Introduction to Narrative

Christopher Lovell

July 6, 2015

Introduction

The *Narrative* package is designed to extend on the ubiquitous *tm* package, making time series analysis of corpora easier, as well additional functionality.

If you have not done any text mining in R before I recommend reading the tm introductory vignette first. The following document provides only a quick introduction to the *tm* package before moving on to demonstrate the additional features provided by *Narrative*.

tm

For this demo we'll be using a corpus of Reuters documents provided with the *tm* package. To load your own data, Narrative provides the `readSeparateText()` function for reading a directory full of .txt files. This is discussed in detail later.

```
library(tm)
```

```
## Loading required package: NLP
```

```
## Warning: package 'NLP' was built under R version 3.2.1
```

```
reut21578 <- system.file("texts", "crude", package = "tm")
reuters <- VCorpus(DirSource(reut21578),readerControl = list(reader = readReut21578XMLasPlain))
```

```
reuters
```

```
## <<VCorpus>>
## Metadata:  corpus specific: 0, document level (indexed): 0
## Content:  documents: 20
```

The corpus contains 20 documents. To look at a document in detail, use `as.character()`.

```
as.character(reuters[[1]])
```

```
## [1] "Diamond Shamrock Corp said that\neffective today it had cut its contract prices for crude oil by
```

The meta data contains the title, language, some high level topics, place of origin, and a time stamp; this will be important later when we begin to use the *Narrative* package.

```
meta(reuters[[1]])
```

```
##   author       : character(0)
##   datetimestamp: 1987-02-26 17:00:56
##   description  :
##   heading      : DIAMOND SHAMROCK (DIA) CUTS CRUDE PRICES
##   id           : 127
##   language     : en
##   origin       : Reuters-21578 XML
##   topics       : YES
##   lewissplit   : TRAIN
##   cgisplit     : TRAINING-SET
##   oldid        : 5670
##   topics_cat   : crude
##   places       : usa
##   people       : character(0)
##   orgs         : character(0)
##   exchanges    : character(0)
```

After reading in your data the next stage is to cleanse it and stem it. *tm* provides a numnber of functions for doing so.

```
reuters <- tm_map(reuters, content_transformer(tolower))
reuters <- tm_map(reuters, content_transformer(removeNumbers))
reuters <- tm_map(reuters, content_transformer(removePunctuation))
reuters <- tm_map(reuters, content_transformer(removeWords), stopwords("english"))
reuters <- tm_map(reuters, stemDocument)
reuters <- tm_map(reuters, content_transformer(stripWhitespace))
```

```
as.character(reuters[[1]])
```

```
## [1] "diamond shamrock corp said effect today cut contract price crude oil dlrs barrel reduct bring p
```

An incredibly useful representation of the corpus is as a *document term matrix*. This is a *td* matrix, where t is a vector of all terms used in the corpus, and d is a vector of all documents in the corpus. Each row then represents the frequency of a term in each document, and each row shows a documents constituent terms and their frequency.

```
dtm <- DocumentTermMatrix(reuters)
inspect(dtm[1:5,1:20])
```

```
## <<DocumentTermMatrix (documents: 5, terms: 20)>>
## Non-/sparse entries: 6/94
## Sparsity           : 94%
## Maximal term length: 10
## Weighting          : term frequency (tf)
##
##      Terms
## Docs  abdulaziz abil abl abroad accept accord across act activity add
##   127         0    0   0      0      0      0      0   0        0   0
##   144         0    2   0      0      0      0      0   0        0   0
##   191         0    0   0      0      0      0      0   0        0   0
##   194         0    0   0      0      0      0      0   0        0   0
##   211         0    0   0      0      0      0      0   0        0   0
```

```
##      Terms
## Docs  added address adher advantag advisers agenc agr agre agreement
##   127     0       0     0        0        0     0   0    0         0
##   144     1       4     0        1        0     0   1    0         1
##   191     0       0     0        0        0     0   0    0         0
##   194     0       0     0        0        0     0   0    0         0
##   211     0       0     0        0        0     0   0    0         0
##      Terms
## Docs  agricultur
##   127          0
##   144          0
##   191          0
##   194          0
##   211          0
```

