Data Science Specialization Course: Capstone Project Report *

by Kamran Rasim Asgarov

Abstract

This paper describes the process of building a probabilistic English language model, capable of predicting words based on previous context. The model is trained on a corpus consisting of blog posts, online newspaper articles and twitter feeds. The basic model was smoothed by Kneser-Ney and pruned by Seymore-Rosenfeld methods. In addition, a simple Shiny application was built to showcase the model's capabilities. The end result is a 5-gram model, which recognizes over 40,000 words and 12,000,000 word combinations, is able to handle both unknown words and unknown combinations of words, and has a perplexity score of 205.

*The author gratefully acknowledges the assitance of various members of the stackoverflow community.

Table of Contents

1. Introduction

This model started out as a Capstone Project (final assignment) for the Data Science specialization course, offered by Johns Hopkins Bloomberg School of Public health via Coursera.org [1]. The original assignment was designed in collaboration with SwiftKey[2], who provided a large, multi-lingual corpus of random blog posts, newspaper articles and twitter feeds.

The students were tasked to use R in order to train an ngram model on said corpus. The model had to be capable of predicting the next word in a sentence, based on the preceding words, with reasonable speed and accuracy. Once the model was complete, students were asked to create an interactive Shiny[3], where a user could enter some words and have the next one predicted by the application. Similar technology is currently being used in numerous smartphone keyboard applications, such as the one designed by SwiftKey.

After the assignment has been submitted, the author has continued to work on the project, expanding the model from 3-grams to 5-grams, incorporating a new pruning method and optimizing the R code to run faster and more efficiently.

For those wishing to replicate this project, please note that the model was built on a computer running Windows 7, with 8GB of RAM. While this is technically sufficient to generate the desired results, it does require RStudio to be restarted during certain steps due to "out of memory" errors. Thus, a higher amount of RAM is strongly recommended.

2. Project goal

This paper aims to present the code necessary to construct a probabilistic 5-gram English language model. It has capable of predicting the next word in a sentence, given some previous context. The model should be able to correctly handle words outside of its vocabulary and make predictions for unknown combinations of known words.

Furthermore, the model must be lean enough in terms of memory and RAM requirements to operate on an average modern computer, and to generate predictions reasonably fast.

3. Ngram overview

An ngram model is a common way of estimating the conditional probability of a word in a sentence, where "ngram" refers to a sequence of "n" words. Given a sentence "s" with "l" words in it, the probability of that sentence can be calculated as a product of the conditional probabilities of each word [5]:

[1]The information about the course may be found here: https://www.coursera.org/specializations/jhu-data-science.

[2]SwiftKey is a company famous for its Android/iOS keyboard application. The company profile is available here: https://swiftkey.com/en/company/.

[3]Shiny is an R package, which allows to create interactive web applications. More information is available here: https://shiny.rstudio.com/.

$$p(s) = p(w_1) \times p(w_2|w_1) \times p(w_3|w_2w_1)...p(w_l|w_1...w_{l-1})$$

For long sentences, it will be quite difficult to calculate the conditional probability of the last word, due to their long histories (i.e. the number of words preceding them). Hence, we can approximate the conditional probability of these words by considering only the latest part of their histories[5]. Below is an example of a bigram model, where the conditional probability is based on a single preceding word:

$$p(s) \approx \prod p(w_i|w_{i-1})$$

Hence, a 5-gram model will consist of 5 word ngrams, consisting of 4 word history "h" and the fifth word "w". The conditional probability of ngram "hw" can be calculated by dividing the count of "hw", by the sum of counts of all 5-grams with the same history:

$$p(hw) = C(hw) \sum_{i=1}^{n} C(hw_i)$$

Thus, the first major step would be to generate a frequency table, listing the counts of all unique 5grams occurring within the corpus (body of text used to train the model).

4. Reading the corpus

The first step is to load the required R packages, download the corpus and unzip it.

```
library(quanteda)
library(parallel)
library(caret)
library(readr)
library(data.table)
library(stringr)
library(gtools)

URL <- "https://d396qusza40orc.cloudfront.net/dsscapstone/dataset/
Coursera-SwiftKey.zip"

download.file(url = URL, destfile = "swift.zip", method = "libcurl")

rm(URL)

unzip(zipfile = "swift.zip")
```

The corpus contains text data in English, Russian, German and Finnish, but for the purposes of this project, we will only be working with the English texts. There is a separate file for each source of text :

- News
- Blogs
- Twitter

Note that some of the sentences contain escape characters, which cause certain read functions to exit at that sentence and ignore the rest. Through some trial and error it was determined that "read_lines" function from the "readr" package does not have that issue.

```
fun1<- function(x){

    temp <- read_lines(x, locale = locale(encoding = "UTF-8"))

    paste(temp, sep = "", collapse = "")
}

cl<-makeCluster(detectCores()-1)

invisible(clusterEvalQ(cl, c(library(readr), library(quanteda),
                             library(dplyr))))

clusterExport(cl, "fun1")

tok.s <- parLapply(cl = cl, X = c("~/final/en_US/en_US.blogs.txt",
                                  "~/final/en_US/en_US.news.txt",
                                  "~/final/en_US/en_US.twitter.txt"),
                   fun = fun1)
```

In order to limit the complexity of the final model, it was decided to omit the paragraph structures and train the model on individual sentences. Hence, the next step would be to tokenize the text into separate sentences. It should be noted that "read_lines" does not do a very good job of that, occasionally reading a whole paragraph as a sentence. Thus, after the text was read, all paragraphs were merged together and split into sentences again with the help of the "tokens" function from "quanteda" package.

The end result were 3 corpora, one for each source, split into sentences.

```
tok.s <- parLapply(cl = cl, X = tok.s, fun = tokens, what = "sentence")

stopCluster(cl)

rm(fun1)
```

5. Preparing train and test sets

While it would be interesting to study the differences in writing styles between blogs, articles and twitter, the aim of this project was to create a general language model. Hence

it would be best to combine these three corpora into one, while taking note each sentence's original source. A numeric vector is created, which assigns "1" to sentences from blogs, 2 for news articles and 3 for twitter posts.

```
source <- c(rep(1, length(tok.s[[1]][[1]])),
            rep(2, length(tok.s[[2]][[1]])),
            rep(3, length(tok.s[[3]][[1]])))

tok.s <- c(tok.s[[1]][[1]], tok.s[[2]][[1]], tok.s[[3]][[1]])

tok.s <- data.table(text = unlist(tok.s), source = source)

rm(source)
```

Next, this large corpus needs to be split into the test and train sets, which must have an equal proportion of sentences from each source.

