Text Analytics with R

Contents

Introduction

Here is initial data processing and manipulation.

```
############################################
# Text Analytics (learning from D.Langer)
# Mainly following playlist at : https://www.youtube.com/watch?v=4vuw0AsHeGw&list=PL8eNk_zTBST8olxIRFoo
############################################

#install.packages(c("quanteda","irlba"))

spam.raw <- read.csv(file = 'spam.csv' , stringsAsFactors =  FALSE)
spam.raw <- spam.raw[,1:2]

names(spam.raw) <- c("Label","Text")

length(which(!complete.cases(spam.raw)))
```

```
## [1] 0
```

```
# It is useful to convert class labels to factors early on
spam.raw$Label <- factor(spam.raw$Label)


# Prop table express them as fractions
prop.table(table(spam.raw$Label))
```

```
##
##       ham      spam
## 0.8659368 0.1340632
```

```r
table(spam.raw$Label)
```

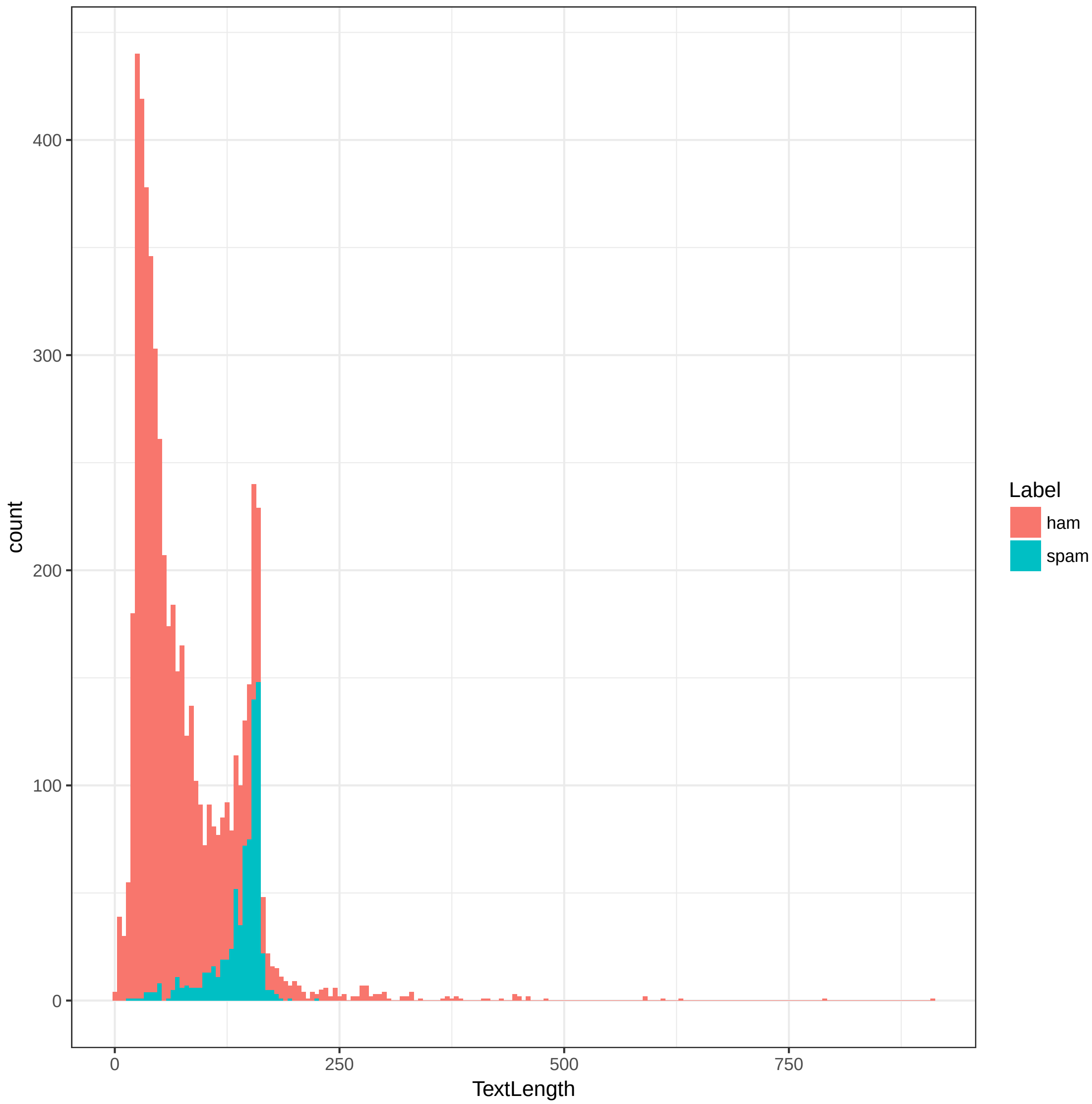
```
##
##  ham spam
## 4825  747
```

```r
# Let's look at the relative lengths of texts

spam.raw$TextLength <- nchar(spam.raw$Text)
summary(spam.raw$TextLength)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    2.00   36.00   61.00   80.12  121.00  910.00
```

```r
# This turns out to be a great feature for classification
library(ggplot2)
ggplot(spam.raw,aes(x = TextLength, fill = Label))+
    geom_histogram(binwidth = 5)+
    theme_bw()
```




```r
# Data split (stratifed split: ie: class proportions are maintained during the split)
library(caret)

set.seed(32984)
indexes <- createDataPartition(y = spam.raw$Label,
                               p = 0.7, list = FALSE)

train <- spam.raw[indexes,]
test <- spam.raw[-indexes,]

# Note that class label proportions are maintained
prop.table(table(train$Label))
```

```
##
```

```
##       ham      spam
## 0.8659318 0.1340682
```

```r
prop.table(table(test$Label))
```

```
##
##       ham      spam
## 0.8659485 0.1340515
```

