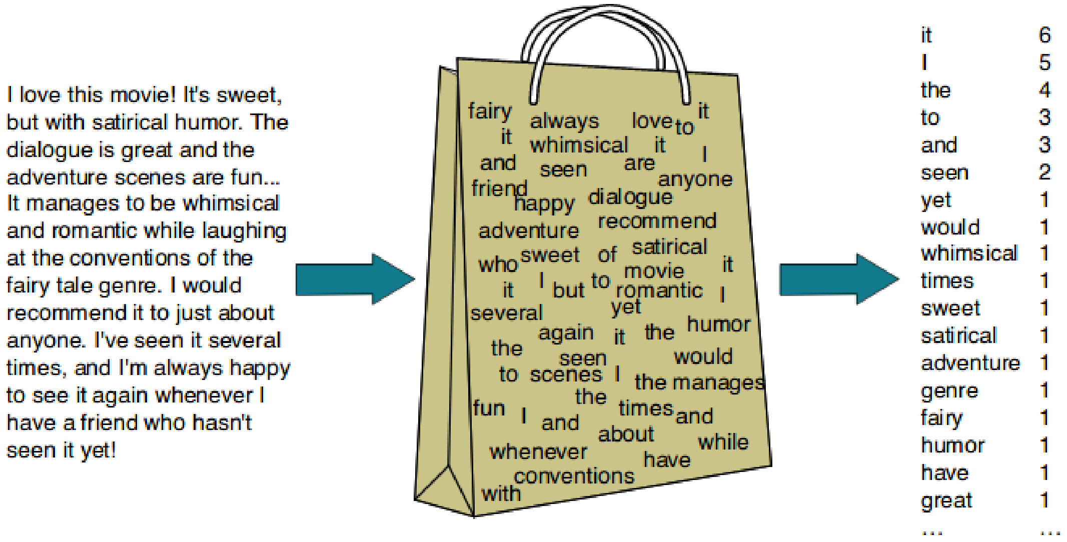
Supervised Learning with Naive Bayes - Case study

Case Study - Filtering mobile phone spam

Marcus Birkenkrahe

April 11, 2023

README



I love this movie! It's sweet, but with satirical humor. The dialogue is great and the adventure scenes are fun... It manages to be whimsical and romantic while laughing at the conventions of the fairy tale genre. I would recommend it to just about anyone. I've seen it several times, and I'm always happy to see it again whenever I have a friend who hasn't seen it yet!

it	6
I	5
the	4
to	3
and	3
seen	2
yet	1
would	1
whimsical	1
times	1
sweet	1
satirical	1
adventure	1
genre	1
fairy	1
humor	1
have	1
great	1
...	...

Figure 1: Bag of words technique illustrated

- This lecture and practice follows the case developed by Lantz (2019) and the Bag-of-Words method detailed in Kwartler (2017).

- We use the `tm` R package for text mining originally developed by Feinerer (2008).
- To code along with the lecture, download `5_naive_bayes_practice.org` from GitHub, complete the file and upload it to Canvas by the deadline.
- Exercises for extra credit will be provided, as well as a graded test.

Rationale

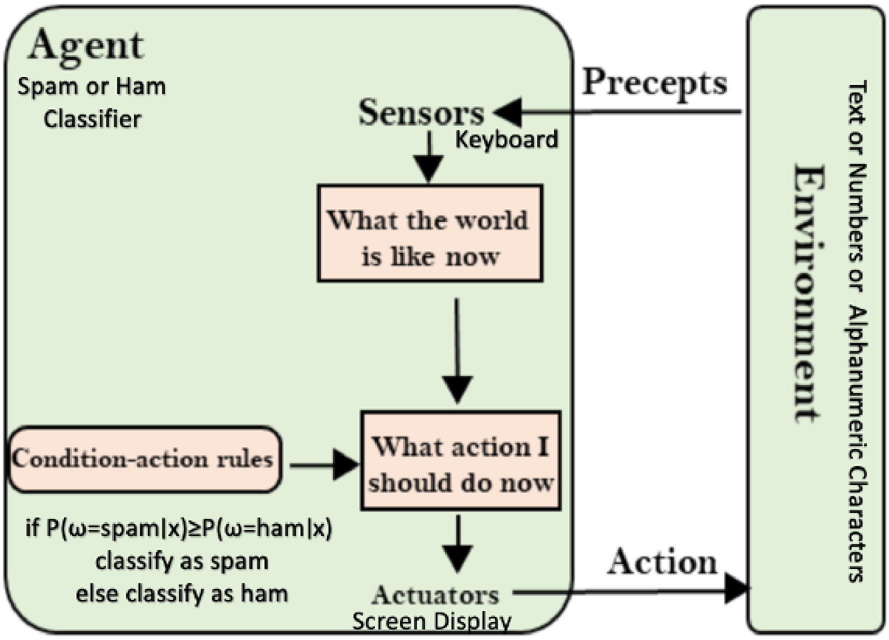


Figure 2: Classifier as AI agent. Illustration from: Deep, 2020

- SMS spam is particularly disruptive because phones are "always on". Cellular phone providers and private users need protection!
- SMS messages are limited to 160 characters: less text available and shorthand lingo makes spam identification more difficult.

- In addition to the regular ML workflow, we meet a multitude of text mining methods needed to prepare the data before we can train a model - these same methods are used to "data engineer" ChatGPT.

- There are many different approaches to solving this problem - e.g. using a Random Forest Algorithm (a supervised decision tree), see Sjarif et al (2019).

- For more details on the text mining methods employed, see Kwartler (2017) but also a complete DataCamp course.

ML workflow = Text Mining + Naive Bayes

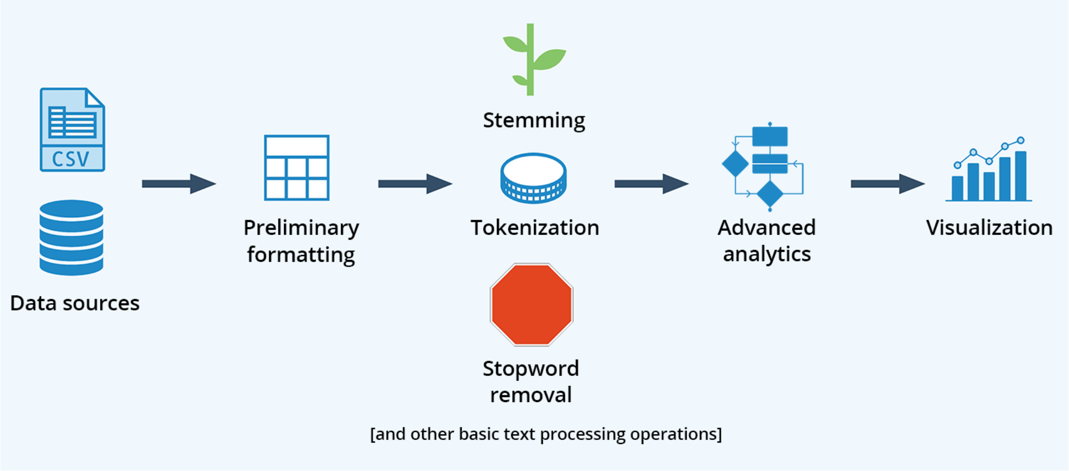
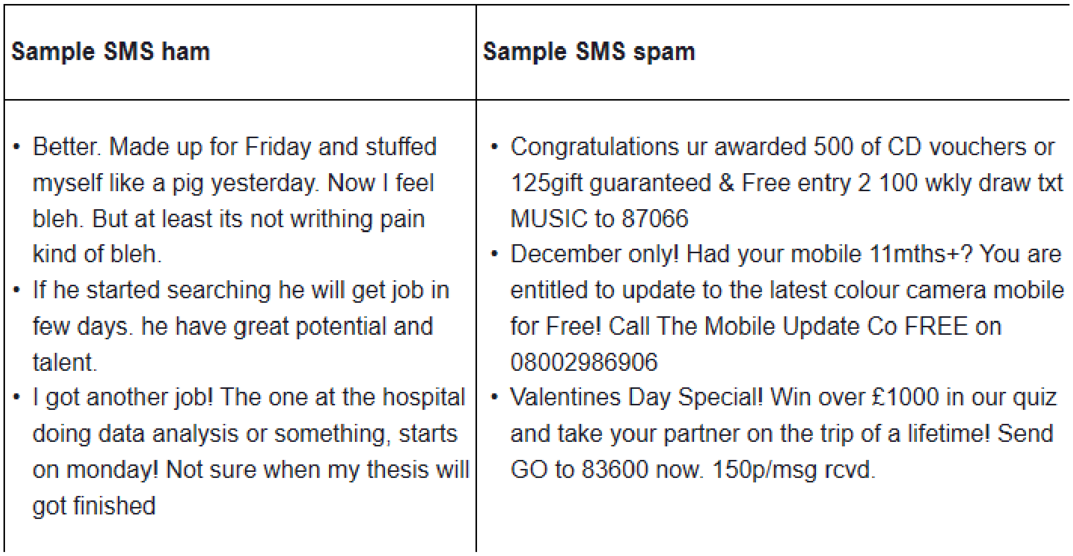


Figure 3: Basic (technical) text processing pipeline (Harris, 2016)

1. Getting the data (CSV to data frame) and creating a text corpus
2. Formatting - remove punctuation, whitespace, make lower case etc.
3. Tokenization - split messages into individual words
4. Stemming - remove word endings and re-complete
5. Stopwords - remove unnecessary words
6. Analytics - Document Term and Term Document matrices

7. Visualization - Word clouds and barcharts
8. Training the model
9. Testing the model
10. Improving the model

SMS Spam Collection Data



Sample SMS ham	Sample SMS spam
• Better. Made up for Friday and stuffed myself like a pig yesterday. Now I feel bleh. But at least its not writhing pain kind of bleh. • If he started searching he will get job in few days. he have great potential and talent. • I got another job! The one at the hospital doing data analysis or something, starts on monday! Not sure when my thesis will got finished	• Congratulations ur awarded 500 of CD vouchers or 125gift guaranteed & Free entry 2 100 wkly draw txt MUSIC to 87066 • December only! Had your mobile 11mths+? You are entitled to update to the latest colour camera mobile for Free! Call The Mobile Update Co FREE on 08002986906 • Valentines Day Special! Win over £1000 in our quiz and take your partner on the trip of a lifetime! Send GO to 83600 now. 150p/msg rcvd.

Figure 4: Sample spam vs. sample ham messages (Lantz, 2019).

- The SMS spam collection contains 5,574 SMS messages in English tagged as "spam" or "ham"[1].

☐ Do you notice any distinguishing spam signals in the table?[2]

- The spam vs. ham labelling relies on word frequency and context detection to identify potentially malicious patterns.

[1] The spam collection used here was modified by Lantz (2019). The original is from Gomez (2012) - the URL is no longer accessible, and I referenced the dataset from kaggle.com instead.

[2] Two of three spam messages use the word "free". Two of the ham messages cite specific days of the week, none of the spam.

- Ask ChatGPT what the "SMS Spam Collection" is:

  ```
  ask_chatgpt("What is the SMS Spam Collection?")
  ```

  > "The SMS Spam Collection is a dataset consisting of text messages that have been identified as spam. It was created to help researchers develop spam detection techniques and improve the accuracy of existing algorithms. The dataset includes over 5,500 messages in English and has been tagged with labels indicating whether each message is spam or not. The SMS Spam Collection is widely used in the research community and has helped advance the field of spam detection in recent years."

