Text Mining Foundations

Motivation

What is “text”?

Motivation

What is “text”?

Can be written or spoken.

Here we assume text is written and available in a suitably machine-readable format (more on this later).

So no spoken only with no transcripts.

Motivation

What can we already do with text?

Motivation

What can we already do with text?

We already mastered basic text/string functionality including rexexps.

To some extent, can search for patterns or search for similar texts. (Did you ever hear about “fuzzy matching”?)

Motivation

What do we want to do with text?

Motivation

What do we want to do with text?

A. Search for patterns and similar texts

B. Group texts according to their similarity

C. Use text to improve predictive modeling (i.e., use text as predictor)

D. Model differences in text (i.e., use text as response)

Task A is a problem in information retrieval.

Task B is a unsupervised learning problem (clustering).

Tasks C and D are supervised learning problems (with text as predictor or response).

In finance, focus is on task C: e.g., use sentiment when predicting returns.

How can we deal with text?

All tasks face the following fundamental challenge: how should we represent text?

Usually, cannot take text “as is”, i.e., as nomimal factors: then different texts would be different, with no notion of similarity.

Maybe split into their individual characters (or bytes)?

Gives sequences of character/bytes with varying lengths.

Usually, we need sequences with **fixed** lengths.

Maybe truncate (or pad with missings)?

Maybe there are ways of getting fixed length representations of such sequences? This is actually a good idea, see later.

How can we deal with text?

Maybe simply tabulate? Well, what can you expect from working with the character frequencies?

Actually, one can use these to detect the language of the texts, based on the frequencies of char/byte **trigrams** (i.e., triples of consecutive chars/bytes). See CRAN package textcat.

But otherwise these frequencies are not a useful representation: they do not allow to capture the “meaning”.

What do we usually do with text?

For most purposes, the individual characters are not the right building blocks. Natural language texts are based on “words” which make up “sentences” and perhaps “paragraphs” etc.

Maybe split texts into words (or words within sentences, . . .)?

This is a better idea. However:

- not so easy
- not so clear what this precisely means
- still gives sequences of words of varying lengths.

Words

Consider the following example:

```
R> (s1 <- "This is some text.")
```

```
[1] "This is some text."
```

What are the words? And how can we get them?

The simplest idea is splitting into syntactic **tokens** by splitting on whitespace: a whitespace **tokenizer**.

DIY:

```
R> unlist(strsplit(s1, "[[:space:]]+"))
```

```
[1] "This"  "is"    "some"  "text."
```

Alternatively,

```
R> scan(text = s1, what = character())
```

```
[1] "This"  "is"    "some"  "text."
```

Not so good: the sentence end period should not be part of 'text'.

Words

Maybe we should first use a sentence tokenizer (“split into sentences”)?

Well, this could take care of sentence end ‘.’, ‘!’ or ‘?’.

But what about

```
R> (s2 <- "You are late, she said.")
```

```
[1] "You are late, she said."
```

Maybe we should split on whitespace or punctuation characters?

```
R> unlist(strsplit(s2, "[[:space:][:punct:]]+"))
```

```
[1] "You"  "are"  "late" "she"  "said"
```

Or equivalently, remove all punctuation characters before splitting on whitespace?

Words

Consider

Financial returns usually have heavy-tailed distributions.

Clearly, reducing to ‘heavytailed’ is wrong: this is not a word.

But what about ‘heavy-tailed’: one word or two words?

And while we are at this, consider

distribution function

New York City

In German, the former would be

Verteilungsfunktion

so clearly one word (“compound noun”). In English, this is done differently, but then there are also ‘football’, ‘railway’, . . .).

See, e.g., `https://en.wikipedia.org/wiki/Compound_(linguistics)`.

Suggests that actual “words” may be syntactic word bigrams or trigrams (or other *n*-grams).

Words and lemmas

And finally,

```
R> (s3 <- "You are late, she said.  He thinks differently.")
```

```
[1] "You are late, she said.  He thinks differently."
```

where we are mostly happy with

```
R> unlist(strsplit(s3, "[[:space:][:punct:]]+"))
```

```
[1] "You"          "are"          "late"         "she"          "said"
[6] "He"           "thinks"       "differently"
```

In the above text,

are said thinks

are derived forms with respective base forms (**lemmas**) given, resptively, by

be say think

Some applications work better when using the lemmas. But even in English having a good **lemmatizer** is not so easy.

