[DATA-05] Text mining

Miguel-Angel Canela, IESE Business School

June 16, 2016

Introduction

A **corpus** is a large and structured set of texts. Broadly speaking, **text mining** is data mining applied to a corpus of documents. The text mining process can start when we already have the corpus, or building the corpus may be part of it. I assume in this lecture that the documents have already been captured and stored in electronic form, as plain text.

Most of the text data on which data scientists work has been captured from Internet. Specially relevant, for business, are the data captured from social networks. So, **web mining** has emerged as a new, promising subfield, similar to text mining but taking advantage of the extra information available in Web documents. These documents come in XML, HTML or JSON format, and include structural **markup**, which distinguishes parts of the document. The markup can be used in the mining process. Although it sometimes involves arbitrary and unpredictable choices by individual web designers, this disadvantage is offset by the amount of data available.

Preprocessing the corpus

To make text data useful, unstructured text data is converted into structured data (a table) for further processing. This conversion is performed by counting terms. But, previous to this, some transformations are be performed. This is called, generically, **preprocessing**. Tools for preprocessing text are available in many programming languages, including R.

Typical steps involved in the preprocessing step are:

- Conversion to lower case simplifies the preprocessing task and helps to avoid confusion when counting the occurrences of terms in the documents.
- Dropping numbers and punctuation.
- Dropping postal and email addresses, telephone numbers and URL's, since they only affect one particular document (there are exceptions to this).
- Eliminating **stopwords**, that is, articles and other words that convey little or no information. Lists of stopwords are available in Internet for practically any language you may be interested in. For the English language, a popular choice is the list of the SMART information retrieval system (`http://jmlr.csail.mit.edu/papers/volume5/lewis04a/a11-smart-stop-list/english.stop`), which I use in the example. Note that, although dropping stopwords is regarded as a preprocessing routine, stopwords can also be discarded after counting terms.
- Users of systems such as SMS or e-mail have created shortenings for communication. The resulting genre of communication is called **texting**. This has to be taken into account when mining text that comes from these sources. For instance, the word 'text' may appear as 'txt', or 'your' as 'ur'. Note that the last example is a stopword, so it would not be removed unless we add it to our stopword list.
- Replacing words that are synonyms, and plural and other variants of words by a single term. This is typically based on **stemming**. Stemmers for the English language use algorithms that remove common word endings, such as 'es', 'ed' and 's'. There is plenty of choice, but **Snowball** is probably the most popular stemmer. Stemming can be easier than in English in some languages, like Spanish, but it can involve infixes and prefixes in other languages like German. For instance, stemming transforms

'angegeben' (declared) into 'angeben' (declare). The stems are usually completed, using the first occurrence, or the most frequent term for every stem. I do not use stemming in the example, preferring to group terms in a more careful way.

Extra packages

I do not use in this example any specific package for text manipulation, to simplify the presentation (which is technical enough). So, all the functions used in the example are plain R. But it is worth mentioning two packages which are quite popular.

- The package `tm` has functions that render operations like dropping stopwords or stemming very easy, and also has a function for building term-document matrices. But this comes at the price of working with complex objects. Nevertheless, many people use `tm`, and other packages, such as `tm.plugin.webmining`, for capturing news, use `tm` objects. So, if you plan to work on this, it may be worth learning a bit about these objects.
- The package `stringr` contains the basic functions used in the example for preprocessing and counting terms, but under names which are intuitively clear and a syntax which is a bit more straightforward. For instance, the function `gsub`, which I use many times when cleaning the text, is called `string_replace_all` in `stringr, a name that suggests what the function does. Also, in this package, the order of the arguments is consistent across functions.

Regular expressions

Some of these transformations performed in the preprocessing stage are dramatically simplified by using **regular expressions**. A regular expression is a pattern that describes a set of strings. For instance, `[0-9]` stands for any digit, and `[A-Z]` for any capital letter. The square brackets indicate *any* of the characters enclosed. This is called a **character class**. Followed by a *plus* sign (`+`) these expressions indicate a sequence of any length. For instance, `[0-9]+` indicates any sequence of digits, so any number. Regular expressions are a whole chapter of programming, with entire books devoted to them, such as Friedl (2007). Beginners may also try the `regexr` website at `http://www.regexr.com`, which makes fun of learning regular expressions.

