Intelligence Scanning of Internet Resources

Analytical procedures involve the study of plausible relationships between both financial and non-financial data. Before the mid-2000s, sources of such information were spotty, unreliable and scarce, and that, indeed, was considered a prime reason for markets needing annual, audited financial statements. Today, there are numerous social network, news and discussion boards that offer both raw and curated, streaming sources of useful business intelligence. The auditor who does not scan for such client and industry intelligence both increases the cost of auditing, and can be considered negligent in not investigating all available information about the financial health of their client.

Sentiment analysis with tidy data

When human readers approach a text, we use our understanding of the emotional intent of words to infer whether a section of text is positive or negative, or perhaps characterized by some other more nuanced emotion like surprise or disgust. We can use the tools of text mining to approach the emotional content of text programmatically. We start by representing text in R's "tidy" structure:

- Each variable is a column
- Each observation is a row
- Each type of observational unit is a table

We thus define the tidy text format as being a table with one-token-per-row. A token is a meaningful unit of text, such as a word, that we are interested in using for analysis, and tokenization is the process of splitting text into tokens. This one-token-per-row structure is in contrast to the ways text is often stored in current analyses, perhaps as strings or in a document-term matrix. For tidy text mining, the token that is stored in each row is most often a single word, but can also be an n-gram, sentence, or paragraph. R's `tidytext` package provides functionality to tokenize by commonly used units of text like these and convert to a one-term-per-row format.

One way to analyze the sentiment of a text is to consider the text as a combination of its individual words and the sentiment content of the whole text as the sum of the sentiment content of the individual words. This is not the only way to approach sentiment analysis, but it is an often-used approach, and an approach that naturally takes advantage of the tidy tool ecosystem. The tidytext package contains several sentiment lexicons in the sentiments dataset. For example, consider the following code chunk.

```
library(tidytext)
sentiments
```

The three general-purpose lexicons are

- `bing` from Bing Liu and collaborators at University of Illinois - Chicago,
- `AFINN` from Finn Årup Nielsen, and
- `nrc` from Saif Mohammad and Peter Turney.

All three of these lexicons are based on unigrams, i.e., single words. These lexicons contain many English words and the words are assigned scores for positive/negative sentiment, and also possibly emotions like joy, anger, sadness, and so forth. All three were constructed via either crowdsourcing (using, for example,

Amazon Mechanical Turk) or by the labor of one of the authors, and were validated using some combination of crowdsourcing again, restaurant or movie reviews, or Twitter data.

The nrc lexicon categorizes words in a binary fashion ("yes"/"no") into categories of positive, negative, anger, anticipation, disgust, fear, joy, sadness, surprise, and trust.

The `bing` lexicon categorizes words in a binary fashion into positive and negative categories.

The `AFINN` lexicon assigns words with a score that runs between -5 and 5, with negative scores indicating negative sentiment and positive scores indicating positive sentiment.

All of this information is tabulated in the sentiments dataset, and tidytext provides a function get_sentiments() to get specific sentiment lexicons without the columns that are not used in that lexicon.

```
get_sentiments("afinn")

get_sentiments("bing")

get_sentiments("nrc")
```

There are also some domain-specific sentiment lexicons available, constructed to be used with text from a specific content area -- e.g., for accounting and finance. Dictionary-based methods like the ones we are discussing find the total sentiment of a piece of text by adding up the individual sentiment scores for each word in the text. Not every English word is in the lexicons because many English words are pretty neutral. It is important to keep in mind that these methods do not take into account qualifiers before a word, such as in "no good" or "not true"; a lexicon-based method like this is based on unigrams only.

One last caveat is that the size of the chunk of text that we use to add up unigram sentiment scores can have an effect on results. A text the size of many paragraphs can often have positive and negative sentiment averaged out to about zero, while sentence-sized or paragraph-sized text often works better.