Words and stems

A poor person's variant is **stemming**:

> *In linguistic morphology and information retrieval, stemming is the process of reducing inflected (or sometimes derived) words to their word stem, base or root form . . .*

(`https://en.wikipedia.org/wiki/Stemming`).

For English, this "often" actually gives the lemma:

```
R> tm::stemDocument(c("says", "said", "are", "shoes", "happily"))
```

```
[1] "say"     "said"    "are"     "shoe"    "happili"
```

But clearly the last is not what we would have expected (and presumably also not what we really want).

Disclaimer

In what follows we will typically write calls to a function `fun()` in a non-default package **pkg** as

```
pkg::fun(ARGS)
```

to make clear which package the function comes from.

For production code in packages, you would selectively import these functions into your namespace, and then calls without the `pkg::` prefix.

Words

More food for thought:

This can store 100G of data.

They paid USD 12.34 per share.

So we also need to think about digits and numbers (hello, QFin!).

In

```
R> unlist(strsplit(s3, "[[:space:][:punct:]]+"))
```

```
[1] "You"         "are"         "late"        "she"         "said"
[6] "He"          "thinks"      "differently"
```

clearly 'You' are 'He' are no different from 'you' and 'he'.

Maybe map everything to lower case?

Most languages have **multi-word tokens** (not English): e.g.,

German: zum → zu dem, French: du → de le, . . .

Words

To sum up: even the basic task of splitting texts into words is far from straightforward.

Even if the texts are written in a grammatically simple language such as English.

Of course, we can use prebuilt tokenizers: but these may not do exactly what we want.

E.g., how to precisely deal with punctuation, specifically intra-word dashes, ...

Words

Now suppose we have found a good way to split texts into sequences of words (maybe grouped according to sentence etc.).

What can we do with these sequences?

They still have varying lengths, which is not good.

Simplest idea (again): tabulate and represent by their frequencies.

This is the so-called **bag-of-words** model.

It ignores the actual sequential ordering of the words in texts, which may be ok, or may be very wrong.

E.g., consider the sentiment in

good is not bad
this is not good

Term frequencies

One convenient function function for computing term frequencies is `termFreq()` in package **tm**.

E.g., for

```
R> (s <- "This is some text.  And here is more.")
```

```
[1] "This is some text.  And here is more."
```

the defaults give

```
R> tm::termFreq(s)
```

```
  and  here more.  some text.  this
    1     1     1     1     1     1
attr(,"class")
[1] "term_frequency" "integer"
```

Hmm, not quite what we expected: what about the sentence end period? And where did 'is' go?

Term frequencies

It helps to look at the documentation.

So this does the following:

- Run a pipeline possibly featuring
  tokenize, tolower, removePunctuation, removeNumbers, stopwords, stemming

  (the last four are not done by default),
- Possibly match the terms against a dictionary (default: no)
- Possibly filter terms according to frequency (default: no)
- Possibly filter terms according to length (number of characters, default: min word length of 3 characters)

Stopwords are words to be excluded (typically, high frequency words which are not interesting, see later).

So the words/terms in stopwords are dropped, and the ones in the dictionary (if given) kept.

Term frequencies

Trying again:

```
R> ctrl <- list(removePunctuation =
+                 list(preserve_intra_word_dashes = TRUE),
+               wordLengths = c(2, Inf))
R> tm::termFreq(s, control = ctrl)

 and here   is more some text this
   1    1    2    1    1    1    1
attr(,"class")
[1] "term_frequency" "integer"
```

Cool!

Note 1: the pipeline can be customized via functions (e.g., use our own tokenizer) or options for the default functions.

Note 2: the above would filter out single-character words such as 'I'.

Term frequencies

Representing texts by their term frequencies is the basic **vector space model** from information retrieval.

It allows to measure similarity of texts via the similarity of the corresponding term frequencies.

How should the the latter be measured?

Clearly, scaling frequencies does not change the pattern, so Euclidean distances are out, and we want functions of the normalized frequencies.