This was achieved by using "createDataPartition" function from the "caret" package to split up the data set, with 60% of sentences going to train and 40% to test set. Lastly, in order to reduce the future vocabulary size, all letters in each set were converted to lowercase.

```
set.seed(768943)

Intrain <- createDataPartition(y = tok.s$source, p = 0.6, list = FALSE)

train <- tok.s[Intrain, 1]

test <- tok.s[-Intrain, 1]

test<- char_tolower(test$text)

write(x = test, file = "test.txt")

rm(Intrain, tok.s, test)

train <- char_tolower(train$text)

write(x = train, file = "train.txt")
```

6. Setting vocabulary

The training set will be tokenized by word in order to create a document-feature matrix, which will show the respective frequency of each word in the corpus. The matrix is in turn converted to a data table for easier handling.

```
temp <- tokens(x = paste(train, collapse = " "), what = "word",
               remove_punct = TRUE, remove_hyphens = TRUE)

n1grams <- dfm(temp)

rm(temp)

n1grams <- data.table(words = as.character(colnames(n1grams)),
                      count = colSums(n1grams))

dim(n1grams)
```

```
## [1] 1000678       2
```

The following step was part of the original assignment of the Data Science Specialization course, which asked the students to filter out profanities from the vocabulary of the model, so that they do not appear among the future word predictions. This can be achieved with some regular expressions and the "str_which" function from the "stringr" package. Together they allow to filter out all possible versions of a given profane word from the vocabulary.

```
profanity <- c("^ass(holes?|face|es|munchers?)?$", "^buttholes?$",
               "^cunts?$", "^cumm?(ing|shots?|stains?)?$",
               "^bitche?(s|z)?$", "^sluts?(bags?)$", "^jackass?(es)?$",
               "^fuck(s|er|ing|ed(up)?|tards?|heads?)$]?", "^bastards?$",
               "^cock(s|block(s|ers?)?|suckers?)?$", "^whores?$",
               "^shit(s|z|heads?)?$", "^piss?(ed|er)?$", "^wankers?$",
               "^bollocks?$", "^turds?$", "^faggots?$", "^nigg?ers?$",
               "^gooks?$", "^dick(s|heads?|rid(e|ing)?|wads?)?$",
               "^boob(s|ies)?$")

z <- c(0)

invisible(lapply(X = 1:length(profanity), FUN = function(i){

    x <- str_which(string = n1grams$words, pattern = profanity[i])

    z <<- c(z, x)
}))

z <- z[-1]

n1grams<-n1grams[-z]

rm(profanity, z)
```

Furthermore, to simplify the model, all words with the apostrophe sign are removed. At this stage, the 1-gram discount is calculated as well, to be used in a future stage of the project.

```
n1grams <- n1grams[-str_which(string = n1grams$words,
                              pattern = "[^a-z//']+"),]

n1grams <- n1grams[order(count, decreasing = TRUE),]

D1 <-  sum(n1grams$count == 1) / (sum(n1grams$count == 1) +
                                    2 * sum(n1grams$count == 2))

dim(n1grams)
```

```
## [1] 699977      2
```

At this stage, the vocabulary has close to 700,000 words, which is way too large. A model with such a large vocabulary will produce an even greater number of 5-grams, which would be far beyond the processing capabilities of the authors computer. Hence, the vocabulary shall be further reduced, but removing words which have a low frequency count in the corpus. This cutoff point can be determined by building a table, which calculates the percentage of left over counts and vocabulary length, if all words below a given count were removed.

Due to the structure of English language, words such as "the", "I", "to" and etc occur much more frequently than the rest. Referred to as "stopwords", they will heavily skew the leftover counts in their favor and as such, will be temporarily removed from the vocabulary.

```
temp <- n1grams[!(n1grams$words %in% stopwords())]

table <- data.table(i = 0, ratio.count = 0, ratio.length = 0)

invisible(lapply(X = 1:50, FUN = function(i){

    z <- sum(temp$count[temp$count >= i]) / sum(temp$count)

    l <- length(temp$count[temp$count >= i]) / length(temp$count)

    table <<- rbind(table, data.table(i = i, ratio.count = z,
                                      ratio.length = l))
}))

table[25:35,]
```

```
##       i ratio.count ratio.length
```

```
##  1: 24   0.9555262   0.06694322
##  2: 25   0.9547754   0.06549281
##  3: 26   0.9540749   0.06419387
##  4: 27   0.9533985   0.06298782
##  5: 28   0.9526828   0.06175890
##  6: 29   0.9520209   0.06066287
##  7: 30   0.9513800   0.05963830
##  8: 31   0.9507650   0.05868803
##  9: 32   0.9500761   0.05765774
## 10: 33   0.9494222   0.05671033
## 11: 34   0.9488342   0.05588438
```

```
rm(temp, table)

n1grams <- n1grams[count >= 32]
```

As can be seen from the table, removing words which occur less than 32 times will only reduce the total word count by 5%, but vocabulary length will be reduced by 94%.

```
dim(n1grams)
```

```
## [1] 40523     2
```

7 Encoding words.

The primary way of storing ngrams in this project is in the form of a frequency table, with the first column listing all unique ngrams which can be found in the corpus and the second column listing their respective frequencies. Due to considerations discussed in an upcoming section of this project, it is necessary to store and generate the frequency table from 5-grams, as well as 4-, 3, 2- and 1-grams, referred to as lower order ngrams.

Since R loads all its objects directly into RAM, these frequency will quickly become prohibitively large, due to the following reasons.

Firstly, the ngrams themselves get longer with each order (combinations of 5 words are longer than combinations of 4 words), thus requiring more memory to be stored. Secondly, there are more unique ngrams of higher order than lower order (for example - there are more unique 5 word combinations that 4 word), so the frequency tables of higher order will have more rows, requiring yet more RAM.

One way to circumvent this issue, without further reducing corpus size or vocabulary length, is to replace each vocabulary word with a 3 character alphanumeric code. Since most words are longer than 3 characters, this will lead to shorter ngrams, which will reduce the memory required to store them. Furthermore, there are more than enough possible combinations of digits and lowercase letters to encode the entire vocabulary.

```
length(permutations(n = 36, r = 3, v = c(letters, 0:9)))
```

```
## [1] 128520
```

The "permutations" function from the "gtools" package is well suited to create such a code.

```
wordcode <- permutations(n = 36, r = 3, v = c(letters, 0:9))

wordcode <- apply(X = wordcode, MARGIN = 1, paste, collapse = "")

n1grams <- n1grams[, .(words,  ngrams = wordcode[1:length(n1grams$words)],
                       count)]

rm(wordcode)

head(n1grams)
```

```
##    words ngrams   count
## 1:   the    012 2837026
## 2:    to    013 1653390
## 3:   and    014 1443190
## 4:     a    015 1427074
## 5:    of    016 1204194
## 6:    in    017  985963
```

