Natural Language Processing with Disaster Tweets. Analysis and classification prediction

Carlos Morote García

0. Preliminaries

Check if the libraries that are going to be used along this library are installed. In case it is detected that one is not installed, it will be installed automatically.

```
source("./requirements.R")
```

Once we have made sure that the libraries have been installed, they will be imported in order to run the rest of the notebook.

```
# Load of all the libraries

library(data.table)
library(hunspell)
library(qdap)
```

```
## Loading required package: qdapDictionaries
## Loading required package: qdapRegex
## Loading required package: qdapTools
##
## Attaching package: 'qdapTools'
## The following object is masked from 'package:data.table':
##
##     shift
## Loading required package: RColorBrewer
##
## Attaching package: 'qdap'
## The following objects are masked from 'package:base':
##
##     Filter, proportions
```

```
library(utf8)
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
## The following object is masked from 'package:qdapTools':
##
```

```
##     id
## The following object is masked from 'package:qdapRegex':
##
##     explain
## The following objects are masked from 'package:data.table':
##
##     between, first, last
## The following objects are masked from 'package:stats':
##
##     filter, lag
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(quanteda)
```

```
## Package version: 3.2.0
## Unicode version: 13.0
## ICU version: 68.2
## Parallel computing: 8 of 8 threads used.
## See https://quanteda.io for tutorials and examples.
##
## Attaching package: 'quanteda'
## The following object is masked from 'package:hunspell':
##
##     dictionary
```

```
library(quanteda.textmodels)
library(quanteda.textplots)

library(wordcloud)
library(tm)
```

```
## Loading required package: NLP
##
## Attaching package: 'NLP'
## The following objects are masked from 'package:quanteda':
##
##     meta, meta<-
## The following object is masked from 'package:qdap':
##
##     ngrams
##
## Attaching package: 'tm'
## The following object is masked from 'package:quanteda':
##
##     stopwords
## The following objects are masked from 'package:qdap':
```

```
##
##     as.DocumentTermMatrix, as.TermDocumentMatrix
```

```
library(caret)
```

```
## Loading required package: ggplot2
```

```
##
## Attaching package: 'ggplot2'
```

```
## The following object is masked from 'package:NLP':
##
##     annotate
```

```
## The following object is masked from 'package:qdapRegex':
##
##     %+%
```

```
## Loading required package: lattice
```

We will load the variables, lists and functions defined in the `healpers.R` file. This is done in an external file to keep the notebook as readable as possible.

```
source("./helper.R")
```

Finally, we will establish a seed to control the generation of numerous random samples in order to make the experiments reproducible.

```
set.seed(957735)
```

1. Import data

We load the data sets, both the training set (for which we know the classification) and the test set (for which we do not know the classification).

```
df.train <- fread("./data/train.csv")
df.test <- fread("./data/test.csv")
```

As this work is intended to explore natural language processing, we will eliminate those variables that are not related. In this case we eliminate for both data sets: `keyword` and `location`.

```
# Train dataset
df.train$id <- NULL
df.train$keyword <- NULL
df.train$location <- NULL

# Test dataset
df.test$keyword <- NULL
df.test$location <- NULL
```

Cast the target variable into a factor

```
df.train$target <- as.factor(df.train$target)
```

Analyzing very superficially the resulting DataFrame we observe that there are almost a thousand more cases where the tweet does not correspond to a natural disaster.

```
summary(df.train)
```

```
##      text              target
##  Length:7613        0:4342
##  Class :character   1:3271
##  Mode  :character
```

We check if the character encoding is correct (*utf-8*). In this case we check that it is in this format.

```
df.train$text[!utf8_valid(df.train$text)]
```

```
## character(0)
```

```
NFC_df <- utf8_normalize(df.train$text)
sum(NFC_df != df.train$text) # It is normalized
```

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

```
NFC_df <- utf8_normalize(df.test$text)
sum(NFC_df != df.test$text) # It is normalized
```

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

2. Data and Corpus preprocessing

In this section we will generate the corpus for the training and test datasets. We will also process these corpus to make them ready and clean to be analyzed by the subsequent models. Additionally, these corpus will be used to perform a basic analysis of the texts.