More data exploration and processing

How do we represent text as Data Frame? We achieve this by **TOKENIZATION**.

Once Tokenization is completed we can create a **Document-Frequency Matrix** (DFM).

- Each row represents a distinct **document**.
- Each column represents a distinct **token**.(Distinct Tokens across all documents are also called **Terms**)
- Each cell(matrix entry) contains the **counts** of that token for a given document.
- one-grams produce **bag-of-words model**, this is where we point we start, it is possible to preserve word order by adding n-grams to make our models even stronger or accurate.

Some considerations

- Do we want all tokens to be terms in our DFM?
  - How about case-sensitivity/capitalization
  - Punctuation? Do I want them in my DFM? Typically you don't.
  - Do you want numbers/digits in your DFM?
  - Do you want evey word? No: don't use stop words (e.g: the,and..)
  - Symbols (sometimes very important)
  - What about similar words? **STEMMING** Is it possible to COLLAPSE similar word to a common stem (single representation).

Data Pipelines

Notice we will have some dirty text, such as **&amp** here. Our pipelines should take these into consideration, possibly adding domain knowledge into it. Replacing, stripping/removing these are all important decisions.

```r
train$Text[21]
```

```
## [1] "I'm back &amp; we're packing the car now, I'll let you know if there's room"
```

```r
train$Text[38]
```

```
## [1] "A gram usually runs like  &lt;#&gt; , a half eighth is smarter though and gets you almost a whol
```

It is reccomended to streamline the steps below as a Pipeline:

Tokenization

```r
# Quite powerful package
library(quanteda)

# Tokenize
train.tokens <- tokens(train$Text,what = "word",
                       remove_numbers = TRUE,
                       remove_punct = TRUE,
                       remove_symbols = TRUE,
                       remove_hyphens = TRUE)

# Returns a list-like object that contains tokens
train.tokens[[357]]
```

```
##  [1] "Your"              "credits"           "have"
##  [4] "been"              "topped"            "up"
##  [7] "for"               "http"              "www.bubbletext.com"
## [10] "Your"              "renewal"           "Pin"
## [13] "is"                "tgxxrz"
```

Notice that based on our preferences the tokenization is performed.

If we want 3-grams, simply use ngrams argument:

(Note that the default concetanator is "_")

```r
tokens(train$Text,what = "word",
                       remove_numbers = TRUE,
                       remove_punct = TRUE,
                       remove_symbols = TRUE,
                       remove_hyphens = TRUE,
       ngrams = 3)[[357]]
```

```
##  [1] "Your_credits_have"                "credits_have_been"
##  [3] "have_been_topped"                 "been_topped_up"
##  [5] "topped_up_for"                    "up_for_http"
##  [7] "for_http_www.bubbletext.com"      "http_www.bubbletext.com_Your"
##  [9] "www.bubbletext.com_Your_renewal" "Your_renewal_Pin"
## [11] "renewal_Pin_is"                   "Pin_is_tgxxrz"
```

Transform Tokens

Let's convert the tokens to lowercase (for our use-case)

```r
train.tokens <- tokens_tolower(train.tokens)
train.tokens[[357]]
```

```
##  [1] "your"              "credits"           "have"
##  [4] "been"              "topped"            "up"
##  [7] "for"               "http"              "www.bubbletext.com"
## [10] "your"              "renewal"           "pin"
## [13] "is"                "tgxxrz"
```

Removing stopwords

This is a tricky step for any text analytics pipeline. We need to understand what is in the stop words library of each package we might be using. Depending on our domain, the list might contain words that we may actually want to maintain in our DFM.