The next step is encoding all words in the corpus, so that in future they can be tokenized into ngrams. This process is rather time consuming and is a prime target for parallel processing. Hence, in order to accommodate the higher RAM demands of parallel processing, the corpus will be converted to a single string of text, which occupies far less memory. In order to preserve the original sentences, special word markers will be added to the beginning and end of the sentences, so that they maybe split up again later on.

```
object.size(train)

train <- as.character(lapply(X = train, function(x){

    paste("ZdZstrYzX", x, "ZdZendYzX")
}))

object.size(train)
```

Now that all sentences have beginning and end markers, they shall be encoded along with all the other words.

```
n1grams <- rbind(n1grams, data.table(words = "ZdZendYzX", ngrams = "eee",
                                     count = length(train)))

n1grams <- rbind(n1grams, data.table(words = "ZdZstrYzX", ngrams = "sss",
                                     count = length(train)))
```

One last task remains to accomplish, before the corpus can be encoded - teaching the model to handle unknown words. One way to accomplish this is to change all out-of-vocabulary words in the corpus into a special "unknown" token and treat it as a regular word when building ngrams (see chapter 4, p36 of [2]). In future, if a user inputs and unknown word into the interactive application, the model will convert it to the "unknown" token and use ngrams containing it to make relevant predictions.

```
n1grams <- rbind(n1grams, data.table(words = "UNK", ngrams = "zzz",
                                     count = 0))
```

Previously, a large number of words has been excluded from our vocabulary. Most of these words are slang, wrongly spelled, URLs, hash tags or come from foreign languages. Hence, it makes intuitive sense to use such words to simulate "unknown" inputs into the model.

Next, the actual process of encoding may begin. Firstly, the entire corpus is combined into a single text string and a tokens function is used to separate it into individual words.

```
t <- length(train)

train <- list(train[1 : round(t/3)], train[round(t/3+1) : round(t*2/3)],
              train[round(t*2/3+1) : t])

train <- lapply(X = train, FUN = function(x){

    as.character(tokens(x = paste(x, collapse = " "), what = "word",
        remove_numbers = TRUE, remove_punct = TRUE, remove_hyphens = TRUE))
})
```

Secondly, each word in the corpus is replaced with its corresponding 3 character code. Words without a match in the vocabulary are replaced with the "unknown" token.

```
tocode <- function(data){

    l <- length(data)

    cl<-makeCluster(detectCores()-1)

    clusterExport(cl, c("data", "n1grams"))
```

```
    clusterEvalQ(cl, library(stringr))

    result <- parLapply(cl = cl, X = list(data[1 : round(l/3)],
                                          data[round(l/3+1) : round(l*2/3)],
                                          data[round(l*2/3+1) : l]),
                                          fun = function(x){

                                              words <- n1grams$words

                                              ngrams <- n1grams$ngrams

                                              lapply(x, function(i){

                                                  ngrams[words == i]
                                                  })})
    stopCluster(cl)

    result <- as.character(c(result[[1]], result[[2]], result[[3]]))

    result[result == "character(0)"] <- "zzz"

    paste(result, collapse = " ")
}
train <- lapply(X = train, FUN = tocode)

write(as.character(train), file = "codetrain.txt")
```

8. Generating ngrams

With the corpus encoded, the 1-grams can be generated from it, by using the "quanteda" package.

```
my.dfm <- function(data, n) {

    ngrams <- tokens(x = data, what = "word", ngrams = n, concatenator = "")

    ngrams <- dfm(ngrams)

    data.table(ngrams = as.character(colnames(ngrams)),
               count = colSums(ngrams))
}
```

Due to limited available RAM, the corpus was split up in 3 parts and tokenized one by one.

```
n1grams.a1 <- my.dfm(data = train[[1]], n = 1)

n1grams.a2 <- my.dfm(data = train[[2]], n = 1)

n1grams.a3 <- my.dfm(data = train[[3]], n = 1)
```

This produces 3 1-gram frequency tables, which need to be merged.

```
n1grams.a <- merge(x = n1grams.a1, y = n1grams.a2, by = "ngrams", all = TRUE)

n1grams.a <- merge(x = n1grams.a, y = n1grams.a3, by = "ngrams", all = TRUE)

rm(n1grams.a1, n1grams.a2, n1grams.a3)
```

Once the tables are merged, zeros are assigned to the missing counts and they are summed together to get the total count for each unique 1gram in the corpus.

```
names(n1grams.a) <- c("ngrams", "count1", "count2", "count3")

n1grams.a$count1[is.na(n1grams.a$count1)] <- 0

n1grams.a$count2[is.na(n1grams.a$count2)] <- 0

n1grams.a$count3[is.na(n1grams.a$count3)] <- 0

n1grams.a <- n1grams.a[, .(ngrams, count = rowSums(x = n1grams.a[, -1]))]
```

Lastly, the newly made 1gram frequency table is merged with the vocabulary.

```
n1grams <- merge(x = n1grams, y = n1grams.a, by = "ngrams", all.y = TRUE)

n1grams <- n1grams[, -3]

names(n1grams) <- c("ngrams", "words", "count")

rm(n1grams.a)

write.csv(n1grams, "n1grams.csv")

rm(n1grams)
```

```
fread("backup/n1grams.csv")[, -1]

head(n1grams)
```

```
rm(n1grams)
```

Up next is the code to generate 2-grams. At this stage, it is necessary to mention the importance of "start of sentence" markers, which are "sss" in this model. The final prediction model should be able to offer word predictions for the beginning of a sentence, i.e. before the user has began typing. To achieve this, the "sss" marker is added before the first word in a sentence, so that it can have a history of its own. Then, whenever the final model is asked to predict the first word of a sentence, it can look at the counts of ngrams, whose histories consist of the "sss" marker.

Normally, when generating 2-grams, a single "sss" marker is added, but due to pruning considerations (discussed in upcoming chapter), two are added at this stage.

The rest of the code is the same as for 1-grams, except for additional step to calculate the 2-gram discount at the end, which will be necessary in the next section of the project.

```
train <- lapply(train, function(x){

    str_replace_all(string = x, pattern = "sss", "sss sss")
})

n2grams.1 <- my.dfm(data = train[[1]], n = 2)

n2grams.2 <- my.dfm(data = train[[2]], n = 2)

n2grams.3 <- my.dfm(data = train[[3]], n = 2)

n2grams <- merge(x = n2grams.1, y = n2grams.2, by = "ngrams", all = TRUE)

n2grams <- merge(x = n2grams, y = n2grams.3, by = "ngrams", all = TRUE)

rm(n2grams.1, n2grams.2, n2grams.3)

names(n2grams) <- c("ngrams", "count1", "count2", "count3")

n2grams$count1[is.na(n2grams$count1)] <- 0

n2grams$count2[is.na(n2grams$count2)] <- 0

n2grams$count3[is.na(n2grams$count3)] <- 0

n2grams <- n2grams[, .(ngrams, count = rowSums(x = n2grams[, -1]))]

z <- str_which(string = n2grams$ngrams, pattern = "eee")

n2grams <- n2grams[-z]
```

```
D2 <- sum(n2grams$count == 1) / (sum(n2grams$count == 1) +
                                   2 * sum(n2grams$count == 2))

write.csv(n2grams, "n2grams.csv")

rm(n2grams)
```

The code for generating 3-grams, 4-grams and 5-grams is largely the same, so it will be omitted for the sake of brevity.