Before we start we will use the `hunspell` library to detect grammatical errors in the text. We perform this check since these texts come from Twitter and since they are not formal texts, but texts from various sources, it is more than likely that there are multiple grammatical errors. In addition, on Twitter, due to the limited number of characters that can be used per tweet, we have to cut words or use contributions that are not grammatically correct.

The results provided by this method are given in a table format where TRUE means that **no** error exists, while FALSE means that **yes** error exists. Each value corresponds to a word. This notebook considers as a word any sequence of characters separated by a space.

We note that 24308 out of 113650 words have some grammatical error. This means that 21% of the words have problems.

```
# Detects spelling errors

summary(unlist(strsplit(as.character(df.train$text), split = " ")) %>%
hunspell_check() )
```

```
##    Mode   FALSE    TRUE
## logical   24308   89342
```

On the other hand, the test data set presents 10703 errors out of 48876 words. This is another 21%.

```
summary(unlist(strsplit(as.character(df.test$text), split = " ")) %>%
hunspell_check() )
```

```
##    Mode   FALSE    TRUE
## logical   10703   38173
```

Next we generate the **Corpus** using the `tm` library method.

```
df.train.corpus.original <- Corpus(VectorSource(df.train$text))
df.test.corpus.original <- Corpus(VectorSource(df.test$text))
```

First we transform all generated tokens, words in this case, to lowercase.

```
df.train.corpus <- tm_map(df.train.corpus.original, content_transformer(tolower))
df.test.corpus <- tm_map(df.test.corpus.original, content_transformer(tolower))
```

Twitter has a tag system to allow quick searches by tags, as well as to allow grouping tweets by the same topic. These are the hashtags, which are identified with the hash symbol (#). These tags may contain useful information, but by their nature they tend to group multiple words without spaces, causing the algorithm to detect them as a single one. To extract the maximum knowledge from these tags we have made use of regular expressions by which it will detect the different words within a Hashtag, as long as they are differentiated with the first letter of each word capitalized. Therefore, the hashtag #SpainOnFire would be transformed into the three words that compose them: Spain, On, Fire. Additionally, they will be transformed into lowercase letters to match the transformation previously made.

```
df.train.corpus <- tm_map(df.train.corpus, content_transformer(function(text){gsub("[#]{1,}([A-Z][^A-Z]
df.test.corpus <- tm_map(df.test.corpus, content_transformer(function(text){gsub("[#]{1,}([A-Z][^A-Z]*)
```

The mention of users (made with the @ symbol followed by the user name) is much more difficult to extract the words they may contain. In many cases, nicknames do not correlate with reality, since the original names of people are not unique, many times the numbers are manipulated to create a user name that is unique. Therefore, usernames will be removed from the Corpus.

```
df.train.corpus <- tm_map(df.train.corpus, content_transformer(function(text){gsub("@\\S+ ", "", text)}
df.test.corpus <- tm_map(df.test.corpus, content_transformer(function(text){gsub("@\\S+ ", "", text)}))
```

The urls as a text source do not provide any useful information since they could be considered a succession of random characters that only have in common the beginning (*http...*). If one wanted to go deeper into this problem, these links are reverencing an image on the web, therefore, they could be extracted and analyzed with other algorithms in order to generate derived variables. As this is not the object of this work, the latter will not be implemented. Therefore, the urls will be eliminated.

```
df.train.corpus <- tm_map(df.train.corpus, content_transformer(function(text){gsub("\\S*http+\\S*", "",
df.test.corpus <- tm_map(df.test.corpus, content_transformer(function(text){gsub("\\S*http+\\S*", "", t
```

As was the case with urls, emojis also do not provide relevant information regarding the text. Hence, they will be removed. The definition of what is an emoji is composed in the `healper.R` file.

```
df.train.corpus <- tm_map(df.train.corpus, content_transformer(function(text){mgsub(text, pattern = emo
df.test.corpus <- tm_map(df.test.corpus, content_transformer(function(text){mgsub(text, pattern = emoji
```

Contractions are often treated as a single token when in fact they represent two or more tokens. They are also often treated as different tokens but on their contracted version. This means for example that it will differentiate between the *is* token and the *'s* token (from *He's* for example). To solve this, a list of equivalences of a contraction with its extended version has been made. Based on these references the captured contractions have been discarded.

```
df.train.corpus <- tm_map(df.train.corpus, content_transformer(function(text){replace_contraction(text,
df.test.corpus <- tm_map(df.test.corpus, content_transformer(function(text){replace_contraction(text, c
```