What works “best” is **cosine similarity**, the inner product of the normalized vectors:

$$\frac{\langle x, y \rangle}{\|x\| \|y\|} = \left\langle \frac{x}{\|x\|}, \frac{y}{\|y\|} \right\rangle .$$

In particular, one can search via finding texts with highest cosine similarity to the the feature vectors of queries (no regexps, no fuzzy).

Term frequencies

If we have several texts (a **corpus** of text **documents**), we need to compute the term frequencies for each text/document.

Typically this is combined into a matrix (**document-term matrix** or **term-document matrix**).

Typically, this is a very **sparse** matrix.

Let us illustrate this using the Financial Phrase Bank data set.

Financial Phrase Bank data

```
R> load("fpb.rda")
R> dim(fpb)
[1] 4846    3
R> names(fpb)
[1] "sentence" "label"    "agree"
R> summary(fpb)
   sentence              label          agree
 Length:4846        negative: 604   Min.   : 50.00
 Class :character   neutral :2879   1st Qu.: 66.00
 Mode  :character   positive:1363   Median : 66.00
                                    Mean   : 71.51
                                    3rd Qu.: 75.00
                                    Max.   :100.00
```

Financial Phrase Bank data

This has the text also pre-processed to some extent:

```
R> writeLines(strwrap(head(fpb$sentence, 4), exdent = 2))
```

```
According to Gran , the company has no plans to move all production to
  Russia , although that is where the company is growing .
Technopolis plans to develop in stages an area of no less than 100,000
  square meters in order to host companies working in computer
  technologies and telecommunications , the statement said .
The international electronic industry company Elcoteq has laid off tens
  of employees from its Tallinn facility ; contrary to earlier layoffs
  the company contracted the ranks of its office workers , the daily
  Postimees reported .
With the new production plant the company would increase its capacity
  to meet the expected increase in demand and would improve the use of
  raw materials and therefore increase the production profitability .
```

Financial Phrase Bank data

So let's try a variant which removes punctuation and numbers, and otherwise using the defaults.

See `?DocumentTermMatrix` for details.

However, there is one catch.

Package tm really prefers to work with text **corpora** rather than character vectors of texts.

These corpora are obtained by applying suitable **readers** to suitable **sources**, and can feature both texts and **metadata**, at both document and corpus level.

Financial Phrase Bank data

The simpest way forward is creating a corpus from the character vector of texts only:

```
R> x <- tm::Corpus(tm::VectorSource(fpb$sentence))
R> x
```

```
<<SimpleCorpus>>
Metadata:  corpus specific: 1, document level (indexed): 0
Content:  documents: 4846
```

As we can see, this has no document level metadata, and one meta datum at corpus level:

```
R> NLP::meta(x, type = "c")
```

```
$language
[1] "en"

attr(,"class")
[1] "CorpusMeta"
```

(Correct, but where from? Default.)

Financial Phrase Bank data

Now we can compute our document-term matrix:

```
R> m <- tm::DocumentTermMatrix(x,
+                              control = list(removePunctuation = TRUE,
+                                             removeNumbers = TRUE))
R> m
```

```
<<DocumentTermMatrix (documents: 4846, terms: 9300)>>
Non-/sparse entries: 67972/44999828
Sparsity           : 100%
Maximal term length: 25
Weighting          : term frequency (tf)
```

Financial Phrase Bank data

As we can see, this has close to 100% sparsity!

R has several packages for sparse matrices. The most widely used is package **Matrix** (S4, bad), **tm** uses package **slam** (clever acronym, made @ WU, S3, good!):

```
R> class(m)
```

```
[1] "DocumentTermMatrix"    "simple_triplet_matrix"
```

Knowing this allows to compute the marginal term frequencies in the corpus (well, for the terms we obtained and kept) via

```
R> s <- slam::col_sums(m)
```

and summarize via:

```
R> summary(s)
```

```
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
   1.000    1.000    1.000    8.072    4.000 6066.000
```

Financial Phrase Bank data

In fact,

```
R> z <- vapply(1 : 10, function(i) mean(s == i), 0)
R> rbind(round(z, 2), round(cumsum(z), 2))
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,] 0.51 0.15 0.07 0.04 0.03 0.02 0.02 0.01 0.01  0.01
[2,] 0.51 0.66 0.73 0.78 0.80 0.83 0.85 0.86 0.87  0.88
```

shows that most of the 9300 distinct terms occur very rarely.