These are the stopwords removed by the quanteda package:

```r
quanteda::stopwords()
```

```
##   [1] "i"          "me"         "my"         "myself"     "we"
##   [6] "our"        "ours"       "ourselves"  "you"        "your"
##  [11] "yours"      "yourself"   "yourselves" "he"         "him"
##  [16] "his"        "himself"    "she"        "her"        "hers"
##  [21] "herself"    "it"         "its"        "itself"     "they"
##  [26] "them"       "their"      "theirs"     "themselves" "what"
##  [31] "which"      "who"        "whom"       "this"       "that"
##  [36] "these"      "those"      "am"         "is"         "are"
##  [41] "was"        "were"       "be"         "been"       "being"
##  [46] "have"       "has"        "had"        "having"     "do"
##  [51] "does"       "did"        "doing"      "would"      "should"
##  [56] "could"      "ought"      "i'm"        "you're"     "he's"
##  [61] "she's"      "it's"       "we're"      "they're"    "i've"
##  [66] "you've"     "we've"      "they've"    "i'd"        "you'd"
##  [71] "he'd"       "she'd"      "we'd"       "they'd"     "i'll"
##  [76] "you'll"     "he'll"      "she'll"     "we'll"      "they'll"
##  [81] "isn't"      "aren't"     "wasn't"     "weren't"    "hasn't"
##  [86] "haven't"    "hadn't"     "doesn't"    "don't"      "didn't"
##  [91] "won't"      "wouldn't"   "shan't"     "shouldn't"  "can't"
##  [96] "cannot"     "couldn't"   "mustn't"    "let's"      "that's"
## [101] "who's"      "what's"     "here's"     "there's"    "when's"
## [106] "where's"    "why's"      "how's"      "a"          "an"
## [111] "the"        "and"        "but"        "if"         "or"
## [116] "because"    "as"         "until"      "while"      "of"
## [121] "at"         "by"         "for"        "with"       "about"
## [126] "against"    "between"    "into"       "through"    "during"
## [131] "before"     "after"      "above"      "below"      "to"
## [136] "from"       "up"         "down"       "in"         "out"
## [141] "on"         "off"        "over"       "under"      "again"
## [146] "further"    "then"       "once"       "here"       "there"
## [151] "when"       "where"      "why"        "how"        "all"
## [156] "any"        "both"       "each"       "few"        "more"
## [161] "most"       "other"      "some"       "such"       "no"
## [166] "nor"        "not"        "only"       "own"        "same"
## [171] "so"         "than"       "too"        "very"       "will"
```

```r
# Remove stopwords
train.tokens <- tokens_select(x = train.tokens,
                              pattern = stopwords(),
                              selection = "remove")
train.tokens[[357]]
```

```
## [1] "credits"            "topped"             "http"
## [4] "www.bubbletext.com" "renewal"            "pin"
## [7] "tgxxrz"
```

Stemming tokens

```
train.tokens <- tokens_wordstem(train.tokens,
                                language = "english")
train.tokens[[357]]
```

```
## [1] "credit"            "top"               "http"
## [4] "www.bubbletext.com" "renew"             "pin"
## [7] "tgxxrz"
```

What we have gone through is almost a typical text preprocessing pipeline:

1. Tokenize (specify what to remove; numbers, symbols, hyphens…)
2. Lovercase
3. Remove stopwords (or custom common words?)
4. Stemming

Create Document Frequency Matrix

In our case this is a bag-of-words model since we used one-grams;

```
# Use quantida function dfm
train.tokens.dfm <- dfm(x = train.tokens,
                        tolower = FALSE)

# Generates a fairly large matrix:
dim(train.tokens.dfm)
```

```
## [1] 3901 5742
```

```
# Convert to standard matrix:
train.tokens.matrix <- as.matrix(train.tokens.dfm)

head(train.tokens.matrix[,1:30])
```

```
##        features
## docs    go jurong point crazi avail bugi n great world la e buffet cine
##   text1  1      1     1     1     1    1 1     1     1  1 1      1    1
##   text2  0      0     0     0     0    0 0     0     0  0 0      0    0
##   text3  0      0     0     0     0    0 0     0     0  0 0      0    0
##   text4  0      0     0     0     0    0 0     0     0  0 0      0    0
##   text5  0      0     0     0     0    0 0     0     0  0 0      0    0
##   text6  0      0     0     0     0    0 0     0     0  0 0      0    0
##        features
## docs    got amor wat u dun say earli hor c alreadi nah think goe usf live
##   text1   1    1   1 0   0   0     0   0 0       0   0     0   0   0    0
##   text2   0    0   0 2   1   2     1   1 1       1   0     0   0   0    0
##   text3   0    0   0 0   0   0     0   0 0       0   1     1   1   1    1
##   text4   0    0   0 0   0   0     0   0 0       0   0     0   0   0    0
##   text5   0    0   0 0   0   0     0   0 0       0   0     0   0   0    0
##   text6   0    0   0 1   0   0     0   0 0       0   0     0   0   0    0
##        features
## docs    around though
##   text1      0      0
##   text2      0      0
```

```
##   text3      1       1
##   text4      0       0
##   text5      0       0
##   text6      0       0
```

Note that our feature space/dimentionality increased dramatically.

