Topic Modeling on the tweets dataset

David Lorenzo Alfaro

Introduction

The increasingly overwhelming amount of available natural language motivates the pressing need to find efficient and reliable computational techniques capable of processing and analysing this type of data to achieve human-like natural language understanding for a wide range of downstream tasks. Over the last decade, Natural Language Processing (NLP) has seen impressively fast growth, primarily favoured by the increase in computational power and the progress on unsupervised learning in linguistics.

Dealing with unannotated data poses as an unsupervised learning problem, a paradigm within Machine Learning in which no outputs or target variables are provided, i.e., the model is only trained with inputs x, $D = \{x_i\}_{i=1}^{N}$, with the goal being to discover or extract patterns present in data. Often referred to as knowledge discover in the literature, this class of problem is much less defined and since it lacks from ground truth, it is not trivial to find efficient metrics to use for the model to learn about relationships in data and to assess goodness of the identified groups.

In this assignment, we use Latent Dirichlet Allocation (LDA) (Blei et al., 2003), a generative probabilistic model for collections of discrete data such as text corpora in which each item is modelled as a finite mixture over an underlying set of topics. In this context, we attempt to obtain meaningful representations of documents according to the topic probabilities yielded by a LDA model.

Problem description

For this work we will use a dataset composed of 1.6 million tweets, first introduced (Go et al., 2009). Originally proposed for sentiment classification, in this assignment we are mainly concerned with the text of the messages and not in the label, from which the most prominent topics will be captured via LDA.

Import all dependencies

Check whether all necessary packages are installed, and install those that are not automatically from the `CRAN` repository.

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

```
## Loading required package: Matrix
```

```
##
## Attaching package: 'textmineR'
```

```
## The following object is masked from 'package:Matrix':
##
##     update
```

```
## The following object is masked from 'package:stats':
##
##     update

## Package version: 3.2.0
## Unicode version: 13.0
## ICU version: 69.1

## Parallel computing: 8 of 8 threads used.

## See https://quanteda.io for tutorials and examples.
```

Packages `quanteda` and `spacyr` are used for preprocessing purposes, `textmineR` to induce a topic model from data using LDA and to perform queries at inference time; packages `readr` and `rlist` to serialize information allowing to load (store) data from disk to provide with faster execution times by avoiding repeating calculations; and `plotly`, used to generate interactive graphs to visualize topic assignations for words and documents.

```
library(textmineR)
library(quanteda)
library(readr)
library(rlist)
library("spacyr")
source("utils.R")
```

```
## Loading required package: ggplot2

##
## Attaching package: 'plotly'

## The following object is masked from 'package:ggplot2':
##
##     last_plot

## The following object is masked from 'package:stats':
##
##     filter

## The following object is masked from 'package:graphics':
##
##     layout
```

Load data

Let us first define some variables to handle data directories:

- `data_dir` holds the relative path where all data is stored.
- `serial_dir` is the relative path where all serialized data is stored, allowing for faster execution by eschewing recomputing some key components.

```
data_dir <- "data\\"
serial_dir <- paste0(data_dir, "serialized\\")
```

Next, we define the size of the training and test sets, composed of $2 \mathring{u} 10^5$ and 10^5 instances, respectively, where an instance is a preprocessed tweet (i.e., not in raw form).

```
train_size <- 200000
test_size <- 100000
```

Similarly, in order to grant reproducibility of all experiments hereby conducted we set a seed. This will induce deterministic execution of directives using randomness (e.g., the `sample` function).

```
seed <- 0
```

Let us generate the subsequent output filenames for both the training and test set. As aforementioned, this will come handy to avoid repeating preprocessing steps that should provide a deterministic execution, due to the static nature of the data.

```
out_file_train <- paste0(data_dir, "tweets-train-",floor(train_size/1000),"k-prep.txt")
out_file_test <- paste0(data_dir, "tweets-test-",floor(test_size/1000),"k-prep.txt")
print(out_file_train)
```

```
## [1] "data\\tweets-train-200k-prep.txt"
```

```
print(out_file_test)
```

```
## [1] "data\\tweets-test-100k-prep.txt"
```