Collecting the data

- Take a look at the raw file to check if there's a header: bit.ly/sms_spam_csv
- Import the CSV data and save them to a data frame `sms_raw`. Do not automatically convert `character` to `factor` vectors. Use the appropriate function arguments:

  ```
  ## save CSV data as data frame sms_raw
  sms_raw <- read.csv(file="https://bit.ly/sms_spam_csv",
                      header=TRUE,
                      stringsAsFactors=FALSE)
  ```

- Check that the data frame was loaded:

  ```
  ls()
  ```

  ```
   [1] "a"                        "api_key"                  "ask_chatgpt"
   [4] "b"                        "bar"                      "chardonnay_corpus"
   [7] "chardonnay_df"            "chardonnay_src"           "chardonnay_vec"
  [10] "clean_chardonnay"         "clean_chardonnay_corpus" "clean_coffee"
  [13] "clean_coffee_corpus"      "coffee_corpus"            "coffee_df"
  [16] "coffee_src"               "coffee_vec"               "convert_counts"
  [19] "foo"                      "ham"                      "launch"
  [22] "load_packages"            "m"                        "r"
  [25] "reg"                      "sms_classifier"           "sms_classifier_"
  ```

```
[28] "sms_corpus"            "sms_corpus_clean"       "sms_dtm"
[31] "sms_dtm_freq_test"     "sms_dtm_freq_train"     "sms_dtm_test"
[34] "sms_dtm_train"         "sms_dtm2"               "sms_freq_words"
[37] "sms_raw"               "sms_test"               "sms_test_labels"
[40] "sms_test_pred"         "sms_train"              "sms_train_labels"
[43] "spam"                  "string"
```

Exploring the data

- Check the data structure:

```
str(sms_raw) ## check the data structure
head(sms_raw,2)
```

```
'data.frame': 5559 obs. of  2 variables:
 $ type: chr  "ham" "ham" "ham" "spam" ...
 $ text: chr  "Hope you are having a good week. Just checking in" "K..give back my
  type                                              text
1  ham Hope you are having a good week. Just checking in
2  ham                          K..give back my thanks.
```

- Convert the spam vs. ham label to a `factor` and confirm the conversion:

```
## convert class character vector to factor
factor(sms_raw$type) -> sms_raw$type
## confirm conversion to factor
is.factor(sms_raw$type)
str(sms_raw)
```

```
[1] TRUE
'data.frame': 5559 obs. of  2 variables:
 $ type: Factor w/ 2 levels "ham","spam": 1 1 1 2 2 1 1 1 2 1 ...
 $ text: chr  "Hope you are having a good week. Just checking in" "K..give back my
```

- Examine the frequency of spam vs. ham messages in the dataset:

```
table(sms_raw$type)  ## examine frequency of spam vs ham
```

```

 ham spam
4812  747
```

Getting the `tm` R package

Figure 5: tm is a tools package for text mining

- SMS messages are *strings* of text composed of words, spaces, numbers, and punctuation, with many uninteresting words like *but*, *and* etc.
- The text mining package `tm` (Feinerer et al, 2008) provides a bunch of functions to deconstruct text.
- Install and load `tm` (load it from the terminal if you haven't set `options()$repos` in your `~/.Rprofile` file). This is an actively developed package so re-installation will never do any harm:

  ```
  ## install tm
  install.packages("tm")
  ## load tm
  library(tm)
  ```

```
Installing package into 'C:/Users/birkenkrahe/AppData/Local/R/win-library/4.2'
(as 'lib' is unspecified)
trying URL 'https://cloud.r-project.org/bin/windows/contrib/4.2/tm_0.7-11.zip'
Content type 'application/zip' length 989797 bytes (966 KB)
downloaded 966 KB

package 'tm' successfully unpacked and MD5 sums checked
Warning: cannot remove prior installation of package 'tm'
Warning: restored 'tm'

The downloaded binary packages are in
C:\Users\birkenkrahe\AppData\Local\Temp\Rtmpuuq3NZ\downloaded_packages
Warning message:
In file.copy(savedcopy, lib, recursive = TRUE) :
  problem copying C:\Users\birkenkrahe\AppData\Local\R\win-library\4.2\00LOCK\tm\
Loading required package: NLP

Attaching package: 'NLP'

The following object is masked from 'package:httr':

    content

Warning message:
package 'tm' was built under R version 4.2.3
```

- Check that the package has been loaded and look at the methods (functions) and datasets included in `tm`:

```
search() ## check package has been loaded
ls('package:tm') ## list functions in tm
data(package="tm")
```

```
 [1] ".GlobalEnv"        "package:tm"        "package:NLP"       "ESSR"
 [5] "package:stats"     "package:graphics"  "package:grDevices" "package:utils"
 [9] "package:datasets"  "package:stringr"   "package:httr"      "package:methods"
[13] "Autoloads"         "package:base"
 [1] "as.DocumentTermMatrix"   "as.TermDocumentMatrix"   "as.VCorpus"
 [4] "Boost_tokenizer"         "content_transformer"     "Corpus"
 [7] "DataframeSource"         "DirSource"               "Docs"
```

```
[10] "DocumentTermMatrix"      "DublinCore"              "DublinCore<-"
[13] "eoi"                     "findAssocs"              "findFreqTerms"
[16] "findMostFreqTerms"       "FunctionGenerator"       "getElem"
[19] "getMeta"                 "getReaders"              "getSources"
[22] "getTokenizers"           "getTransformations"      "Heaps_plot"
[25] "inspect"                 "MC_tokenizer"            "nDocs"
[28] "nTerms"                  "PCorpus"                 "pGetElem"
[31] "PlainTextDocument"       "read_dtm_Blei_et_al"     "read_dtm_MC"
[34] "readDataframe"           "readDOC"                 "reader"
[37] "readPDF"                 "readPlain"               "readRCV1"
[40] "readRCV1asPlain"         "readReut21578XML"        "readReut21578XMLasPlain'
[43] "readTagged"              "readXML"                 "removeNumbers"
[46] "removePunctuation"       "removeSparseTerms"       "removeWords"
[49] "scan_tokenizer"          "SimpleCorpus"            "SimpleSource"
[52] "stemCompletion"          "stemDocument"            "stepNext"
[55] "stopwords"               "stripWhitespace"         "TermDocumentMatrix"
[58] "termFreq"                "Terms"                   "tm_filter"
[61] "tm_index"                "tm_map"                  "tm_parLapply"
[64] "tm_parLapply_engine"     "tm_reduce"               "tm_term_score"
[67] "URISource"               "VCorpus"                 "VectorSource"
[70] "weightBin"               "WeightFunction"          "weightSMART"
[73] "weightTf"                "weightTfIdf"             "writeCorpus"
[76] "XMLSource"               "XMLTextDocument"         "Zipf_plot"
[79] "ZipSource"
Warning message:
In file.show(outFile, delete.file = TRUE, title = paste("R", tolower(x$title))) :
  '"c:/PROGRA~1/R/R-42~1.2/bin/pager"' not found
```

DATA PREPARATION I - TEXT MINING AND VISUALIZATION

Building a document text corpus

- A *corpus* is a collection of text documents. It is a list of lists with a lot of meta-data slapped on to it.

- In order to be able to work with large text corpora, they need to be suitably organized and cleaned.

- Example: this corpus contains 150 billion Google Book documents.

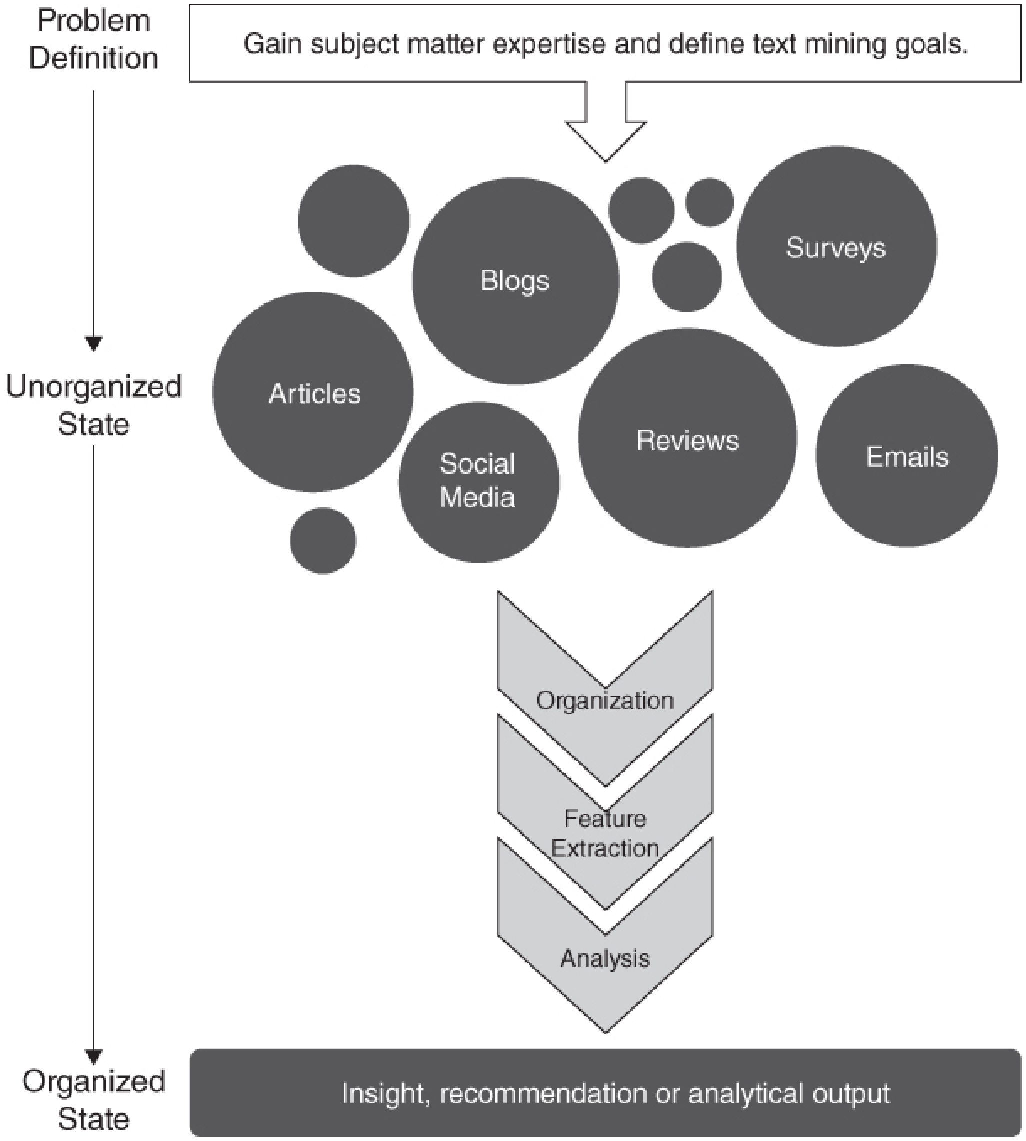


Figure 6: Tex mining workflow from Kwartler (2019)

- Three steps lead from a data frame with text to a corpus:

  1. Isolate the text vector
  2. Turn the vector into a source
  3. Turn the source into a corpus
  4. Check that the corpus is there

```
sms_corpus <- VCorpus(VectorSource(sms_raw$text))
ls()
```

```
 [1] "a"                       "api_key"                 "ask_chatgpt"
 [4] "b"                       "bar"                     "chardonnay_corpus"
 [7] "chardonnay_df"           "chardonnay_src"          "chardonnay_vec"
[10] "clean_chardonnay"        "clean_chardonnay_corpus" "clean_coffee"
[13] "clean_coffee_corpus"     "coffee_corpus"           "coffee_df"
[16] "coffee_src"              "coffee_vec"              "convert_counts"
[19] "foo"                     "ham"                     "launch"
[22] "load_packages"           "m"                       "r"
[25] "reg"                     "sms_classifier"          "sms_classifier_"
[28] "sms_corpus"              "sms_corpus_clean"        "sms_dtm"
[31] "sms_dtm_freq_test"       "sms_dtm_freq_train"      "sms_dtm_test"
[34] "sms_dtm_train"           "sms_dtm2"                "sms_freq_words"
[37] "sms_raw"                 "sms_test"                "sms_test_labels"
[40] "sms_test_pred"           "sms_train"               "sms_train_labels"
[43] "spam"                    "string"
```

- The VCorpus function creates a volatile, in-memory list that is not permanent (not for writing to an external database):

```
sms_corpus  # print the corpus label
```

```
<<VCorpus>>
Metadata:  corpus specific: 0, document level (indexed): 0
Content:  documents: 5559
```

Explore the text corpus

- The corpus is a list structure and its own R object class:

```
typeof(sms_corpus)
class(sms_corpus)

[1] "list"
[1] "VCorpus" "Corpus"
```

- You can see its content element-wise using list indexing. For example for message no. 1, `tm::inspect` returns meta data + content:

```
inspect(sms_corpus[[1]])

<<PlainTextDocument>>
Metadata:  7
Content:  chars: 49

Hope you are having a good week. Just checking in
```

- To extract a message, e.g. the first message, you can use the index operator `[[` subset by `[1]`, or you can use the function `tm::content`, or `as.character`:

```
sms_corpus[[1]][1]  ## extract msg content from corpus with [ ]
content(sms_corpus[[1]])   ## extrxact msg content from corpus
as.character(sms_corpus[[1]])

$content
[1] "Hope you are having a good week. Just checking in"
[1] "Hope you are having a good week. Just checking in"
[1] "Hope you are having a good week. Just checking in"
```

- While `tm::meta` returns only the meta information, which can be subset, too:

```
meta(sms_corpus)          # corpus metadata
meta(sms_corpus[[1]])     # metadata of first corpus element
meta(sms_corpus[[1]])[2]  # "datetimestamp" metadata of 1st element
```

```
data frame with 0 columns and 5559 rows
  author       : character(0)
  datetimestamp: 2023-04-11 15:55:40
  description  : character(0)
  heading      : character(0)
  id           : 1
  language     : en
  origin       : character(0)
$datetimestamp
[1] "2023-04-11 15:55:40 GMT"
```

- To see several list elements at once, `lapply` will apply its `FUN`

argument to all `list` members - for the first three messages:

```
lapply(sms_corpus[1:3], FUN=as.character)

$`1`
[1] "Hope you are having a good week. Just checking in"

$`2`
[1] "K..give back my thanks."

$`3`
[1] "Am also doing in cbe only. But have to pay."
```

Cleaning the text corpus: lower case, numbers

- The corpus contains the raw text of 5,559 messages. It needs to be standardized, which includes transforming all words to lower case, removing numbers and punctuation.
- Transformation of the whole corpus is done with the `tm_map` function, which accepts a corpus and a function as an argument:

  ```
  args(tm_map)

  function (x, FUN, ...)
  NULL
  ```

- To transform words to lower case, we use `base::tolower`

```
tolower("WHY ARE YOU YELLING AT ME!?")
```

```
[1] "why are you yelling at me!?"
```

- Since `tolower` is not in `tm`, we need to wrap it in another function, `tm::content_transformer`:

  ```
  tm_map(x=sms_corpus,
         FUN = content_transformer(tolower)) -> sms_corpus_clean
  ```

- Let's check that the transformation worked: print the `content` of the first message from the original and the transformed corpus:

  ```
  content(sms_corpus[[1]])
  content(sms_corpus_clean[[1]])
  ```

  ```
  [1] "Hope you are having a good week. Just checking in"
  [1] "hope you are having a good week. just checking in"
  ```

- To remove numbers from the SMS messages, use `tm::removeNumbers` on the new corpus object:

  ```
  tm_map(x=sms_corpus_clean,
         FUN=removeNumbers) -> sms_corpus_clean
  ```

- Compare the `content` of the original and transformed corpus for message 4:

  ```
  content(sms_corpus[[4]])
  content(sms_corpus_clean[[4]])
  ```

  ```
  [1] "complimentary 4 STAR Ibiza Holiday or £10,000 cash needs your URGENT collecti
  [1] "complimentary  star ibiza holiday or £, cash needs your urgent collection.  r
  ```

- To see all `tm` functions that can be used with `tm_map`, check the help for `getTransformations`. They are: `removeNumbers`, `removePunctuation`, `removeWords` and `stemDocument` (in connection with a dictionary), and `stripWhitespace`.

Removing stopwords and punctuation

- We need to remove filler words like *to*, *and*, *but* etc. These are known as *stopwords* and are removed before text mining.
- The tm package provides a stopwords function to access various sets of stop words from different languages. Check its arguments.

```
args(stopwords)
```

```
function (kind = "en")
NULL
```

- Which language contains the most stopwords? Compare the length of english, spanish and german tm::stopword dictionaries:

```
length(stopwords("english"))
length(stopwords("spanish"))
length(stopwords("german"))
```

```
[1] 174
[1] 308
[1] 231
```

- To apply stopwords to the corpus, run removeWords on it. The stopwords function is an additional parameter (cp. args(tm_map)):

```
tm_map(x=sms_corpus_clean,
       FUN=removeWords,
       c(stopwords("en"),"just")) -> sms_corpus_clean
```

- Compare the content of the first message of the original and the cleaned corpus:

```
content(sms_corpus[[1]])
content(sms_corpus_clean[[1]])
```

```
[1] "Hope you are having a good week. Just checking in"
[1] "hope     good week.  checking "
```

- Now remove the punctuation with `removePunctuation`, save the result in a new `sms_corpus_clean` object, and compare before/after for message 16 :

  ```
  tm_map(sms_corpus_clean, removePunctuation) -> sms_corpus_clean
  content(sms_corpus[[16]])
  content(sms_corpus_clean[[16]])

  [1] "Ha ha cool cool chikku chikku:-):-DB-)"
  [1] "ha ha cool cool chikku chikkudb"
  ```

- There are subtleties here: e.g. `removePunctuation` strips punctuation characters completely, with unintended consequences[3]:

  ```
  removePunctuation("hello...world")

  [1] "helloworld"
  ```

Word stemming with `SnowballC`

- Word stemming involves reducing words to their root form. It reduces words like "learning", "learned", "learns" to "learn".
- In this way, the classifier does not have to learn a pattern for each variant of what is semantically the same feature.
- `tm` integrates word-stemming with the `SnowballC` package which needs to be installed separately, alas. Load the package and check its content:

  ```
  library(SnowballC)
  search()
  ls('package:SnowballC')

   [1] ".GlobalEnv"        "package:SnowballC" "package:tm"        "package:NLP"
   [5] "ESSR"              "package:stats"     "package:graphics"  "package:grDevice
   [9] "package:utils"     "package:datasets"  "package:stringr"   "package:httr"
  [13] "package:methods"   "Autoloads"         "package:base"
  [1] "getStemLanguages" "wordStem"
  ```

[3] To work around this default, you can write your own function using `gsub`, which substitutes a pattern - in this case any punctuation is simply replaced by the string " " instead of remove altogether: `replacePunctuation <- function(x){gsub("[[:punct:]]+"," ",x)}`

- Which languages are available for stemming?

  ```
  getStemLanguages()
  ```

  ```
   [1] "arabic"     "basque"     "catalan"    "danish"     "dutch"      "english"
   [7] "finnish"    "french"     "german"     "greek"      "hindi"      "hungarian"
  [13] "indonesian" "irish"      "italian"    "lithuanian" "nepali"     "norwegian"
  [19] "porter"     "portuguese" "romanian"   "russian"    "spanish"    "swedish"
  [25] "tamil"      "turkish"
  ```

- Let's check the `SnowballC::wordStem` function on an example:

  ```
  library(SnowballC)
  wordStem(c("learn", "learned", "learning", "learns", "learner"))
  args(wordStem)
  ```

  ```
  [1] "learn"   "learn"   "learn"   "learn"   "learner"
  function (words, language = "porter")
  NULL
  ```

- The Porter algorithm used by `wordStem` does not recognize "learner" because it is not a word that can be broken down in its root and affixes using the algorithm's rules!

- To apply `wordStem` to the cleaned corpus with `tm_map`, use the `stemDocument` function, and check another message (25) for success[4]:

  ```
  tm_map(sms_corpus_clean, stemDocument) -> sms_corpus_clean
  content(sms_corpus[[25]])
  content(sms_corpus_clean[[25]])
  ```

  ```
  [1] "Could you not read me, my Love ? I answered you"
  [1] "read love answer"
  ```

- Lastly, remove additional whitespace using `stripWhitespace`, and check the first three messages for success:

[4] If you receive an error message `all scheduled cores encountered errors"` with `stemDocument`, add the parameter `mc.cores=1` to `tm_map`.

```
tm_map(sms_corpus_clean, stripWhitespace) -> sms_corpus_clean
lapply(sms_corpus[1:3],content)
lapply(sms_corpus_clean[1:3],content)

$`1`
[1] "Hope you are having a good week. Just checking in"

$`2`
[1] "K..give back my thanks."

$`3`
[1] "Am also doing in cbe only. But have to pay."
$`1`
[1] "hope good week check"

$`2`
[1] "kgive back thank"

$`3`
[1] "also cbe pay"
```

Tokenization - word splitting

- The final step is to split the messages into individuals terms or tokens, a single element of a text string - in this case, a word.
- The `DocumenTermMatrix` function takes a corpus and creates a document-term matrix (DTM) with rows as docs and columns as terms:

  ```
  sms_dtm <- DocumentTermMatrix(sms_corpus_clean)
  ```

- The DTM's transpose is the TDM (term-document matrix) - if the list of documents (columns) is small and the word list (rows) is large, TDM displays more easily.
- To look at the DTM, transform to a matrix:

  ```
  m <- as.matrix(sms_dtm)
  m[100:105, 100:108]
  ```

```
     Terms
Docs  adsens adult advanc adventur advic advis advisor aeronaut aeroplan
  100      0     0      0        0     0     0       0        0        0
  101      0     0      0        0     0     0       0        0        0
  102      0     0      0        0     0     0       0        0        0
  103      0     0      0        0     0     0       0        0        0
  104      0     0      0        0     0     0       0        0        0
  105      0     0      0        0     0     0       0        0        0
```

- Not much to see, is there? It's a sparse matrix with very few non-zero entries. How sparse exactly?

  ```
  dim(m)
  100 * length(which(m!=0))/(nrow(m)*ncol(m))
  ```

  ```
  [1] 5559 6536
  [1] 0.1149183
  ```

- In fact, the sparsity is contained in the meta-data of the DTM:

  ```
  sms_dtm
  ```

  ```
  <<DocumentTermMatrix (documents: 5559, terms: 6536)>>
  Non-/sparse entries: 41754/36291870
  Sparsity           : 100%
  Maximal term length: 40
  Weighting          : term frequency (tf)
  ```

- You can also create a DTM directly from the raw, unprocessed SMS corpus:

  ```
  sms_dtm2 <- DocumentTermMatrix(sms_corpus,
                                 control = list(
                                   tolower = TRUE,
                                   removeNumbers = TRUE,
                                   stopwords = TRUE,
                                   removePunctuation = TRUE,
                                   stemming = TRUE))
  dim(sms_dtm2)
  ```

```
[1] 5559 6940
```

- You notice a difference in the number of terms: this is due to the fact that `DocumentTermMatrix` uses a different `stopwords` function[5].
- This illustrates an important text mining principle: the order of operations matters!