2 facts to notice:

1. Text Analytics suffers from **curse of dimensionality**.
2. Text Analytics creates a matrix with mostly zeros (**sparsity problem**), which we will try to deal with using **feature extraction**.

```
# Investigating the effects of stemming
colnames(train.tokens.matrix)[1:50]
```

```
##  [1] "go"       "jurong"  "point"   "crazi"   "avail"   "bugi"    "n"
##  [8] "great"    "world"   "la"      "e"       "buffet"  "cine"    "got"
## [15] "amor"     "wat"     "u"       "dun"     "say"     "earli"   "hor"
## [22] "c"        "alreadi" "nah"     "think"   "goe"     "usf"     "live"
## [29] "around"   "though"  "freemsg" "hey"     "darl"    "week"    "now"
## [36] "word"     "back"    "like"    "fun"     "still"   "tb"      "ok"
## [43] "xxx"      "std"     "chgs"    "send"    "å"       "rcv"     "winner"
## [50] "valu"
```

This pretty much completes the standard text data processing pipeline.

Building our First Model

We will build our model using cross-validation.

DFM is contains our corpus (corpus is a fancy name for a collection of documents) and terms(features).

We set up a feature data frame with labels:

```
# Collecting everything in a standard dataframe:
train.tokens.df <- cbind(Label= train$Label,
                         convert(train.tokens.dfm, to = "data.frame"))
head(train.tokens.df[,1:10])
```

```
##   Label document go jurong point crazi avail bugi n great
## 1   ham    text1  1      1     1     1     1    1 1     1
## 2   ham    text2  0      0     0     0     0    0 0     0
## 3   ham    text3  0      0     0     0     0    0 0     0
## 4  spam    text4  0      0     0     0     0    0 0     0
## 5  spam    text5  0      0     0     0     0    0 0     0
## 6  spam    text6  0      0     0     0     0    0 0     0
```

Fix the names of data frame:

Note that terms we generated by tokenization requires some additional processing:

```
names(train.tokens.df)[c(146,148,235,238)]
```

```
## [1] "try:wal" "4txt"    "2nd"     "8am"
```

Note that these are not **valid column names** for data frames in R. Machine learning algorithms will throw error unless we transform them using **makes.names()** function:

```
names(train.tokens.df) <- make.names(names(train.tokens.df))
names(train.tokens.df)[c(146,148,235,238)]
```

```
## [1] "try.wal" "X4txt"   "X2nd"    "X8am"
```

Setting up cross-validation

We need to perform **stratified cross-validation** given the class-imbalance in our data set. In other words, in our CV folds, we need to make sure the representation of the individual classes are similar to the entire training data.

```
library(caret)
set.seed(48743)
cv.folds <- createMultiFolds(y = train$Label,
                             k = 10,
                             times = 3)

# 3 times means we will create 30 random stratified samples

cv.cntrl <- trainControl(method = "repeatedcv",
                         number = 10,
                         repeats = 3,
                         index = cv.folds)

# Note that setting index = cv.folds makes us ensure that
# trainControl will use the stratified folds we specified above
# If we don't specifiy this, the model will still train but
# it will use random folds, which may not be the desired stratified
# folds we wish to obtain due to class imbalance
```

The 10 fold cross-calidation repeated 3 times gives a more robust estimate of performance (but will be computationally expensive). Particularly when there is class-imbalance like in this example, **repeatedcv** might be useful.

Note that the size of our data-frame is not trivial, hence we will perform parallel multi-core computing using doSNOW package:

```
# Check the number of cores
parallel:::detectCores()
```

```
## [1] 8
```

doSNOW works for both Mac and Windows:

```
library(doSNOW)
start.time <- Sys.time()

# Create a cluster to work on 7 logical cores (keep at least 1 core for the operating system)
# type = "SOCK" Socket cluster
# Behind the scenes it will create 7 jobs
cl <- makeCluster(spec = 7,type = "SOCK")
# Register the cluster:
registerDoSNOW(cl)
# Once registered, caret will recognize this cluster and parallelize the job

rpart.cv.1 <- caret::train(Label ~ ., data = train.tokens.df,
```

```
                             method = 'rpart',trControl = cv.cntrl,
                             tuneLength = 7)

# Processing (training) is done, so we stop the cluster:
stopCluster(cl)

end.time <- Sys.time()
```

We are using **rpart**, single decision tree for an initial model to develop an intuition. We use ALL the features in the DFM.

Note that:

```
**tuneLength = 7** performs effectively hyperparameter tuning using 7 different model fits comparing 7 c
```

```
end.time - start.time

# Time difference of 8.363868 mins
```

Save the model object and reload for future use:

```
#saveRDS(rpart.cv.1,"rpart_cv_1.rds")
rpart.cv.1 <- readRDS("rpart_cv_1.rds")
rpart.cv.1
```

```
## CART
##
## 3901 samples
## 5743 predictors
##    2 classes: 'ham', 'spam'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 3 times)
## Summary of sample sizes: 3511, 3510, 3511, 3511, 3511, 3511, ...
## Resampling results across tuning parameters:
##
##   cp          Accuracy   Kappa
##   0.02103250  0.9425410  0.7115882
##   0.02294455  0.9398008  0.6933445
##   0.02868069  0.9351165  0.6636735
##   0.03059273  0.9328171  0.6479571
##   0.03824092  0.9296348  0.6244737
##   0.05098789  0.9154079  0.5146486
##   0.32504780  0.8801296  0.1534660
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was cp = 0.0210325.
```

Note the results are presented quite intuitively,**cp** is the rpart hyperparameter being tuned and 7 distinct values of that parameter was presented. The best cp value is choosen based on accuracy. 94% accuracy is actually quite good out of the box. Next we will try to improve this benchmark by:

- using TF-IDF (by transforming the nature of the data we have)
- adding n-grams into our set of features (hoping to get more useful features)
- extract features using SVD, to reduce the dimensionality
- and finally trying more powerful algorithms like randomForest

Using TF-IDF: changing representation of DFM

So far we realized that the bag-of-words model with document frequency matrices could work! 94% Accuracy is quite remarkable in simplest terms.