In order to generate the training and test datasets in an unbiased manner:

1. Load the whole dataset and obtain a (random) permutation. Randomness can be controlled via a specific random state.
2. Obtain the training dataset, composed of `train_size` instances, denoted $data_{training} = \{x_1, ..., x_i, ..., x_{train\_size}\}$
3. Obtain the test dataset, composed of `test_size` instances, denoted $data_{test} = \{x_1, ..., x_i, ..., x_{test\_size}\}$
4. In order to leverage memory management, remove the `df.tweets` object (no longer necessary).

```
input_file <- paste0(data_dir,"tweets.csv")
set.seed(seed)
df.tweets <- sample(read.csv(input_file, header=FALSE)$V6)
df.tweets.train <- df.tweets[1:train_size]
df.tweets.test <- df.tweets[train_size+1: (test_size+1)]

# Liberate memory space
rm(df.tweets)

head(df.tweets.train, 10)
```

```
##  [1] "Will never cease to be amazed that there is a giant beautiful park in the middle of a giant bea
##  [2] "@EllieBeck Actually I was on a laptop at the time, have both PC and Laptop but no Mac  Want one
##  [3] "@shinobistalin so snakes are incognito!!!!!!!!!!!!!! that niggah is gonna bite coco "
##  [4] "90210... tut mich sorry "
##  [5] "is enjoying House. rah rah rah "
##  [6] "I love my hairdresser Sylvia at Actpoint salon - shaw towers. I finally found someone who can l
##  [7] "@NattyKnits Re Ghost - pretty good but agree rounder would be better. Just slightly condom-like
##  [8] "@JamieLynnWright: what happened? "
##  [9] "sunny !!!  im happy he won and he's so adorable."
## [10] "people I am saddened as I just learned Facebook found the story of Alex's lemonade stand offens
```

Preprocessing

Tweets are normally written in an informal register Thus, the data preparation phase is arguably the most challenging one, entailing standardization of the language, dealing with slang words and phrases and wrong grammar and spelling, to name a few. In this work we propose a rather naive workflow to deal with the heterogeneous nature of the data, albeit benefiting from further preprocessing could potentially enhance performance of the subsequent models.

Tokenization

Preprocessing is largely conducted at word level. We utilize `quanteda` to *tokenize* tweets, i.e., decomposing an arbitrarily long sentence or text into its constituent words and/or symbols.

First, we define a corpus from the original training set. Note that we impose texts to be encoded in ASCII, removing non-ASCII characters like those refering to emojis or non-English graphemes.

Next, sentences are broken down into tokens. Illegal tokens will be filtered out according to the following criteria:

1. `remove_punct`: removes punctuation characters, e.g., "!", ".", ";".
2. `remove_url`: eliminates URLs beginning with http(s)
3. `remove_symbols`: removes special symbols.
4. `remove_numbers`: deletes tokens that consist only of numbers, but not words that start with digits, e.g., `2day`.
5. `tokens_tolower()`: convert all tokens to lower case.
6. Remove English stop words, yielded by invoking the `stopwords("en")` function, where "en" stands for the language code.

Furthermore, we delete the ampersand (&) and quot symbols, and define some regular expressions to delete usernames (`@\\S+`), and URLs starting with "www" (`www\\.\\S+`)