Load `.RData` and `save.image`

At the start load the data generated so far: in the code block below, you need to adapt the path to the `.RData` file to your own computer:

```
load("~/Downloads/ml_RData")
search()
ls()
```

```
 [1] ".GlobalEnv"        "package:SnowballC" "package:tm"        "package:NLP"
 [5] "ESSR"              "package:stats"     "package:graphics"  "package:grDevices"
 [9] "package:utils"     "package:datasets"  "package:stringr"   "package:httr"
[13] "package:methods"   "Autoloads"         "package:base"
 [1] "a"                        "api_key"                  "ask_chatgpt"
 [4] "b"                        "bar"                      "chardonnay_corpus"
 [7] "chardonnay_df"            "chardonnay_src"           "chardonnay_vec"
[10] "clean_chardonnay"         "clean_chardonnay_corpus" "clean_coffee"
[13] "clean_coffee_corpus"     "coffee_corpus"            "coffee_df"
[16] "coffee_src"               "coffee_vec"               "convert_counts"
[19] "foo"                      "ham"                      "launch"
[22] "load_packages"            "m"                        "r"
[25] "reg"                      "sms_classifier"           "sms_classifier_"
[28] "sms_corpus"               "sms_corpus_clean"         "sms_dtm"
[31] "sms_dtm_freq_test"        "sms_dtm_freq_train"       "sms_dtm_test"
[34] "sms_dtm_train"            "sms_dtm2"                 "sms_freq_words"
[37] "sms_raw"                  "sms_test"                 "sms_test_labels"
[40] "sms_test_pred"            "sms_train"                "sms_train_labels"
[43] "spam"                     "string"
```

At the end:

[5] To force the two prior DTMs to be identical, we can override the default with our own anonymous function: set `stopwords=function(x){removeWords(x),stopwords())}`

```
save.image(".RData")
shell("DIR .RData")

 Volume in drive C is OS
 Volume Serial Number is 0654-135C

 Directory of c:\Users\birkenkrahe\Documents\GitHub\ml\org

04/11/2023  10:55 AM         2,980,147 .RData
               1 File(s)      2,980,147 bytes
               0 Dir(s)  201,528,180,736 bytes free
```

Text visualization with `wordcloud`

- Word clouds visually show the frequency of words in text data.
- Words appearing more/less often are shown in larger/smaller font
- We use the `wordcloud` package to compare the clouds for "spam" and "ham" messages to gauge if our spam filter is working or not[6].
- Install and load the package:

  ```
  ## Do this only if options()$repos is set to cloud.r-project.org/
  options()$repos
  ## install.packages("wordcloud")
  library(wordcloud)
  search()

  [1] "https://cloud.r-project.org/"
  Loading required package: RColorBrewer
  Warning message:
  package 'wordcloud' was built under R version 4.2.3
   [1] ".GlobalEnv"          "package:wordcloud"    "package:RColorBrewer"
   [4] "package:SnowballC"    "package:tm"           "package:NLP"
   [7] "ESSR"                 "package:stats"        "package:graphics"
  [10] "package:grDevices"    "package:utils"        "package:datasets"
  [13] "package:stringr"      "package:httr"         "package:methods"
  [16] "Autoloads"            "package:base"
  ```

[6] For more info on the package, visit blog.fellstat.com - this San Diego CA company has developed a few interesting packages including `OpenStreetMap` for GIS, `wordcloud`, `deducer` and a poker-playing program.

- If you want to know more about the R loading process, look at `help(Startup)`
- Check out the functions in the package:

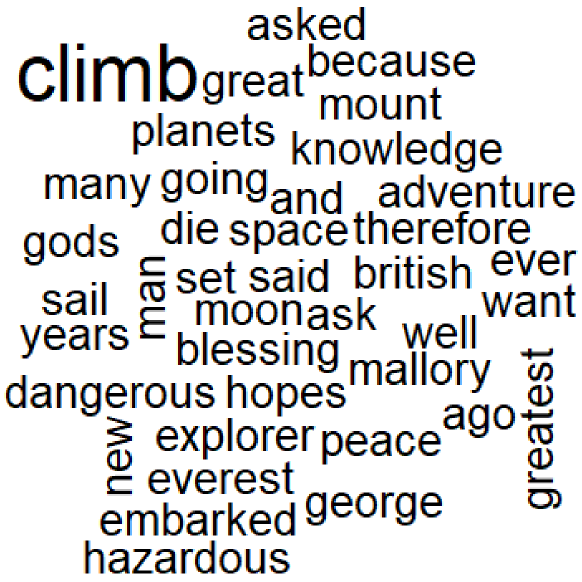
  ```
  ls(’package:wordcloud’)

  [1] "commonality.cloud" "comparison.cloud"  "textplot"          "wordcloud"
  [5] "wordlayout"
  ```

- Check out the arguments of the `wordcloud` function:

  ```
  args(wordcloud::wordcloud)

  function (words, freq, scale = c(4, 0.5), min.freq = 3, max.words = Inf,
      random.order = TRUE, random.color = FALSE, rot.per = 0.1,
      colors = "black", ordered.colors = FALSE, use.r.layout = FALSE,
      fixed.asp = TRUE, ...)
  NULL
  ```

- A simple example: running the function on a string:

  ```
  string <- "Many years ago the great British explorer George Mallory,
  who was to die on Mount Everest, was asked why did he want to climb it.
  He said, \"Because it is there.\" Well, space is there,
  and we’re going to climb it, and the moon and the planets
  are there, and new hopes for knowledge and peace are there.
  And, therefore, as we set sail we ask God’s blessing on the
  most hazardous and dangerous and greatest adventure on which
  man has ever embarked."
  wordcloud(words=string, ,random.order=TRUE)
  ```

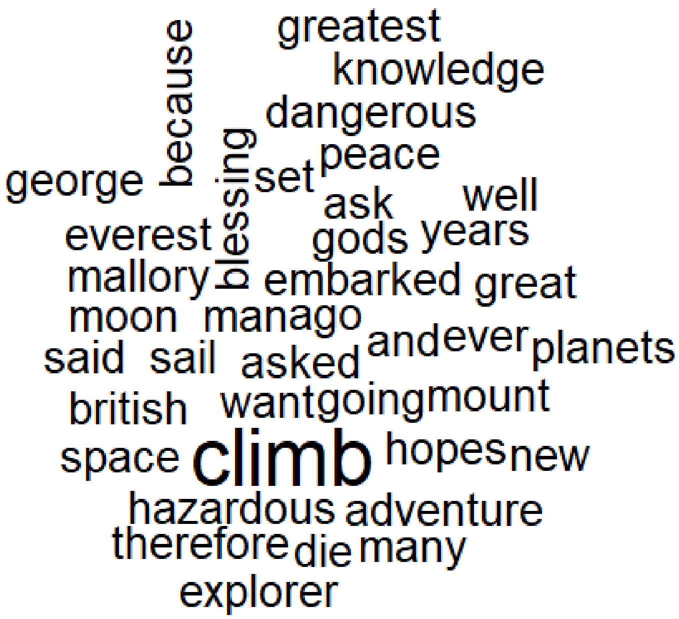


- The function has evidently applied some cleaning and tokenizing operations automatically!
- Let's do the tokenization explicitly with:
    1. `qdap::bracketX` to remove brackets
    2. `tm::removePunctuation` to remove punctuation
    3. `strsplit` to tokenize
    4. `unlist` to transform the `list` result to a vector

```
library(qdap)
library(tm)
bracketX(string) -> stringX
stringX |>
  removePunctuation() |>
  strsplit(split=" ") |>
  unlist() -> tokens
tokens
## same as:
## tokens <- unlist(strsplit(removePunctuation(stringX),split=" "))

Loading required package: qdapDictionaries
Loading required package: qdapRegex
Loading required package: qdapTools

Attaching package: 'qdap'

The following objects are masked from 'package:tm':

    as.DocumentTermMatrix, as.TermDocumentMatrix

The following object is masked from 'package:NLP':

    ngrams

The following objects are masked from 'package:base':

    Filter, proportions
 [1] "Many"      "years"     "ago"       "the"       "great"     "British"  
 [7] "explorer"  "George"    "Mallory"   "who"       "was"       "to"       
[13] "die"       "on"        "Mount"     "Everest"   "was"       "asked"    
[19] "why"       "did"       "he"        "want"      "to"        "climb"    
[25] "it"        "He"        "said"      "Because"   "it"        "is"       
[31] "there"     "Well"      "space"     "is"        "there"     "and"      
[37] "were"      "going"     "to"        "climb"     "it"        "and"      
[43] "the"       "moon"      "and"       "the"       "planets"   "are"      
[49] "there"     "and"       "new"       "hopes"     "for"       "knowledge"
[55] "and"       "peace"     "are"       "there"     "And"       "therefore"
[61] "as"        "we"        "set"       "sail"      "we"        "ask"      
[67] "Gods"      "blessing"  "on"        "the"       "most"      "hazardous"
[73] "and"       "dangerous" "and"       "greatest"  "adventure" "on"       
[79] "which"     "man"       "has"       "ever"      "embarked" 
```

- Now we get a different cloud (meaning that the internal tokenization of wordcloud works differently):

```
wordcloud(tokens)
```



Spam vs ham visualization

- Back to our spam filter! Look at the arguments of `wordcloud` again: you'll need to change `words`, `min.freq` and `random.order`:

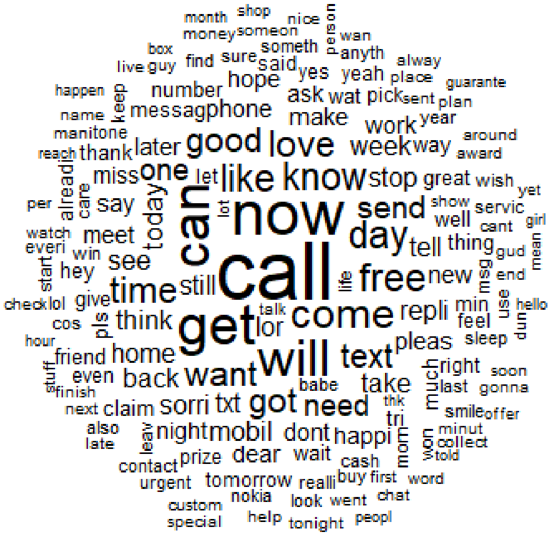
```
args(wordcloud)

function (words, freq, scale = c(4, 0.5), min.freq = 3, max.words = Inf,
    random.order = TRUE, random.color = FALSE, rot.per = 0.1,
    colors = "black", ordered.colors = FALSE, use.r.layout = FALSE,
    fixed.asp = TRUE, ...)
NULL
```