9. Calculating conditional probabilities

At this stage, all ngrams frequency tables have already been created and saved on the drive and it is now the time to calculate the conditional probability of each ngram. Unfortunately, the previously discussed way of calculating it has a serious drawback of making the probabilities of all ngrams sum to 1.

This is an issue, because our corpus is not large enough to contain all possible combinations of the words in the vocabulary. Hence, if the probability of ngrams present in the system sums up to 1, the model will assign zero probability to unseen ngrams. Thus, if a user inputs a combination of words, which was not found in the training corpus, the model will be unalbe to predict a word.

This problem can be solved by discounting some probability mass from the seen ngrams and redistributing it to unseen ngrams. There are a number of different approaches to this and one of the most effective smoothing algorithms is the interpolated Kneser-Ney smoothing [5]. The basic working principle of this model is illustrated by an example:

Without going into great detail here, this algorithm calculates conditional probability by the following formula[6]:

$$P_{KN}(w|h) = \frac{max(\{0, C'(hw) - D_{|h|}\})}{S(h)} + \gamma P_{KN}(w|\hat{h})$$

$$C'(hw) = \begin{cases} 0, & \text{if } |hw| > N \\ C(hw), & \text{if } |hw| = N \\ |\{v : C(vhw) > 0\}|, & \text{otherwise} \end{cases}$$

$$S(h) = \sum_{v} C'(hw)$$

$$\gamma(h) = \frac{|\{v : C'(hv) > 0\}| D_{|h|}}{S(h)}$$

where "w" is a single word, "h" is history, i.e. other words preceding "w", "$\hat{h}w$" is the ngram "hw" with the first word removed, "N" is the highest order of ngrams in

the model (5 in this case), "$D_{|h|}$" is model discount for a given history and "C'(hw)" is modified Kneser-Ney count of an ngram.

Note that in the case of 5-grams, "S(h)" is the actual count of an ngram's history. For all ngrams of lower order, "S(h)" is the sum of unique prefixes for every ngram, which has "h" as its history.

Subsequently, the term "prefix" will be used to denote a single word which appears before a given sequence of words and "suffix" to denote a single word which appears after a given sequence of words. As shown in the formula, the Kneser-Ney (KN) count of ngram is :

- The actual count of the ngram, when working with ngrams of the highest order (5grams in this case)
- Number of unique prefixes of a given ngram for all lower orders.

With that in mind, S(h) is:

- The actual count of the ngram's history, when working with ngrams of the highest order (5grams in this case)
- Sum of unique prefixes of all ngrams, which contain this given history, otherwise.

Lastly, the first term of the numerator of γ is the number of unique suffixes of a given history, for both higher and lower order ngrams, because in this model all ngrams have at least 1 unique prefix.

Thus in order to calculate the KN (Kneser-Ney) conditional probability of an ngram, we need the following components :

- KN count of the ngram.
- S(h)
- Discount.
- Suffix count of the ngram's history.
- Backoff probability.

One way to calculate the number of unique prefixes/suffixes that an ngram has, is to take the list of unique ngrams present in the model, remove the first/last words respectively and count the number of times each ngram appears. For example, if we remove the first word of all 5-grams and discover that the 4-gram "to eat lots of" appears 5 times, it means that it must have 5 unique prefixes. The prefixes themselves are irrelevant, just their number.

These are the formulas used to remove the prefix/suffix of an ngram:

```
rmprefix <- function(x){

    str_replace(string = x, pattern = "^[a-z0-9]{3}", replacement = "")
}

rmsuffix <- function(x){

    str_replace(string = x, pattern = "[a-z0-9]{3}$", replacement = "")
}
```

The following code shows the prefix/suffix calculations performed on 5-grams. It begins by loading in the full list of unique 5-grams and splitting it into 4 parts, as it occupies too much RAM to be processed at once. Subsequently, the first quarter of 5-grams remains loaded in RAM, while the 3 quarters are saved to the drive and unloaded to free up additional RAM for running the code.

```
ngrams <- n5grams$ngrams

save(n5grams, file = "n5grams")

rm(n5grams)

l <- length(ngrams)

ngrams <- list(ngrams[1 : round(l/4)],
               ngrams[(round(l/4) + 1) : round(l*2/4)],
               ngrams[(round(l*2/4) + 1) : round(l*3/4)],
               ngrams[(round(l*3/4) + 1) : l])

temp <- ngrams[[1]]

ngrams <- ngrams[-1]

save(ngrams, file = "ngrams")

rm(ngrams)
```

The following formula takes the list of 5-grams as an input, removes the prefix/suffix and outputs the result as a character vector.

```
getaddon <- function(data){

    cl <- makeCluster(detectCores()-1)
```

```
    clusterExport(cl =cl, varlist = c("data", "rmprefix", "rmsuffix"))

    clusterEvalQ(cl = cl, expr = library(stringr))

    prefix <- parLapply(cl = cl, X = data, fun = function(x){

        lapply(X = x, FUN = rmprefix)
    })

    prefix <- paste(unlist(prefix), collapse = " ")

    suffix <- parLapply(cl = cl, X = data, fun = function(x){

        lapply(X = x, FUN = rmsuffix)
    })

    stopCluster(cl)

    suffix <- paste(unlist(suffix), collapse = " ")

    c(prefix, suffix)
}
```

The formula is applied to all 4 quarters of the 5-gram list. Once done, the results are collated into two list :

1. 5-grams with prefixes removed.
2. 5-grams with suffixes removed.

```
raw.1 <- getaddon(temp)

load("ngrams")

temp <- ngrams[[1]]

ngrams <- ngrams[-1]

save(ngrams, file = "ngrams")

rm(ngrams)

raw.2 <- getaddon(temp)

load("ngrams")
```

```
temp <- ngrams[[1]]

ngrams <- ngrams[-1]

save(ngrams, file = "ngrams")

rm(ngrams)

raw.3 <- getaddon(temp)

load("ngrams")

temp <- ngrams[[1]]

raw.4 <- getaddon(temp)

rm(temp, ngrams)

n5prefix <- paste(raw.1[[1]], raw.2[[1]], raw.3[[1]], raw.4[[1]],
                  collapse = " ")

n5suffix <- paste(raw.1[[2]], raw.2[[2]], raw.3[[2]], raw.4[[2]],
                  collapse = " ")

rm(raw.1, raw.2, raw.3, raw.4)
```