Note: the `quanteda` library provides a directive which remove Twitter characters `@` and `#`, denoted `remove_twitter`. We do not use it here because we deal with such symbols in a more meaningful fashion.

```r
df.corpus <- iconv(corpus(df.tweets.train), from = "UTF-8", to = "ASCII", sub = "")

df.tokens <- tokens(df.corpus,
                    remove_punct = TRUE,
                    remove_url = TRUE,
                    remove_symbols = TRUE,
                    remove_numbers = TRUE
                    ) %>% tokens_tolower(
                    ) %>% tokens_select(
                      pattern = stopwords("en"), selection = "remove"
                    ) %>% tokens_select(
                      pattern = "@\\S+", selection="remove", valuetype ="regex"
                    ) %>% tokens_select(
                      pattern = "www\\.\\S+", selection="remove", valuetype ="regex"
                    ) %>% tokens_select(
                      pattern = "amp", selection="remove"
                    ) %>% tokens_select(
                      pattern = "quot", selection="remove"
                    )

head(df.tokens, 5)
```

```
## Tokens consisting of 5 documents.
## text1 :
##  [1] "never"     "cease"     "amazed"    "giant"     "beautiful" "park"
##  [7] "middle"    "giant"     "beautiful" "city"
##
## text2 :
##  [1] "actually" "laptop"   "time"     "pc"       "laptop"   "mac"
##  [7] "want"     "one"      "artyness" "though"
##
## text3 :
## [1] "snakes"    "incognito" "niggah"    "gonna"     "bite"      "coco"
##
## text4 :
## [1] "tut"   "mich"  "sorry"
##
## text5 :
## [1] "enjoying" "house"    "rah"      "rah"      "rah"
```

SpaCy is an open source and state-of-the-art performant library that features what they call *linguistically-motivated tokenization*. In this work, we aim at finding topics on tweets written in English. We chose the `en_core_web_sm` language model, which provides simple and fast preprocessing pipelines, with nearly negligible accuracy differences with respect to the so-called *large* model, `en_core_web_trf`.

```r
spacy_initialize(model = "en_core_web_sm")
```

```
## Found 'spacy_condaenv'. spacyr will use this environment

## successfully initialized (spaCy Version: 3.1.3, language model: en_core_web_sm)

## (python options: type = "condaenv", value = "spacy_condaenv")
```

The `spacy_parse` function calls spaCy to both tokenize and tag the texts, and returns a `data.table` of the results. Notice that, prior to feeding the parser with inputs, we assemble tokens into texts. We found by empirical experimentation that parsing text in this manner provides faster computations, probably because fewer invocations to the parser are required.

According to the package documentation, while running spaCy on Python through R, a Python process is always running in the background and `Rsession` will take up a lot of memory (typically over 1.5GB). In order to free up the memory allocated to such process, we call the `spacy_finalize` function.

```
df.spacy <- lapply(df.tokens, reduce) %>% unlist(use.names = F) %>% spacy_parse
spacy_finalize()
```

```
head(df.spacy, 10)
```

```
##    doc_id sentence_id token_id     token     lemma  pos entity
## 1   text1           1        1     never     never  ADV
## 2   text1           1        2     cease     cease VERB
## 3   text1           1        3    amazed    amazed  ADJ
## 4   text1           1        4     giant     giant  ADJ
## 5   text1           1        5 beautiful beautiful  ADJ
## 6   text1           1        6      park      park NOUN
## 7   text1           1        7    middle    middle  ADJ
## 8   text1           1        8     giant     giant  ADJ
## 9   text1           1        9 beautiful beautiful  ADJ
## 10  text1           1       10      city      city NOUN
```

Lemmatisation in linguistics refers to the process of gathering altogether the inflected forms of a word so they can be analysed as a single item, identified by the word's lemma, or dictionary form. In order to reduce the number of tokens effectively, we dump all meanings of formal verbs and nouns into their dictionary form.

```
df.processed <- aggregate(df.spacy$lemma, list(df.spacy$doc_id), FUN=reduce)$x

head(df.processed, 10)
```

```
##  [1] "never cease amazed giant beautiful park middle giant beautiful city"
##  [2] "people saddened just learn facebook find story alex 's lemonade stand offense block link"
##  [3] "fall asleep gon na watch hannah montana till get tired love miley"
##  [4] "damn think make"
##  [5] "foot hurt"
##  [6] "nope never ever hear look I m unknown fun"
##  [7] "home sick sick little sick just way wanna spend last week work kick"
##  [8] "lol anymore I m still sick suck ass"
##  [9] "oh thank"
## [10] "beautiful prince oh goodness"
```