With data in a tidy format, sentiment analysis can be done as an inner join. This is another of the great successes of viewing text mining as a tidy data analysis task; much as removing stop words is an antijoin operation, performing sentiment analysis is an inner join operation. Let's look at the words with a joy score from the NRC lexicon. For this example, we capture an HTML formatted General Motors' 10-K report for 2017 from SEC's ECGAR database, demonstrating that HTML documents may be used in the workpapers, as well as those in XBRL format.

```
library(htm2txt)
library(kableExtra)
library(tokenizers)
library(wordcloud)
library(tidyverse)
library(tidytext)
```

```
txt <-
gettxt("https://www.sec.gov/Archives/edgar/data/1467858/000146785818000022/
gm201710k.htm", encoding = "UTF-8")

text_stuff <- htm2txt(txt) %>%
 tokenize_words() %>%
  unlist() %>%
  as.data.frame()

colnames(text_stuff) <- "word"

stuff_sentiment <- text_stuff %>%
  inner_join(get_sentiments("bing"), by="word")

text_stuff %>%
  anti_join(get_stopwords()) %>%
  inner_join(stuff_sentiment) %>%
  count(word)%>%
  with(wordcloud(word, colors=rainbow(3), rot.per=.15, n, max.words =
1000))


net_sentiment <- text_stuff %>%
  inner_join(get_sentiments("bing"), by="word") %>%
  count(sentiment) %>%
  spread(sentiment, n, fill = 0) %>%
  mutate(net_positive = positive - negative,
         proportion__positive = positive / negative - 1)


net_sentiment %>%
  kable(longtable=T,"latex", booktabs = T) %>%
  kable_styling(bootstrap_options = c("striped", "hover", "condensed"),
full_width = F, font_size=10)
```

The size of a word's text here is in proportion to its frequency within its sentiment. We can use this visualization to see the most important positive and negative words, but the sizes of the words are not comparable across sentiments.

In other functions, such as `comparison.cloud()`, you may need to turn the data frame into a matrix with `reshape2`'s `acast()`. Let's do the sentiment analysis to tag positive and negative words using an inner join, then find the most common positive and negative words. Until the step where we need to send the data to `comparison.cloud()`, this can all be done with joins, piping, and `dplyr` because our data is in tidy format.

```
library(reshape2)
library(wordcloud)

stuff_sentiment %>%
```

```
  inner_join(get_sentiments("bing")) %>%
  count(word, sentiment, sort = TRUE) %>%
  acast(word ~ sentiment, value.var = "n", fill = 0) %>%
  comparison.cloud(colors = c("gray20", "gray80"),
                   max.words = 100)
```

With several options for sentiment lexicons, you might want some more information on which one is appropriate for your purposes. Let's use all three sentiment lexicons and examine how the sentiment changes. I use `inner_join()` to calculate the sentiment in different ways. The `AFINN` lexicon measures sentiment with a numeric score between -5 and 5, while the other two lexicons categorize words in a binary fashion, either positive or negative. Use integer division (`%/%`) to define larger sections of text that span multiple lines, and we can use the same pattern with the `tidyverse` library's pipe (`%>%`), `count()`, `spread()`, and `mutate()` to find sentiment.

```
library(tidyverse)
library(Hmisc)


cat("\n\n afinn")

get_sentiments("afinn") %>%
  describe()

cat("\n\n nrc")
get_sentiments("nrc") %>%
     filter(sentiment %in% c("positive",
                             "negative")) %>%
     describe()

cat("\n\n bing")
get_sentiments("bing") %>%
       filter(sentiment %in% c("positive",
                             "negative"))  %>%
       describe()
```

Both `bing` and `NEC` lexicons have more negative than positive words, but the ratio of negative to positive words is higher in the `bing` lexicon than the `NRC` lexicon. This will contribute to the effect we see in the plot above, as will any systematic difference in word matches.