Next, a 4-gram frequency table is built from each list.

```
save(n5prefix, file = "n5prefix")

save(n5suffix, file = "n5suffix")

rm(n5suffix)

prefix <- dfm(tokens(x = n5prefix, what = "fastestword"))

prefix <- data.table(ngrams = as.character(colnames(prefix)),
                     prefix = colSums(prefix))

rm(n5prefix)

load("n5suffix")

suffix <- dfm(tokens(x = n5suffix, what = "fastestword"))
```

```
suffix <- data.table(ngrams = as.character(colnames(suffix)),
                     suffix = colSums(suffix))

rm(n5suffix)
```

The two frequency tables are merged together. Resulting table shows the number of unique prefixes/suffixes for each 4-gram.

```
n4addon <- merge(x = prefix, y = suffix, by = "ngrams", all = TRUE)

rm(prefix, suffix)

n4addon$prefix[is.na(n4addon$prefix)] <- 0

n4addon$suffix[is.na(n4addon$suffix)] <- 0

save(n4addon, file = "n4addon")
```

Next we merge the prefix/suffix counts with the 4-gram frequency table.

```
n4grams <- fread("backup/n4grams.csv")[, -1]

n4grams <- merge(x = n4grams, y = n4addon, by = "ngrams", all = TRUE)

rm("n4addon")
```

Currently, there is a problem with prefix counts for 4-grams which begin with the sentence start marker ("sss"), because they have only one unique prefix in the model - "sss". This tends to greatly downplay the importance of these ngrams and fails to account for the fact that there is usually another sentence in the text preceding the current one. Hence, in order to correct for this, the prefix count for such ngrams is substituted with their actual counts [3]. Hence, the actual counts will be used for 5gram and all ngrams starting with "sss". All other counts will be replaced by the number of unique prefixes an ngram has, as illustrated by this formula[3]:

$$a(w_1^n) = \begin{cases} C(w_1^n), & \text{if } n = N \text{ or } w_1 = \text{"sss"} \\ |v : c(vw_1^n) > 0|, & \text{otherwise} \end{cases}$$

The relevant code for 4-grams is shown below.

```
z <- str_which(string = n4grams$ngrams, pattern = "^sss")

n4grams$prefix[z] <- n4grams$count[z]
```

```
rm(z)

save(n4grams, file = "n4grams")

rm(n4grams)
```

In order to calculate the Kneser-Ney conditional probability for 5-grams, it is necessary to know the suffix count of an ngram's history, as well as the S(h), which in this case the the count of an 5-grams history. This can be accomplished by first, merging the 5-grams frequency table with the 5-gram suffix vector. Since the suffix vector consists of 5-gram with the suffix removed, it is essentially contains the history of every corresponding 5-gram.

However, the current RAM constraints do not allow to do this directly, so in order to circumvent this obstable the "ngrams" column of 5-gram table is temporarily removed and replaced with the suffix/history vector.

```
load("n5grams")

temp <- n5grams$ngrams

save(temp, file = "temp")

rm(temp)

n5grams <- n5grams[, -1]

gc()

load("n5suffix")

n5suffix <- as.character(tokens(x = n5suffix, what = "fastestword"))

n5grams <- data.table(n5grams, ngrams = n5suffix)

rm(n5suffix)

gc()
```

Subsequently, The "left_join" function from "dplyr" package is used to merge 5-gram and 4-gram tables below, because unlike the regular merge function, this one preserves the row order of the 5-gram table.

```
load("n4grams")
```

```
n5grams <- left_join(x = n5grams, y = n4grams[, -c("prefix")], by = "ngrams")

names(n5grams) <- c("count", "ngrams", "history.count", "history.suffix")

n5grams <- n5grams[, c("count", "history.count", "history.suffix")]

rm("n4grams")

gc()
```

Finally, once merging is complete, "ngrams" column is restored to the 5-gram table.

```
load("temp")

n5grams <- data.table(ngrams = temp, n5grams)

rm(temp)

save(n5grams, file = "n5grams")

rm(n5grams)
```

```
##             ngrams count history.count history.suffix
## 1: 01201201204w469     1             1              1
## 2: 0120120121p601b     1             1              1
## 3: 012012012zzz1jb     1             1              1
## 4: 01201201401v02q     1             1              1
## 5: 01201201403p1wn     1             1              1
## 6: 0120120149u730x     1             1              1
```

Very similar code is ran from 4-grams to 1-grams and will be omitted for the sake of brevity.

At this stage each ngram frequency table will be updated with the relevant prefix/suffix counts, which allows for the KN conditional probabilities to be calculated for all ngrams. Starting from 1grams. It should be noted that the formula for calculating 1-gram Kneser-Ney conditional probability is slightly different from other ngrams[3]:

$$p(w_n) = u(w_n) + \gamma(\epsilon) \times \frac{1}{|vocabulary|}$$

where "ϵ" is the empty string.

```
load("n1grams")

## Individual ngram counts are no longer necessary.
```

```
n1grams <- n1grams[, -c("count")]

## Removing the leftover "end of sentence"" token.

z <- str_which(string = n1grams$ngrams, pattern = "eee")

n1grams <- n1grams[-z]

rm(z)

getprob.n1 <- function(data){

    prefix.all <- sum(n2grams$prefix)

    suffix.all <- sum(data$suffix)

    V <- dim(data)[1] # Number of words in the vocabulary

    result <- lapply(X = data$prefix, FUN = function(x){

        (x - D1) / prefix.all + D1 * suffix.all / prefix.all/ V
    })

    unlist(result)
}

prob <- getprob.n1(n1grams)

n1grams <- cbind(n1grams, prob)

rm(prob)

save(n1grams, file = "n1grams")
```

Calculating the probability for 2-grams is a bit more involved, because once more, there isn't enough RAM to work with the entire table. Hence, prior to calculating the probabilities the "ngrams" and "count" columns are saved on the drive and removed from the table, since they are not necessary for this calculation.

```
load("n2grams")

temp <- n2grams[, c(1, 2)]

save(temp, file = "temp")
```

```
rm(temp)

n2grams <- n2grams[, -c(1, 2)]
```