There is still room for improvement:

- Longer documents will tend to have higher term counts.
- Terms that appear frequently across the corpus aren't as important (nearly zero variance).

Therefore, we can improve upon the DFM representation if we can achieve the following:

- Normalize documents based on their length
- Penalize terms that occur frequently across the corpus (set of available documents)

So, if we can adjust our DFM to accomodate these 2 things, it will be much more powerful.

This is exactly what TF-IDF address:

TF: Term Frequency

- Let freq(t,d) be the count of the instances of the term t in document d.
- Let TF(t,d) be the proportion of the count of term t in document d. e.g: if I have a term that appeared in a document 4 times and the document has a total of 10 words (after text processing pipeline), then TF of that term becomes 0.4.

Mathematically:

```
TF(t,d) = freq(t,d) / Sum i (freq(ti,d))
```

Therefore, TF achieves the first goal, that is the normalization to text length.

IDF: Inverse Document Frequency

- Let N be the count of distinct documents in the corpus.
- Let count(t) be the count of documents in the corpus in which the term t is present.

Then,

```
IDF(t) = log( N / count(t) )

# log10 is commonly used.
```

Notice that if a term appears in ALL of the documents, IDF of that term will be $\log 1 = 0$, which will be a penalizing weight for that term. Hence, if the term appears in every single document, this would mean the information in that term is not useful, because it does not explain any variability. This way we achieved our second goal, that is penalizing the terms that occur frequently across the corpus.

TF-IDF idea:

IF we combine TF and IDF, we can enhance the document-term frequency matrices. TF-IDF is simply multiplication of these two amounts:

```
TF-IDF(t,d) = TF(t,d) * IDF(t)
```

In most cases TF-IDF is prototypical for text processing pipelines as it can enhance the features of the DFM. Hence, along with other steps described above, TF-IDF is often incorporated in to text-processing pipeline.

Let's run the TF-IDF using our DFM:

Note that we are writing our own functions for this because:

1. its educational.
2. we can cache the IDF from the training data and we can use the same weights to transform the test data set to get the consistent representation.

```
# Our Function for calculating Term Frequency(TF)

term.frequency <- function(row){
    return(row / sum(row))
}

# Our function for calculating Inverse Document Frequency (IDF)
inverse.doc.freq <- function(col){
    corpus.size <- length(col)
    doc.count <- length(which(col > 0))

    return(log10(corpus.size/doc.count))
}

# Our function for calculating TF-IDF
tf.idf <- function(tf,idf){
    return (tf * idf)
}

# First step, normalize all documents via TF
# Matrix transformations:

train.tokens.df <- apply(train.tokens.matrix,1,term.frequency)
dim(train.tokens.df)
```

```
## [1] 5742 3901
```

```
dim(train.tokens.matrix)
```

```
## [1] 3901 5742
```

Note that the matrix got transposed as a result of the tf transformation:

```
head(train.tokens.df[,1:10],20)
```

```
##          docs
## features  text1     text2 text3 text4 text5  text6 text7 text8 text9
##   go     0.0625 0.0000000     0     0     0 0.0000     0     0     0
##   jurong 0.0625 0.0000000     0     0     0 0.0000     0     0     0
##   point  0.0625 0.0000000     0     0     0 0.0000     0     0     0
##   crazi  0.0625 0.0000000     0     0     0 0.0000     0     0     0
##   avail  0.0625 0.0000000     0     0     0 0.0000     0     0     0
##   bugi   0.0625 0.0000000     0     0     0 0.0000     0     0     0
##   n      0.0625 0.0000000     0     0     0 0.0000     0     0     0
##   great  0.0625 0.0000000     0     0     0 0.0000     0     0     0
##   world  0.0625 0.0000000     0     0     0 0.0000     0     0     0
##   la     0.0625 0.0000000     0     0     0 0.0000     0     0     0
```

```
##    e      0.0625 0.0000000      0      0      0 0.0000      0      0      0
##    buffet 0.0625 0.0000000      0      0      0 0.0000      0      0      0
##    cine   0.0625 0.0000000      0      0      0 0.0000      0      0      0
##    got    0.0625 0.0000000      0      0      0 0.0000      0      0      0
##    amor   0.0625 0.0000000      0      0      0 0.0000      0      0      0
##    wat    0.0625 0.0000000      0      0      0 0.0000      0      0      0
##    u      0.0000 0.2222222      0      0      0 0.0625      0      0      0
##    dun    0.0000 0.1111111      0      0      0 0.0000      0      0      0
##    say    0.0000 0.2222222      0      0      0 0.0000      0      0      0
##    earli  0.0000 0.1111111      0      0      0 0.0000      0      0      0
##          docs
## features text10
##    go          0
##    jurong      0
##    point       0
##    crazi       0
##    avail       0
##    bugi        0
##    n           0
##    great       0
##    world       0
##    la          0
##    e           0
##    buffet      0
##    cine        0
##    got         0
##    amor        0
##    wat         0
##    u           0
##    dun         0
##    say         0
##    earli       0
```

Notice that now the documents are columns and terms are rows.