Special characters

Text imported from PDF or HTML documents and from devices like mobile phones may contain special symbols like the n-dash (`-`), the left/right quotation marks (‘‘, ‘, etc), or the three-dot character (…), which is better to control, to avoid confusion. Since the example is technical enough, I have eliminated these symbols in the SMS data used in the example of this lecture, or replaced them by ordinary characters (e.g. the right quote by an apostrophe). But if you capture text data on your own, you will probably find some of this in your documents. Even if the documents are expected to be in English, they can be contaminated by other languages: Han characters, German umlaut, Spanish ñ, etc.

Another source of trouble is that these special symbols can be encoded by different computers or different text editors in different ways. The typical **encodings** in the Western world are UTF-8 and Latin-1. R can deal with these two, but may have trouble with other encodings. I do not discuss encodings in this course, but, if you are interested, you may take a look at Korpela (2006).

Example: Spam SMS

Short Message Service (SMS) is the text communication service component of phone, web or mobile communication systems. It uses standardized communication protocols which allow the exchange of short text messages between fixed line or mobile phone devices. The 160 character limit of SMS was designed to accommodate the *communication of most thoughts* in English and Latin languages. According to the International Telecommunication Union (ITU), SMS has become a massive commercial industry. As of 2010,

there were 5.3 billion active users of SMS, globally sending 6.1 trillion messages. In the same year, Americans alone sent 1.8 trillion messages. Even in China, 98.3% of phone owners use SMS service, averaging 42.6 received messages weekly.

The growth of mobile phone users has lead to a dramatic increasing of **SMS spam** messages. SMS spam is any junk message delivered to a mobile phone as text messaging. Although this practice is rare in the West, it has been very common in some parts of Asia. In practice, fighting mobile phone spam is difficult, due to several factors, including the lower rate of SMS, which has allowed many users and service providers to ignore the issue, and the limited availability of mobile phone spam-filtering software.

Apparently, SMS spam is not as cost-prohibitive to spammers as it used to be, as the popularity of SMS has led to messaging charges dropping below USD 0.001 in markets like China, and even free of charge in others. According to Cloudmarkstats, the amount of mobile phone spam varies widely from region to region. For instance, in North America, much less than 1% of SMS messages were spam in 2010, while, in parts of Asia, up to 30% of messages were by spam.

The data sets for this example contain the SMS messages (file `message.txt`), tags (ham/spam) indicating whether the message is legitimate or spam (file `class.txt`) and the SMART stopwords (file `stopwords.txt`). There are 5,574 SMS messages in English, of which 4,827 SMS are legitimate messages (86.6%) and 747 spam messages (13.4%). The messages are not chronologically sorted. Note that, owing to the way in which the data set was built, the spam rate in this corpus is not the same as in the real world.

These data have been gathered by Almeida *et al.* (2011). The two parts of the data set have the following origins:

- The legitimate messages were collected for research at the Department of Computer Science at the National University of Singapore. They largely originate from Singaporean students. These messages were collected from volunteers who were made aware that their contributions were going to be made publicly available.
- A collection of SMS spam messages extracted manually from the Grumbletext Web site (`http://www.grumbletext.co.uk`). This is a UK forum in which cell phone users make public claims about SMS spam messages.