Some useful operations can be applied to the term document matrix, such as finding the most frequent terms, or terms that are highly correlated.

```
findFreqTerms(dtm,lowfreq = 15)
```

```
##  [1] "barrel" "bpd"    "crude"  "dlrs"   "kuwait" "last"   "market"
##  [8] "mln"    "offici" "oil"    "one"    "opec"   "price"  "reuter"
## [15] "said"   "saudi"  "will"
```

```
findAssocs(dtm, "opec" ,corlimit = 0.85)
```

```
## $opec
##      emerg       meet    analyst production        oil
##       0.92       0.92       0.91       0.87       0.86
```

Narrative

Now that I have summarised the basic text mining functionality provided by *tm*, I'll move on to the additional functionality written in *Narrative*. I will use the `Narrative::function` syntax to distinguish between functions written in Narrative or other packages.

Word & Character Counts Word and character counts are often useful for normalising statistics generated from your corpus. We can generate vectors of each using the `characterCount` and `wordCount` functions. `characterCount` accepts the corpus as an argument, whereas `wordCount` uses a document term matrix.

```
wc <- Narrative::wordCount(dtm)
cc <- Narrative::characterCount(reuters)
```

Each function returns a numeric vector giving the word or character count for each document in our corpus.

Meta data

These vectors can be added to our corpus metadata using the `addToMetaData` function.

```
reuters <- Narrative::addToMetaData(corpus = reuters, vector = wc, tag = "word_count")
reuters <- Narrative::addToMetaData(corpus = reuters, vector = cc, tag = "character_count")
meta(reuters[[1]])
```

```
##   author          : character(0)
##   datetimestamp   : 1987-02-26 17:00:56
##   description     :
##   heading         : DIAMOND SHAMROCK (DIA) CUTS CRUDE PRICES
##   id              : 127
##   language        : en
##   origin          : Reuters-21578 XML
##   topics          : YES
##   lewissplit      : TRAIN
##   cgisplit        : TRAINING-SET
##   oldid           : 5670
##   topics_cat      : crude
##   places          : usa
##   people          : character(0)
##   orgs            : character(0)
##   exchanges       : character(0)
##   word_count      : 57
##   character_count: 353
```

IO

Narrative provides a couple of IO functions that make handling meta data associated with a corpus easier. To demonstrate, I'll first write our *reuters* corpus to disk using the `saveCorpus` function. This function takes a corpus, and writes it to the specified directory, along with an .Rdata file containing all associated metadata.

```
library(Narrative)
Narrative::saveCorpus(reuters,"..\\directory", save_metadata = T)
```

```
## Narrative v0.2
```

```
## metadata saved to " C:\Users\324240\Desktop\test_files " as " reuters_metadata.RDS "
```

We can then read this data back in to R, including the metadata, using the `readSeparateText` function.

```
reuters2 <- Narrative::readSeparateText("..\\directory", load_metadata = T, metadata_filename = "reuters
```

Term Frequency Analysis

A document term matrix provides a ready made data structure for searching. In our example corpus, we can find the number of occurences of the term "oil" by subsetting the document term matrix on columns.

```
inspect(dtm[,c("oil")])
```

```
## <<DocumentTermMatrix (documents: 20, terms: 1)>>
## Non-/sparse entries: 20/0
## Sparsity           : 0%
```

```
## Maximal term length: 3
## Weighting          : term frequency (tf)
##
##      Terms
## Docs  oil
##   127   5
##   144  12
##   191   2
##   194   1
##   211   1
##   236   7
##   237   3
##   242   3
##   246   5
##   248   9
##   273   5
##   349   4
##   352   5
##   353   4
##   368   3
##   489   4
##   502   5
##   543   3
##   704   3
##   708   1
```

This can be used to create a time series, showing how the usage of this term has changed over time. To do this, we convert the subsetted document term matrix in to a numeric vector.

```
search.result <- as.matrix(dtm[,c("oil")])
```