Overall, the total number of terms is

```
R> sum(m)
```

```
[1] 75074
```

where more than half of the terms occurs only once, and 80% at most five times.

On the other hand, some terms occur very often:

```
R> head(sort(s, decreasing = TRUE), 10)
```

```
  the   and   eur   for  will company  from   its   has  with
 6066  2593  1310  1151   850     848   768   646   579   573
```

Financial Phrase Bank data

Most of these are stopwords that are typically often found in texts:

```
R> intersect(names(head(sort(s, decreasing = TRUE), 10)), tm::stopwords())
[1] "the"  "and"  "for"  "from" "its"  "has"  "with"
```

Alternatively,

```
R> z <- sort(s, decreasing = TRUE)
R> z <- z[setdiff(names(z), tm::stopwords())]
R> head(z, 10)
  eur  will company  said finnish  sales million   net profit finland
 1310   850     848   544     512   453     441   412    409     337
R> sum(z)
[1] 57619
```

so the fraction of non-stopwords in the corpus is

```
R> sum(z) / sum(s)
[1] 0.7674961
```

In “normal” (non-short-message) texts it is typically much lower.

Financial Phrase Bank data

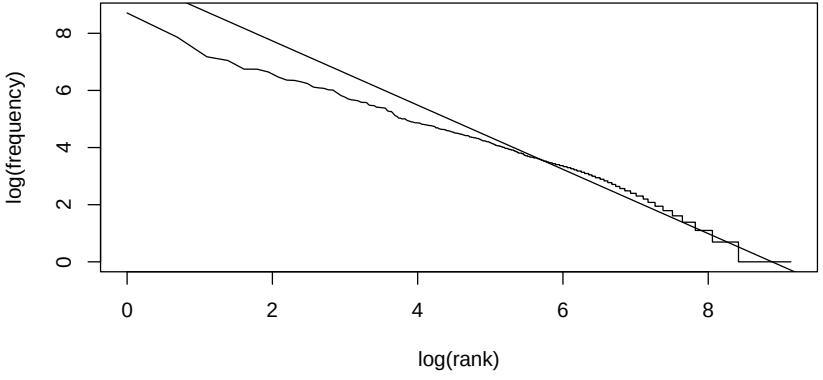
Our exploration has illustrated a key stylized fact from corpus linguistics: the frequency distributions have very long tails, and typically follow power law distributions.

This is Zipf's law (`https://en.wikipedia.org/wiki/Zipf%27s_law`).

Another empirical finding is Heaps' law (`https://en.wikipedia.org/wiki/Heaps%27_law`) which status that vocabulary size (the number of different terms employed) grows polynomially with the text size (the total number of terms in the texts).

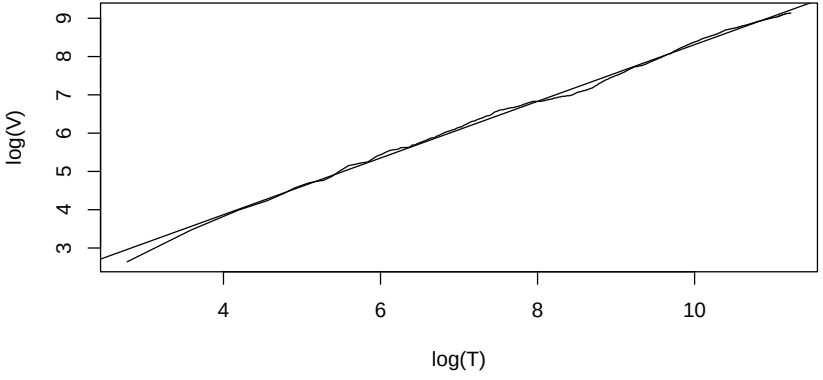
Financial Phrase Bank data: Zipf's law

```
R> tm::Zipf_plot(m)
(Intercept)           x
   9.981945   -1.124237
```



Financial Phrase Bank data: Heaps' law

```
R> tm::Heaps_plot(m)
(Intercept)           x
  0.9094633   0.7401668
```



Term frequencies

As a consequence of these findings, one often drops both the very frequent terms (not being informative as they occur “everywhere”) and the very infrequent terms (not being informative as theu occur “nowhere”).