The second step, calculate the **IDF vector** that we will use for transforming both training and test data. This is very important, because we will train the model using these set of IDFs, then when we want to transform the test data we should be able to transform the data to exactly the same space.

```r
# Apply the idf function over the columns
train.tokens.idf <- apply(train.tokens.matrix,2,inverse.doc.freq)
str(train.tokens.idf)
```

```
##  Named num [1:5742] 1.11 3.59 2.23 2.64 2.55 ...
##  - attr(*, "names")= chr [1:5742] "go" "jurong" "point" "crazi" ...
```

Note that the idf is a single numeric vector of idfs as we expected.

Finally, calculate the TF-IDF for our training corpus:

```r
# Note that since train.tokens.df was transposed during the normalization, we apply the tf.idf function
train.tokens.tfidf <- apply(train.tokens.df,2,tf.idf,
                            idf = train.tokens.idf)
dim(train.tokens.tfidf)
```

```
## [1] 5742 3901
```

I still maintain the transposed matrix state, but the data is now multiplied by the IDF weights for each term:

```r
head(train.tokens.tfidf[,1:10],20)
```

```
##          docs
## features       text1     text2 text3 text4 text5      text6 text7 text8
##   go      0.06953809 0.0000000     0     0     0 0.00000000     0     0
##   jurong  0.22444850 0.0000000     0     0     0 0.00000000     0     0
##   point   0.13934051 0.0000000     0     0     0 0.00000000     0     0
##   crazi   0.16480834 0.0000000     0     0     0 0.00000000     0     0
##   avail   0.15936145 0.0000000     0     0     0 0.00000000     0     0
##   bugi    0.17162987 0.0000000     0     0     0 0.00000000     0     0
##   n       0.10230834 0.0000000     0     0     0 0.00000000     0     0
##   great   0.10322854 0.0000000     0     0     0 0.00000000     0     0
##   world   0.13934051 0.0000000     0     0     0 0.00000000     0     0
##   la      0.18681975 0.0000000     0     0     0 0.00000000     0     0
##   e       0.11617389 0.0000000     0     0     0 0.00000000     0     0
##   buffet  0.20563412 0.0000000     0     0     0 0.00000000     0     0
##   cine    0.17581404 0.0000000     0     0     0 0.00000000     0     0
##   got     0.08635381 0.0000000     0     0     0 0.00000000     0     0
##   amor    0.22444850 0.0000000     0     0     0 0.00000000     0     0
##   wat     0.10385981 0.0000000     0     0     0 0.00000000     0     0
##   u       0.00000000 0.1822942     0     0     0 0.05127025     0     0
##   dun     0.00000000 0.2302958     0     0     0 0.00000000     0     0
##   say     0.00000000 0.3585450     0     0     0 0.00000000     0     0
##   earli   0.00000000 0.2456627     0     0     0 0.00000000     0     0
##          docs
## features text9 text10
##   go         0      0
##   jurong     0      0
##   point      0      0
##   crazi      0      0
##   avail      0      0
##   bugi       0      0
##   n          0      0
##   great      0      0
##   world      0      0
##   la         0      0
##   e          0      0
##   buffet     0      0
##   cine       0      0
##   got        0      0
##   amor       0      0
##   wat        0      0
##   u          0      0
##   dun        0      0
##   say        0      0
##   earli      0      0
```

After this transformations, we achieved our goals. Now the values also reflect the impact of how often a particular term is being seen in the document. As a result, if the TF-IDF value is low, that means the term is relatively frequent across the corpus and that is reflected (e.g: value of the word **go** is quite low, which is intuitive because it would be a common word. In contrast, the word **jurong** has a higher value in text1, which could imply its possibly higher predictive value relative to word **go**).