One advantage of having the data frame with both sentiment and word is that we can analyze word counts that contribute to each sentiment. By implementing `count()` here with arguments of both `word` and `sentiment`, we find out how much each word contributed to each sentiment.

```
bing_word_counts <- stuff_sentiment %>%
  inner_join(get_sentiments("bing")) %>%
  count(word, sentiment, sort = TRUE) %>%
  ungroup()

head(bing_word_counts)
```

This can be shown visually, and we can pipe (`%>%`) straight into `ggplot2`, if we like, because of the way we are consistently using tools built for handling tidy data frames.

```
bing_word_counts %>%
  group_by(sentiment) %>%
  top_n(10) %>%
  ungroup() %>%
  mutate(word = reorder(word, n)) %>%
  ggplot(aes(word, n, fill = sentiment)) +
  geom_col(show.legend = FALSE) +
  facet_wrap(~sentiment, scales = "free_y") +
  labs(y = "Contribution to sentiment",
       x = NULL) +
  coord_flip()
```

This lets us spot an anomaly in the sentiment analysis; the words “loss”, "debt" and "fair" are generic words used to describe many accounting situations and accounts. If it were appropriate for our purposes, we could easily add “loss” to a custom stop-words list using `bind_rows()`. We could implement that with a strategy such as this.

```
library(tidytext)
library(tidyverse)
library(wordcloud)

custom_stop_words <-
  bind_rows(tibble(word = c("loss", "debt", "fair"),
  lexicon = c("custom")),
  get_stopwords())

head(custom_stop_words)

## here is our prior wordcloud with custom_stop_words

text_stuff %>%
  anti_join(custom_stop_words) %>%
  inner_join(stuff_sentiment) %>%
```

```
    count(word) %>%
    with(wordcloud(word,
                   colors = rainbow(3),
                   rot.per = 0.15, n,
                   max.words = 1000)
         )
```

Much useful work can be done by tokenizing at the word level, but sometimes it is useful or necessary to look at different units of text. For example, some sentiment analysis algorithms look beyond only unigrams (i.e. single words) to try to understand the sentiment of a sentence as a whole. These algorithms try to understand that "I am not having a good day" is a sad sentence, not a happy one, because of negation. R packages included `coreNLP`, `cleanNLP`, and `sentimentr` are examples of such sentiment analysis algorithms. For these, we may want to tokenize text into sentences.

Scanning of Uncurated News Sources from Social Networks

Twitter is arguably the most important of all the social networks for market analysis. Facebook, MySpace, Instagram, Snapchat LinkedIn, WeChat, and so forth potentially could offer more useful marketing information, but regulations either prohibit the distribution of data completely (e.g., LinkedIn) or the significantly limit it to just the people you may know directly (Facebook). This section shows you how to gain access to Twitter's API (application programming interface).

Twitter has around 130 million daily active users, a number that vastly dwarfs any of the finance specific social platforms (e.g., StockTwits) which have usage measured in hundreds of thousand. Even though it is not finance specific, the volume of financial information exchanged on Twitter is substantially larger than on bespoke finance platforms. In addition, Twitter is more likely to report the market, competitor and product specific problems that tend to drive valuations today. I use the `rtweet` package here to extract information from Twitter.

The "#" symbol is used to refer to individuals on Twitter. A # symbol before a word tells Twitter to "index" all transactions that have this hashtag. Twitter accumulates all posts that have it and retrieves them more quickly than would search with a query engine.

The "@" symbol is used to refer to individuals on Twitter. It is combined with a username and inserted into tweets to refer to that person or send them a public message. When @ precedes a username, it automatically gets linked to that user's profile page. Users may be individuals or firms.

It is not necessary to receive a token to use `rtweet`; only a user account is required. But the `OAuth` authentication gives access to more functions on Twitter, such as posting. If you need full access, go to *https://developer.twitter.com* (Twitter Developer Site) create an app and apply for a consumer key and consumer secret; as the 'Callback URL' enter:* http://127.0.0.1:1410*.

The authentication process of Twitter involves creating a *Twitter app* and doing a *handshake*. You have to *handshake* every time you want to get data from Twitter with R. Since Twitter released the Version 1.1 of their API an OAuth handshake is necessary for every request you do. The following steps will get you onto the Twitter API:

1. Create an *app* at Twitter: Go to *apps.twitter.com/* and log in with your Twitter Account. from your Profile picture in the upper right corner select the drop-down menu and look for *"My Applications"*. Click on it and then on *"Create new application"*. As the Callback URL enter:* http://127.0.0.1:1410*. Click on *Create* you´ll get redirected to a screen with all the OAuth setting of your new App.