In doing so, one may want to take into account the variability within documents versus the variability across documents.

The most popular scheme for this is the tf-idf weighting (e.g., `https://en.wikipedia.org/wiki/Tf%E2%80%93idf`), which is based on the following idea:

> *The specificity of a term can be quantified as an inverse function of the number of documents in which it occurs.*

Term frequencies

With $n_{t,d}$ the frequency of term t in document d and D the set of documents, one has

$$\mathrm{tf}_{t,d} = \frac{n_{t,d}}{\sum_\tau n_{\tau,d}}, \qquad \mathrm{idf}_t = \log \frac{|D|}{|\{d \in D : t \in d\}|} = \log \frac{|D|}{\sum_d I(n_{t,d} > 0)}$$

and uses weight

$$\mathrm{tfidf}_{t,d} = \mathrm{tf}_{t,d} \mathrm{idf}_t.$$

See `? tm::weigthtTfIdf`.

Term frequencies

Representing texts by term frequencies is the traditional vector space representation for texts.

Moving from texts to document-term matrix is the traditional **text engineering pipeline**.

Issues:

- it makes a bag-of-words “assumption”
- it cannot distinguish **homonyms**, e.g., ‘good’ as adjective and ‘good’ as noun. These could be disambiguated via their **part of speech**.
- it has no idea of semantic relations (**synonyms**, . . .).

Can we do better? Stay tuned.

Natural Language Processing

Getting complicated? Want to know more? Read

Dan Jurafsky and James H. Martin,

Speech and Language Processing.

https://web.stanford.edu/~jurafsky/slp3/

Currently draft of 3rd edition, including deep learning architectures for sequence processing.

Has PDFs for all chapters.

NLP tasks

In addition to splitting into sentences or words (tokenization), common NLP tasks include

- part of speech (POS) tagging
- lemmatizing
- named entity recognition (NER)
- parsing and chunking
- dependency parsing
- coference resolution
- ...

NLP packages for R

We already saw **tm**, which builds on top of **NLP** (more later).

Many more packages, including

Software	ProgLang	R package
Stanford CoreNLP	Java	**StanfordCoreNLP**
OpenNLP	Java	**OpenNLP**
spaCy	Python	**spacyr**
UDPipe	C++	**udpipe**
Stanza	Python	work in progress

and interfaces to NLP web services by Google, Microsoft et al.

NLP packages for R

All support English, UDpipe supports many languages.

All need pre-trained models for each language and task: often huge!

For English we have found Stanford CoreNLP to work best.

However, has huge model files (as jars), hence not on CRAN.

To install:

```
install.packages(c("NLP", "rJava"))
install.packages("StanfordCoreNLP",
                 repos = "https://datacube.wu.ac.at/",
                 type = "source")
```

To use:

```
options(java.parameters = "-Xmx4g")
```

Stanford CoreNLP pipeline

The main function in package **StanfordCoreNLP** is

```
StanfordCoreNLP::StanfordCoreNLP_Pipeline
```

This sets up an **annotator pipeline** for the desired tasks (default: pos and lemma and what is needed for these) and language (default: English, other need additional model packages from datacube).

What does that mean?

When given a text string *s*, it creates annotations for *s* (no simple transforming and splitting as for the classical pipeline).

Stanford CoreNLP pipeline

```
R> p <- StanfordCoreNLP::StanfordCoreNLP_Pipeline()
R> p
An annotator inheriting from classes
  Annotator
with description
  Computes annotations using a Stanford CoreNLP annotator pipeline with
  the following annotators: tokenize, ssplit, pos, lemma.
```

Stanford CoreNLP pipeline

```
R> a <- p("This is some text.")
R> a
```

```
 id type      start end features
  1 sentence     1  18 constituents=<<integer,5>>
  2 word         1   4 word=This, POS=DT, lemma=this
  3 word         6   7 word=is, POS=VBZ, lemma=be
  4 word         9  12 word=some, POS=DT, lemma=some
  5 word        14  17 word=text, POS=NN, lemma=text
  6 word        18  18 word=., POS=., lemma=.
```

WOW!