We then use this to generate an `xts` object, which is an R time series data type. To do this, we pass the search result as well as the date-time data from the corpus meta data to the `xtsGenerate` function.

```
xts.search <- Narrative::xtsGenerate(time = do.call(c,meta(reuters,"datetimestamp")),
                                     value = search.result)
xts.search
```

```
##                     [,1]
## 1987-02-26 17:00:56    5
## 1987-02-26 17:34:11   12
## 1987-02-26 18:18:00    2
## 1987-02-26 18:21:01    1
## 1987-02-26 19:00:57    1
## 1987-03-01 03:25:46    7
## 1987-03-01 03:39:14    3
## 1987-03-01 05:27:27    3
## 1987-03-01 08:22:30    5
## 1987-03-01 18:31:44    9
## 1987-03-02 01:05:49    5
## 1987-03-02 07:39:23    4
## 1987-03-02 07:43:22    5
## 1987-03-02 07:43:41    4
```

```
## 1987-03-02 08:25:42    3
## 1987-03-02 11:20:05    4
## 1987-03-02 11:28:26    5
## 1987-03-02 12:13:46    3
## 1987-03-02 14:38:34    3
## 1987-03-02 14:49:06    1
```

The xts data type allows us then to aggregate our data by time bin. Below, we aggregate by day to return the total number of occurences across all documents on this day.

```
xts.search.aggregate <- Narrative::xtsAggregate(xts.search,
                                                time_aggregate = "daily",
                                                normalisation = F)
xts.search.aggregate
```

```
##                     [,1]
## 1987-02-26 19:00:57   21
## 1987-03-01 18:31:44   27
## 1987-03-02 14:49:06   37
```

If you have a longer time series you can aggregate by week, month, quarter and year.

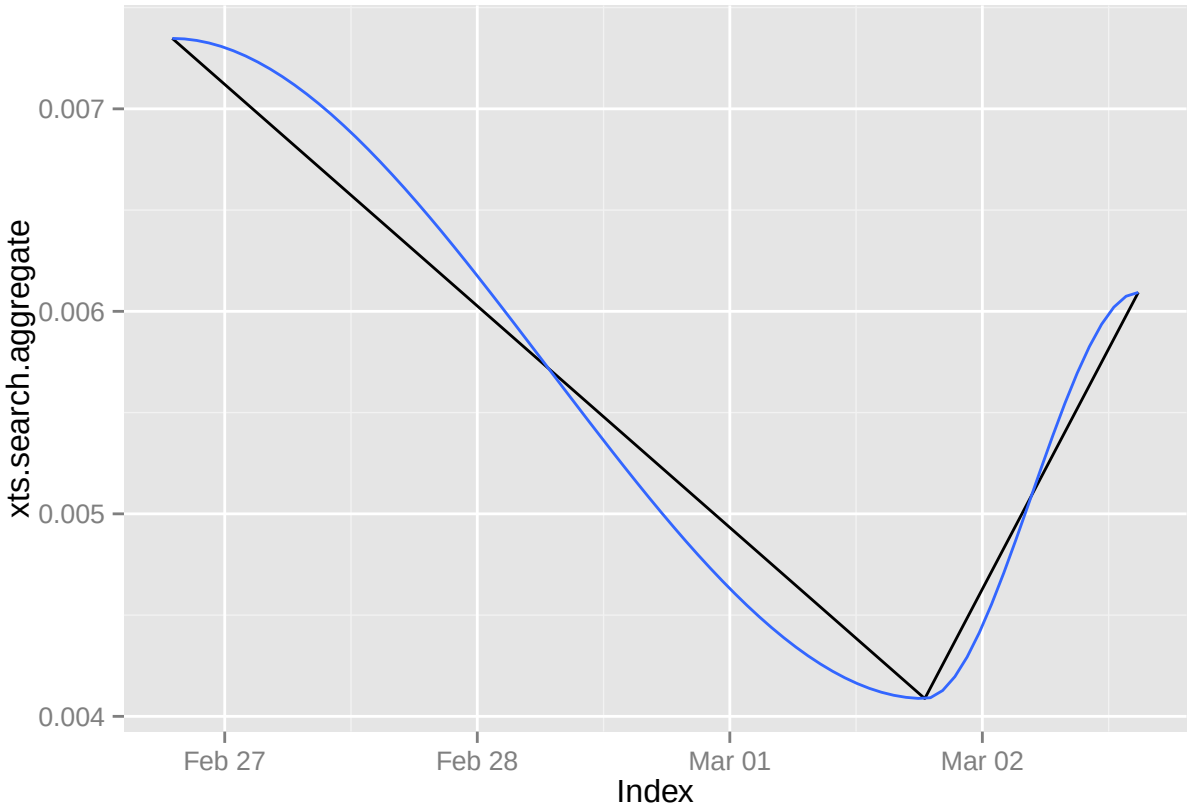
There are a number of different normalisation options when aggregating your time series data. If you wish to normalise by the number of documents in a given window, set the normalisation flag to true.

```
xts.search.aggregate <- Narrative::xtsAggregate(xts.search,
                                                time_aggregate = "daily",
                                                normalisation = T)
xts.search.aggregate
```

```
##                     [,1]
## 1987-02-26 19:00:57  4.2
## 1987-03-01 18:31:44  5.4
## 1987-03-02 14:49:06  3.7
```