2. Use the *setup_twitter_OAuth()* function which uses the *httr* package. Get your *api_key* and your *api_secret* as well as your *access_token* and *access_token_secret* from your app settings on Twitter (click on the *"API key"* tab to see them). Here is an example.

```
## install devtools package if it's not already
if (!requireNamespace("devtools", quietly = TRUE)) {
  install.packages("devtools")
}

## install dev version of rtweet from github
devtools::install_github("mkearney/rtweet")
install.packages("maps")
## load rtweet package
library(rtweet)
library(maps)


## access token method: create token and save it as an environment variable
create_token(
  app = "Test_of_the_API_platform",
  consumer_key = 'AWsZc3pjFsgAFlBK4OHRlyGtK',
  consumer_secret =  'DTRvorcjSaQQ1goWzynZ2tc226mgRvQ1JPxGur7nQMTesuXw3z',
  access_token = '14122740-FWlOwlo4qvhiy6oTcRypgVaIyvmlg1OZLudAToO6c',
  access_secret = 'sYjzQMjFKQFvMVRCU9gYx7bOteiS4XCoLvCgodTJZVm7y')

## Google API key for accessing geo location data through Google Maps
westland_api <- 'AIzaSyCErk3aBmPoG1FAKEqNUz6elhD6ZrR2MQtN7W0'


# To test your authentication, search for 18000 tweets using the rstats
hashtag

rt <- search_tweets(
  "#rstats", n = 18000, include_rts = FALSE
)
```

Example: Extracting tweets about General Motors and the Auto Industry

```
library(tidyverse)
library(rtweet)
```

```
##Query used to select and customize streaming collection method.
## There are four possible methods.

## (1) The default, q = "",
## returns a small random sample of all publicly available Twitter
statuses.

## (2) To filter by keyword, provide
## a comma separated character string with the desired phrase(s) and
keyword(s).

## (3) Track users by providing a comma separated list of user IDs or
screen names.

## (4) Use four latitude/longitude bounding box points to stream by geo
location.
##This must be provided via a vector of length 4, e.g., c(-125, 26, -65,
49).


stream_tweets(
  q = "auto, car, general motors, GM",
  timeout = 100,    ## the number of seconds that you will access.
                    ## Max 18000 tweets / 15 min
  parse = FALSE,    ## we'll do this later
  file_name = "tweetsaboutGM.json"
)

## read in the data as a tidy tbl data frame
djt <- parse_stream("tweetsaboutGM.json")

djt <- djt[,3:6]  ## just a few things we'd like to see
glimpse(djt)

# To get an idea of what you should be seeing before you
# actually signup for a Twitter OAuth code,
# you can bring in the dataset *trump_tweet.RData
# provided with this workout
```

Example: zxtracting Tweets about Roland Musical Instruments and their Industry

For the following examples of R code, we will be interested in Internet intelligence scanning to support a presumed analytical review of the Roland Corporation. Headquartered in Hamamatsu, Japan, Roland's 3000 employees design, manufacture and market electronic musical instruments. The firm was publicly traded until 2014 as TYO: 7944. Since it is not a US publicly listed firm, the auditor would not have any access to SEC-EDGAR filings.

```
library(rtweet)
library(httpuv)

#Twitter rate limits cap the number of search results returned to 18,000
every 15 minutes.
## To request more than that,  set retryonratelimit = TRUE
## and rtweet will wait for rate limit resets for you.
## Here we search for 18000 tweets using the Roland hashtag


roland_tweets <- search_tweets(
  "#Roland", n = 18000, include_rts = FALSE, retryonratelimit = TRUE
)


## plot time series of tweets
roland_tweets %>%
  ts_plot("3 hours") +
  ggplot2::theme_minimal() +
  ggplot2::theme(plot.title = ggplot2::element_text(face = "bold")) +
  ggplot2::labs(
    x = NULL, y = NULL,
    title = "Frequency of #Roland Twitter statuses from past 9 days",
    subtitle = "Twitter status (tweet) counts aggregated using three-hour
intervals",
    caption = "\nSource: Data collected from Twitter's REST API via rtweet"
  )
```