Next, the backoff 1-gram probabilities are loaded and merged with the 2-gram table.

```
load("n2prefix")

n2prefix <- as.character(tokens(x = n2prefix, what = "fastestword"))

n2grams <- data.table(ngrams = n2prefix, n2grams)

rm(n2prefix)

n2grams <- left_join(x = n2grams, y = n1grams[, c("ngrams", "prob")],
                     by = "ngrams")

rm(n1grams)

## Remove the "ngrams" column.

n2grams <- n2grams[, -1]

names(n2grams) <- c("prefix", "history.prefix", "history.suffix",
                    "backoff.prob")
```

Now the probabilities can be calculated with the function below.

```
getprob <- function(data, D){

    cl <- makeCluster(detectCores()-1)

    on.exit(stopCluster(cl))

    clusterExport(cl = cl, varlist = c("data", "D2", "D3", "D4", "D5"))

    ## Note the column names of n2grams in previous code chunk.

    result <- parApply(cl = cl, X = data, MARGIN = 1, FUN = function(i){

        (i[[1]] - D) / i[[2]] + D * i[[3]] / i[[2]] * i[[4]]
        })

    as.numeric(result)
```

```
}

prob <- getprob(n2grams, D2)

n2grams <- cbind(n2grams, prob)

rm(prob)
```

Lastly the "ngrams" and "count" columns are restored to the table.

```
load("temp")

n2grams <- data.table(temp, n2grams)

save(n2grams, file = "n2grams")

rm(temp)
```

Probabilities for higher order models are calculated in a similar fashion.

10. Pruning the model

With the 5-gram probabilities fully calculated, the ngram model is complete. Unfortunately, it would require 15 GB of RAM just to load it and even then it would be quite slow at predicting words. Hence, the model needs to be cut down to a manageable size. This process is called "pruning" and there are a number of different techniques used to select which ngrams are to be removed.

It has been confirmed through repeated experimentation [1] that one technique which works well with Kneser-Ney (KN) smoothin in particular is the Seymore-Rosenfeld method. The SR (Seymore-Rosenfeld) pruning method [4] uses the following formula to calculate the weighted difference factor for a given ngram:

$$w.d.f. = K \times (log(\text{original prob}) - log(\text{backoff prob}))$$

where K is the Good-Turing discounted ngram count. The intuition behind this approach is that if the probability of the higher order ngram is very similar to the backoff probability, it makes sense to prune the higher order ngram and use the backoff probability instead.

The Good-Turing counts are calculated by the following formula[5]:

$$r^* = (r+1)\frac{N_{r+1}}{N_r}$$

where "r" is the actual count of a given ngram and "N_r" is the total number of ngrams which occur exactly "r" times in the corpus. This formula works well for ngrams with

low counts, but fails when it comes to ngrams with high counts. For example, there might be an ngram occurring 2679 times, but no ngram which occurs exactly 2680 times. Hence, prior to calculating Good-Turing counts, the "N_r" counts need to be smoothed.

A relatively simple way to smooth the "N_r" has been suggested by William Gale [7].It consists of fitting a linear regression of "N_r" on "r" and generating a set for predicted "N_r" counts ("pN_r") for each value of "r". Then, he proposed to use the actual "N_r" counts until they were no longer sufficiently different from "pN_r", then switch to "pN_r" and keep on using them. An "N_r" count is considered to be sufficiently different from a "pN_r" count if their difference is greater than 1.65 * standard derivation of the Good-Turing estimate. The variance of a Good-Turing estimate can be approximated by the following formula[7]:

$$Var(r_T^*) \approx (r+1)^2 \frac{N_{r+1}}{N_r^2}(1 + \frac{N_{r+1}}{N_r})$$

The code below follows these steps in reverse order:

1. Retrieve existing "r" and "N_r" counts for a given ngram table.
2. Fit linear regression of "r" on "N_r".
3. Generate "pN_r".
4. Calculate the point when "N_r" are switched to "pN_r".
5. Calculate discounted Good-Turing counts.
6. Calculate weighted difference factor.
7. Set up a pruning threshold.
8. Prune the ngram frequency table.

First, the "table" function is used with the "count" column to calculate existing "r" and "N_r" counts.

```
temp <- n5grams$count

rm(n5grams)

temp <- data.table(table(temp))

names(temp) <- c("r", "Nr")

temp$r <- as.numeric(temp$r)
```

Next, a function is defined to calculate the Good-Turing counts of each ngram.

```
gt.count <- function(data){

    ## This step averages out sparse ngram counts with the neighboring zeros.

    Zr <- sapply(X = 2 : (dim(data)[1] - 1), FUN = function(i){

        data$Nr[i] / (0.5 * (data$r[i+1]-data$r[i-1]))
    })

    last <- data$Nr[dim(data)[1]]/(0.5 * data$r[dim(data)[1]])

    Zr <- c(data$Nr[1], Zr, last)

    data <- data.table(data, Zr = Zr)

    ## Fitting simple linear regression.

    fit <- lm(formula = log10(Zr) ~ log10(r), data = data)

    r <- 1 : (max(data$r)+1)

    data <- merge(x = data, y = data.table(r), by = "r", all = TRUE)

    data$Nr[is.na(data$Nr)] <- 0

    data$Zr[is.na(data$Zr)] <- 0

    ## Generating pNr values.

    smooth <- predict(fit, newdata = data.table(r = r))

    smooth <- 10^as.numeric(smooth)

    data <- data.table(data, smooth = smooth)

    ## Next code section generates regular Good-Turing counts with
    ## existing Nr values and Smoothed Good-Turing counts with pNr
    ## values.

    gt.count <- as.numeric(lapply(X = 1 : dim(data)[1], FUN = function(i){

        (data$r[i]+1)*data$Nr[i+1]/data$Nr[i]
    }))
```

```
    sgt.count <- as.numeric(lapply(X = 1 : dim(data)[1], FUN = function(i){

        (data$r[i] + 1) * data$smooth[i+1] / data$smooth[i]
    }))

    ## Variance of Good-Turing counts is calculated next

    gt.var <- as.numeric(lapply(X = 1 : dim(data)[1], FUN = function(i){

        (data$r[i]+1)^2 * data$Nr[i+1] / data$Nr[i]^2 *
            (1 + data$Nr[i+1] / data$Nr[i])
    }))

    data <- data.table(data, gt.count = gt.count,
                       sgt.count = sgt.count, gt.var = gt.var)

    ##loop until we find counts with low difference

    for(i in 1 : dim(data)[1]){

        a <- abs(data$gt.count[i]-data$sgt.count[i])

        b <- 1.65*sqrt(data$gt.var[i])

        if(a <= b) {q <- i; break}
    }

    ## Original Good-Turing count is useq until "q", then only the predicted
    ## smoothed counts are used.

    final.count <- c(data$gt.count[1 : (q-1)],
                     data$sgt.count[q : dim(data)[1]])

    data.table(count = data$r, sgt.count = final.count)
}

temp <- gt.count(temp)
```

Now that the set of Good-Turing counts has been obtained, it is possible to calculate the weighted difference factor.

```
load("n5grams")

n5grams <- left_join(x = n5grams, y = temp, by = "count")
```

```
rm(temp)

wdf <- n5grams$sgt.count * (log(n5grams$prob) - log(n5grams$backoff.prob))
```

Lastly, a cutoff w.d.f. point is selected to reduce the table size by a 90%.

```
n5grams <- n5grams[, c("ngrams", "prob")]

n5grams <- data.table(n5grams)

gc()

n5grams <- n5grams[wdf >= 0.12 ,]

save(n5grams, file = "n5grams-final")

rm(n5grams, wdf)
```

Similar procedure is repeated for 4gram, 3gram and 2gram tables.