Alternatively, you can normalise by any other dimension associated with your data. Just pass a numeric vector of normalisation values to the `normalisation` argument. Below, we use the character count calculated previously to normalise by size of document, then plot this along with a Loess smoother.

```
xts.search.aggregate <- Narrative::xtsAggregate(xts.search,
                                                time_aggregate = "daily",
                                                normalisation = do.call(c,meta(reuters,"character_count

p <- zoo::autoplot.zoo(xts.search.aggregate) + ggplot2::stat_smooth()
p
```

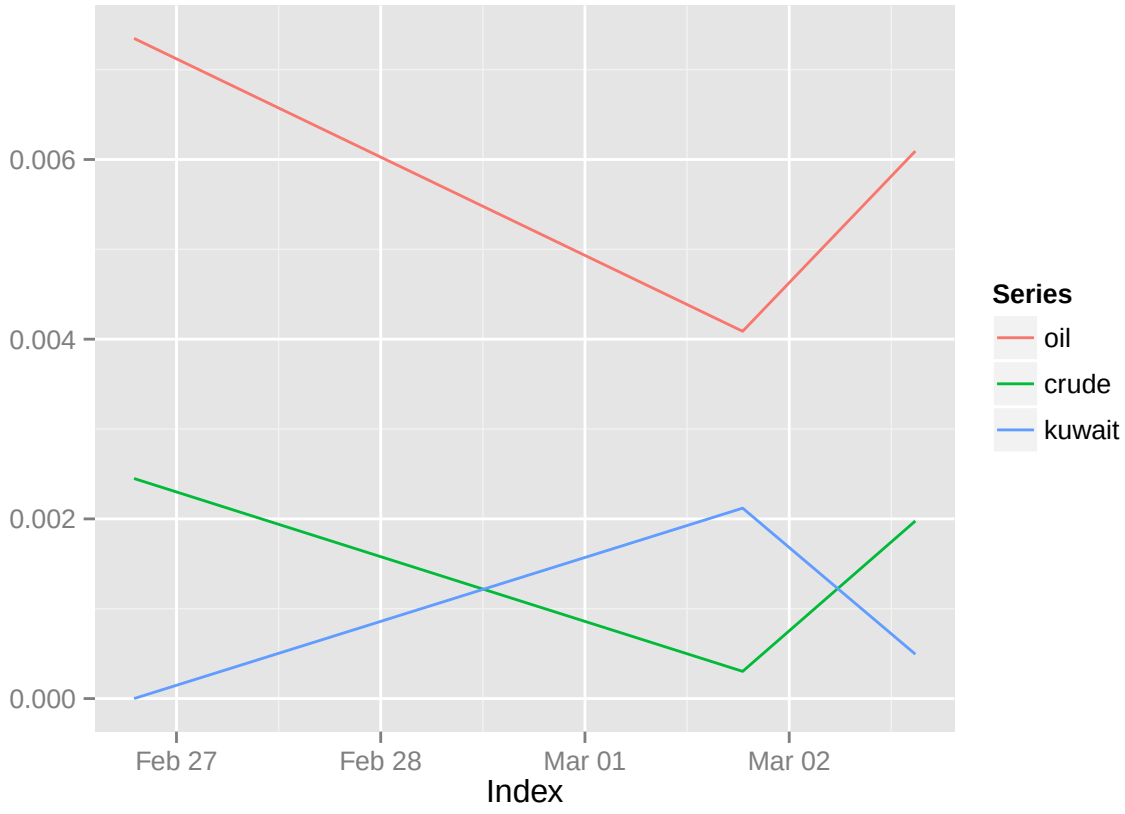


You can search for and compare multiple terms at the same time.

```
terms <- c("oil","crude","kuwait")
search.result <- as.matrix(dtm[,terms])

xts.search <- Narrative::xtsGenerate(time = do.call(c,meta(reuters,"datetimestamp")),
                                     value = search.result)

xts.search.aggregate <- Narrative::xtsAggregate(xts.search,
                                                time_aggregate = "daily",
                                                normalisation = do.call(c,meta(reuters,"character_count
```

```
## Warning in if (normalisation == F) {: the condition has length > 1 and only
## the first element will be used

## Warning in if (normalisation == T) {: the condition has length > 1 and only
## the first element will be used
```

```
p <- zoo::autoplot.zoo(xts.search.aggregate, facets = NULL)
p
```



Logical search

When searching for multiple terms, it is often useful to combine those searches logically. The `logicalMatch` function supports 'AND' and 'OR' logical operations on search results. I'll use the search reult from the previous section to demonstrate.

```
head(search.result)
```

```
##      Terms
## Docs  oil crude kuwait
##   127   5     2      0
##   144  12     0      0
##   191   2     2      0
##   194   1     3      0
##   211   1     0      0
##   236   7     2     10
```

If we want to find all documents where both oil AND crude appear, we pass the first two columns of our search reult and specify the "AND" logical operation.

```
Narrative::logicalMatch(search.result[,1:2],"AND")
```

```
##   127   144   191   194   211   236   237   242   246   248   273   349
##  TRUE FALSE  TRUE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE
```