- A word cloud can be created directly from a `tm` corpus[7]:

[7] If you an error message that R could not fit all words on the figure, increase

1. We use the cleaned corpus of SMS messages
2. Words must be found in > 1% of the corpus (50/5000)
3. Place higher-frequency words closer to the center:

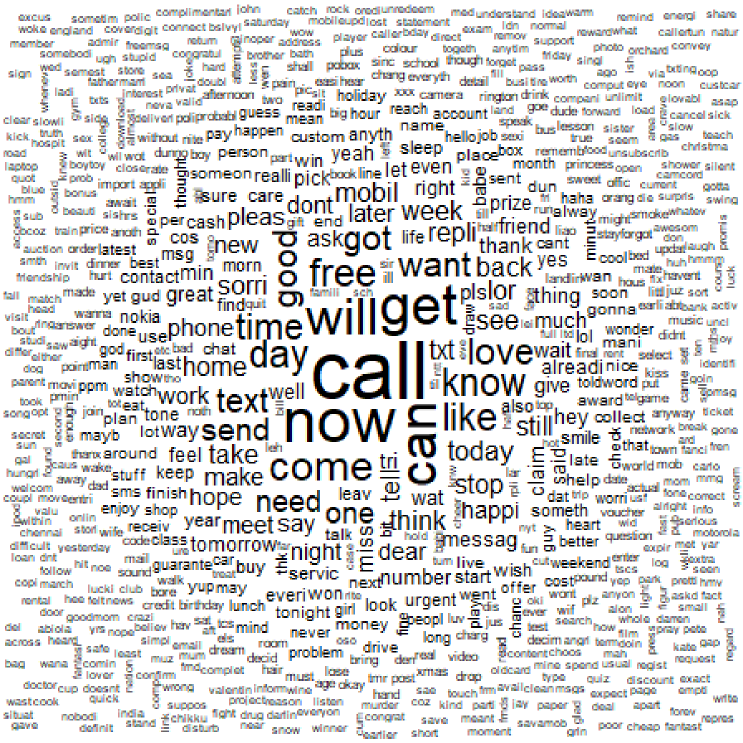
```
wordcloud(words=sms_corpus_clean,
          min.freq=50,
          random.order=FALSE)
```



- Redraw the word cloud with altered arguments: change
  1. the minimum frequency `min.freq` to 200 and 10
  2. the `scale` (`c(font,cex)`) to different values (`font` takes values 1 to 4, and `cex` takes any value. The default is `c(4,0.5)`.

`min.freq` to reduce the number of words in the cloud, or reduce the font size with `scale=c(font,cex)`.

```
wordcloud(sms_corpus_clean,
          min.freq=10,
          scale=c(4,0.5),
          random.order=FALSE)
```



- More interesting is a comparison of the clouds for spam and ham. `wordcloud` will automatically preprocess so we can use `sms_raw`.
- Split the data into spam and ham messages using `subset`:

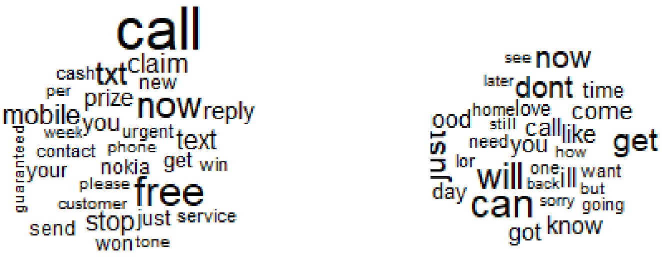
```
spam <- subset(sms_raw, type == "spam")
ham <- subset(sms_raw, type == "ham")
```

- Create two wordclouds side by side looking only at the 30 most common words in each of the two sets - can you guess which is which?

  1. set `max.words` to 30

2. set the `spam scale` to `c(3,0.5)`
3. set the `ham scale` to `c(2,0.2)`

```
par(mfrow=c(1,2),pty='m')
wordcloud(spam$text, max.words=30, scale=c(3,0.5))
wordcloud(ham$text, max.words=30, scale=c(2,0.1))
```

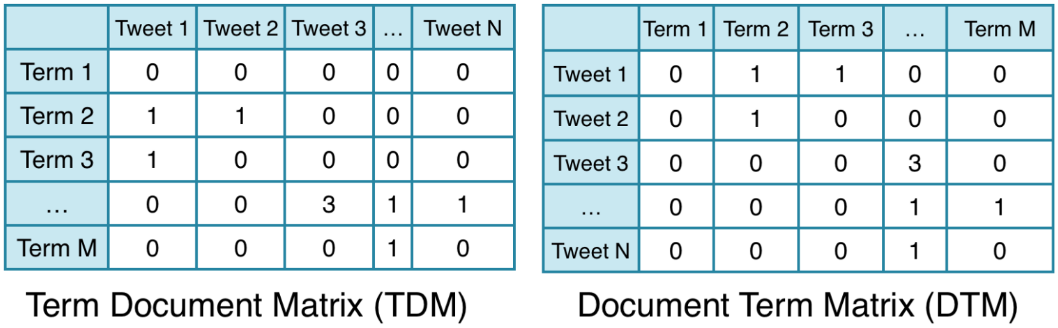


- Because of the randomization process in the function, the clouds will look different each time you run the function, and you can pick the cloud that looks most appealing for presentation purposes.
- **Spam** SMS messages include words like "free", "stop", "cash", "guaranteed", while **ham** SMS messages contain words like "can", "time", "will" and "just".
- The `wordcloud` package has other interesting functions like `comparison.cloud` and `commonality.cloud` to visualize dis/similar words in two data sets,

but they are not applicable to our spam/ham scenario, which is based on disjoint term sets.

DATA PREPARATION II - TRAINING AND TEST DATA

Creating training and test data



	Tweet 1	Tweet 2	Tweet 3	…	Tweet N
Term 1	0	0	0	0	0
Term 2	1	1	0	0	0
Term 3	1	0	0	0	0
…	0	0	3	1	1
Term M	0	0	0	1	0

Term Document Matrix (TDM)

	Term 1	Term 2	Term 3	…	Term M
Tweet 1	0	1	1	0	0
Tweet 2	0	1	0	0	0
Tweet 3	0	0	0	3	0
…	0	0	0	1	1
Tweet N	0	0	0	1	0

Document Term Matrix (DTM)

Figure 7: Term-Document and Document-Term Matrix for a corpus of tweets

- Split data into training and test datasets to allow for creation and evaluation of the model.
- It is important that the data are split **after** the data have been cleaned and processed - both training and test data need to have undergone exactly the same treatment.
- For the visualization, we used the TDM - summing over the columns (documents) returned the frequency for each word, which lead to bargraphs (word size of the word cloud is the `height` of `barplot`).
- For the prediction, we'll go back to the DTM whose columns are our features (word) to whom we attach probabilities so that we can compute the conditional probabilities P(spam|word).
- The DTM object is structured very much like a data frame and can be split using the familiar `[row,col]` operation where rows are messages and columns are words:

```
str(sms_dtm)  # rows = documents, columns = terms

List of 6
 $ i       : int [1:41754] 1 1 1 1 2 2 2 3 3 3 ...
 $ j       : int [1:41754] 958 2269 2568 6187 428 2973 5607 193 907 4088 ...
 $ v       : num [1:41754] 1 1 1 1 1 1 1 1 1 1 ...
 $ nrow    : int 5559
 $ ncol    : int 6536
 $ dimnames:List of 2
  ..$ Docs : chr [1:5559] "1" "2" "3" "4" ...
  ..$ Terms: chr [1:6536] "‘morrow" "‘rent" "’llspeak" "’re" ...
 - attr(*, "class")= chr [1:2] "DocumentTermMatrix" "simple_triplet_matrix"
 - attr(*, "weighting")= chr [1:2] "term frequency" "tf"
```

- Since the SMS messages are already sorted randomly, we simply take the first 75% (4,169) messages for training and leave 25% (1,390) for testing:

```
sms_dtm_train <- sms_dtm[1:4169, ]
sms_dtm_test  <- sms_dtm[4170:5559, ]
```

- Save a pair of vectors with the class labels "spam" or "ham" for each message - these labels are not stored in the DTM (remember that we used `sms_raw$text` to define the corpus) but in the raw data frame in the `type` column:

```
str(sms_raw)

'data.frame': 5559 obs. of  2 variables:
 $ type: chr  "ham" "ham" "ham" "spam" ...
 $ text: chr  "Hope you are having a good week. Just checking in" "K..give back my
```

- Extract the corresponding rows for training and testing labels:

```
sms_train_labels <- sms_raw[1:4169, ]$type
sms_test_labels <- sms_raw[4170:5559, ]$type
```

- To confirm that the subsets are representative of the complete set of SMS data, compute the proportion of spam and ham labels:

```
prop.table(table(sms_train_labels))
prop.table(table(sms_test_labels))
```

```
sms_train_labels
      ham      spam
0.8647158 0.1352842
sms_test_labels
      ham      spam
0.8683453 0.1316547
```

- Spam is evenly divided between training and test dataset (13%).

Reducing training features with `findFreqTerms`

- The sparse matrix currently contains over 6,500 features - one feature for every word that appears in at least one SMS message:

```
dim(sms_dtm)    # documents = rows, words = columns = features
dim(t(sms_dtm))
```

```
[1] 5559 6536
[1] 6536 5559
```

- It's unlikely that all of these are useful for classification so we reduce the features by eliminating any word appearing in < 5 (0.1%) of the messages.
- The `tm::findFreqTerms` function takes a DTM and returns a `character` vector containing words with frequencies in the interval `[lowfreq,highfreq]`:

```
args(findFreqTerms)
```

```
function (x, lowfreq = 0, highfreq = Inf)
NULL
```

- We save the vector in `sms_freq_words`:

```
library(tm)
findFreqTerms(sms_dtm_train, lowfreq = 5) -> sms_freq_words
```

- Check the structure of `sms_freq_words`:

  ```
  str(sms_freq_words)
  ```

  ```
  chr [1:1137] "£wk" "abiola" "abl" "abt" "accept" "access" "account" "across" ...
  ```

- There are 1,137 words appearing in at least 5 SMS messages - we've reduced the dimension of our features by 83%.
- Some of the terms show the result of word-stemming without re-completion ("abl"), and not having removed abbreviations and symbols ("£wk")[8].
- We narrow our training and test features already stored using `sms_freq_words`: we use the column index to include all rows:

  ```
  sms_dtm_freq_train <- sms_dtm_train[ ,sms_freq_words]
  sms_dtm_freq_test <- sms_dtm_test[ ,sms_freq_words]
  ```