Stanford CoreNLP pipeline

Equivalently,

```
R> s <- "This is some text."
R> a <- NLP::annotate(s, p)
R> a
```

```
 id type     start end features
  1 sentence     1  18 constituents=<<integer,5>>
  2 word         1   4 word=This, POS=DT, lemma=this
  3 word         6   7 word=is, POS=VBZ, lemma=be
  4 word         9  12 word=some, POS=DT, lemma=some
  5 word        14  17 word=text, POS=NN, lemma=text
  6 word        18  18 word=., POS=., lemma=.
```

This gives an **Annotation** object.

Clearly, `start` and `end` give the spans of the annotations, and `features` their features.

Stanford CoreNLP pipeline

We can work such objects like data frames:

```
R> head(subset(a, type == "word")$features, 1L)

[[1]]
[[1]]$word
[1] "This"

[[1]]$POS
[1] "DT"

[[1]]$lemma
[1] "this"
```

Stanford CoreNLP pipeline

Hence, to extract words and corresponding POS tags:

```
R> feats <- subset(a, type == "word")$features
R> vapply(feats, `[[`, "", "word")
```

```
[1] "This" "is"   "some" "text" "."
```

```
R> vapply(feats, `[[`, "", "POS")
```

```
[1] "DT"  "VBZ" "DT"  "NN"  "."
```

Stanford CoreNLP pipeline

Note that the annotation does not keep the string it annotates.

We can use the annotation to extract substrings:

```
R> s <- NLP::String("This is some text.")
R> a <- NLP::annotate(s, p)
R> s[subset(a, type == "word")]
```

```
[1] "This" "is"   "some" "text" "."
```

Even better to combine the text and its annotation into an annotated plain text document:

```
R> d <- NLP::AnnotatedPlainTextDocument(s, a)
R> d
```

```
<<AnnotatedPlainTextDocument>>
Metadata:  0
Annotations:  length: 6
Content:  chars: 18
```

Stanford CoreNLP pipeline

Now we can simply do

```
R> NLP::words(d)
```

```
[1] "This" "is"   "some" "text" "."
```

```
R> NLP::tagged_words(d)
```

```
This/DT
is/VBZ
some/DT
text/NN
./.
```

And if we want Universal POS tags instead of the default ones:

```
R> NLP::tagged_words(d, NLP:::Universal_POS_tags_map)
```

```
This/DET
is/VERB
some/DET
text/NOUN
./.
```

Stanford CoreNLP pipeline

Stanford CoreNLP can also perform parsing:

```
R> s <- NLP::String("This is not good.")
R> p <- StanfordCoreNLP::StanfordCoreNLP_Pipeline("parse")
R> d <- NLP::AnnotatedPlainTextDocument(s, NLP::annotate(s, p))
R> NLP::parsed_sents(d)
```

```
[[1]]
(ROOT (S (NP (DT This)) (VP (VBZ is) (RB not) (ADJP (JJ good))) (. .)))
```

This could be used for recognizing negations beyond regexp magic.

Stanford CoreNLP pipeline

Stanford CoreNLP can also sentiment scoring:

```
R> p <- StanfordCoreNLP::StanfordCoreNLP_Pipeline("sentiment")
R> (s <- fpb$sentence[1L])

[1] "According to Gran , the company has no plans to move all production to Russia

R> a <- NLP::annotate(s, p)
R> feats <- subset(a, type == "sentence")$features
R> names(feats[[1L]])

[1] "constituents"                      "parse"
[3] "basic-dependencies"                "collapsed-dependencies"
[5] "collapsed-ccprocessed-dependencies" "enhanced-dependencies"
[7] "enhanced-plus-plus-dependencies"    "sentimentValue"
[9] "sentiment"

R> feats[[1]]$sentiment

[1] "Negative"
```

Stanford CoreNLP pipeline

Scoring the texts, somewhat defensively:

```
R> score <- function(s) {
+     a <- NLP::annotate(s, p)
+     vapply(subset(a, type == "sentence")$features, `[[`, "", "sentiment")
+ }
R> vals <- lapply(head(fpb$sentence, 13), score)
R> table(unlist(vals), head(fpb$label, 13))
```

```
           negative neutral positive
  Negative        0       2        8
  Neutral         1       0        1
  Positive        0       0        1
```

Oh dear . . .