Preprocessing the input is a costly process, even for a rather small dataset. The most straightforward to alleviate computational needs, one can serialize information, allowing for loading (or storing) data from disk whilst maintaining all meaningful info. In this case, it can be convenient to keep the processed tweets in a human-readable format, hence we simply sink the tweets into a `txt` file using the `write_lines` function of the `readr` package. Similarly, to load data from, we use the `read.delim` function.

```
if (file.exists(out_file_train)){
  df.processed <- read.delim(out_file_train, header=F)$V1
} else {
  write_lines(df.processed, out_file_train)
}
```

Further, the `utils` model features some functions that enables applying this very processing pipeline. Namely, they are the `preprocessing` and the `tokenize` directives. As the name suggests, the former covers all the preprocessing process, whereas the later is solely concerned with the tokenization process. Refer to the code for further details, should you require it.

Model induction via LDA.

Latent Dirichlet allocation constitutes one of the main approaches for topic modeling. It is roughly based on these two principles:

- *Every document is a mixture of topics.* A document may contain words belonging for several topics. LDA assigns documents to topics in a probabilistic fashion, i.e., membership to topics is inferred from the likelihood estimates.
- *Every topic is a mixture of words.* A topic can be characterized by a set of words. For example, given the topic of "entertainment", one could think of words such as "music", "singer", "actress", "movie", etc. A word may belong to more than one topic.

This allows documents to interact with each other, i.e., they can share semantics, rather than being segmented into mutually-exclusive groups (e.g., a binary classification problem), in a way that resembles the organic use of natural language.

Term co-ocurrence matrix.

A term co-ocurrence matrix (TCM) is a square matrix whereby rows and columns are characterized by each of the tokens in the vocabulary of the corpus. There are to main approaches:

1. **term-context matrix**, where the whole document serves as a context. If two terms occur in the same context, they are said to have occurred in the same occurrence context.
2. **k-skip-n-gram** approach, where a sliding window will include the (k+n) words, both to the left and the right. This window will serve as the context now. Terms that co-occur within this context are said to have co-occurred.

Because length of the tweets are bounded, we chose the first approach, which slightly reduces the overhead introduced by the windowing functions. This is a particular case of the skip gram model, where the size of the window equals the size of the document.

The TCM is based on the rationale that terms that appear conjointly across the corpus are prone to be semantically related. Of course, removal of stop words avoids appearance of spurious correlations that may arise. Similarly, in order to better grasp those relations (and to reduce complexity of the matrix), the lemmatisation dumps all forms of a verb/noun into its dictionary form.

As the cardinality of the vocabulary increases, correlations between words tend to be more marginal, in the sense that out of all the words, only a small proportion of them are meaningfully related. Therefore, sparse representations are more suitable (the equivalent dense representations have a quadratic order of

space complexity). In this problem, we have a vocabulary size of 68984 tokens, which leads to a TCM with 4,758,792,256 entries. As for the preprocessed dataset, we serialize it to reduce the massive computational overhead.

```
tcm_filename <- paste0(serial_dir, "tcm_tweets.dat")

if (file.exists(tcm_filename)){
  tcm <- list.unserialize(tcm_filename)
} else{
  tcm <- CreateTcm(doc_vec = df.processed,
                   #skipgram_window = 3,
                   verbose = FALSE,
                   cpus = 4)
  list.serialize(tcm, tcm_filename)
}
```

Fitting the model.

Once we have a TCM, we can use the same procedure to make an embedding model. It is worth noting that this process is very costly because of dimensionality of the matrix. We fit a LDA model from the data with the following hyperparameters:

