  if (words$Next[i] %in% sugs){
    model2valcount <- model2valcount + 1
  }
}
model2valcount / x
```

```
## [1] 0.41
```

Model2 performs with similar accuracy using the validation set.