Importantly, we need to **transpose this matrix back** into the original form of the matrix, where columns are features and rows are documents.

```r
# Transpose the matrix to original shape
train.tokens.tfidf <- t(train.tokens.tfidf)
dim(train.tokens.tfidf)
```

```
## [1] 3901 5742
```

```r
head(train.tokens.tfidf[,1:10],20)
```

```
##         features
## docs             go    jurong     point     crazi     avail      bugi
##   text1  0.06953809 0.2244485 0.1393405 0.1648083 0.1593615 0.1716299
##   text2  0.00000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##   text3  0.00000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##   text4  0.00000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##   text5  0.00000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##   text6  0.00000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##   text7  0.00000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##   text8  0.00000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##   text9  0.00000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##   text10 0.00000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##   text11 0.00000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##   text12 0.00000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##   text13 0.00000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##   text14 0.00000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##   text15 0.00000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##   text16 0.13907618 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##   text17 0.00000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##   text18 0.00000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##   text19 0.00000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##   text20 0.00000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##         features
## docs             n     great     world        la
##   text1  0.1023083 0.1032285 0.1393405 0.1868197
##   text2  0.0000000 0.0000000 0.0000000 0.0000000
##   text3  0.0000000 0.0000000 0.0000000 0.0000000
##   text4  0.0000000 0.0000000 0.0000000 0.0000000
##   text5  0.0000000 0.0000000 0.0000000 0.0000000
##   text6  0.0000000 0.0000000 0.0000000 0.0000000
##   text7  0.0000000 0.0000000 0.0000000 0.0000000
##   text8  0.0000000 0.0000000 0.0000000 0.0000000
##   text9  0.0000000 0.0000000 0.0000000 0.0000000
##   text10 0.0000000 0.0000000 0.0000000 0.0000000
##   text11 0.1091289 0.0000000 0.0000000 0.0000000
##   text12 0.0000000 0.0000000 0.0000000 0.0000000
##   text13 0.0000000 0.0000000 0.0000000 0.0000000
##   text14 0.0000000 0.0000000 0.0000000 0.0000000
##   text15 0.0000000 0.0000000 0.0000000 0.0000000
##   text16 0.0000000 0.0000000 0.0000000 0.0000000
##   text17 0.0000000 0.0000000 0.0000000 0.0000000
##   text18 0.0000000 0.0000000 0.0000000 0.0000000
##   text19 0.0000000 0.0000000 0.0000000 0.0000000
##   text20 0.0000000 0.0000000 0.0000000 0.0000000
```

Check for incomplete cases:

```r
length(which(!complete.cases(train.tokens.tfidf)))
```

```
## [1] 6
```

Note that as a result of the pre-processing pipeline, it is possible to end up with empty strings in our matrix. Imagine that some words could be just punctuations or similar characters that were stripped, hence missing values in this matrix do occur, because any time there is an error in the calculation of TF-IDF there would be NAN errors.

Fix the incomplete cases:

```r
incomplete.cases <- which(!complete.cases(train.tokens.tfidf))
train$Text[incomplete.cases]
```

```
## [1] "What you doing?how are you?" "645"
## [3] ":) "                         "What you doing?how are you?"
## [5] ":( but your not here...."    ":-) :-)"
```

Note that we expect that all these cases will be stripped off by our pre-processing pipeline. We need to correct these documents.

```r
# We fill these documents with zero values, instead of removing them. This is because these messages co
train.tokens.tfidf[incomplete.cases,] <- rep(0,0,ncol(train.tokens.tfidf))
dim(train.tokens.tfidf)
```

```
## [1] 3901 5742
```

```r
sum(which(!complete.cases(train.tokens.tfidf)))
```

```
## [1] 0
```

We have now fixed these incomplete cases, so that machine learning algorithms will not throw error.

Lastly, lets combine this matrix with labels and make the column names legitimate as we have done for the DFM previously:

```r
train.tokens.tfidf.df <- cbind(Label = train$Label,
                               data.frame(train.tokens.tfidf))

names(train.tokens.tfidf.df) <- make.names(names(train.tokens.tfidf.df))

head(train.tokens.tfidf.df[,1:10],10)
```

```
##        Label         go    jurong     point     crazi     avail      bugi
## text1    ham 0.06953809 0.2244485 0.1393405 0.1648083 0.1593615 0.1716299
## text2    ham 0.00000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
## text3    ham 0.00000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
## text4   spam 0.00000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
## text5   spam 0.00000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
## text6   spam 0.00000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
## text7    ham 0.00000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
## text8   spam 0.00000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
## text9    ham 0.00000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
## text10   ham 0.00000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
##                n     great     world
## text1  0.1023083 0.1032285 0.1393405
## text2  0.0000000 0.0000000 0.0000000
## text3  0.0000000 0.0000000 0.0000000
## text4  0.0000000 0.0000000 0.0000000
```

```
## text5  0.0000000 0.0000000 0.0000000
## text6  0.0000000 0.0000000 0.0000000
## text7  0.0000000 0.0000000 0.0000000
## text8  0.0000000 0.0000000 0.0000000
## text9  0.0000000 0.0000000 0.0000000
## text10 0.0000000 0.0000000 0.0000000
```