In the file `message.txt` , each message takes one line, meaning that they are separated by a return character. To import such files, it is faster (and simpler) to use the function `readLines` .

```
message <- readLines("message.txt")
str(message)
```

```
##  chr [1:5574] "Go until jurong point, crazy.. Available only in bugis n great world la e b
uffet... Cine there got amore wat..." ...
```

Now, `message` is a vector of length 5,574. The length will be used later, so I save it.

```
N <- length(message)
```

The file `class.txt` has the same structure, so I use also `readLines` . I check that `class` has two values, `spam` and `ham` , with the expected frequencies.

```
class <- readLines("class.txt")
table(class)
```

```
## class
##  ham spam
## 4827  747
```

In this presentation, I use the first 10 messages (messages 3, 6, 9 and 10 are spam) to track the transformations performed in the subsequent steps of the processing stage.

```
message[1:10]
```

```
##  [1] "Go until jurong point, crazy.. Available only in bugis n great world la e buffet...
 Cine there got amore wat..."
##  [2] "Ok lar... Joking wif u oni..."

##  [3] "Free entry in 2 a wkly comp to win FA Cup final tkts 21st May 2005. Text FA to 87121
 to receive entry question(std txt rate)T&C's apply 08452810075over18's"
##  [4] "U dun say so early hor... U c already then say..."

##  [5] "Nah I don't think he goes to usf, he lives around here though"

##  [6] "FreeMsg Hey there darling it's been 3 week's now and no word back! I'd like some fun
 you up for it still? Tb ok! XxX std chgs to send, £1.50 to rcv"
##  [7] "Even my brother is not like to speak with me. They treat me like aids patent."

##  [8] "As per your request 'Melle Melle (Oru Minnaminunginte Nurungu Vettam)' has been set
 as your callertune for all Callers. Press *9 to copy your friends Callertune"
##  [9] "WINNER!! As a valued network customer you have been selected to receive a £900 prize
 reward! To claim call 09061701461. Claim code KL341. Valid 12 hours only."
## [10] "Had your mobile 11 months or more? U R entitled to Update to the latest colour mobil
es with camera for Free! Call The Mobile Update Co FREE on 08002986030"
```

I perform a set of transformations before removing punctuation. Some of these transformations, due to the special nature of the documents, are specific of this example. In most cases, I add extra white space to prevent terms merging, now or in later transformations. The conversion to lower case is performed with the function `tolower`.

```
message <- tolower(message)
```

Substitutions are performed in R with the function `gsub`, which has three arguments with no default, `pattern` (what we wish to be replaced), `replacement` (what replaces), and the vector on which the substitution takes place. First, I drop the apostrophe,

```
message <- gsub(message, pattern="'", replacement="")
```

Next, I remove all numbers. The expression `[0-9]` has been already commented when introducing regular expressions.

```
message <- gsub(message, pattern="[0-9]", replacement=" ")
```

You may have noticed, in the initial 10-message list, that the three-dot token (`...`) appears frequently in ham but not in spam messages. You have probably seen (and used) this token in SMS, WhatsApp or e-mail messages. So, since punctuation will be dropped later, it should be included as a term. I replace it by the term ‘dots’. In the code line below, note the number 3 between curly braces, which indicates that the character

between square brackets is transformed only if it comes in a sequence of length 3. Also, note that the dot (`.`) is between square brackets because, if not, it will be taken as a **metacharacter**. In regular expressions, it stands for any character that is not listed.

```
message <- gsub(message, pattern="[.]{3}", replacement=" dots ")
```

The conversion of the pound (£) and dollar ($) symbols into words may be relevant in this example, because a quick exploration suggests that most messages explicitly mentioning amounts of money are spam. Pounds are more frequent than dollars. I replace both symbols by the word 'pound'.

```
message <- gsub(message, pattern="[£$]", replacement=" pound ")
```

Emoticons, such as ':)' may help to discard spamness. Here, I transform ':)' into 'smiley'.

```
message <- gsub(message, pattern=":)", replacement=" smiley ")
```

The expression 'terms and conditions' may appear in SMS messages (specially in spam) in many versions, such as 'T&C', 'T&Cs', etc. I replace all these by the term 'tconditions' (padded with white space), using a regular expression. Here, `[ s]` stands for either white space or s, and the question mark (`?`) is a **quantifier** indicating that the preceding character or character class may occur or not.

```
message <- gsub(message, pattern="t[s ]?& ?cs?", replacement=" tconditions ")
```