```
##   352   353   368   489   502   543   704   708
## FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE
```

This is useful to see where terms of interest appear, but if we want to know the total occurence of both terms in these document we need to pass a function to apply to the search vectors. In this case, we wish to sum the occurences of both terms, so we pass the `sum` function from base R to the `FUN` argument.

```
Narrative::logicalMatch(search.result[,1:2], "AND", FUN = sum)
```

```
## 127 144 191 194 211 236 237 242 246 248 273 349 352 353 368 489 502 543
##   7   0   4   4   0   9   0   0   0   0  10   6   0   6   0   0   0   5
## 704 708
##   0   2
```

We may only wish to find the number of matching pairs of terms, in which case we can pass the `min` function.

```
Narrative::logicalMatch(search.result[,1:2], "AND", FUN = min)
```

```
## 127 144 191 194 211 236 237 242 246 248 273 349 352 353 368 489 502 543
##   2   0   2   1   0   2   0   0   0   0   5   2   0   2   0   0   0   2
## 704 708
##   0   1
```

We can also perform an "OR" logical match. This time we'll look for all occurences of crude OR kuwait, and summ thos eoccurences.

```
Narrative::logicalMatch(search.result[,2:3], "OR", FUN = sum)
```

```
## 127 144 191 194 211 236 237 242 246 248 273 349 352 353 368 489 502 543
##   2   0   2   3   0  12   0   1   0   3   5   3   0   4   0   0   0   2
## 704 708
##   0   1
```

Sentiment

Narrative provides a dictionary based sentiment calculator, inspired by work by Rickard Nyman et al.. Two dictionaries of positive and negative terms from a financial perspective are provided, and can be read in using the Narrative IO functions.

```
dictionaries <- Narrative::readSeparateText("inst//extdata//dictionaries")
```

We then call the `corpusSentiment` function, providing the term document matrix, each dictionary, and a meta data field over which to normalise. This generates a numeric vector of sentiment values, calculated by counting occurences of words from each dictionary, taking the difference, then normalising, in this case by word count. This results in a score with value ± 1, where 0 represents a neutral document.

$$\frac{|positive| - |negative|}{charactercount}$$

The occurence counts for each dictionary are also returned.