I can perform a quick sentiment analysis of the Roland tweets using the *NRC Word-Emotion Association* lexicon (available in R in the `textdata` library) which associates words with ten sentiments:

```
- positive
- negative
- anger
- anticipation
- disgust
- fear
- joy
- sadness
- surprise
- trust
```

```
library(tidytext)
library(tidyverse)
library(textdata)
```

```
library(ggplot2)

reg <- "([^A-Za-z\\d#@']|'(?![A-Za-z\\d#@]))"
tweet_words <-
  roland_tweets %>%
  select(user_id,source,created_at,text) %>%
  filter(!str_detect(text, '^"')) %>%
  mutate(text = str_replace_all(text, "https://t.co/[A-Za-z\\d]+|&amp;",
"")) %>%
  unnest_tokens(word, text, token = "regex", pattern = reg) %>%
  filter(!word %in% stop_words$word,
         str_detect(word, "[a-z]"))


sentmnt <- inner_join(
  get_sentiments("nrc"),
  tweet_words,
  by="word") %>%
  count(sentiment)


ggplot(sentmnt, aes(sentiment, n)) +
  geom_col() +
  theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
  xlab("Sentiment") +
  ylab("Frequency expressed in tweets")
```

We see from this brief analysis that sentiment is overall very positive for Roland. This clearly provides some insight into the sentiments of consumers and investors, but there is an important caveat. Like most social platforms, Twitter curates very little, and the quality of information conveyed about businesses is highly variable. It is best used to identify potential problems or exceptions that may not appear in other sources. The auditor will want to augment Twitter analysis with curated news sources, and this is the subject of the next section.

Intelligence Scanning of Curated News Streams

Curated intelligence sources such as Google News, MacRumours, and other news feeds, offer prepackaged information that can be My favorite is `Feedly` (stylized as `feedly`), a news aggregator application that compiles news feeds from a variety of online sources for the user to customize and share with others. Feedly is emblematic of the sort of cloud based information services that are revolutionizing the audit process.

Start by going to https://developer.feedly.com/ and then page to https://feedly.com/v3/auth/dev to sign in, get a user ID and then follow steps to get your access token. This requires either a “Pro” account or a regular account and you manually requesting OAuth codes. Store it in your ~/.Renviron in FEEDLY_ACCESS_TOKEN.

Feedly will return your access token and refresh token, which looks like the tokens below, and which you can save on your computer, leaving you free to easily access Feedly content.

```
Congratulations, you have successfully generated a developer access token.

Your user id is 448b1736-7c91-45d2-8a06-1cd797b12edc
Your access token:
AwPMVtPDxTUC7FAKEIb6_9P:feedlydev

(expires on 2019-03-15)

Your refresh token (help):
AwPMVtPDxTUC44wAQ

Do not share your tokens!
```

```
knitr::opts_chunk$set(echo = TRUE)


#' Simplifying some example package setup for this non-pkg example
.pkgenv <- new.env(parent=emptyenv())
.pkgenv$token <- Sys.getenv("FEEDLY_ACCESS_TOKEN")

#' In reality, this is more complex since the non-toy example has to
#' refresh tokens when they expire.
.feedly_token <- function() {
  return(.pkgenv$token)
}
```