11. Calculating perplexity

At this stage the language model has been reduced to a manageable size and is ready to be assessed on it's accuracy. One way to accomplish this is to use the model to calculate perplexity of the test set. This metric can be calculated in the following way [5]:

1. Using the chain rule to calculate the probability of each sentence "t", by multiplying the conditional probabilities of each word in the sentence. Then, multiplying the sentence probabilities to get the total probability of the test set, "T".

$$p(t) = \prod_{i=1}^{t} p(w_i)$$

$$p(T) = \prod_{i=1}^{T} p(t_i)$$

2. Calculating cross entropy "$H_p(T)$" of the test set "T", where "W_t" is the number of words in the test set:

$$H_p(T) = -\frac{1}{W_T} log_2 p(T)$$

3. Lastly the perplexity is calculated as follows:

$$PP_p(T) = 2^{H_p(T)}$$

In order to calculate the probability of the test set, it first needs to be read.

```
test <- read_lines("test.txt", locale = locale(encoding = "UTF-8"),
                   n_max = 100000)
```

Next, the "beginning of sentence markers" are added and the set is tokenized by word.

```
test<- as.character(lapply(1:length(test), function(i){

    paste("sss sss sss sss", test[i])
}))

test <- tokens(x = test, what = "word", remove_numbers = TRUE,
               remove_punct = TRUE, remove_twitter = TRUE,
               remove_hyphens = TRUE, remove_url = TRUE)

test <- as.list(test)
```

The following chunk of code converts the words in test set to the 3 character alphanumeric code and creates 5-grams.

```
cl<-makeCluster(detectCores()-1)

clusterExport(cl, c("test", "n1grams"))

clusterEvalQ(cl = cl, expr = c(library(stringr), library(data.table),
                               library(quanteda)))

test <- parLapply(cl = cl, X = test, fun = function(x){  .

    as.character(lapply(X = x, FUN = function(i){

        temp <- n1grams[words == i, ngrams]

        if(length(temp) == 0) {temp <- "zzz"}

        temp
        }))})

ngrams <- parLapply(cl = cl, X = test, fun = function(x){

    as.character(tokens(x = paste(x, collapse = " "), what = "fastestword",
                        concatenator = "", ngrams = 5L))
})

stopCluster(cl)
```

Next, a function is defined to calculate the KN conditional probability of a given ngram.

```
ngram.prob <- function(x) {

    prob <- n5grams[ngrams == x, prob]

    if(length(prob) == 0) {

        x <- rmprefix(x)

        prob <- n4grams[ngrams == x, prob]

        if(length(prob) == 0) {

            x <- rmprefix(x)

            prob <- n3grams[ngrams == x, prob]

            if(length(prob) == 0) {

                x <- rmprefix(x)

                prob <- n2grams[ngrams == x, prob]

                if(length(prob) == 0) {

                    x <- rmprefix(x)

                    prob <- n1grams[ngrams == x, prob]
                }}}}
    prob
}
```

Subsequently, the probability of each sentence is calculated.

```
cl<-makeCluster(detectCores()-1)

clusterEvalQ(cl, c(library(stringr), library(quanteda)))

clusterExport(cl, c("ngrams", "n1grams", "n2grams", "n3grams", "n4grams",
                    "n5grams", "ngram.prob", "rmprefix"))

prob.test <- parLapply(cl = cl, X = ngrams, fun = function(x){

   temp <- as.numeric(lapply(X = x, FUN = ngram.prob))
```

```
   sum(log(temp, 2))
})

stopCluster(cl)

prob.test <- as.numeric(prob.test)
```

Finally, the perplexity of test set is calculated.

```
2^-(sum(prob.test)/length(unlist(ngrams)))  ## Calculating perplexity.
```

```
## [1] 205.274
```

12. Building the Shiny app

The ultimate goal of this assignment was to create a Shiny application, which would allow the user to submit a word or a sequence of words and have the model predict the next word. The first half of the application is the user interface, which consists of a set instructions for the app, a text field to input the text and 10 buttons, which show the words predicted by the model, from most to least likely.

```
library(shiny)
shinyUI(fluidPage(
    titlePanel(h1("Four-gram word prediction model")),
    sidebarLayout(

        ## Instructions to use the model

        sidebarPanel(h3("Instructions"), p("1. Please wait for the model to
            load for the first time."), p("2. Once the first 10 suggested
            words appear, you may begin by either typing some text or
            selecting one of the suggested starting words."), p("3. The model
            will update the choice of word predictions based on your input."),
            width = 4),

        mainPanel(

            ## Text input field.

            textInput("usertext", NULL, value = "", width = NULL,
                      placeholder = "Please type your text here."),

            ##Prediction buttons
```

```
            actionButton("pred1", ""), actionButton("pred2", ""),
            actionButton("pred3", ""), actionButton("pred4", ""),
            actionButton("pred5", ""), actionButton("pred6", ""),
            actionButton("pred7", ""), actionButton("pred8", ""),
            actionButton("pred9", ""), actionButton("pred10", "")
        ))))
```