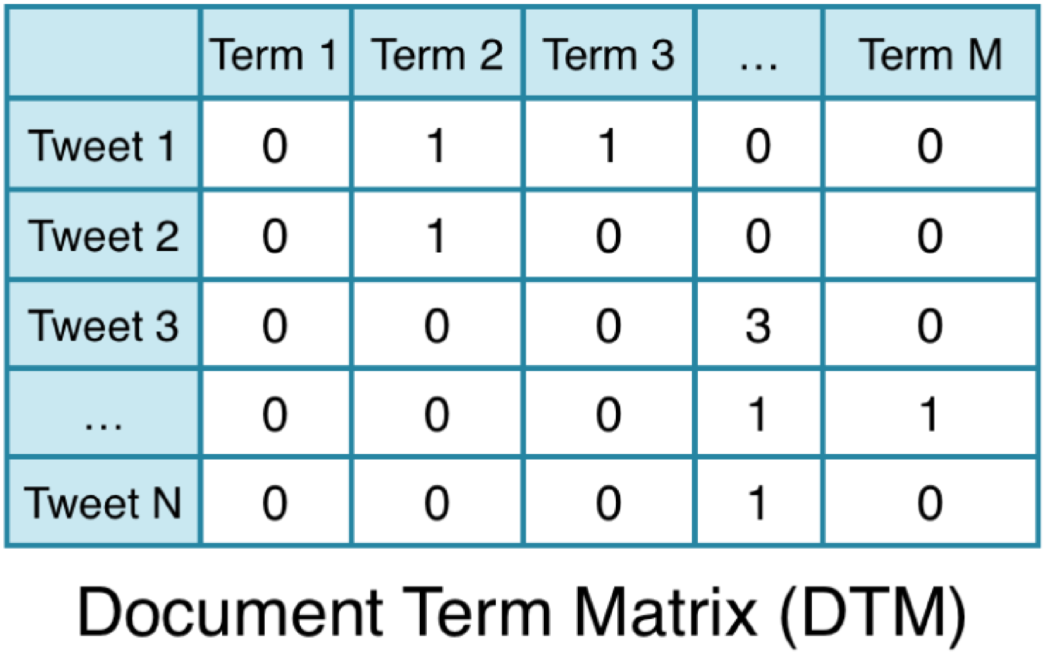
Convert `numeric` counts to categorical features

- The Naive Bayes classifier is trained on data with categorical features ("spam" vs. "ham") but the DTM cells record the number of times a word appears in a message:
- We convert the counts to "Yes" or "No" strings with a simple function, and apply the function to the whole matrix with `apply`.
- The conversion function uses `ifelse` as a way of testing a condition (`x > 0`) for all elements of a vector:

  ```
  ## function definition
  convert_counts <- function (x) { x <- ifelse(test = (x > 0),
                                               yes = "Yes",
                                               no = "No") }
  ```

- The `apply` function applies its function argument `FUN` to all elements of an array by row (`MARGIN=1`) or by column (`MARGIN=2`) - here, we're interested in columns:

[8] The `qdap` text cleaning package contains plenty of functions for additional corpus cleaning. In a real scenario, we'd run those functions on our corpus, too.



	Term 1	Term 2	Term 3	…	Term M
Tweet 1	0	1	1	0	0
Tweet 2	0	1	0	0	0
Tweet 3	0	0	0	3	0
…	0	0	0	1	1
Tweet N	0	0	0	1	0

Document Term Matrix (DTM)

Figure 8: Document-Term-Matrix for a corpus of tweets

```
## function definition
convert_counts <- function (x) { x <- ifelse(test = (x > 0),
                                             yes = "Yes",
                                             no = "No") }
apply(sms_dtm_freq_train,2,convert_counts) -> sms_train
apply(X = sms_dtm_freq_test,
      MARGIN = 2,
      FUN = convert_counts) -> sms_test
```

- The result are our final training and test data in the form of two matrices with "No" for 0 and "Yes" for non-zero frequencies:

```
dim(sms_train)
dim(sms_test)
sms_train[2:3,2:3]  # head of the training data matrix
sms_test[100:102,1135:1137]  # tail of the test data matrix
```

```
[1] 4169 1137
[1] 1390 1137
    Terms
Docs abiola abl
   2 "No"   "No"
   3 "No"   "No"
      Terms
Docs   yet  yoga yup
  4269 "No" "No" "No"
  4270 "No" "No" "No"
  4271 "No" "No" "No"
```

- Taking stock! The `ls()` function has a pattern argument. Use it to list all objects you've defined so far for the SMS messages (all of these objects begin with "sms"):

```
ls(pattern="^sms")   # regular expression "^sms"
```

```
 [1] "sms_classifier"     "sms_classifier_"    "sms_corpus"
 [4] "sms_corpus_clean"   "sms_dtm"            "sms_dtm_freq_test"
 [7] "sms_dtm_freq_train" "sms_dtm_test"       "sms_dtm_train"
[10] "sms_dtm2"           "sms_freq_words"     "sms_raw"
[13] "sms_test"           "sms_test_labels"    "sms_test_pred"
[16] "sms_train"          "sms_train_labels"
```

Training a classifier on the data

- We have transformed the raw SMS messages into a format that can be represented by a statistical model.
- The Naive Bayes algorithm uses the presence or absence of words to estimate the probability that a given SMS message is spam.
- We use the algorithm implemented in the imaginatively named `e1071` package from the TU Wien[9]:

  1. Install the package (unless you already did that)
  2. Load the package with `library`
  3. Make sure it's loaded with `search`
  4. Take a look at the functions contained in it with `ls`:

```
## Do this only if options()$repos is set to cloud.r-project.org/
options()$repos
## install.packages("e1071")
library(e1071)
search()
ls('package:e1071')
```

```
[1] "https://cloud.r-project.org/"
Warning message:
package 'e1071' was built under R version 4.2.3
 [1] ".GlobalEnv"               "package:e1071"            "package:qdap"
 [4] "package:qdapTools"        "package:qdapRegex"        "package:qdapDictionari
 [7] "package:wordcloud"        "package:RColorBrewer"     "package:SnowballC"
[10] "package:tm"               "package:NLP"              "ESSR"
[13] "package:stats"            "package:graphics"         "package:grDevices"
[16] "package:utils"            "package:datasets"         "package:stringr"
[19] "package:httr"             "package:methods"          "Autoloads"
[22] "package:base"
 [1] "allShortestPaths"      "bclust"                "best.gknn"
 [4] "best.nnet"             "best.randomForest"     "best.rpart"
 [7] "best.svm"              "best.tune"             "bincombinations"
[10] "bootstrap.lca"         "centers.bclust"        "classAgreement"
```

[9] An almost identical alternative is the `NaiveBayes` function in the `klaR` package from the TU Dortmund, Germany. Both are well maintained.

```
[13] "clusters.bclust"       "cmeans"                "compareMatchedClasses"
[16] "countpattern"          "cshell"                "d2sigmoid"
[19] "ddiscrete"             "dsigmoid"              "element"
[22] "extractPath"           "fclustIndex"           "gknn"
[25] "hamming.distance"      "hamming.window"        "hanning.window"
[28] "hclust.bclust"         "hsv_palette"           "ica"
[31] "impute"                "interpolate"           "kurtosis"
[34] "lca"                   "matchClasses"          "matchControls"
[37] "moment"                "naiveBayes"            "pdiscrete"
[40] "permutations"          "probplot"              "qdiscrete"
[43] "rbridge"               "rdiscrete"             "read.matrix.csr"
[46] "rectangle.window"      "rwiener"               "scale_data_frame"
[49] "sigmoid"               "skewness"              "stft"
[52] "svm"                   "tune"                  "tune.control"
[55] "tune.gknn"             "tune.knn"              "tune.nnet"
[58] "tune.randomForest"     "tune.rpart"            "tune.svm"
[61] "write.matrix.csr"      "write.svm"
```

- Unlike the k-NN algorithm, training and using the Naive Bayes algorithm occurs in several steps:

- The training with `naiveBayes` includes a parameter for Laplace correction and returns a model `m`:

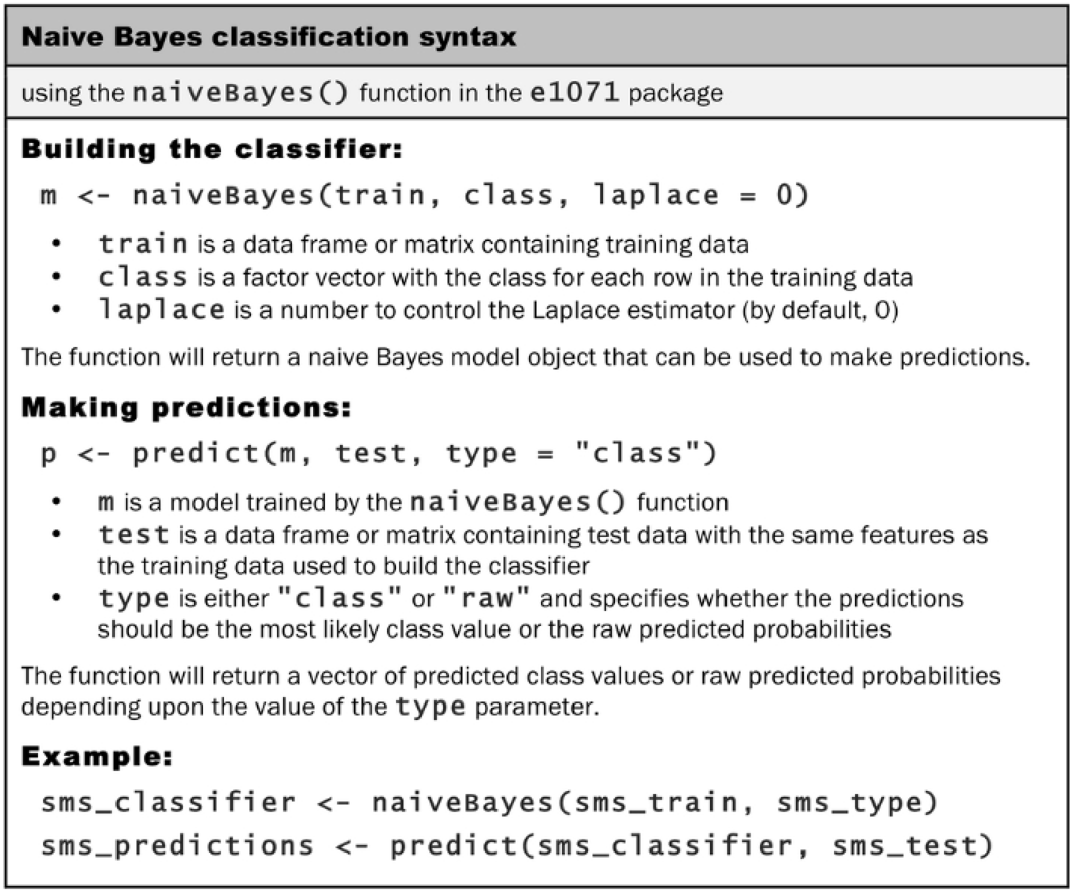
- The `predict` function runs the model (`object`) `m` on the (unseen) test data (`newdata`) and returns a vector of predicted labels.

- We build our model `sms_classifier` on the `sms_train` matrix with the associated `sms_train_labels` vector:

  ```
  library(e1071)
  naiveBayes(x = sms_train,
             y = sms_train_labels) -> sms_classifier
  ```

- The `sms_classifier` variable now contains a `naiveBayes` classifier `list` object that can be used to make predictions: let's look at

  1. the class of the model
  2. the data structure of the model
  3. the probabilities for two words from the "spam" and "ham" pile, "free" and "come" - as a `table`:



Naive Bayes classification syntax

using the `naiveBayes()` function in the `e1071` package

Building the classifier:

```
m <- naiveBayes(train, class, laplace = 0)
```

- `train` is a data frame or matrix containing training data
- `class` is a factor vector with the class for each row in the training data
- `laplace` is a number to control the Laplace estimator (by default, 0)

The function will return a naive Bayes model object that can be used to make predictions.