Sometimes the message has been transmitted in a way that it gets contaminated with **HTML conventions**. A typical example is the occurrence of '&' which is a **character entity** that stands for the ampersand symbol. Other character entities are '<' and '>' which mean '<' and '>', respectively. These symbols are used as markup in HTML. Also, in this corpus, the expressions '<#>', '<DECIMAL>' and '<TIME>' stand for numbers. I replace all this by white space. Here, I deal with several patterns at the same time by using a vertical bar (`|`), which means OR.

```
message <- gsub(message, pattern="&lt;(#|decimal|time)&gt", replacement=" ")
message <- gsub(message, pattern="&[lg]t;", replacement=" ")
message <- gsub(message, pattern="&amp;", replacement=" ")
```

Another point is that we identify the terms 'text' and 'txt'. A quick exploration shows that they have the same meaning and are frequently used in spam messages.

```
message <- gsub(message, pattern="txt", replacement="text")
```

I list again the first 10 messages, to see the effect of these transformations.

```
message[1:10]
```

```
##  [1] "go until jurong point, crazy.. available only in bugis n great world la e buffet dot
s  cine there got amore wat dots "
##  [2] "ok lar dots  joking wif u oni dots "

##  [3] "free entry in   a wkly comp to win fa cup final tkts   st may     . text fa to
 to receive entry question(std text rate) tconditions  apply           over  s"
##  [4] "u dun say so early hor dots  u c already then say dots "

##  [5] "nah i dont think he goes to usf, he lives around here though"

##  [6] "freemsg hey there darling its been   weeks now and no word back! id like some fun yo
u up for it still? tb ok! xxx std chgs to send,  pound  .   to rcv"
##  [7] "even my brother is not like to speak with me. they treat me like aids patent."

##  [8] "as per your request melle melle (oru minnaminunginte nurungu vettam) has been set as
 your callertune for all callers. press *  to copy your friends callertune"
##  [9] "winner!! as a valued network customer you have been selected to receive a  pound
 prize reward! to claim call            . claim code kl   . valid    hours only."
## [10] "had your mobile    months or more? u r entitled to update to the latest colour mobil
es with camera for free! call the mobile update co free on            "
```

The next step is to remove the punctuation: period (.), comma (,), colon (:), semicolon (;), question mark (?), etc. I have already taken care of the apostrophe, which was just dropped (I wished to transform 'are'nt' into 'arent', not into two separate words). I replace the rest of the punctuation marks by white space, to prevent mergers. I use a regular expression that stands for all the punctuation marks. Note the double brackets: single brackets would be read as *any of characters within the brackets*.

```
message <- gsub(message, pattern="[[:punct:]]", replacement=" ")
```

In addition to the unnecessary white space that the original messages may contain, I have added more. Now, I remove it, to get a cleaner corpus. In the code lines below, the first line removes extra white space in the middle, by replacing any sequence of white spaces by one. The plus sign (+) is a quantifier that indicates one or more occurrences of the preceding character. The second line removes white space at the beginning and at the end of a string, respectively. The caret symbol (^), at the beginning of a regular expression, refers to the initial position of a string. The dollar symbol ($), at the end of the expression, to the last position.

```
message <- gsub(message, pattern=" +", replacement=" ")
message <- gsub(message, pattern="^ | $", replacement="")
```

I list again the first 10 messages.

```
message[1:10]
```

```
##  [1] "go until jurong point crazy available only in bugis n great world la e buffet dots c
ine there got amore wat dots"
##  [2] "ok lar dots joking wif u oni dots"

##  [3] "free entry in a wkly comp to win fa cup final tkts st may text fa to to receive entr
y question std text rate tconditions apply over s"
##  [4] "u dun say so early hor dots u c already then say dots"

##  [5] "nah i dont think he goes to usf he lives around here though"

##  [6] "freemsg hey there darling its been weeks now and no word back id like some fun you u
p for it still tb ok xxx std chgs to send pound to rcv"
##  [7] "even my brother is not like to speak with me they treat me like aids patent"

##  [8] "as per your request melle melle oru minnaminunginte nurungu vettam has been set as y
our callertune for all callers press to copy your friends callertune"
##  [9] "winner as a valued network customer you have been selected to receive a pound prize
 reward to claim call claim code kl valid hours only"
## [10] "had your mobile months or more u r entitled to update to the latest colour mobiles w
ith camera for free call the mobile update co free on"
```

