Data Science Project: Word Prediction

David Pham

07.10.2014

Contents

1 Introduction

This is the reproducible report of the data science capstone project on 2014. The code is split into three main parts:

1. Data preprocessing, with installation of the required packages. In this part, we mostly clean the input to avoid to much noise in the modelling/fitting part of the process. Basically, we read each line of the text file and use a cleaning algorithm described later.
2. Modelling part, where all the transformation and adjustement are made in order to compute the probabilities. We used N-grams with skipped words (*n=2,3, skip=0,1*) as features for prediction and Good Turing adjusting and adjusted counts have been used to account for unseen words. The exact descriptions of these algorithms can be found on the github page of the project. Note that thanks to a careful implementation having in mind memory leaks, the fitting part on the whole data set takes about 1 hour.

3. Prediction part, where we try to predict the following word of a user input. The user argument needs to be cleaned as well and then we create a simple lookup in a huge table in order to provide the most likely next words.

Data visualisation (with ggplot2) is left to the reader as they have been check previously in the other report.

For lisibilty, descriptions and codes have been split, the descriptions of algorithms following as a linear flow, whereas code is appended at the end.

1.1 Pacakges requirements

This steps makes sure the required pacakges are installed.

```
pkgs <- c('tm', 'openNLP', 'RWeka', 'RcmdrPlugin.temis',
          'lsa','lda', 'kernlab', 'RTextTools', 'wordcloud', 'data.table',
          'tau')

install.packages(pkgs)
```

2 Data Preprocessing

The data have been cleaned in the following way:

- Removeal of punctation (except for the apostroph and the '-' as these might describe something different);
- Deletion numbers as they provide no help for prediction;
- Removal of most badwords and most stopwords;
- White spaces are replace by spaces;
- No stemming, because this requires to much memory to keep all the data (the stemmed and unstemmed one).

The result of this steps is then used in the modeling part.

3 Modelling

Here are the most important features about the modelling part.

- Bi- and trigrams have been counted and simulation of 4- and 5-grams have been fit by computing skip 1 words bigram and trigrams (function *CountNgramSkip* which is faster than *txtcount* from **tau**)

- Adjustement probability through Good Turing algorithm and adjusted count. Smoothing by back-off, but unnescesary for predition. Everything is in the function *Conditional-ProbNgram*;
- Heavily rely on parallel computing and kill cluster to avoid memory leaks, in the *CountNgramSkip*;
- Data.table are used extensively throught the *data.table.keyby* function;
- However it can not predict anything if the last word was unseen.

The code takes 1 hour to fit the model by computing the (skip) n-gram.

As stressed previously, the exact descriptions of the Good Turing, adjusted count and Back-off algorithms are found on the github page in the slides pdf, where the concept are explained clearly.

4 Prediction

Basically, the function cleans the string argument in order to have a common structure as the previous results. Then it splits the strings by spaces and find the last two words (or the last two word with a skip words in between) and then a lookup operation is performed in a table to provide the most likely output. The function returns the 20 most probables following words as a table.

It works in 99% of the case as long as the user does not want to game the function.

5 Possible improvements

In the following, some possible ideas worth following are described.

- Levenstein distance could be used to predict unfinished words;
- Faster implementation in c++ or java;
- Use of maybe 4-gram and also skip 2 words;
- Better assesment on how precise the prediction is.

6 Conclusion

It was a fun project and the main part of the project is inside the modelling part. Maybe it would require some tweaking on the part of the prediction part to account for unfinished words, but I think it would require a lot more time to proceed this. One appreciable feature is the speed of fitting of the algorithm, taking into account how **R** is built.

7 Code

In the following, the code used to do the project is appended for completeness. The comments should be helpful to understand how the functions are written.