Making predictions:

```
p <- predict(m, test, type = "class")
```

- `m` is a model trained by the `naiveBayes()` function
- `test` is a data frame or matrix containing test data with the same features as the training data used to build the classifier
- `type` is either `"class"` or `"raw"` and specifies whether the predictions should be the most likely class value or the raw predicted probabilities

The function will return a vector of predicted class values or raw predicted probabilities depending upon the value of the `type` parameter.

Example:

```
sms_classifier <- naiveBayes(sms_train, sms_type)
sms_predictions <- predict(sms_classifier, sms_test)
```

Figure 9: Naive Bayes classification syntax (Lantz, 2019)

```
class(sms_classifier)
typeof(sms_classifier)
which(sms_freq_words=="free") -> foo  # index of "free" labels
which(sms_freq_words=="come") -> bar  # index of "come" labels
sms_classifier$table[[foo]]
sms_classifier$table[[bar]]
```

```
[1] "naiveBayes"
[1] "list"
free
sms_train_labels         No        Yes
    ham  0.98751734 0.01248266
    spam 0.76950355 0.23049645
come
sms_train_labels          No         Yes
    ham  0.942579750 0.057420250
    spam 0.991134752 0.008865248
```

- Just for fun, how does this compare with `klaR::NaiveBayes`?

```
library(klaR)
sms_classifier_ <- NaiveBayes(sms_train, factor(sms_train_labels))
sms_classifier_$table[[which(sms_freq_words=="free")]]
sms_classifier_$table[[which(sms_freq_words=="come")]]
```

```
Loading required package: MASS
Warning message:
package 'klaR' was built under R version 4.2.3
var
grouping         No        Yes
    ham  0.98751734 0.01248266
    spam 0.76950355 0.23049645
var
grouping          No         Yes
    ham  0.942579750 0.057420250
    spam 0.991134752 0.008865248
```

Evaluating model performance

- To evaluate the classifier `sms_classifier`, we test its predictions on the unseen messages in the test data stored in the matrix `sms_test`, with associated class labels stored in sms_test_labels.

- The predict function is part of the base R installation in the stats package - it only needs a model object (the classifier) and a test dataset - this will take a while to execute:

```
predict(sms_classifier,
        sms_test) -> sms_test_pred  # this holds our predictions!
```

- Let's get an overview of the proportional probabilities:

```
prop.table(table(sms_test_pred))
prop.table(table(sms_test_labels))
```

```
sms_test_pred
      ham      spam
0.8856115 0.1143885
sms_test_labels
      ham      spam
0.8683453 0.1316547
```

- How accurate is our classifier? Average over the misidentified message labels with mean:

```
paste("Misidentified messages: ",
      format((
        mean(sms_test_pred!=sms_test_labels))*100,
        digits=2),"%")
```

```
[1] "Misidentified messages:  2.6 %"
```

- For a confidence matrix overview, we use gmodels::CrossTable with reduced cell output (suppressing various proportions):

```
library(gmodels)
CrossTable(x = sms_test_pred,
           y = sms_test_labels,
           prop.chisq=FALSE,prop.c=FALSE,prop.r=FALSE,
           dnn = c('predicted', 'actual'))
```

```
   Cell Contents
|-------------------------|
|                       N |
|         N / Table Total |
|-------------------------|


Total Observations in Table:  1390


     | actual
   predicted |       ham |      spam | Row Total |
-------------|-----------|-----------|-----------|
 ham |      1201 |        30 |      1231 |
     |     0.864 |     0.022 |           |
-------------|-----------|-----------|-----------|
spam |         6 |       153 |       159 |
     |     0.004 |     0.110 |           |
-------------|-----------|-----------|-----------|
Column Total |      1207 |       183 |      1390 |
-------------|-----------|-----------|-----------|
```

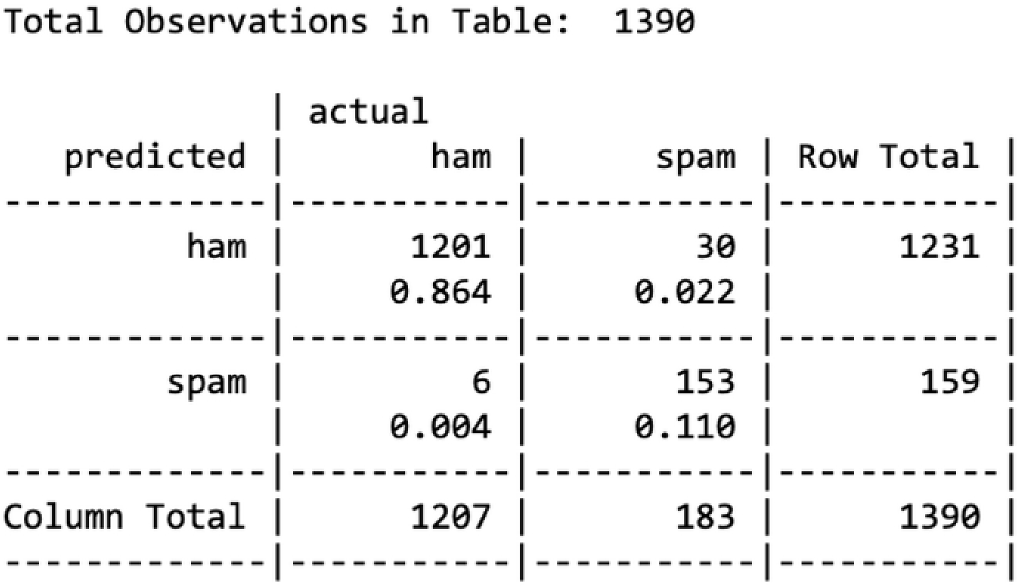
- Let's look at the results:

  1. Only 30 false negatives (actual spam classified as ham)
  2. Only 6 + 30 = 36 of 1,390 messages (2.6%) misidentified
  3. Only 6 false positives (actual ham classified as spam)
  4. 6 wrongly filtered messages could mean important messages!

- For the relatively little effort we made, this out of the box result is pretty impressive! Next stop: tweak the model.

Improving model performance

- Since we kept the Laplace correction at 0 during training, words that appeared in zero spam or zero ham messages influenced the result.



```
Total Observations in Table:  1390

             | actual
   predicted |       ham |      spam | Row Total |
-------------|-----------|-----------|-----------|
         ham |      1201 |        30 |      1231 |
             |     0.864 |     0.022 |           |
-------------|-----------|-----------|-----------|
        spam |         6 |       153 |       159 |
             |     0.004 |     0.110 |           |
-------------|-----------|-----------|-----------|
Column Total |      1207 |       183 |      1390 |
-------------|-----------|-----------|-----------|
```

Figure 10: Naive Bayes spam filter results as gmodels::CrossTable.

- Just because a word like "ringtone" only appeared in spam messages in the training data, does not mean that every message with this word should be classified as spam.

- We build a new classifier with `laplace=0.1` adding a small correction to the conditional probabilities:

```
sms_classifier2 <- naiveBayes(x = sms_train,
                              y = sms_train_labels,
                              laplace = 0.1)
```

- We repeat our prediction with the new classifier:

```
sms_test_pred2 <- predict(sms_classifier2, sms_test)
```

- Check new accuracy:

```
paste("Misidentified messages: ",
      format((mean(sms_test_pred2!=sms_test_labels))*100,
             digits=2),"%")
```

```
[1] "Misidentified messages:  2.2 %"
```

- Check new confidence matrix:

```
library(gmodels)
CrossTable(x = sms_test_pred2,
           y = sms_test_labels,
           prop.chisq=FALSE,prop.c=FALSE,prop.r=FALSE,
           dnn = c('predicted', 'actual'))
```

```
   Cell Contents
|-------------------------|
|                       N |
|         N / Table Total |
|-------------------------|


Total Observations in Table:  1390


     | actual
   predicted |       ham |      spam | Row Total |
-------------|-----------|-----------|-----------|
 ham |      1202 |        26 |      1228 |
     |     0.865 |     0.019 |           |
-------------|-----------|-----------|-----------|
spam |         5 |       157 |       162 |
     |     0.004 |     0.113 |           |
-------------|-----------|-----------|-----------|
Column Total |      1207 |       183 |      1390 |
-------------|-----------|-----------|-----------|
```

- We've improved the result a little - we have reduced the number of false positive (ham classified as spam) from 6 to 5, and the number of false negatives (spam classified as ham) from 30 to 26.

- When tweaking further, we need to be careful because we need to strike a balance between overly aggressive (strong filter) and overly passive (weak filter): users would prefer that a small number of spam messages gets through rather than losing too many ham messages.

Glossary of code

COMMAND	MEANING
`tm`	text mining package
`tm::VectorSource`	turn vector into source
`tm::VCorpus`	turn source into volatile corpus
`tm::inspect`	look at corpus elements
`tm::content`	look at corpus content
`tm::meta`	look at corpus meta data
`as.character`	convert value to `character`
`lapply(X,FUN)`	apply function to list elements
`apply(X,MARGIN,FUN)`	apply function to arrays
`tm::tm_map`	run function on whole corpus
`base::tolower`	convert characters to lower case
`tm::content_transformer`	transform function to run on corpus
`tm::removeNumbers`	remove numbers
`tm::stripWhitespace`	remove white space
`tm::stopwords`	get stop words dictonary
`tm::stopwords("en")`	English stop words dictionary
`tm::removePunctuation`	remove punctuation
`SnowballC`	word stemming package
`SnowballC::getStemLanguages`	languages available for word stemming
`SnowballC::wordStem`	stem words (default: English)
`tm::DocumentTermMatrix`	make matrix of docs x terms (DTM)
`tm::TermDocumentMatrix`	make matrix of terms x docs (TDM)
`wordcloud`	package for word cloud visualization
`options()$repos`	package download repository URL
`wordcloud::wordcloud`	make wordcloud from R object
`par(mfrow=c(1,2)`	create 1 x 2 panel for plots
`tm::findFreqTerms`	find frequent terms in DTM
`ifelse (test,yes,no)`	apply condition to vector
`ls(pattern="^a")`	list objects beginning with "a"
`e1071`, `klaR`	Naive Bayes algorithm packages
`e1071::naiveBayes`	create Naive Bayes classifier
`stats::predict`	run model object on new data set

Summary

Text Mining and Naive Bayes Classification:

- Preparing the text data for analysis requires specialized R packages for text processing (`tm`, `qdap`, `SnowballC`) and visualization (`wordcloud`).
- An out-of-the-box classification using the `e1071` or `klaR` algorithm packages yields a 97% success rate for an SMS message spam filter with NB.

