Reproducible Report Submission

Rodrigo Bertollo de Alexandre

Friday, October 10, 2014

Contents

1 About the project

This aim of this capstone project is to develop a data scientist mind. Therefore, we are supposed to learn how to acquire, clean, explore, interpret, develop workarounds, create logical reasoning, develop codes, improve predictive accuracy, reduce computational runtime and model complexity, and for last, present your final findings.
In this report, we are supposed to present all we learn with this course up to the final goal. Use the acquired knowledge through the entire specialization from coding to a final end user application.

2 About the data

2.1 Brief description of the data

The data was acquired from the capstone project website under the *Announcements* page within the *Task 0 - Understanding the problem*. Within this file, existed different databases from different languages. The option for this project was to work with the *en_US* folder containing three files: *en_US.blogs.txt*, *en_US.news.txt*, *en_US.twitter.txt*.
These text files contain a massive amount of English written scripts from different origins. All together they sum up to 3 million and 337 thousand lines and 556 Mega Bytes.

2.2 Data cleaning

2.2.1 Non English words

Overall the data was full of "not-English" words and characters. Furthermore, since some of the data came from social media there as an immense amount of misspelling. As I previously shown in the milestone project, about 92% of the data had words that showed up 35 times or less. After the assemble of a dictionary containing *655110* English words (including nouns, verbs, names, names of cities and countries, swearwords and others), I was able to conclude that about 79% of the words with frequency of 35 times or lower was considered as "not-English" words or trash data. However, these low frequency non-English words were only removed from the n-grams. This approach was chosen because removing these words from the middle of sentences could alter some of the prediction model. Therefore, removing the whole n-grams containing the affected words would result in a more clean database and a more accurate prediction.
Choosing the number of occurrence of 35 was due a visual analysis in the data, aiming to remove a large amount of non-real words and still keeping some less frequent real English words that could also exist and not be present at the assembled dictionary.

2.2.2 Punctuations

Punctuations that would represent an end of sentence (*! ? ,*) where transformed to end periods. After this step, new lines were created for every period. In this way, during the n-gram phase, there were not going to be issues resulted by the merging two different phrases. Other unknown and unused punctuations where removed from the data. The dash (-) was kept in the data since it is also part of the English vocabulary.
Furthermore, in some cases there was a mixture of different apostrophes like 'and' which was interfering in the data prediction. Therefore all apostrophes where converted to the straight form.
For further details about the cleaning and processing of the files, please see attachment 1.

3 The prediction model

3.1 Assembling the n-grams

Due to memory RAM limit the clean database was divided into 8 different files with 1 million lines each.

```
clean_data1 <- matrix(clean_data[1:1000000])
clean_data2 <- matrix(clean_data[1000001:2000000])
clean_data3 <- matrix(clean_data[2000001:3000000])
clean_data4 <- matrix(clean_data[3000001:4000000])
clean_data5 <- matrix(clean_data[4000001:5000000])
clean_data6 <- matrix(clean_data[5000001:6000000])
clean_data7 <- matrix(clean_data[6000001:7000000])
clean_data8 <- matrix(clean_data[7000001:7644814])
```

Each of these 8 files where divided into n-grams using the **ngram package**. Since the files were line based, the function tryCatch({}, error=function(e){}) was used to avoid errors due to phrases with less words than the minimal required for each n-gram. New files were created based on n-grams of size 3-gram, 4-gram and 5-gram.

```
library("ngram")
#notice that the following steps were used for each of the 8 files, with n-grams of size 3, 4 and 5.
clean_data1 <- matrix(clean_data[1:1000000])
rm(clean_data)
```

```
ngram3_list1 <- apply(clean_data1, 1, function(x) tryCatch({ngram(x , n =3)}, error=function(e){}))
rm(clean_data1)
ngram3_1 <- rapply(ngram3_list1, function(x) as.matrix(get.ngrams(x)))
rm(ngram3_list1)
save(ngram3_1, file="ngram3_1.RData")
rm(ngram3_1)
```

Afterwards more cleaning and processing of each files was performed.

```
#load all ngram saved files back
#notice that the following steps were used for each of the 8 files, with n-grams of size 3, 4 and 5.
load("ngram3_1.RData")
#removing words that starts with "-".
ngram3_1 <- gsub("^[-]+", "", ngram3_1)
#removing words that starts with spaces.
ngram3_1 <- gsub("^[[:blank:]]+", "", ngram3_1)
#removing multiple spaces
ngram3_1 <- gsub("[[:blank:]]+", " ", ngram3_1)
#saving
save(ngram3_1, file="ngram3_1.RData")
#keeping n-grams with only the amount of words that it should have
library("qdap")
load("ngram3_1.RData")
ngram3_1 <- ngram3_1[wc(ngram3_1)==3]
save(ngram3_1, file="ngram3_1.RData")
rm(ngram3_1)
```