Finally, a series of typical transformations have been carried out, such as:

- Eliminating the numbers
- Removing characters and words that define the end of a sentence.
- Removing punctuation symbols
- Removing the necessary sequence of blank characters. That is, between words there is only a single space.
- Stem the document

```
df.train.corpus <- tm_map(df.train.corpus, content_transformer(removeNumbers))
df.train.corpus <- tm_map(df.train.corpus, content_transformer(removeWords), stopwords())
df.train.corpus <- tm_map(df.train.corpus, content_transformer(removePunctuation))
df.train.corpus <- tm_map(df.train.corpus, content_transformer(stripWhitespace))
df.train.corpus <- tm_map(df.train.corpus, content_transformer(stemDocument))

df.test.corpus <- tm_map(df.test.corpus, content_transformer(removeNumbers))
df.test.corpus <- tm_map(df.test.corpus, content_transformer(removeWords), stopwords())
df.test.corpus <- tm_map(df.test.corpus, content_transformer(removePunctuation))
df.test.corpus <- tm_map(df.test.corpus, content_transformer(stripWhitespace))
df.test.corpus <- tm_map(df.test.corpus, content_transformer(stemDocument))
```

To conclude this section we will contrast the transformations made by comparing an original record against a modified one.

```
df.train.corpus.original[['32']][['content']]
```

```
## [1] "@bbcmtd Wholesale Markets ablaze http://t.co/lHYXEOHY6C"
```

```
df.train.corpus[['32']][['content']]
```

```
## [1] "wholesal market ablaz"
```

```
df.test.corpus.original[['25']][['content']]
```

```
## [1] "SETTING MYSELF ABLAZE http://t.co/6vMe7P5XhC"
```

```
df.test.corpus[['25']][['content']]
```

```
## [1] "set ablaz"
```

3. Term Document Matrix

In this third section we will generate the Term Document Matrix (TDM). We will also analyze the most frequent tokens while eliminating the less frequent tokens to remove irrelevant variables that provide (probably) the least information to our problem.

```
tdm <- TermDocumentMatrix(df.train.corpus, control = list(weighting = weightTfIdf))
tdm
```

```
## <<TermDocumentMatrix (terms: 11552, documents: 7613)>>
## Non-/sparse entries: 62545/87882831
## Sparsity           : 100%
## Maximal term length: 49
## Weighting          : term frequency - inverse document frequency (normalized) (tf-idf)
```

First we will eliminate those tokens that are less frequent.

```
tdm <- removeSparseTerms(tdm, 0.99)
tdm
```