7.1 Data Preprocessing

```
library('tm')

library('parallel')
options('mc.cores' =  4)

### Set the source directory here
#src.dir <- 'final/en_US_short/'  # testing purpose
src.dir <- 'final/en_US/'

src <- DirSource(src.dir)
en.t <- Corpus(src, readerControl = list(reader = readPlain, language= "en",
                        load = TRUE)) # A collection of 4 text

length(en.t)

### tm_Map(x, fn) apply f to all element of x
### fn can be  removeNumbers, removePunctuation, removeWords, stemDocument,
### stripWhitespace

### tmFilter and tmIndex for filtering.
### searchFullText function look for regular expresions inside the text
### doclevel = T, for usage to document level.

### We will perform white space elminiation and lowercase conversion with stopword removal.

### Tockenzation will be perform by the tm::TermDocumentMatrix function.
### /usr/share/dict/words for the dictionnary
RemovePunctuation <- function(plainTxtDoc){
  s <- plainTxtDoc$content
  s <- gsub("(\\w)-(\\w)", "\\1\1dd\\2", s)
  s <- gsub("(\\w)'(\\w)", "\\1\2dd\\2", s)
  s <- gsub("[[:punct:]]+", "", s)
  s <- gsub("\1dd", "-", s, fixed = TRUE)
  s <- gsub("\2dd", "'", s, fixed = TRUE)
  plainTxtDoc$content <- s
  return(plainTxtDoc)
}
```

```
CleanText <- function(corp.txt){
  transformations <- getTransformations()
  for (f in transformations){
    if (f == 'removeWords'){
      bw <- scan('bad_words.txt', character(), sep = '\n')
      corps.txt <- tm_map(corp.txt, get(f), bw)
      next
    }

    if (f == 'removePunctuation'){
      corp.txt <- tm_map(corp.txt, function(txt) RemovePunctuation(txt))
      next
    }

    if (f == 'stemDocument') next

    corps.txt <- tm_map(corp.txt, get(f))
  }
  return(corp.txt)
}


txt.clean <- CleanText(en.t)
save(txt.clean, file='final/en_US/en_US_corpus_clean.RData')
# save(txt.clean, file='final/en_US_short/en_US_corpus_clean_short.RData')
```