Afterwards a loop for separating files in alphabetical order was performed.

```
#separate by alphabetical order in different files (notice that a similar loop function was performed f
for(i in letters){
    for(n in 1:8){
        test <- get(paste("ngram3_", n, sep=""))
        #if the file does not exist create the file
        if(!exists(paste("With_",i,sep=""))){
            #if it starts with i and I
            if(i == "i"){
                assign(paste("With_",i,sep=""), test[grepl(test, pattern=paste("^[","iI","]", sep=""))]
            }
            #all other letters are lowercased
            else{
                assign(paste("With_",i,sep=""), test[grepl(test, pattern=paste("^[",i,"]", sep=""))])
            }
        }
        #if the file already exists join them toguether
        else if(exists(paste("With_",i,sep=""))){
            #if it starts with i and I
            if(i == "i"){
                assign(paste("With_",i,sep=""), c(get(paste("With_",i,sep="")), test[grepl(test, patter
            }
            #all other letters are lowercased
            else{
                assign(paste("With_",i,sep=""), c(get(paste("With_",i,sep="")), test[grepl(test, patter
```

```
            }
        }
    }
}
#files were saved depending on its n-gram name
#3-gram
save(With_a, file="With_a.RData")
#4-gram
save(With4_a, file="With4_a.RData")
#5-gram
save(With5_a, file="With5_a.RData")
```

After the execution of these loops and savings, n-grams with composed of single characters only where removed from the data. The same step was also used for transforming the data in a table with frequency and near its final form. At last all files where saved as **.CSV** with an unique function.

```
for(n in 3:5){
    if(n == 3){
        t = ""
    }
    else{
        t = n
    }
    for(l in letters){
        load(paste("With",t,"_",l,".RData", sep=""))
        ngram <- get(paste("With",t,"_",l, sep=""))
        ngram <- data.frame(table(ngram))
        ngram <- ngram[character_count(ngram$ngram) > n,]
        write.csv(ngram, file = paste("ngram",n,"_DF_",l,".csv", sep=""))
        rm(ngram)
        rm(list = paste("With",t,"_",l, sep=""))
        print(c("ngram n:",n))
        print(c("letter:",l))
    }
}
```

3.2 Removing "not-English" words

At last, the final n-gram was generated by removing the "not-English" as previously explained in the data-cleaning step (please see codes at the attachment 1).

3.3 Data storage

For testing of fastest performance of size and reading time of each different extension, multiple tests were performed.
At first, all data was saved as **.CSV** to facilitate reading with external programs like *Notepad++* and *Excel*. At first, multiple formats like **.CSV**; **.RData**, **bin.dat** and **.GZ** where tested for amount of disk space used. The result of this test concluded that a same file compressed as **.GZ** and **.RData** where the best way to store the data. These two formats occupy almost 75% less space than the **.CSV** previously used. Afterwards a test for reading speed was conducted with the two formats that occupy the least amount of space and the **.CSV**.

```
t1() #Rdata
user  system elapsed
   0.59    0.00    0.56
t2() #CSV
user  system elapsed
   2.47    0.04    2.53
t3() #GZ
user  system elapsed
   7.11    0.08    7.22
```

This result conclude that the best way to store and read files was with the use of standard R saving method: **.RData**.

3.4 The prediction function.

The prediction function is based on a few steps. First the software recognize which is the first letter in the word to be searched for. Afterwards, it recognizes if it is a 3-gram, 4-gram or a 5-gram search by the number of words present in the phrase. Then it loads the right n-gram that represent that exact first letter.
Afterwards the search for similar words occurs in 3 major steps:
1. If you have written 2 words, the software will search in a database composed by 3-grams.
2. If you have written 3 words, the software will search in a database composed by 4-grams, if there is less than 3 words for the prediction, the software will transform to a 2 word phrase and execute step 1.
3. If you have written 4 words or more, the software will select the last 4 words and search in a database composed by 5-grams, if there is less than 3 words for the prediction, the software will transform to a 3 word phrase and execute step 2.

Therefore, searches with 4 words or more are slower than the others; however, its accuracy also increase. In each of these last 3 steps, if any of the returned word is a swearword present in the "swearword dictionary" the function will remove this curse word as one possible next word prediction. Furthermore, if the new search result has any repeated word, these words are automatically removed from the prediction and a new step is followed until it completes the minimal 3 word necessary. If the function ends, and the minimal 3 words is not found, it returns the amount of words that were found ultil that last step.

Each n-gram search, besides searching for the exact word, also tries to find similar words within the database. This search consist by three steps:

1. `If the desired sequence is not found, last written word is divided in half, and a similar word with`
2. `If the desired sequence is not found, last written word is divided in half, and a similar word with`
3. `If the desired sequence is not found, the software picks only the last 3 characters and search for`

If none of these conditions is meet for each of the n-grams searches, the software will return:

`"Sorry no prediction for your word".`

This function could be modified to a reclusive function; however, due to lack of time I was not able to modify the function and simplify the codes.

To see the entirely function for the predicting model please see the attachment 2.

4 Attachments

Attachment 1 link 1 Attachment 1 link 2

Attachment 2 link 1 Attachment 2 link 2