- `dtm` : document term matrix or term co-occurrence matrix of class dgCMatrix.
- `iterations` (200): Integer number of iterations for the *Gibbs* sampler to run.
- `burnin` (): Integer number of burnin iterations. If `burnin` is greater than -1, the resulting "phi" and "theta" matrices are an average over all iterations greater than `burnin`
- `alpha` (): Vector of length `k` for asymmetric or a number for symmetric. This is the prior for topics $T = \{t_1, ..., t_n\}$ over documents $D = \{d_1, \ldots, d_n\}$, i.e., $\{P(t_i|d_i) \forall t_i \in T \wedge d_i \in D\}$.
- `beta` (): Vector of length `ncol(dtm)` for asymmetric or a number for symmetric. This is the prior for words $W = \{w_1, ..., w_n\}$ over topics $T = \{t_1, ..., t_n\}$, i.e., $\{P(w_i|T)|w_i \in W\}$.
- `optimize_alpha` (`True`): adjust α every 10 *Gibbs* iterations.
- `calc_coherence` (`True`): calculate probabilistic coherence of topics after the model is trained.
- `calc_r2` (`True`): calculate R-squared after the model is trained.
- `cpus` (4): number of CPUs to use to compute the model.

```
k_topics <- 20
embeddings_filename <- paste0(serial_dir, "lda_model_k", k_topics, ".dat")

if (file.exists(embeddings_filename)){
  embeddings <- list.unserialize(embeddings_filename)
} else{
  print(embeddings_filename)
  embeddings <- FitLdaModel(dtm = tcm,
                            k = k_topics,
                            iterations = 200,
                            burnin = 175,
                            alpha = 0.1,
```

```
                            beta = 0.05,
                            optimize_alpha = TRUE,
                            calc_coherence = TRUE,
                            calc_r2 = TRUE,
                            cpus = 4)

  list.serialize(embeddings, embeddings_filename)
}
```

R-squared (R^2) comes handy to get an estimate of the overall goodness of fit of the model. A possible interpretation of this metric is as the amount of variance explained by the model respect to the original term co-ocurrence matrix. A property verified by R-squared is that $0 \leq R^2 \leq 1$, where $R^2 = 1$ denote a perfect fit, hence values closer to 1 are preferrable. In this context, a very low value of R^2 may reveal (1) deficiencies in the training process, e.g., because of bad election of the model hyperparametrization, mainly due to suboptimal number of the topics to identify or the number of iterations to perform; or (2) absence of correlations in data, which does not allow for pattern identification.

The yielded value for R-squared is $R^2 = 0.304$, which far from being ideal, largely enhances the one obtained without any preprocessing (~ 0.113). It is worth noting that, in this scenario, the *optimal* number of topics, denoted k, is likely to be larger than 20 (remember that the cardinality, denoted $|\mathring{u}|$, of the training set is $|data_{training}| = 2\mathring{u}10^5$). That notwithstanding, large values for k can derive in computationally intractable problems and increase the complexity to interpret the topics.

```
# Get an R-squared for general goodness of fit
embeddings$r2
```

```
## [1] 0.3044259
```

Now, let us retrieve the top k most representative terms of each topic, using the `GetTopTerms` function, which takes as input the matrix containing the likelihood for words to belong to each of the topics (`embeddings$phi`) and the number of top terms to extract, `top_k_terms`.

Then, we create a summary including the name of the topics and the prevalence, coherence and top k terms of each of the topics.

```
top_k_terms = 5
# Get top terms, no labels we only have unigrams (words treated as a unique entity).
embeddings$top_terms <- GetTopTerms(phi = embeddings$phi,
                                    M = top_k_terms)


embeddings$summary <- data.frame(
  topic = rownames(embeddings$phi),
  prevalence = round(colSums(embeddings$theta), 3),
  coherence = round(embeddings$coherence, 3),
  top_terms = apply(embeddings$top_terms, 2, paste, collapse=", "),
  stringsAsFactors = FALSE
  )

# alternatively, you can use the ldaSummary function:
# embeddings$summary <- ldaSummary(embeddings)
```