NLP pipelines

Possible uses of NLP pipelines include:

- Better tokenizers
- Use POS tags for filtering or disambiguation
- Good lemmatizers (remember stemming)
- Generate features for measuring **readability** (syntactic depth, length of coreference chains, . . .)

Embeddings

In the document-term matrix, we only count unique “words”.

This has no notion about the semantic relations between the words, in particular, their “similarity”/“relatedness”.

Basic idea: related words co-occur (in context).

We could use the DTM to compute a word co-occurence matrix (if the context is the document).

Usually, we want smaller contexts (short word sequences within sentences).

Such co-occurence matrices will be huge: size $V \times V$ when V is the vocabulary/dictionary size.

Traditional idea from proximity scaling and friends: find suitably low-dimensional Euclidean representations which preserve similarities “well”.

Embeddings

Simplest idea: write v_t for the vector of term frequencies for term t.

In a DTM, this is the column corresponding to term t.

We could look at the best **linear** "compression" $e_t = Lv_t$ and (again linear) reconstruction $r_t = Me_t$, with $r = \mathrm{nrow}(L) = \mathrm{ncol}(M) \ll V$.

Best possible: e.g.,

$$\sum_t \|v_t - MLv_t\|_2^2 \to \min .$$

Equivalently, with D the DTM

$$\|D - MLD\|_F^2 \to \min .$$

Then MLD must be the best rank r approximation to D: we know that this is the reduced SVD of D.

So the optimal (linear) "embeddings" are obtained via the SVD: this is **Latent Semantic Analysis** (LSA).

Discuss pros and cons.

Embeddings

In modern terminology: LSA does embeddings via finding the best all-linear auto-encoder.

Could think of this as a simple (all-linear) neural networks, and generalize.

However: perhaps there are “better” tasks for finding the embeddings? E.g.,

- Predict the next word in a sentence (or its end): think of typing with your mobile phone.
- Predict a word from the surrounding context
- Predict the surrounding context (e.g., co-occurrence in context)

All of these do not need explicit labels, so give **self-supervised** learning tasks.

Good because we have huge text corpora (Wikipedia, Twitter, financial disclosures, . . .) for finding good embeddings for such tasks!

word2vec

A family of approaches to obtain word embeddings from large datasets, with main reference Mikolov et al (2013), `https://arxiv.org/abs/1301.3781`.

The one that's easiest to describe is skip-gram with negative sampling.

For all occurrences of a word (hmm, term?) we look at the context (e.g., the one word before and the one word after).

This gives us positive examples: all pairs (w, c) of words and context words.

But there are no negative examples? Don't worry: just randomly sample from the words not in context (negative sampling).

word2vec

Now suppose we have representations v_w for the words and $\tilde{v}_c$ for the contexts.

One idea is to try to have the inner products (or cosine similarities if we maintain length one) $v_w' \tilde{v}_c$ large when (w, c) is a positive example, and small when it is a negative one.

Even better, if we have a sigmoid (activation function) σ which maps the reals to $(0, 1)$, we want $\sigma(v_w' \tilde{v}_c)$ close to one for positive, and close to zero for negative examples.

If we choose $\sigma = \mathrm{plogis}$, we get something similar to logistic regression (treating all examples as independent):

$$\prod_{(w,c) \text{ positive}} \mathrm{plogis}(v_w' \tilde{v}_c) \times \prod_{(w,c) \text{ negative}} \big(1 - \mathrm{plogis}(v_w' \tilde{v}_c)\big) \to \max .$$

word2vec

We can then use the optimal w, or the sums of the optimal w and $\tilde{w}$, as the embeddings.

Usually, this gives embeddings which are close/similar when the corresponding words are related/similar.