The second part is the server, which accepts input from the u.i., runs it through the ngram model and displays the predictions back on the u.i. Since the majority of this code is part of single function, it will be displayed in whole, with comments explaining various steps.

```
## Loading the packages and ngram probability tables

library(shiny)
library(quanteda)
library(data.table)
library(stringr)

load("n1grams-final"); load("n2grams-final")
load("n3grams-final"); load("n4grams-final")
load("n5grams-final")

vocab <- c(n1grams$words)

## Most common words are used as default predictions, before the
## user enters anything.

unigrams <- c("the","and", "to", "a", "in", "of", "is", "for", "that", "on")

rmprefix <- function(x){
    str_replace(string = x, pattern = "^[a-z0-9]{3}", replacement = "")
}

getpred <- function(x){
    str_extract(string = x, pattern = "[a-z0-9]{3}$")
}

## Following code triggers when user types something

shinyServer(function(input, output, session) {

    data <- reactiveValues(predictions = unigrams)
```

```
observeEvent(input$usertext, {

    ## If the user inputs more than 1 sentence, only the latest
    ## senctence is selected.

    text <- unlist(str_split(string = input$usertext,
                             pattern = "[\\.\\?\\!\\;]"))

    ## The sentence is tokenized into words.

    text <- as.character(tokens(x = tail(text, 1), what = "word",
                remove_numbers = TRUE, remove_punct = TRUE,
                remove_twitter = TRUE, remove_hyphens = TRUE,
                remove_url = TRUE))

    ## Captials are converted to lowercase

    text <- char_tolower(text)

    ## Individual words are encoded

    text <- as.character(lapply(X = 1 : length(text), FUN = function(i){

        t <- paste("^", text[i], "$", sep = "")

        z <- str_which(string = n1grams$words, pattern = t)

        if(length(z) == 0) {"zzz"}

        else {n1grams$ngrams[z]}

    }))

    ## Beginnig of sentence markers are added, if there are not enough
    ## words to form a 4-gram.

    if(length(text) <= 4) {

        text <-c("sss", text)

        if(length(text) <= 4){

            text <-c("sss", text)
```

```
            if(length(text) <= 4){

                text <-c("sss", text)

                if(length(text) <= 4){

                    text <-c("sss", text)
                    }}}}

    text <- paste(text, collapse = " ")

    ## 4grams are formed from the encoded words

    temp <- tokens(x = text, what = "word", n = 4,
                   concatenator = "")$text1

    ## Last 4gram is chosen

    temp <- paste("^", tail(temp, 1), sep = "")

    ## All 5grams starting with this 4gram are selected

    temp <- n5grams[str_detect(string = ngrams, pattern = temp), ngrams]

    ## Keep the 15 most likely ngrams

    temp <- head(temp$ngrams, 15)

    ## Get the last word in 5gram sequence (the prediction)

    temp <- as.character(lapply(temp, getpred))

    ## Rremove predictions if they start of sentence or unknown tokens.

    predictions <- temp[temp != "zzz" & temp != "sss"]

    ## If there is less than 10 predictions, repeat the process
    ## with 4grams.

    if (length(predictions) >= 10) {

        data$predictions <<- head(predictions, 10)}

    else {
```

```
        temp <- tokens(x = text, what = "word", n = 3,
                       concatenator = "")$text1

        temp <- paste("^", tail(temp, 1), sep = "")

        temp <- n4grams[str_detect(string = ngrams,
                                   pattern = temp), prob]

        temp <- head(temp$ngrams, 15)

        temp <- as.character(lapply(temp, getpred))

        temp <- temp[temp != "zzz" & temp != "sss"]

        predictions <- unique(c(predictions, temp))

        if (length(predictions) >= 10) {
            data$predictions <<- head(predictions, 10)}

        #### TRUNCATED: the code repeats for 3-grams and 2-grams ####


                else {

                    temp <- unigrams

                    predictions <- unique(c(predictions, temp))

                    data$predictions <<- head(predictions, 10)}
                }}}

    ## Decode predictions into words.

    data$predictions <- as.character(lapply(X = 1 : 10,
                                            FUN = function(i){

        z <- str_which(string = n1grams$ngrams,
                       pattern = data$predictions[i])

        n1grams$words[z]

    }))

    ## Following code assigns 1 predition to each button, from
```

```
        ## most to least likely.

        updateActionButton(session = session, inputId = "pred1",
                           label = paste("1.", data$predictions[1]))
        updateActionButton(session = session, inputId = "pred2",
                           label = paste("2.", data$predictions[2]))

        #### TRUNCATED: the code repeats for buttons 3-9 ####

        updateActionButton(session = session, inputId = "pred10",
                           label = paste("10.", data$predictions[10]))
    })

    ## The next chunk pastes a prediction into the text input field,
    ## if the corresponding button is pressed.

    observeEvent(input$pred1,{
        updateTextInput(session = session, "usertext",
                        value = paste(input$usertext, data$predictions[1]))
    })
    observeEvent(input$pred2,{
        updateTextInput(session = session, "usertext",
                        value = paste(input$usertext, data$predictions[2]))

        #### TRUNCATED: the code repeats for buttons 3-9 ####

    })
    observeEvent(input$pred10,{
        updateTextInput(session = session, "usertext",
                        value = paste(input$usertext, data$predictions[10]))
    })
})
```

13. Bibliography

[1] Chelba Ciprian et al. "Study on Interaction between Entropy Pruning and Kneser-Ney Smoothing". In: *Proceedings of Interspeech*. 2010, pp. 2242–2245. URL: https://research.google.com/pubs/pub36472.html (visited on 02/25/2018).

[2] Jurafsky Daniel and Martin James H. *Speech and Language Processing. An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. 3rd ed. Draft. URL: https://web.stanford.edu/~jurafsky/slp3/ed3book.pdf (visited on 02/25/2018).

[3] Kenneth Heafield et al. "Scalable Modified Kneser-Ney Language Model Estimation". In: *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Sofia, Bulgaria: Association for Computational Linguistics, 2013, pp. 690–696. URL: http://www.aclweb.org/anthology/P13-2121 (visited on 02/25/2018).

[4] Seymore Kristie and Rosenfeld Ronald. "Scalable backoff language models". In: *Spoken Language, 1996. ICSLP 96. Proceedings., Fourth International Conference on*. Vol. 1. Oct. 1996, pp. 232–235. DOI: 10.1109/ICSLP.1996.607084. URL: citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.46.8369&rep=rep1&type=pdf (visited on 02/25/2018).

[5] Chen Stanley F. and Goodman Joshua. *An Empirical Study of Smoothing Techniques for Language Modeling*. Harvard Computer Science Group Technical Report. Harward, 1998. DOI: TR-10-98. URL: https://dash.harvard.edu/bitstream/handle/1/25104739/tr-10-98.pdf?sequence=1 (visited on 02/25/2018).

[6] Siivola Vesa, Hirsimäki Teemu, and Virpioja Sami. "On Growing and Pruning Kneser-Ney Smoothed *N*-Gram Models". In: *IEEE Transactions on Audio, Speech, and Language Processing* 15.5 (July 2007), pp. 1617–1624. DOI: 10.1109/TASL.2007.896666. URL: users.ics.aalto.fi/vsiivola/papers/TASLP2007.pdf (visited on 02/25/2018).

[7] Gale William A. "Good-Turing Smoothing Without Tears." In: *Journal of Quantitative Linguistics* 2 (1995). URL: l2r.cs.uiuc.edu/~danr/Teaching/CS598-05/Papers/Gale-Sampson-smoothgoodturing.pdf (visited on 02/25/2018).