Now, I transform each document into a **bag of words**. This is easily done with the function `strsplit`, whose syntax is straightforward: the argument `split` is used to specify the string used for the split. Since I have already dropped the extra white space and the punctuation, I can use a single white space as the split string.

```
term.list <- strsplit(message, split=" ")
```

The split is a trivial operation, but there is a technical point here. Splitting the first document gives us a bag of 22 words, that in R is managed as a character vector of length 22. But the second document has 8 words, the third document 28 words, etc. So, `term.list` cannot be a matrix nor any rectangular object. R creates here a **list** (a list is like a vector, but its elements can have any structure). Here, `term.list` is a list of length 5,574, whose elements are character vectors of different lengths.

```
term.list[1:5]
```

```
## [[1]]
##  [1] "go"        "until"     "jurong"    "point"     "crazy"
##  [6] "available" "only"      "in"        "bugis"     "n"
## [11] "great"     "world"     "la"        "e"         "buffet"
## [16] "dots"      "cine"      "there"     "got"       "amore"
## [21] "wat"       "dots"
##
## [[2]]
## [1] "ok"     "lar"    "dots"   "joking" "wif"    "u"      "oni"    "dots"
##
## [[3]]
##  [1] "free"        "entry"       "in"          "a"           "wkly"
##  [6] "comp"        "to"          "win"         "fa"          "cup"
## [11] "final"       "tkts"        "st"          "may"         "text"
## [16] "fa"          "to"          "to"          "receive"     "entry"
## [21] "question"    "std"         "text"        "rate"        "tconditions"
## [26] "apply"       "over"        "s"
##
## [[4]]
##  [1] "u"       "dun"     "say"     "so"      "early"   "hor"     "dots"
##  [8] "u"       "c"       "already" "then"    "say"     "dots"
##
## [[5]]
##  [1] "nah"    "i"      "dont"   "think"  "he"     "goes"   "to"
##  [8] "usf"    "he"     "lives"  "around" "here"   "though"
```

My next step is to prune the vectors of this list by dropping the stopwords. First, I import the stopwords list.

```
stopwords <- readLines("stopwords.txt")
str(stopwords)
```

```
##  chr [1:571] "a" "a's" "able" "about" "above" "according" ...
```

Now, `stopwords` is a character vector of length 571. Even if this is a well known and respectable collection, I am not entirely happy with it. After a quick inspection, I find that some typical texting stopwords, like `ur`, are missing.

```
"ur" %in% stopwords
```

```
## [1] FALSE
```

So I decide to enlarge the stopwords catalogue with some of my own.

```
stopwords <- c(stopwords, "arent", "aint", "cant", "couldnt", "didnt", "doesnt", "dont",
"im", "ive", "pls", "ü", "ur")
```

Next, I define a function that removes any term of the stopwords collection from a character vector. Note that the exclamation mark (`!`) turns `TRUE` into `FALSE` and conversely.

```
stopRemove <- function(x) x[!(x %in% stopwords)]
```

Next, I apply this function to every element of `term.list`. Since `term.list` is a list, I use then function `lapply`, which returns the list that results from applying `stopRemove` to every element of the input list.

```
term.list <- lapply(term.list, stopRemove)
term.list[1:5]
```

```
## [[1]]
##  [1] "jurong" "point"  "crazy"  "bugis"  "great"  "world"  "la"
##  [8] "buffet" "dots"   "cine"   "amore"  "wat"    "dots"
##
## [[2]]
## [1] "lar"    "dots"   "joking" "wif"    "oni"    "dots"
##
## [[3]]
##  [1] "free"        "entry"       "wkly"        "comp"        "win"
##  [6] "fa"          "cup"         "final"       "tkts"        "st"
## [11] "text"        "fa"          "receive"     "entry"       "question"
## [16] "std"         "text"        "rate"        "tconditions" "apply"
##
## [[4]]
## [1] "dun"   "early" "hor"   "dots"  "dots"
##
## [[5]]
## [1] "nah"   "usf"   "lives"
```