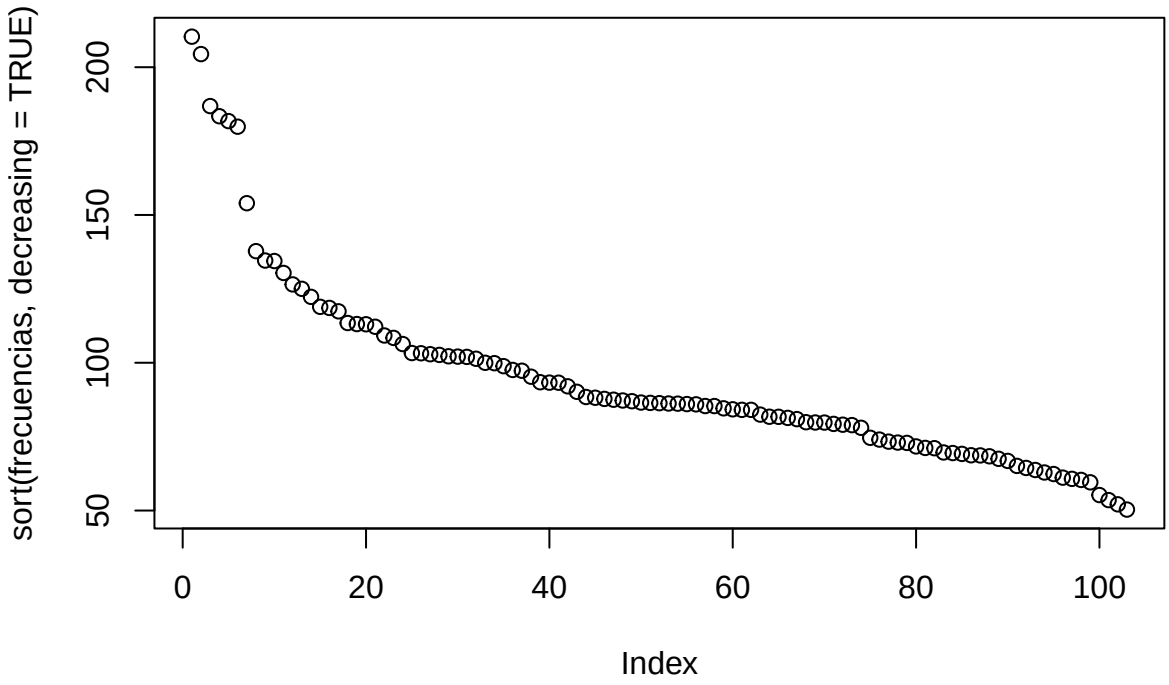
```
## <<TermDocumentMatrix (terms: 103, documents: 7613)>>
## Non-/sparse entries: 13233/770906
## Sparsity           : 98%
## Maximal term length: 10
## Weighting          : term frequency - inverse document frequency (normalized) (tf-idf)
```

```
inspect(tdm)
```

```
## <<TermDocumentMatrix (terms: 103, documents: 7613)>>
## Non-/sparse entries: 13233/770906
## Sparsity           : 98%
## Maximal term length: 10
## Weighting          : term frequency - inverse document frequency (normalized) (tf-idf)
## Sample             :
##         Docs
## Terms    2598 3418 3732 5185 6134 6523 7186 7471 7473 7579
##   amp       0    0    0    0    0    0    0    0    0    0
##   bomb      0    0    0    0    0    0    0    0    0    0
##   fatal     0    0    0    0    0    0    0    0    0    0
##   fire      0    0    0    0    0    0    0    0    0    0
##   get       0    0    0    0    0    0    0    0    0    0
##   just      0    0    0    0    0    0    0    0    0    0
##   like      0    0    0    0    0    0    0    0    0    0
##   now       0    0    0    0    0    0    0    0    0    0
##   scream    0    0    0    0    0    0    0    0    0    0
##   will      0    0    0    0    0    0    0    0    0    0
```

We will also visualize in a scatter plot the frequency of all variables. In this way we can study if there is a big difference between the most frequent variables and the less frequent ones. We can clearly see that there is such a differentiation. There are a handful of these tokens that are much more frequent than the rest, while the rest seem to be repeated with more or less the same frequency.

```
frecuencias <- rowSums(as.matrix(tdm))
plot(sort(frecuencias, decreasing = TRUE))
```