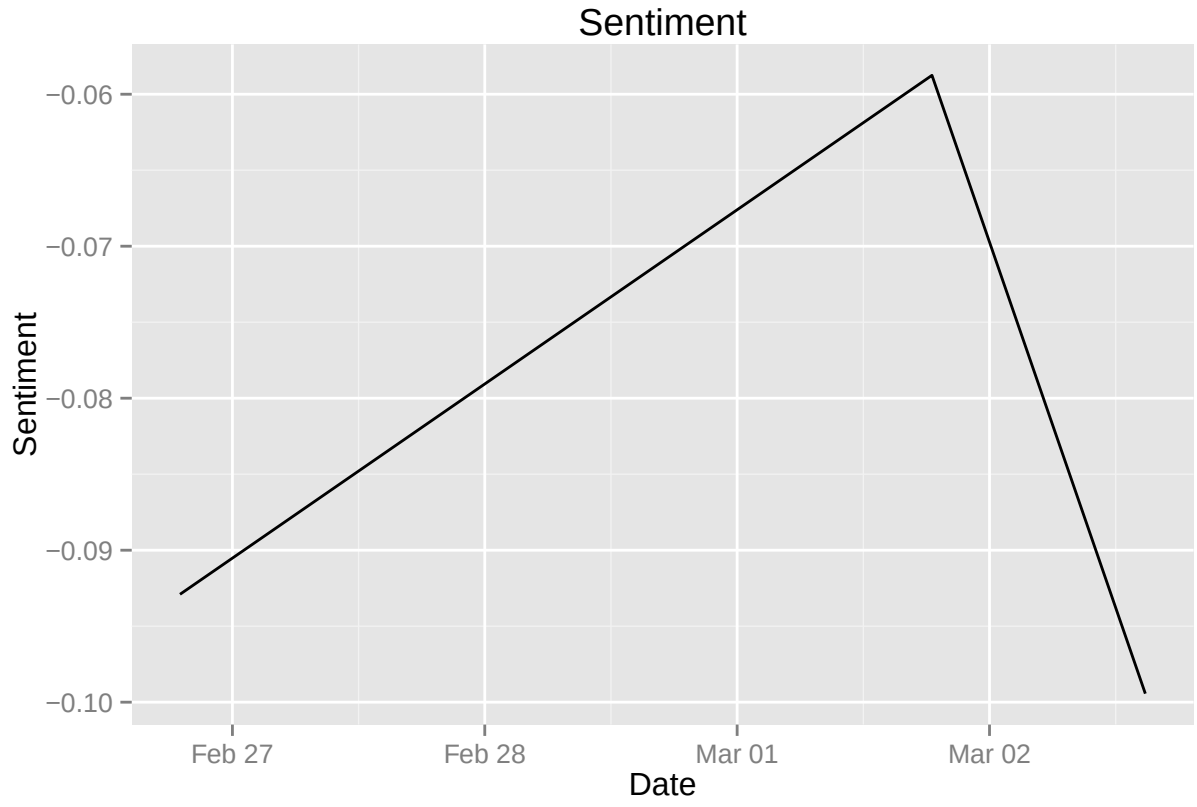
```
v <- Narrative::corpusSentiment(tdm = t(dtm),
                                dict.positive = tolower(dictionaries[meta(dictionaries,tag="id")=="exci
                                dict.negative = tolower(dictionaries[meta(dictionaries,tag="id")=="anxi
                                normalisation.meta = do.call(c,meta(reuters,"word_count")))

head(v)
```

```
##        sentiment positive.vector negative.vector
## 127 -0.070175439               0               4
## 144 -0.022727273               2               8
## 191  0.000000000               0               0
## 194  0.000000000               0               0
## 211  0.000000000               0               0
## 236 -0.007692308               1               3
```