I build a table of top frequent terms, sorted by frequency. First, I transform `term.list` into a vector with the function `unlist`. Then, I count the number of times every term occurs with `table`. Finally, I sort this with the function `sort`. This produces a very long vector, with 7,228 entries, most of them irrelevant. I pick, for my first analysis, the top-10 terms.

```
freq.term <- sort(table(unlist(term.list)), decreasing=T)
length(freq.term)
```

```
## [1] 7408
```

```
freq.term[1:10]
```

```
##
##   dots   call   text  pound   free smiley   good    day    ill   love
##   1261    608    379    361    286    252    247    226    215    214
```

Since I only have to deal with a low number of terms, I create a list of equivalent terms manually, without stemming.

```
top10.list <- list("dots", c("call", "calling", "calls"), c("text", "texts", "texting"),
  c("pound", "pounds"), "free", "smiley", "good", c("day", "days"), c("ill", "illness"),
  c("love", "loves", "lovely", "loving", "lovingly"))
```

I build a binary term-document matrix, whose columns are dummies for the occurrence of these 10 terms in the documents of the corpus. First, I define a matrix of the adequate dimensions,

```
TD <- matrix(nrow=N, ncol=10)
```

Then, I replace every term of the matrix by the corresponding 1/0 value. This can be done in many ways, but using a `for` loop is probably the simplest way. Note that the elements of a list are identified with double brackets (`[[i]]`).

```
for(i in 1:N) for(j in 1:10) TD[i,j] <- max(top10.list[[j]] %in% term.list[[i]])
colnames(TD) <- names(freq.term[1:10])
TD[1:5,]
```

```
##      dots call text pound free smiley good day ill love
## [1,]    1    0    0     0    0      0    0   0   0    0
## [2,]    1    0    0     0    0      0    0   0   0    0
## [3,]    0    0    1     0    1      0    0   0   0    0
## [4,]    1    0    0     0    0      0    0   0   0    0
## [5,]    0    0    0     0    0      0    0   0   0    0
```

The rest of the example is an exercise of predictive modelling. First, I create a data frame by binding the vector `class` with the 10 columns of the matrix TD.

```
sms.TD <- data.frame(class, TD)
```

The columns of `TD` will be the attributes that I will use to predict the first column of `sms.TD` . I specify this as an R formula, taking one half of the messages for training and the other half for testing.

```
train <- sample(1:N, size=N/2, replace=F)
fm <- class ~ .
```

I train two models on these data, a logistic regression model and a decision tree. The code for training the models and getting the predictive scores for the test set is as follows.

```
mod1 <- glm(formula=fm, data=sms.TD[train,], family="binomial")
pred1 <- predict(mod1, newdata=sms.TD[-train,], type="response")
library(rpart)
mod2 <- rpart(formula=fm, data=sms.TD[train,], cp=0.001)
pred2 <- predict(mod2, newdata=sms.TD[-train,])[,"spam"]
```

I restrict the evaluation to the test set. To be more efficient, I define a function with two arguments, a vector of scores and a cutoff, to calculate the true positive and false positive rates.

```
eval <- function(pred, cut) {conf <- table(pred>cut, sms.TD[-train,]$class=="spam")
  tp <- conf["TRUE", "TRUE"]/sum(conf[,"TRUE"])
  fp <- conf["TRUE", "FALSE"]/sum(conf[,"FALSE"])
  return(c(tp, fp))
}
```

I try two cutoffs, 0.5 and the spam rate 0.14, even if it does not make sense here, because the data set has been produced artificially, and this rate is not the same as the spam rate in the real world. But the objective of this example is just to describe the text mining technology. I pack the results of the three models in a table with following code.

```
matrix(c(eval(pred1, 0.5), eval(pred1, 0.14), eval(pred2, 0.5), eval(pred2, 0.14)),
  nrow=2, byrow=T,
  dimnames = list(c("Logistic", "Tree"), c("cutoff 0.5", "", "cutoff 0.14", "")))
```

```
##          cutoff 0.5            cutoff 0.14
## Logistic  0.5390836 0.01862583   0.8382749 0.06208609
## Tree      0.5283019 0.02069536   0.7196765 0.05877483
```

So far, with just 10 terms, the logistic regression and the decision tree models are already doing a good job.