Let us have a look to the summary, focusing on the topics ordered by prevalence, which measures the density of tokens along each embedding dimension (i.e., each topic).

```
printSummaryTable("prevalence")
```

```
##      topic prevalence coherence                          top_terms
## t_14  t_14   4149.243     0.305           lol, haha, yeah, nt, ur
## t_6    t_6   4092.678     0.251  work, find, iphone, phone, update
## t_3    t_3   3996.424     0.428       hurt, feel, hair, bad, back
## t_19  t_19   3814.527     0.296    make, thing, people, life, lot
## t_4    t_4   3728.096     0.301 twitter, follow, tweet, read, send
## t_18  t_18   3707.799     0.322     eat, make, food, good, dinner
## t_13  t_13   3700.561     0.172             hehe, la, sa, ko, de
## t_10  t_10   3538.090     0.292      love, song, lt, hey, awesome
## t_1    t_1   3523.064     0.263       day, today, nice, work, rain
## t_5    t_5   3468.122     0.352     miss, friend, fun, girl, love
## t_20  t_20   3462.242     0.345     back, home, week, work, leave
## t_8    t_8   3382.311     0.358   show, tonight, live, meet, night
## t_17  t_17   3303.145     0.320       watch, movie, sad, love, tv
## t_16  t_16   3248.841     0.367       car, house, break, run, sit
## t_9    t_9   3224.035     0.294     work, night, time, sleep, bed
## t_7    t_7   3178.196     0.299     feel, bad, school, hate, sick
## t_12  t_12   3070.464     0.259        buy, day, year, pay, money
## t_11  t_11   2956.934     0.359    good, hope, great, morning, day
## t_15  t_15   2720.268     0.357         na, gon, play, game, lose
## t_2    t_2   2717.960     0.297     nt, ca, wait, happy, birthday
```

Similarly, we can order the table by topic coherence.

```
printSummaryTable("coherence")
```

```
##      topic prevalence coherence                          top_terms
## t_3    t_3   3996.424     0.428       hurt, feel, hair, bad, back
## t_16  t_16   3248.841     0.367       car, house, break, run, sit
## t_11  t_11   2956.934     0.359    good, hope, great, morning, day
## t_8    t_8   3382.311     0.358   show, tonight, live, meet, night
## t_15  t_15   2720.268     0.357         na, gon, play, game, lose
## t_5    t_5   3468.122     0.352     miss, friend, fun, girl, love
## t_20  t_20   3462.242     0.345     back, home, week, work, leave
## t_18  t_18   3707.799     0.322     eat, make, food, good, dinner
## t_17  t_17   3303.145     0.320       watch, movie, sad, love, tv
## t_14  t_14   4149.243     0.305           lol, haha, yeah, nt, ur
## t_4    t_4   3728.096     0.301 twitter, follow, tweet, read, send
## t_7    t_7   3178.196     0.299     feel, bad, school, hate, sick
## t_2    t_2   2717.960     0.297     nt, ca, wait, happy, birthday
## t_19  t_19   3814.527     0.296    make, thing, people, life, lot
## t_9    t_9   3224.035     0.294     work, night, time, sleep, bed
## t_10  t_10   3538.090     0.292      love, song, lt, hey, awesome
## t_1    t_1   3523.064     0.263       day, today, nice, work, rain
## t_12  t_12   3070.464     0.259        buy, day, year, pay, money
## t_6    t_6   4092.678     0.251  work, find, iphone, phone, update
## t_13  t_13   3700.561     0.172             hehe, la, sa, ko, de
```

Global predictions

After inspecting the model, a naive method to test whether the topics are able to convey the semantic information described by the top terms is by asking the model to classify other related terms. A constraint to this approach is that the *query* terms should belong to the vocabulary of the model, similar to all approaches treating words as "indices", rather than a set of features (semantic-driven embeddings).