We can then add these vectors to our corpus meta data, and plot using the same process detailed above for word counts, passing the corpus sentiment meta data rather than a vector of search results.

```
reuters <- Narrative::addToMetaData(reuters, v[,1], tag="sentiment")
reuters <- Narrative::addToMetaData(reuters, v[,2], tag="positive.count")
reuters <- Narrative::addToMetaData(reuters, v[,3], tag="negative.count")
rm(v)
```

```
xts.sentiment <- Narrative::xtsGenerate(do.call(c,meta(reuters,tag="datetimestamp")),do.call(c,meta(reu

xts.sentiment.aggregate <- Narrative::xtsAggregate(xts.sentiment, "daily", normalisation = F)

p <- zoo::autoplot.zoo(xts.sentiment.aggregate, main = ("Sentiment"))
p + ggplot2::xlab("Date") + ggplot2::ylab("Sentiment")
```



N-gram Term Document Matrix

To search for phrases an N-gram term document matrix must be constructed. The `tdmGenerator` function handles this; pass the term length and the corpus, and it will return a suitably sized term document matrix.

```
tdm.2 <- Narrative::tdmGenerator(2, reuters)
tdm.2
```

```
## <<TermDocumentMatrix (terms: 1946, documents: 20)>>
## Non-/sparse entries: 2288/36632
## Sparsity           : 94%
## Maximal term length: 24
## Weighting          : term frequency (tf)
```

```
inspect(tdm.2[90:100,1:10])
```

```
## <<TermDocumentMatrix (terms: 11, documents: 10)>>
## Non-/sparse entries: 7/103
## Sparsity           : 94%
## Maximal term length: 14
## Weighting          : term frequency (tf)
##
##                Docs
## Terms           0 1 2 3 4 5 6 7 8 9
```

```
##   announc propos 0 0 0 0 0 0 0 0 0 0
##   announc sinc   0 0 0 0 0 0 1 0 0 0
##   annual studi   0 0 0 0 1 0 0 0 0 0
##   anyth reiter   0 1 0 0 0 0 0 0 0 0
##   appar gulf     0 0 0 0 0 1 0 0 0 0
##   appar lost     0 0 0 0 0 0 0 0 0 0
##   appar refer    0 0 0 0 0 1 0 0 0 0
##   appear divid   0 0 0 0 0 0 1 0 0 0
##   appear near    0 0 0 0 0 0 2 0 0 0
##   appear result  0 0 0 0 0 0 0 0 0 0
##   appear show    0 0 0 0 0 0 0 0 0 0
```

If you pass a numeric vector then a term document matrix containing all term lengths will be generated.

```
Narrative::tdmGenerator(c(1,2,3), reuters)
```

```
## <<TermDocumentMatrix (terms: 4990, documents: 20)>>
## Non-/sparse entries: 6312/93488
## Sparsity           : 94%
## Maximal term length: 32
## Weighting          : term frequency (tf)
```

These large term document matrices can be used as any other typical tdm, for example in all search functions detailed previously.

Context Analyser

Narrative provides a couple of functions for annotating your corpus, for example by sentence, and then performing searches over them.

We first need to recreate our original corpus without removing punctuation, otherwise there will be no periods through which to identify sentences.

```
reuters <- VCorpus(DirSource(reut21578),readerControl = list(reader = readReut21578XMLasPlain))
reuters <- tm_map(reuters, content_transformer(tolower))
reuters <- tm_map(reuters, content_transformer(removeNumbers))
reuters <- tm_map(reuters, content_transformer(removeWords), stopwords("english"))
reuters <- tm_map(reuters, stemDocument)
reuters <- tm_map(reuters, content_transformer(stripWhitespace))
```