7.2 Modelling

```
# setwd('./Dropbox/coursera_capstone_project')

library('tau')
library('tm')
library('Matrix')
library("RWeka")
library('ggplot2')
library('data.table')
library('reshape2')
library('parallel')


load('final/en_US/en_US_corpus_clean.RData') # load txt.clean as a corpus
# load('final/en_US_short/en_US_corpus_clean_short.RData') # load txt.clean as a corpus
```

```r
##' Count n-gram by skipping one word
##'
##' Function to take any text taking lower case and counting the number
##' of sequence of words in the text by skpping one word a the time
##' @title Count skipped ngram
##' @param s, character vector
##' @param n the n-gram to consider
##' @return
##' @author David
CountNgramSkip <- function(s, n=3, skip=1){

  cl <- makeCluster(4)

  clusterEvalQ(cl, {
    library('data.table')
    library('parallel')
  })
  env <- new.env()
  assign('n',  n, env)
  assign('skip',  skip, env)
  clusterExport(cl, c('n', 'skip'), envir=env)

  res <- unlist(parLapply(cl, s, function(txt){
    txt <- tolower(txt)
    t.l <- strsplit(txt, ' ')[[1]]

    if (length(t.l) < 5) {return(data.table())}

    res <- vapply(1:(length(t.l)-(2*n-1)), function(idx){
      paste(t.l[idx+((skip+1)*(1:n)-(skip+1))], collapse=' ')
    }, character(1))

    return(res)
  }))

  stopCluster(cl)
  print('End of cluster')
  print(paste('n:', n, 'skip:', skip))
  res <- data.table(s=res)
  res.DT <- res[, .N, by='s'][order(-N)][N > 1]
  setnames(res.DT, 'N', 'count')
  return(res.DT)
}
```

```
FindNgram <- function(x, n=3){
  content <- iconv(x$content, "utf-8", "ASCII", sub="")
  print('Compute n-gram')
  res.ngram <- CountNgramSkip(content, n, 0)
  if (n==1) return(res.ngram)
  print('Compute skip n-gram')
  res.ngram.skip <- CountNgramSkip(content, n, 1)
  res.ngram.all <- rbind(res.ngram, res.ngram.skip)
  return(res.ngram.all)
}

### A Lot of the small word are used really often
CreateNGramTable <- function(corp, n, v.size=0){

  res.l <- lapply(corp, FindNgram, n=n)

  res.dt <- rbindlist(res.l)[, list(count=sum(count)), by=s]
  res.dt <- res.dt[res.dt$count>2]
  res.dt <- CleanNGram(res.dt)
  res.dt <- res.dt[, list(count=sum(count)), by=s]

  if(n>1 & v.size > 0){
    res.adj.count <- GoodTuringSmoothing(FreqNGramVector(res.dt, n, v.size),
                                         v.size) # Compute the smoothing count
    setkey(res.dt, count) # Join the two datasets
    setkey(res.adj.count, count)
    res.dt <- res.adj.count[J(res.dt)]
  }
  setkey(res.dt, s)
  return(res.dt)
}

CleanNGram <- function(n.gram.table){
  ngt <- n.gram.table
  ngt <- ngt[!grepl('- ', ngt[, s]),]
  ngt[, 's':= gsub('^-','', s)]
  return(ngt)
}

### Function to compute the "frequency of frequencies vector"
# http://en.wikipedia.org/wiki/Good%E2%80%93Turing_frequency_estimation
FreqNGramVector <- function(n.gram.table, n=1, v.size=9e5){
  res <- n.gram.table[, list(inv.freq=.N), by=count][order(count)]
  res <- rbind(data.table(count=0, inv.freq=v.size^2-sum(res$inv.freq)), res)
  return(res)
```

```
}

##' Implements a heuristic version of the Good-Turing Smoothing
##'
##' When there are no n-gram with r+1, one uses the the alpha smoothing version
##' @title Good-Turing Smoothing for adjusted count of n-grams
##' @param DT, result of the function FreqNGramVector
##' @param v.size, vocabulary size (number of row of the unigrams)
##' @param alpha smoothing parameter, default 0.00017
##' @return a data.table with column count, inv.freq and adj.count
##' @author david
GoodTuringSmoothing <- function(DT, v.size, alpha = 0.00017, ngram=3){
  n <- DT[, sum(inv.freq)]
  DT[, adj.count:=0.0]
  DT[1, adj.count:=inv.freq/n]

  for (i in seq_along(DT$count)[-1]){
    if(DT[c(i-1, i), diff(count)!=1]) {
      DT[i, adj.count:=(count+alpha)*n/(n+alpha*v.size^ngram)] # alpha adjustement
    } else {
      DT[i, adj.count:=(DT[i, count])*(DT[i, inv.freq]/DT[i-1, inv.freq])]
    }
  }
  return(DT)
}

##' Compute the Conditional probability of trigram
##'
##' Use 3- and 2-grams to compute the conditional probability of a sentence
##' @title Conditional Probability of Trigram computations
##' @param nb a data.table with the with column "s" and "count" where s is a three words str
##' @param ns  idem as trig but for bigrams ("s" contains two words sentence) n(gram-small)
##' @param n, the dimension of the n-gram of in nb
##' @return a data.table with the conditional probability for each trigrams
##' @author david
ConditionalProbNgram <- function(nb, ns, n){

  if (n == 1){
   res <-  nb[, list(s=s, count=count, adj.count=count,
                        cond.prob=count/sum(count))]
   return(res)
  }

  setkey(ns, s)
```

```
  nb[, key.b:= vapply(strsplit(nb$s, ' '), function(x){
      paste0(x[-length(x)], collapse=' ')
    }, character(1))]

  setkey(nb, key.b)
  ### Join and do the division on the adjusted count
  res <- ns[J(nb)][,list(key.w=s, s=i.s, adj.count=adj.count,
                         cond.prop=i.count/count)]
  setkey(res, s)
  return(res)
}

##' Discount Factor computation
##'
##' Use d as the sum of 1 - \sum_{w_2} \alpha (w_2, w_1)
##' @title Dicount Factor Computation for Back Off model
##' @param DT , result of function CondtionalProbNgram
##' @return a data.table with the discount factor compute as 1 - \sum_{w_2} \alpha(w_2, w_1)
##' @author david
ComputeDiscountFactor <- function(DT){
  res <- DT[, list(disc.f= 1 - sum(cond.prop)), by = key.w]
  setnames(res, 'key.w', 's')
  setkey(res, s)
  return(res)
}


GenerateAdjustedProbTables <- function(txt.clean){

  unigram <- CreateNGramTable(txt.clean, 1L)
  v.size <- nrow(unigram)
  print('End of unigram')

  gc()
  trigram <- CreateNGramTable(txt.clean, 3L, v.size)
  print('End of trigram computation.')

  bigram <- CreateNGramTable(txt.clean, 2L, v.size)
  print('End of bigram computation.')

  bprob <- ConditionalProbNgram(bigram, unigram, 2)
  setkey(bprob, 'key.w')

  gc()
```

```
  tprob <- ConditionalProbNgram(trigram, bigram, 3)
  setkey(tprob, 'key.w')

  print('End of conditional probability computation')


  prob.adj <- lapply(list(n2=bprob, n3=tprob),
                     function(DT) {
                       res <- DT[(DT[c('', ' '), -.I])]
                       res <- res[!grep('\\d', res$s),]
                       res[, adj.count:=NULL]
                       setkey(res, 'key.w')
                     })
  gc()

  return(prob.adj)
}

prob.adj <- GenerateAdjustedProbTables(txt.clean)
print('I finished to compute the probabilities. Now saving them.')
# save(prob.adj, file='prob.adj.Rdata')
save(prob.adj, file='prob.adj.all.Rdata')
```