To that end, let us first predict the membership likelyhood to each of the topics for all words in the vocabulary.

```
word_predictions_filename = paste0(serial_dir, "word_predictions_k", k_topics, ".dat")

# predict on held-out documents using gibbs sampling "fold in"
df.processed.predictions <- predict_word(embeddings,
                                         tcm,
                                         out=word_predictions_filename,
                                         method = "gibbs",
                                         iterations = 200,
                                         burnin = 175)
```

Afterwards, let us test some words. This is the behaviour that we expect from the model:

- Word `college` can be semantically related to topic 7 (feel, bad, school, hate, sick), and to a lesser extent to topic 20 (back, home, week, work, leave) and topic 9 (work, night, time, sleep, bed)
- Word `tired` can be semantically associated to topic 9 (work, night, time, sleep, bed).
- Word `lunch` can be linked to topic 18 (eat, make, food, good, dinner).
- Word `android` can be semantically associated to topic 6 (work, find, iphone, phone, update).
- Word `disease` can be related to topic 3 (hurt, feel, hair, bad, back).

```
plot_words2topic(c("college", "tired", "lunch", "android", "disease"),
                 df.processed.predictions)
```

As it can be seen from the graphs, all our hypothesis are satisfied, i.e., the model is able to leverage relations among words and topics based on the frequency of appearance in the same context.

Working on the test dataset: inference.

In this section, we embed documents under the LDA model to obtain topics of **local** contexts, rather than global ones.

First, let us first preprocess the test set.

```
if (file.exists(out_file_test)){
  df.processed.test <- read.delim(out_file_test, header=F)$V1
} else {
  df.processed.test <- preprocess(df.tweets.test)
  write_lines(df.processed.test, out_file_test)
}
```

Then, let's check whether the tweets seem to be properly preprocessed.

```
set.seed(seed)
sample(df.processed.test, 6)
```

```
## [1] "hear radio jon kate r gon na tell everyone r get divorce monday 's show"
## [2] "watch home alone tehhe"
## [3] "one send either nice surprise see molo search huh true fan"
## [4] "pianooo lesson soon how s everyone afternoon"
## [5] "shame job think work big money pay"
## [6] "nice meet ya"
```

In this case, we are interested in modelling each document as a mixture of topics. To examine the per-document-per-topic probabilities, we first obtain a document-term matrix, which describes the frequency of terms that occur in a collection of documents, i.e., a standard bag-of-words representation.

```
dtm.test <- getDtm(df.processed.test)
```

Then, we project the documents into the embedding space by predicting on held-out documents, using *Gibbs* sampling.

```
doc_predictions_filename <- paste0(serial_dir, "doc_predictions_k", k_topics, ".dat")

df.processed.test.predictions <- predict_doc(embeddings,
                                             dtm=dtm.test,
                                             out=doc_predictions_filename
                                             )
```

Now, let us get four random documents from the test set containing the word "college". Notice that the documents shown here are the preprocessed tweets, not the original, and thus they may not be very interpretable.

```
word_search<-"college"
search_sample <- 4

set.seed(seed)
doc.search <- df.processed.test[
                lapply(df.processed.test,
                       function(x){grepl(word_search, x, fixed=T)}
                       ) %>% unlist(use.names=F)
                ] %>% sample(search_sample)

doc.search
```

```
## [1] "since three month old go college now back year now"
## [2] "sit home reply hundred message multiply good news college english class final"
## [3] "wooo job interview awesome opportunity heart set travel summer need money college"
## [4] "back college least make whether work going do another matter"
```

What is the distribution of the documents respect to the topics? Are they any meaningful?

```
plot_doc2topic(doc.search, predictions=df.processed.test.predictions)
```

As it can be seen, most of the documents seem to be highly related to topic 20, characterized by words back, home, week, work and leave. We argue the model is capable of grasping the overall meaning of the documents, referring to going back to college (preprocessed tweet: since three month old go college now back year now), doing work for college ("back college least make whether work going do another matter") and having a job interview to afford college expenses ("wooo job interview awesome opportunity heart set travel summer need money college").

On the other hand, the model reveals tweet "sit home reply hundred message multiply good news college english class final" is related with topics 7 (feel, bad, school, hate, sick) and topic 4 (twitter, follow, tweet, read, send). Only the latter seems to make sense, as it includes related words such as "reply" or "message".

Now, let us retrieve some random documents

```
set.seed(16)
doc_random_search <- sample(df.processed.test, search_sample)
plot_doc2topic(doc_random_search, predictions=df.processed.test.predictions)
```

The model is able to capture salient information about the documents, whenever they can be somehow linked to a topic, whereas for those that cannot be associated clearly to any topic the per-topic probability estimates are uniformly distributed.