For the purposes of this example, consider a "stream" to be all the historical items in a feed. Maximum "page size" (max number of items returned in a single call) is 1,000. For simplicity, there is a blanket assumption that if `continuation` is actually present, we can ask for a large number of items (e.g. 10,000)

```
devtools::install_github("hrbrmstr/seymour")
devtools::install_github("hrbrmstr/hrbrthemes")

library(seymour)
library(hrbrthemes)
library(tidyverse)


# Find writeups for seymour functions at
https://github.com/hrbrmstr/seymour/tree/master/man
```

```
## useful functions for extracting news are:
# feedly_search_contents: Search content of a stream
# feedly_search_title: Find feeds based on title, url or '#topic'
# feedly_stream: Retrieve contents of a Feedly "stream"
# feedly_subscribe: Subscribe to an RSS feed
# feedly_subscriptions: Retrieve Feedly Subscriptions


## use Feedly to check news about Apple Inc. and its products
## (search "feedburner.com" and then the corporation names to see what is
available)

apple_feed_id <- "feed/http://feeds.feedburner.com/MacRumors"

## Here is the stream function

feedly_stream(stream_id,
              ranked = c("newest", "oldest"),
              unread_only = FALSE,
              count = 1000L,
              continuation = NULL,
              feedly_token = feedly_access_token())

apple_stream <- feedly_stream(apple_feed_id)
glimpse(apple_stream)


## Here is another function

feedly_search_contents(query,
                       stream_id = NULL,
                       fields = "all",
                       embedded = NULL,
                       engagement = NULL,
                       count = 20L,
                       locale = NULL,
                       feedly_token = feedly_access_token())

f_search <-
  feedly_search_contents(q = "ipod",
                         stream_id = "apple_feed_id",
                         fields = "keywords")
glimpse(f_search)

# preallocate space
streams <- vector("list", 10)
streams[1L] <- list(apple_stream)

# catch all of the content

idx <- 2L
while(length(apple_stream$continuation) > 0) {
  cat(".", sep="") #  progress bar, sort of
```

```
  feedly_stream(
    stream_id = apple_feed_id,
    ct = 10000L,
    continuation = apple_stream$continuation
  ) -> rb_stream
  streams[idx] <- list(apple_stream)
  idx <- idx + 1L
}
cat("\n")


str(streams, 1)
str(streams[[1]], 1)
glimpse(streams[[1]]$items)
```

Feedly curates numerous news and blog outlets, which distribute a broad array of news items from formal press publications that would have a bearing on the conduct of an audit. I suggest you use the various `lubridate`, `ggplot` and `tidyverse` tools to analyze and present any insights from this larger dataset.

There are many sources of curated news for conducting analytical reviews in an audit, each with its own merits. The largest consolidator of curated news on the web is arguably Google. The `newsAPI` package is used to access Google's News API using R.

```


## install script
if (!"devtools" %in% installed.packages()) {
  install.packages("devtools")
}
devtools::install_github("mkearney/newsAPI")

## load package
library(newsAPI)

# go to newsapi.org and register to get an API key.
# save the key as an environment variable

## my obscured key
NEWSAPI_KEY <- "079ee9e373894dcfb9a062a85c4e1e7e"

## save to .Renviron file
cat(
  paste0("NEWSAPI_KEY=", NEWSAPI_KEY),
  append = TRUE,
  fill = TRUE,
  file = file.path("~", ".Renviron")
)

## install script
if (!"devtools" %in% installed.packages()) {
```

```
  install.packages("devtools")
}
devtools::install_github("mkearney/newsAPI")

src <-  get_sources(category = "", language = "en", country = "", apiKey =
NEWSAPI_KEY,
  parse = TRUE)

## load package
library(newsAPI)

df <- lapply(src$id, get_articles,apiKey=NEWSAPI_KEY)

## collapse into single data frame
df <- do.call("rbind", df)

## additional functions allow the parsing of streamed news articles

# get_sources(
  #category = "", language = "", country = "", apiKey = NEWSAPI_KEY,
parse = TRUE)


# x <- get_sources(
  #category = "", language = "", country = "", apiKey = NEWSAPI_KEY,
parse = TRUE)
#parse_sources(x)

# y <- get_articles(
  # source, sortBy = "top", apiKey = NEWSAPI_KEY, parse = TRUE)
#parse_articles(y)

# source:Name of news source.
# sortBy: Name of sorting mechanism must be one of
   ## latest, top, or popular. Certain methods only work for certain news
sources.

# apiKey: Character string API token. Default is to grab it from user R
environ.
# parse: Logical indicating whether to parse response object to data frame.

 # where x is a response object from get_sources
```

An alternative to Google News is Feedly's (stylized as `feedly`) news aggregator application that compiles news feeds from a variety of online sources -- especially firm specific, technical and financial news feeds.