New analysis with an enlarged list of terms

The purpose of the preceding analysis was to illustrate the process, not to carry it out in the best possible way. Indeed, it is probable that enlarging the collection of terms involved in the predictive model improves the performance. So, to complete this presentation, I add 15 terms, to get a term-document matrix with 25 columns (this is not really big, for the standards of data scientists).

```
freq.term[11:25]
```

```
##
##   time   send   home   stop    lor   back  today     da  reply mobile
##    214    200    167    165    162    153    153    151    148    144
##  phone   dear   week  night  great
##    126    125    122    118    115
```

```
top11_25.list <- list(c("time", "times"), c("send", "sending", "sender"), "home", "stop",
  "lor", "back", c("today", "todays"), "da", "reply", "mobile", c("phone", "phones"),
  "dear", c("week", "weeks", "weekly"), c("night", "nights"), "great")
```

I calculate a term-document matrix for the new terms, binding it to the previous matrix. So, I get a matrix of 25 columns.

```
extra.TD <- matrix(nrow=N, ncol=15)
for(i in 1:N) for(j in 1:15) extra.TD[i,j] <- max(top11_25.list[[j]] %in% term.list[[i]])
colnames(extra.TD) <- names(freq.term[11:25])
TD <- cbind(TD, extra.TD)
TD[1:5,11:25]
```

```
##      time send home stop lor back today da reply mobile phone dear week
## [1,]    0    0    0    0   0    0     0  0     0      0     0    0    0
## [2,]    0    0    0    0   0    0     0  0     0      0     0    0    0
## [3,]    0    0    0    0   0    0     0  0     0      0     0    0    0
## [4,]    0    0    0    0   0    0     0  0     0      0     0    0    0
## [5,]    0    0    0    0   0    0     0  0     0      0     0    0    0
##      night great
## [1,]     0     1
## [2,]     0     0
## [3,]     0     0
## [4,]     0     0
## [5,]     0     0
```

Both logistic and tree models improve a bit their performance. An intermediate cutoff will probably do a better job than those two used.

```
mod1 <- glm(formula=fm, data=sms.TD[train,], family="binomial")
pred1 <- predict(mod1, newdata=sms.TD[-train,], type="response")
mod2 <- rpart(formula=fm, data=sms.TD[train,], cp=0.001)
pred2 <- predict(mod2, newdata=sms.TD[-train,])[,"spam"]
matrix(c(eval(pred1, 0.5), eval(pred1, 0.14), eval(pred2, 0.5), eval(pred2, 0.14)), nrow=2, b
yrow=T,
  dimnames = list(c("Logistic", "Tree"), c("cutoff 0.5", "", "cutoff 0.14", "")))
```

```
##          cutoff 0.5            cutoff 0.14
## Logistic  0.5390836 0.01862583   0.8382749 0.06208609
## Tree      0.5283019 0.02069536   0.7196765 0.05877483
```

References

[1] TA Almeida, JM Gómez Hidalgo & A Yamakami (2011), Contributions to the study of SMS Spam Filtering — New Collection and Results, *Proceedings of the 2011 ACM Symposium on Document Engineering*, Mountain View, CA.

[2] JEF Friedl (2007), *Mastering Regular Expressions*, O'Reilly.

[3] JK Korpela (2006), *Unicode Explained*, O'Reilly.

[4] Z Markov & DT Larose (2007), *Data Mining the Web*, Wiley.

[5] S Munzert, C Ruba, P Meissner & D Nyhuis (2015), *Automated Data Collection with R*, Wiley.

[6] SM Weiss, N Indurkhya, T Zhang & FJ Damerau (2005), *Text Mining — Predictive Methods for Analyzing Unstructured Information*, Springer.