When querying for tweets containing "Harry Potter", the model identifies the main topic is topic 17 (watch, movie, sad, love, tv).

```
topic_search(df.processed.test, "harry potter", df.processed.test.predictions, seed=seed)
```

When querying for documents containing the word purchase, results are again meaningful:

- The first tweet contains the word "wait", related with topic 2 (nt, ca, wait, happy, birthday) and the word "home", related with topic 20 (back, home, week, work, leave)
- The second tweet contains "browse" related with topic 6 (work, find, iphone, phone, update) and "purchase", with topic 12 (buy, day, year, pay, money).
- The third tweet contains words "purchase" and "buy", associated with topic 12.
- The fourth tweet contains the word "page", related with topic 6, and words "store", "purchase" and "checkout", related with topic 12.

```
topic_search(df.processed.test, "purchase", df.processed.test.predictions, seed=seed)
```

References

Blei, D. M., Ng, A. Y., & Edu, J. B. (2003). Latent Dirichlet Allocation. *Journal of Machine Learning Research, 3*(Jan), 993–1022.

Go, A., Bhayani, R., & Huang, L. (2009). Twitter Sentiment Classification using Distant Supervision. *CS224N Project Report, Stanford, 1*(2009), 1–12.

Annex I. Session information.

```
sessionInfo()
```

```
## R version 4.1.2 (2021-11-01)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 10 x64 (build 19044)
##
## Matrix products: default
##
## locale:
## [1] LC_COLLATE=Spanish_Spain.1252  LC_CTYPE=Spanish_Spain.1252
## [3] LC_MONETARY=Spanish_Spain.1252 LC_NUMERIC=C
## [5] LC_TIME=Spanish_Spain.1252
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] plotly_4.10.0   ggplot2_3.3.5   spacyr_1.2.1    rlist_0.4.6.2
## [5] readr_2.1.1     quanteda_3.2.0  textmineR_3.0.5 Matrix_1.4-0
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_1.0.8             here_1.0.1             lattice_0.20-45
##  [4] tidyr_1.1.4            png_0.1-7              RhpcBLASctl_0.21-247.1
##  [7] assertthat_0.2.1       rprojroot_2.0.2        digest_0.6.29
## [10] utf8_1.2.2             R6_2.5.1               evaluate_0.14
## [13] httr_1.4.2             pillar_1.6.4           rlang_0.4.12
## [16] lazyeval_0.2.2         data.table_1.14.2      reticulate_1.23
## [19] rmarkdown_2.11         stringr_1.4.0          RcppProgress_0.4.2
## [22] htmlwidgets_1.5.4      munsell_0.5.0          compiler_4.1.2
## [25] xfun_0.29              pkgconfig_2.0.3        htmltools_0.5.2
## [28] tidyselect_1.1.1       tibble_3.1.6           lgr_0.4.3
## [31] fansi_0.5.0            viridisLite_0.4.0      crayon_1.4.2
## [34] dplyr_1.0.7            tzdb_0.2.0             withr_2.4.3
## [37] rappdirs_0.3.3         grid_4.1.2             jsonlite_1.7.2
## [40] gtable_0.3.0           lifecycle_1.0.1        DBI_1.1.2
## [43] magrittr_2.0.1         scales_1.1.1           RcppParallel_5.1.5
## [46] stringi_1.7.6          ellipsis_0.3.2         stopwords_2.3
## [49] generics_0.1.1         vctrs_0.3.8            fastmatch_1.1-3
## [52] tools_4.1.2            float_0.2-6            glue_1.6.0
## [55] mlapi_0.1.0            purrr_0.3.4            hms_1.1.1
## [58] crosstalk_1.2.0        text2vec_0.6           parallel_4.1.2
## [61] fastmap_1.1.0          yaml_2.2.2             colorspace_2.0-2
## [64] rsparse_0.5.0          knitr_1.37
```