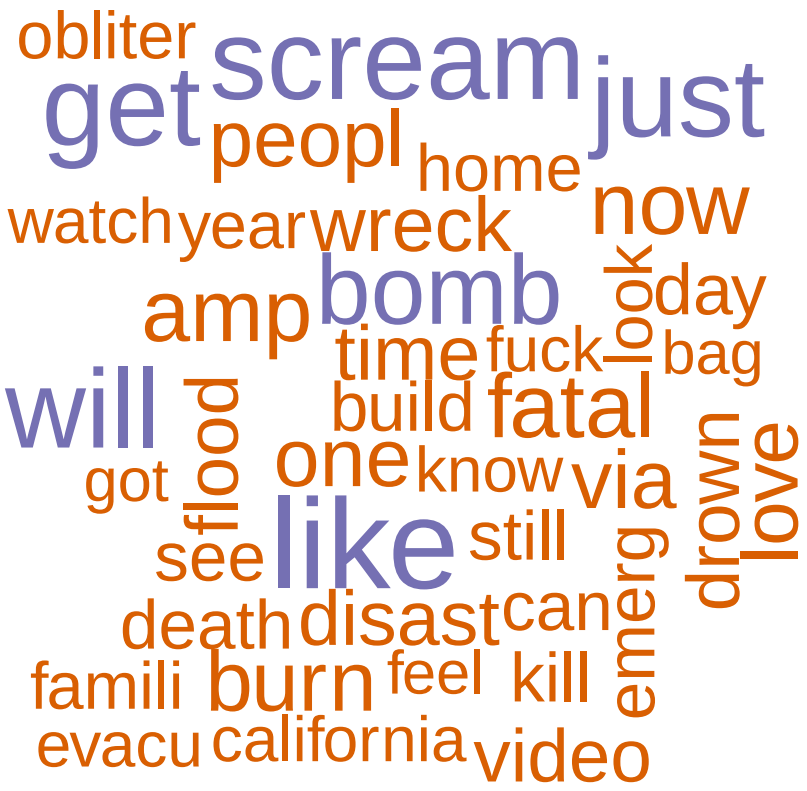
We list these tokens more frequently. We note that we are dealing with words related to natural disasters such as fire, flood or, directly, disaster.

```
tail(sort(frecuencias),n=20)
```

```
##    wreck    crash     time   disast    peopl      new     love      one
## 113.0232 113.0912 113.4464 117.3973 118.5550 118.9228 122.2927 125.0148
##      via     burn      now      amp    fatal     bomb     just     will
## 126.5078 130.3887 134.4261 134.5883 137.7637 153.9875 179.8730 181.7609
##      get   scream     fire     like
## 183.4035 186.8431 204.4429 210.3275
```

We also make use of the word cloud graph to visualize the frequency of these words.

```
freq = data.frame(sort(rowSums(as.matrix(tdm)), decreasing=TRUE))
wordcloud(rownames(freq), freq[,1], max.words=50, colors=brewer.pal(1, "Dark2"))
```



4. Modeling and evaluation

In this final section we will create the models that will be in charge of discerning whether a text corresponds to a natural disaster or not. For this we will make use of the models provided by the `quanteda.textmodels` library:

- SVM Linear
- SVM
- Naive Bayes

This library requires the information to be in a specific format. Therefore, before starting with the models, the corpus will be converted to a document-feature matrix. At the same time that we do this conversion we will generate two data sets (train and test) based on the set from which we know the predictions. Then we will be able to evaluate our models. The division will be 70% for training and the remaining 30% for testing.

```
upper.bound <- round(length(df.train.corpus)*0.7,0)
dfm.train.train <- dfm(corpus(df.train.corpus)[1:upper.bound])
dfm.train.test <- dfm(corpus(df.train.corpus)[upper.bound:length(corpus(df.train.corpus))])
```

First we will use the `textmodel_svm` model. The way to proceed with the models will always be the same. The model will be trained, then the predictions will be generated with the saved data to obtain an evaluation of the model. Finally, we will obtain a series of metrics that will inform us how well our model generalizes.

```
model.svm <- textmodel_svm(dfm.train.train, df.train$target[1:upper.bound])
```

```
predictions.svm <- predict(model.svm, newdata=dfm.train.test)

tab_class <- table(df.train$target[upper.bound:length(corpus(df.train.corpus))], predictions.svm)
confusionMatrix(tab_class, mode = "everything")
```

```
## Confusion Matrix and Statistics
##
##    predictions.svm
##        0    1
##   0 1017  258
##   1  421  589
##
##                Accuracy : 0.7028
##                  95% CI : (0.6836, 0.7215)
##     No Information Rate : 0.6293
##     P-Value [Acc > NIR] : 8.537e-14
##
##                   Kappa : 0.3873
##
##  Mcnemar's Test P-Value : 5.068e-10
##
##             Sensitivity : 0.7072
##             Specificity : 0.6954
##          Pos Pred Value : 0.7976
##          Neg Pred Value : 0.5832
##               Precision : 0.7976
##                  Recall : 0.7072
##                      F1 : 0.7497
##              Prevalence : 0.6293
##          Detection Rate : 0.4451
##    Detection Prevalence : 0.5580
##       Balanced Accuracy : 0.7013
##
##        'Positive' Class : 0
##
```