Solutions

Collecting the data

```
sms_raw <- read.csv(file = "https://bit.ly/sms_spam_csv",
                    header = TRUE,  # this is not the default
                    stringsAsFactors = FALSE) # this is the default
ls()
```

```
 [1] "a"                      "api_key"                "ask_chatgpt"
 [4] "b"                      "bar"                    "chardonnay_corpus"
 [7] "chardonnay_df"          "chardonnay_src"         "chardonnay_vec"
[10] "clean_chardonnay"       "clean_chardonnay_corpus" "clean_coffee"
[13] "clean_coffee_corpus"    "coffee_corpus"          "coffee_df"
[16] "coffee_src"             "coffee_vec"             "convert_counts"
[19] "foo"                    "ham"                    "launch"
[22] "load_packages"          "m"                      "r"
[25] "reg"                    "sms_classifier"         "sms_classifier_"
[28] "sms_classifier2"        "sms_corpus"             "sms_corpus_clean"
[31] "sms_dtm"                "sms_dtm_freq_test"      "sms_dtm_freq_train"
[34] "sms_dtm_test"           "sms_dtm_train"          "sms_dtm2"
[37] "sms_freq_words"         "sms_raw"                "sms_test"
[40] "sms_test_labels"        "sms_test_pred"          "sms_test_pred2"
[43] "sms_train"              "sms_train_labels"       "spam"
[46] "string"                 "stringX"                "tokens"
```

Exploring the data

```
sms_raw <- read.csv(file = "https://bit.ly/sms_spam_csv",
                    header = TRUE,  # this is not the default
                    stringsAsFactors = FALSE) # this is the default
ls()
str(sms_raw)  # data frame structure
factor(sms_raw$type) -> sms_raw$type   # converting type to factor
is.factor(sms_raw$type)  # logical check if type is now factor
str(sms_raw)  # structure after conversion
table(sms_raw$type)   # frequency table for all levels in type
prop.table(table(sms_raw$type))  # proportions
## fancy formatted proportions printout
paste(format(prop.table(table(sms_raw$type)) * 100, digits=4),"%")
```

```
 [1] "a"                       "api_key"                 "ask_chatgpt"
 [4] "b"                       "bar"                     "chardonnay_corpus"
 [7] "chardonnay_df"           "chardonnay_src"          "chardonnay_vec"
[10] "clean_chardonnay"        "clean_chardonnay_corpus" "clean_coffee"
[13] "clean_coffee_corpus"     "coffee_corpus"           "coffee_df"
[16] "coffee_src"              "coffee_vec"              "convert_counts"
[19] "foo"                     "ham"                     "launch"
[22] "load_packages"           "m"                       "r"
[25] "reg"                     "sms_classifier"          "sms_classifier_"
[28] "sms_classifier2"         "sms_corpus"              "sms_corpus_clean"
[31] "sms_dtm"                 "sms_dtm_freq_test"       "sms_dtm_freq_train"
[34] "sms_dtm_test"            "sms_dtm_train"           "sms_dtm2"
[37] "sms_freq_words"          "sms_raw"                 "sms_test"
[40] "sms_test_labels"         "sms_test_pred"           "sms_test_pred2"
[43] "sms_train"               "sms_train_labels"        "spam"
[46] "string"                  "stringX"                 "tokens"
'data.frame':	5559 obs. of  2 variables:
 $ type: chr  "ham" "ham" "ham" "spam" ...
 $ text: chr  "Hope you are having a good week. Just checking in" "K..give back my than
[1] TRUE
'data.frame':	5559 obs. of  2 variables:
 $ type: Factor w/ 2 levels "ham","spam": 1 1 1 2 2 1 1 1 2 1 ...
 $ text: chr  "Hope you are having a good week. Just checking in" "K..give back my than

 ham spam
4812  747

      ham      spam
0.8656233 0.1343767
```

```
[1] "86.56 %" "13.44 %"
```

Getting the tm package

```
install.packages("tm") ## install tm
library(tm) ## load tm
search() ## check package has been loaded
ls('package:tm') ## list functions in tm
data(package='tm')  ## datasets in package

Warning: package 'tm' is in use and will not be installed
 [1] ".GlobalEnv"               "package:gmodels"          "package:klaR"
 [4] "package:MASS"             "package:e1071"            "package:qdap"
 [7] "package:qdapTools"        "package:qdapRegex"        "package:qdapDictionaries"
[10] "package:wordcloud"        "package:RColorBrewer"     "package:SnowballC"
[13] "package:tm"               "package:NLP"              "ESSR"
[16] "package:stats"            "package:graphics"         "package:grDevices"
[19] "package:utils"            "package:datasets"         "package:stringr"
[22] "package:httr"             "package:methods"          "Autoloads"
[25] "package:base"
 [1] "as.DocumentTermMatrix"   "as.TermDocumentMatrix"   "as.VCorpus"
 [4] "Boost_tokenizer"         "content_transformer"     "Corpus"
 [7] "DataframeSource"         "DirSource"               "Docs"
[10] "DocumentTermMatrix"      "DublinCore"              "DublinCore<-"
[13] "eoi"                     "findAssocs"              "findFreqTerms"
[16] "findMostFreqTerms"       "FunctionGenerator"       "getElem"
[19] "getMeta"                 "getReaders"              "getSources"
[22] "getTokenizers"           "getTransformations"      "Heaps_plot"
[25] "inspect"                 "MC_tokenizer"            "nDocs"
[28] "nTerms"                  "PCorpus"                 "pGetElem"
[31] "PlainTextDocument"       "read_dtm_Blei_et_al"     "read_dtm_MC"
[34] "readDataframe"           "readDOC"                 "reader"
[37] "readPDF"                 "readPlain"               "readRCV1"
[40] "readRCV1asPlain"         "readReut21578XML"        "readReut21578XMLasPlain"
[43] "readTagged"              "readXML"                 "removeNumbers"
[46] "removePunctuation"       "removeSparseTerms"       "removeWords"
[49] "scan_tokenizer"          "SimpleCorpus"            "SimpleSource"
[52] "stemCompletion"          "stemDocument"            "stepNext"
[55] "stopwords"               "stripWhitespace"         "TermDocumentMatrix"
[58] "termFreq"                "Terms"                   "tm_filter"
```

```
[61] "tm_index"                "tm_map"                  "tm_parLapply"
[64] "tm_parLapply_engine"     "tm_reduce"               "tm_term_score"
[67] "URISource"               "VCorpus"                 "VectorSource"
[70] "weightBin"               "WeightFunction"          "weightSMART"
[73] "weightTf"                "weightTfIdf"             "writeCorpus"
[76] "XMLSource"               "XMLTextDocument"         "Zipf_plot"
[79] "ZipSource"
Warning message:
In file.show(outFile, delete.file = TRUE, title = paste("R", tolower(x$title))) :
  ’"c:/PROGRA~1/R/R-42~1.2/bin/pager"’ not found
```

Cleaning: lower case and numbers

- Let’s check that the transformation worked: print the `content` of the first message from the original and the transformed corpus:

  ```
  content(sms_corpus[[1]])
  content(sms_corpus_clean[[1]])
  ```

  ```
  [1] "Hope you are having a good week. Just checking in"
  [1] "hope good week check"
  ```

- To remove numbers from the SMS messages, use `tm::removeNumbers` on the new corpus object:

  ```
  tm_map(sms_corpus_clean, removeNumbers) -> sms_corpus_clean
  ```

- Compare the `content` of the original and transformed corpus for message 4:

  ```
  content(sms_corpus[[4]])
  content(sms_corpus_clean[[4]])
  ```

  ```
  [1] "complimentary 4 STAR Ibiza Holiday or £10,000 cash needs your URGENT collect:
  [1] "complimentari star ibiza holiday £ cash need urgent collect now landlin lose
  ```

Removing stopwords and punctuation

- The `tm` package provides a `stopwords` function to access various sets of stop words from different languages. Check its arguments.

```
args(stopwords)

function (kind = "en")
NULL
```

- Which language contains the most stopwords? Compare the `length` of `english`, `spanish` and `german` `tm::stopword` dictionaries:

  ```
  length(stopwords("english"))
  length(stopwords("spanish"))
  length(stopwords("german"))

  [1] 174
  [1] 308
  [1] 231
  ```

- To apply `stopwords` to the corpus, run `removeWords` on it. The `stopwords` function is an additional parameter (cp. `args(tm_map)`):

  ```
  tm_map(sms_corpus_clean,
         removeWords,
         stopwords("en")) -> sms_corpus_clean
  ```

- Compare the `content` of the first message of the original and the cleaned corpus:

  ```
  content(sms_corpus[[1]])
  content(sms_corpus_clean[[1]])

  [1] "Hope you are having a good week. Just checking in"
  [1] "hope good week check"
  ```

- Now remove the punctuation with `removePunctuation`, save the result in a new `sms_corpus_clean` object, and compare before/after for message 16 :

  ```
  tm_map(sms_corpus_clean, removePunctuation) -> sms_corpus_clean
  content(sms_corpus[[16]])
  content(sms_corpus_clean[[16]])

  [1] "Ha ha cool cool chikku chikku:-):-DB-)"
  [1] "ha ha cool cool chikku chikkudb"
  ```

Word stemming

- `tm` integrates word-stemming with the `SnowballC` package which needs to be installed separately, alas. Load the package and check its content:

  ```
  library(SnowballC)
  search()
  ls('package:SnowballC')
  ```

  ```
   [1] ".GlobalEnv"               "package:gmodels"          "package:klaR"
   [4] "package:MASS"             "package:e1071"            "package:qdap"
   [7] "package:qdapTools"        "package:qdapRegex"        "package:qdapDictionari
  [10] "package:wordcloud"        "package:RColorBrewer"     "package:SnowballC"
  [13] "package:tm"               "package:NLP"              "ESSR"
  [16] "package:stats"            "package:graphics"         "package:grDevices"
  [19] "package:utils"            "package:datasets"         "package:stringr"
  [22] "package:httr"             "package:methods"          "Autoloads"
  [25] "package:base"
  [1] "getStemLanguages" "wordStem"
  ```

- Which languages are available for stemming? getStemLanguages() getStemLanguages() #+end_src

  ```
   [1] "arabic"     "basque"     "catalan"    "danish"     "dutch"
   [6] "english"    "finnish"    "french"     "german"     "greek"
  [11] "hindi"      "hungarian"  "indonesian" "irish"      "italian"
  [16] "lithuanian" "nepali"     "norwegian"  "porter"     "portuguese"
  [21] "romanian"   "russian"    "spanish"    "swedish"    "tamil"
  [26] "turkish"
  ```

References

- Data: UCI Machine Learning, SMS Spam Collection Dataset. URL: kaggle.com.

- Deep (2020). The Ultimate Guide TO SMS: Spam or Ham Classifier Using Python. URL: towardsdatascience.com.

- Feinerer et al (2008). Text mining infrastructure in R. In: J Stat Software 25:1-54. URL: cran.r-project.org.

- Gomez et al (2012). On the Validity of a New SMS Spam Collection. In: Proceedings of the 11th IEEE International Conference on Machine Learning and Applications.

- Harris (Oct 3, 2016). What Is Text Analytics? We Analyze the Jargon. URL: softwareadvice.com.

- Kwartler (2017). Text Mining in Practice with R. Wiley. URL: wiley.com.

- Lantz (2019). Machine learning with R (3e). Packt. URL: packtpub.com.

- R Core Team (2022). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL `https://www.R-project.org/`.

- Ripley/Venables (January 23, 2023). Package 'class': Various functions for classification, including k-nearest neighbour, Learning Vector Quantization, and Self-Organizing Maps. URL: cran.r-project.org.

- Sjarif et al (January, 2019). SMS Spam Message Detection using Term Frequency-Inverse Document Frequency and Random Forest Algorithm. In: Procedia Comp Sci 161:509-515. DOI: 10.1016/j.procs.2019.11.150. URL: researchgate.net.

- Warnes (October 13, 2022). Package 'gmodels': Various R Programming Tools for Model Fitting. URL: cran.r-project.org.