Refitting the rpart single decision tree

Let's see if using TF-IDF improved the performance of the same model we fitted previously by using the raw DFM:

```r
library(doSNOW)
start.time <- Sys.time()

# Create a cluster to work on 7 logical cores (keep at least 1 core for the operating system)
# type = "SOCK" Socket cluster
# Behind the scenes it will create 7 jobs
cl <- makeCluster(spec = 7,type = "SOCK")
# Register the cluster:
registerDoSNOW(cl)
# Once registered, caret will recognize this cluster and parallelize the job

rpart.cv.2 <- caret::train(Label ~ ., data = train.tokens.tfidf.df,
                           method = 'rpart',trControl = cv.cntrl,
                           tuneLength = 7)

# Processing (training) is done, so we stop the cluster:
stopCluster(cl)

end.time <- Sys.time()

end.time - start.time
```

```r
#saveRDS(rpart.cv.2,"rpart_cv_2.rds")
rpart.cv.2 <- readRDS("rpart_cv_2.rds")
rpart.cv.2
```

```
## CART
##
## 3901 samples
## 5742 predictors
##    2 classes: 'ham', 'spam'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 3 times)
## Summary of sample sizes: 3511, 3510, 3511, 3511, 3511, 3511, ...
## Resampling results across tuning parameters:
##
##   cp          Accuracy   Kappa
##   0.01529637  0.9456527  0.7592221
##   0.01912046  0.9402661  0.7317088
##   0.02103250  0.9390713  0.7235391
##   0.02294455  0.9372775  0.7157780
```

```
##   0.02868069  0.9375330  0.7155453
##   0.07138305  0.9236217  0.6011774
##   0.32504780  0.8824156  0.1770913
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was cp = 0.01529637.
```

Good! So note that the accuracy is uplifted from 0.942 to 0.945, hence TF-IDF transformation helped to improve model's predictive performance.

N-grams

Now we have TF-IDF transformed Document Term Matrix. Can we further improve upon this representation? N-grams can be one way:

- Our representations so far have been single terms (unigrams)
- We can have more complex N-grams, this will help to capture some signal from word ordering.

Hence, we can add n-grams during the text processing pipeline.

But be careful, by adding even bigrams, we will significantly increase the size of the matrix! This will ever increase the SPARSITY and CURSE OF DIMENSIONALITY problems.

Adding bigrams to our feature matrix:

```
# Notice we can reuse our existing token object:
train.tokens <- tokens_ngrams(train.tokens, n = 1:2)
train.tokens[[357]]
```

```
##  [1] "credit"                     "top"
##  [3] "http"                       "www.bubbletext.com"
##  [5] "renew"                      "pin"
##  [7] "tgxxrz"                     "credit_top"
##  [9] "top_http"                   "http_www.bubbletext.com"
## [11] "www.bubbletext.com_renew" "renew_pin"
## [13] "pin_tgxxrz"
```

Now, we need to run the entire text processing pipeline on this new matrix:

(Note that stopword removal,lowercasing and stemming has been already performed since we used the original train.tokens object.)

```
# Use quantida function dfm
train.tokens.dfm <- dfm(x = train.tokens,
                        tolower = FALSE)

# Convert to standard matrix:
train.tokens.matrix <- as.matrix(train.tokens.dfm)

# Apply the idf function over the columns (need to cache this for test set)
train.tokens.idf <- apply(train.tokens.matrix,2,inverse.doc.freq)

# Need to increase memory limit otherwise won't fit into memory
memory.limit(size=56000)
```

```
## [1] 56000
```

```r
# Note that since train.tokens.df was transposed during the normalization, we apply the tf.idf function
train.tokens.tfidf <- apply(train.tokens.df,2,tf.idf,
                            idf = train.tokens.idf)

# Transpose the matrix to original shape
train.tokens.tfidf <- t(train.tokens.tfidf)


# Fix incomplete cases
incomplete.cases <- which(!complete.cases(train.tokens.tfidf))
train.tokens.tfidf[incomplete.cases,] <- rep(0,0,ncol(train.tokens.tfidf))


# Combine with label and make legitimate names
train.tokens.tfidf.df <- cbind(Label = train$Label,
                              data.frame(train.tokens.tfidf))

names(train.tokens.tfidf.df) <- make.names(names(train.tokens.tfidf.df))
```

Note the nice feature of quantida DFMs, you can type the name of the object and get information:

```r
train.tokens.dfm
```

```
## Document-feature matrix of: 3,901 documents, 29,154 features (99.9% sparse).
```

Note how the dimensions are significantly expanded just by adding bigrams, and the 99.9% of the matrix is sparse!

Important: we should clean up unsued memory:

```r
gc()
```

```
##              used    (Mb) gc trigger    (Mb)   max used    (Mb)
## Ncells    3841475   205.2    6908587   369.0    5033991   268.9
## Vcells 1840462731 14041.7 2913341648 22227.1 2427717124 18522.1
```

This function is sometimes useful to call to clean up unused memory. However, in text analytics we can easily reach the memory boundaries and it may or may not help.

LSA: Latent Semantic Analysis using SVD

We will perform feature extraction!

Two purposes:

1. We want to make our columns more feature rich than the current highly sparse state. We want to shrink the dimension, yet maintaining the variance.
2. By reducing the size of the data, we want to be able to use more powerful (but also more complex and computationally costly) algorithms for our problem (otherwise we will be running big scalability problems).

Vector space model

Vector space model helps to address our current problems related to curse of dimensionality and scaleability.

- Core intuition: we represent documents as **vectors of numbers**.

- Our representation allows us to work with document geometrically.

The idea is explained well here (statring from 12:12):