7.3 Prediction

```
library('data.table')

load(file='prob.adj.all.Rdata') # load the prob.adj variable

##' Clean the String inputs from the users
##'
##' Cleans the strings for prediction
##' @title Clean input string
##' @param s, a string character
##' @return a clean version of the argument (lowercase, no punctuation)
##' @author david
CleanInputString <- function(s){
  s <- tolower(s)
  s <- gsub("(\\w)-(\\w)", "\\1\1dd\\2", s)
  s <- gsub("(\\w)'(\\w)", "\\1\2dd\\2", s)
  s <- gsub("[[:punct:]]+", "", s)
  s <- gsub("\1dd", "-", s, fixed = TRUE)
  s <- gsub("\2dd", "'", s, fixed = TRUE)
```

```
  return(s)
}


##' Word Prediction
##'
##' Predict the following words by looking up on the table and provide
##' the table of words with the adjusted probabilities
##' @title Predict next word
##' @param s, a string and we try to predict the next word
##' @param prob.adj.init a list of adjusted probability
##' @return a table with words
##' @author David
PredictNextWord <- function(s, prob.adj.init){

  s <- CleanInputString(s)

  s.l <- strsplit(tolower(s), ' ')[[1]]
  n <- min(length(s.l), 2)

  prob.adj <- lapply(prob.adj.init, function(x) copy(x))
  ## Predict usual n-gram
  idxs <- list((-(n-1):0) + length(s.l), (-2*(n-1):0) + length(s.l))

  out.l <- list()
  i <- 1 # bad
  for(idx in idxs){
    s. <- paste0(s.l[idx], collapse=' ')

    res <- prob.adj$n3[s.][order(-cond.prop)]
    idx.l <- 3

    if (any(is.na(res$s))){
      s.2 <- paste0(s.l[idx[-1]], collapse=' ')
      res <- prob.adj$n2[s.2][order(-cond.prop)]
      idx.l <- 2
    }

    out.res <- res[, list(predict.word =
                            unlist(lapply(strsplit(s, ' '), '[', idx.l)),
                          cond.prop)]
    out.l[[i]] <- head(data.frame(out.res), 10)
    i <- i + 1
  }
```

```
  ## Predict with skipngram
  out.res <- na.omit(rbindlist(out.l))
  out.res <- data.table(out.res)
  # print(out.res)
  out.res <- out.res[, list(cond.prop=mean(cond.prop)), by=predict.word]


  ## rownames(out.res) <- NULL
  return(data.frame(out.res))
}

s <- 'I expect to write a '
PredictNextWord(s, prob.adj)
```

7.4 Session info

```
sessionInfo()
```

```
## R version 3.1.1 (2014-07-10)
## Platform: x86_64-pc-linux-gnu (64-bit)
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=de_CH.UTF-8        LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=de_CH.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=de_CH.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=de_CH.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## loaded via a namespace (and not attached):
## [1] digest_0.6.4    evaluate_0.5.5  formatR_0.10    htmltools_0.2.4
## [5] knitr_1.6       rmarkdown_0.3.3 stringr_0.6.2   tools_3.1.1
## [9] yaml_2.1.13
```