We continue by analyzing the `textmodel_svmlin` model. The results obtained with this model are worse in all metrics than the previously trained support vector machine.

```
model.svmlin <- textmodel_svmlin(dfm.train.train, df.train$target[1:upper.bound])

predictions.svmlin <- predict(model.svmlin, newdata=dfm.train.test, force = T)

tab_class <- table(df.train$target[upper.bound:length(corpus(df.train.corpus))], predictions.svmlin)
confusionMatrix(tab_class, mode = "everything")
```

```
## Confusion Matrix and Statistics
##
##    predictions.svmlin
##       0   1
##   0 754 521
##   1 378 632
##
```

```
##                Accuracy : 0.6066
##                  95% CI : (0.5862, 0.6267)
##     No Information Rate : 0.5046
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.214
##
##  Mcnemar's Test P-Value : 2.18e-06
##
##             Sensitivity : 0.6661
##             Specificity : 0.5481
##          Pos Pred Value : 0.5914
##          Neg Pred Value : 0.6257
##               Precision : 0.5914
##                  Recall : 0.6661
##                      F1 : 0.6265
##              Prevalence : 0.4954
##          Detection Rate : 0.3300
##    Detection Prevalence : 0.5580
##       Balanced Accuracy : 0.6071
##
##        'Positive' Class : 0
##
```

Finally we tried to train a simpler model such as a Naive Bayes model (`textmodel_nb`).

We observe that it is the model with the best results so far. Moreover, it has the advantage of being a much simpler model than SVMs and it is also interpretable.

```
model.nb <- textmodel_nb(dfm.train.train, df.train$target[1:upper.bound])

predictions.nb <- predict(model.nb, newdata=dfm.train.test, force = T)

tab_class <- table(df.train$target[upper.bound:length(corpus(df.train.corpus))], predictions.nb)
confusionMatrix(tab_class, mode = "everything")
```

```
## Confusion Matrix and Statistics
##
##    predictions.nb
##       0   1
##   0 947 328
##   1 232 778
##
##                Accuracy : 0.7549
##                  95% CI : (0.7367, 0.7724)
##     No Information Rate : 0.516
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.508
##
##  Mcnemar's Test P-Value : 5.958e-05
##
##             Sensitivity : 0.8032
##             Specificity : 0.7034
```

```
##          Pos Pred Value : 0.7427
##          Neg Pred Value : 0.7703
##               Precision : 0.7427
##                  Recall : 0.8032
##                      F1 : 0.7718
##              Prevalence : 0.5160
##          Detection Rate : 0.4144
##    Detection Prevalence : 0.5580
##       Balanced Accuracy : 0.7533
##
##        'Positive' Class : 0
##
```

Finally we are going to use the whole training data set (`df.train`) to train the models again. In this way we will be able to generate the predictions with the data set of which we do not know its classification. After generating these predictions we will be able to upload them to the Kaggle competition from which this information comes from and thus know its accuracy.

```
dfm.train <- dfm(corpus(df.train.corpus))
dfm.test <- dfm(corpus(df.test.corpus))

model.svm <- textmodel_svm(dfm.train, df.train$target)
model.svmlin <- textmodel_svmlin(dfm.train, df.train$target)
model.nb <- textmodel_nb(dfm.train, df.train$target)

predictions.svm <- predict(model.svm, newdata=dfm.test)
predictions.svmlin <- predict(model.svmlin, newdata=dfm.test, force = T)
predictions.nb <- predict(model.nb, newdata=dfm.test, force = T)
```

Once we compute all the predicions we generate a *csv* file to submitt to Kaggle.

```
df.test.svm <- data.frame(
  id = df.test$id,
  target = predictions.svm
)

df.test.svmlin <- data.frame(
  id = df.test$id,
  target = predictions.svmlin
)

df.test.nb <- data.frame(
  id = df.test$id,
  target = predictions.nb
)
```

```
write.csv(df.test.svm,
           "./output/svm.test.csv",
           sep = ",",
           col.names = T,
           row.names = F)

write.csv(df.test.svmlin,
```

```
            "./output/svmlin.test.csv",
            sep = ",",
            col.names = T,
            row.names = F)

write.csv(df.test.nb,
            "./output/nb.test.csv",
            sep = ",",
            col.names = T,
            row.names = F)
```

The accuracy of the submited models are:

- **SVM**: 0.77597
- **SVM Linear**: 0.62212
- **Naive Bayes (nb)**: 0.78915