We can then call `narrativeAnnotator`, passing the corpus we wish to annotate, the desired annotation method, and whether we wish to include meta data in the corpus object returned.

```
reuters.annotated <- Narrative::narrativeAnnotator(reuters, annotator = "sentence", metadata = T)
reuters.annotated[[1]][[1]]
```

```
## [1] "diamond shamrock corp said effect today cut contract price crude oil ."
## [2] "dlrs barrel."
## [3] "reduct bring post price west texas intermedi ."
## [4] "dlrs barrel, copani said."
## [5] "\" price reduct today made light falling oil product price weak crude oil market,\" company spo
## [6] "diamond latest line u.s."
## [7] "oil compani cut contract, posted, price last two days cit weak oil markets."
## [8] "reuter"
```

We can filter by individual sentences by first selecting the document, then the content, then the sentence of interest.

```
reuters.annotated[[1]][[1]][2]
```

```
## [1] "dlrs barrel."
```

```
reuters.annotated[[1]][[1]][4]
```

```
## [1] "dlrs barrel, copani said."
```

To search over this annotated corpus, use the `annotatorSearch` function. This accepts an annotated corpus, a character vector of search terms, and a width parameter detailing the number of adjacent sentences to include in the result. A list of matching sentences is returned.

```
Narrative::annotatorSearch(annotated.corpus = reuters.annotated[c(1,3)], terms = "oil", width = 0)
```

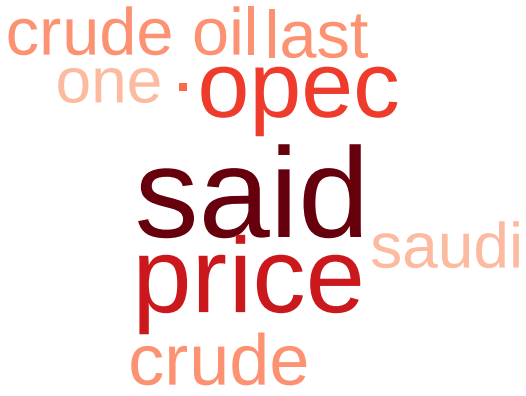
```
## [[1]]
## [1] "diamond shamrock corp said effect today cut contract price crude oil ."
## [2] "\" price reduct today made light falling oil product price weak crude oil market,\" company spol
## [3] "oil compani cut contract, posted, price last two days cit weak oil markets."
##
## [[2]]
## [1] "texaco canada said lower contract price will pay crude oil canadian cts barrel, effect today."
## [2] "texaco canada last chang crude oil post feb ."
```

This list of sentences can be converted in to a document term matrix for further analysis. Below, we use it to create a wordcloud of the content of those matched sentences.

```
terms <- "oil"
matched.sentences <- Narrative::annotatorSearch(reuters.annotated, terms, width=0)

sentences <- tm::Corpus(tm::VectorSource(matched.sentences))
sentences.tdm <- Narrative::tdmGenerator(seq(1, 2, by=1), sentences)

logi <- rownames(sentences.tdm) %in% terms
sentences.tdm <- sentences.tdm[!logi,]  # remove original search term from tdm

m <- as.matrix(sentences.tdm)
v <- sort(rowSums(m),decreasing=TRUE)
d <- data.frame(word = names(v),freq=v,key="word")

library(wordcloud)
```

```
## Loading required package: RColorBrewer
```

```
wordcloud(d$word, d$freq, scale=c(4,1), max.words=200, min.freq=10, random.order=FALSE,
          rot.per=0, fixed.asp=TRUE, use.r.layout=FALSE, colors=brewer.pal(9, "Reds"), random.color=FALS
```




```
rm(m,v,d,sentences,sentences.tdm,logi,search.vector,terms)
```

```
## Warning in rm(m, v, d, sentences, sentences.tdm, logi, search.vector,
## terms): object 'search.vector' not found
```

Miscellaneous

The `corpusAggregate` function is a flexible function allowing you to apply a user defined function over aggregate time periods of a corpus. This can be useful for applying alternative normalisation statistics, such as by number of unique terms.

`weightSort` takes a term document or document term matrix and returns a nested list of terms, sorted by weight, for each document.