The standard example is that

$$v_{\text{King}} - v_{\text{Man}} + v_{\text{Woman}} \approx v_{\text{Queen}}.$$

Don't ask 1: what about word sense disambiguation? Don't ask ...

Don't ask 2: what about documents? Hmm. Take the sums of the embeddings of the words in the sentence, or their "barycenter" (mean).

Hmm, but isn't this a bit like bag of works again? Don't ask!

GloVe

Another nice idea is Global Vectors for Word Representation (GloVe) by Pennington et al (`https://nlp.stanford.edu/pubs/glove.pdf`).

This does a modern version of old-style log-linear ("maximum entropy") modeling.

Write n_{ij} for the number of times word (of course, term) i occurs in the context of word j. A log-linear model would use

$$\log(\mathbb{E}(n_{ij})) = \lambda + \lambda_i^A + \lambda_j^B + \lambda_{ij}^{AB}.$$

Re-using the inner product idea gives

$$\log(\mathbb{E}(n_{ij})) = \beta_i + \tilde{\beta}_j + v_i' \tilde{v}_j.$$

However, this is not estimated by maximum likelihood: instead, one minimizes

$$\sum_{i,j} f(n_{ij})(n_{ij} - (\beta_i + \tilde{\beta}_j + v_i' \tilde{v}_j))^2$$

where f suitably scales frequencies to $[0, 1]$.

CRAN package **rsparse** provides one implementation. You'll like it: R6 again ...

word2vec

CRAN package **word2vec** provides a “standalone” implementation of several word2vec approaches (no need to use keras/tensorflow).

This wraps around efficient C++ code on `https://github.com/maxoodf/word2vec`.

The default tokenizer is very simple (see the github page).

Illustration using skip-grams:

```
R> load("fpb.rda")
R> s <- tolower(fpb$sentence)
R> m <- word2vec::word2vec(s, type = "skip", dim = 42, threads = 4)
R> v <- as.matrix(m)
R> dim(v)
```

```
[1] 2352   42
```

One can inspect the vocabulary using

```
summary(m, type = "vocabulary")
```

This again shows one really needs much better pre-processing.

word2vec

Embeddings can be obtained using `predict()` on the fitted embedding model.

For embedded words, one can also find the most similar ones:

```
R> predict(m, c("wireless", "earnings"), type = "nearest", top_n = 5)
$wireless
     term1      term2 similarity rank
1 wireless  providers  0.9851125    1
2 wireless developers  0.9835103    2
3 wireless      email  0.9817126    3
4 wireless     allows  0.9795173    4
5 wireless navigation  0.9786310    5

$earnings
     term1    term2 similarity rank
1 earnings      eps  0.9822359    1
2 earnings  diluted  0.9591051    2
3 earnings amounted  0.9493228    3
4 earnings     eur0  0.9469179    4
5 earnings     came  0.9318122    5
```

word2vec

One can use `word2vec::doc2vec()` to obtain document vectors (simply the standardized sum of the word vectors).

```
R> x <- word2vec::doc2vec(m, s)
R> dim(x)
```

```
[1] 4846   42
```

word2vec

These document features could now be used as predictors:

```
R> df <- data.frame(y = (fpb$label == "positive"), as.data.frame(x))
R> lrm <- glm(y ~ ., data = df, family = binomial())
R> p <- predict(lrm, df, type = "response")
R> table(df$y, p > 0.5)
```

```
        FALSE TRUE
  FALSE  3217  266
  TRUE    910  453
```

```
R> pROC::auc(df$y, p)
```

```
Area under the curve: 0.7643
```

Hmm, not so good . . .

word2vec

One can also use pre-trained models.

One could create a model for 10-K reports or stocktwits etc.

Transformers

Much enhanced neural network architectures based on the same basic ideas.

E.g., Bidirectional Encoder Representations from Transformers (BERT), see `https://arxiv.org/abs/1810.04805v2`.

The base model uses 12 encoders with 12 bidirectional self-attention heads, with each hidden layer featuring 768 units.

Usually, one takes the embeddings from layers 11 and 12 as the “most semantic ones”.

Training such models needs huge computational resources (TPUs).

For me, even using such models takes “forever”.

CRAN package **text** provides a convenient (but inefficient?) high level interface.

More maybe next year . . .