Start by going to https://developer.feedly.com/ and then page to https://feedly.com/v3/auth/dev to sign in, get a user ID and then follow steps to get your access token. This requires either a “Pro” account or a regular account and you manually requesting OAuth codes. Store it in your ~/.Renviron in

FEEDLY_ACCESS_TOKEN. Feedly will return your access token and refresh token, which looks like the tokens below, and which you can save on your computer, leaving you free to easily access Feedly content.

Feedly is a news aggregator application for various web browsers and mobile devices running 'iOS' and 'Android', also available as a cloud-based service. It compiles news feeds from a variety of online sources for the user to customize and share with others. Methods are provided to retrieve information about and contents of 'Feedly' collections and streams.

Neither `feedly_search()` nor `feedly_stream()` require authentication (i.e. you do not need a developer token) to retrieve the contents of the API call. For `feedly_stream()` You do need to know the Feedly-structured feed id which is (generally) feed/FEED_URL (e.g. feed/http://feeds.feedburner.com/RBloggers).

I've generally found Feedly to be more useful than Google News for business intelligence scanning, because it is less hampered by throttling, and the curation extends to industry specific feeds (rather than Google's algorithmic guess about the topic),

In the following example, consider a "stream" to be all the historical items in a feed. Maximum "page size" (max number of items returned in a single call) is 1,000. For simplicity, there is a blanket assumption that if `continuation` is actually present, we can ask for a large number of items (e.g. 10,000)

```
library(kableExtra)

read.csv("~/audit_analytics_book/aaa_chapters/tables/feedly_functions.csv")
%>%
  kable("latex", booktabs = T) %>%
  kable_styling()
```

```
# devtools::install_github("hrbrmstr/seymour")
# devtools::install_github("hrbrmstr/hrbrthemes")

library(seymour)
library(hrbrthemes)
library(tidyverse)


# Find writeups for seymour functions at
https://github.com/hrbrmstr/seymour/tree/master/man

## useful functions for extracting news are:
# feedly_search_contents: Search content of a stream
# feedly_search_title: Find feeds based on title, url or '#topic'
# feedly_stream: Retrieve contents of a Feedly "stream"
# feedly_subscribe: Subscribe to an RSS feed
# feedly_subscriptions: Retrieve Feedly Subscriptions
```

```
## Let's check what is happening at Apple
## (search Google's "feedburner.com" and then the corporation names to see
what is available)


## the following should retreive your Feedly access token, but if not,
## you can log into the site and request the token.

token <- feedly_access_token()

feedly_search_title("roland")

# prefix the URL with 'feed/'
music_feed_id <- "feed/http://feeds.feedburner.com/MusicRadar"
music_feed_id_2 <- "feed/http://feeds.feedburner.com/MusicTech"


## Here is the stream function
feedly_stream(stream_id,
              unt = 1000L,
              continuation = NULL,
              feedly_token = feedly_access_token())

music_stream <- feedly_stream(music_feed_id_2)
glimpse(music_stream)


## Here is another function
feedly_search_contents(query,
                       stream_id = NULL,
                       fields = "all",
                       embedded = NULL,
                       engagement = NULL,
                       count = 20L,
                       locale = NULL,
                       feedly_token = feedly_access_token())

f_search <-  feedly_search_contents(q = "ipod", stream_id =
"music_feed_id", fields = "keywords")
glimpse(f_search)

# preallocate space
streams <- vector("list", 10)
streams[1L] <- list(music_stream)

# catch all of the content

idx <- 2L
while(length(music_stream$continuation) > 0) {
  cat(".", sep="") #  progress bar, sort of
  feedly_stream(
    stream_id = music_feed_id,
```

```
    ct = 10000L,
    continuation = music_stream$continuation
  ) -> rb_stream
  streams[idx] <- list(music_stream)
  idx <- idx + 1L
}
cat("\n")


str(streams, 1)
str(streams[[1]], 1)
glimpse(streams[[1]]$items)
```

API datastreams from social networks, blogs and other Internet resources tend to be best for qualitative intelligence scanning. They can alert the auditor to information that would not appear in financial news, or in the accounting statements and transactions. Such information is an essential part of the analytical review process, but until the advent of Internet accessible resources and automated tools provided by R, has not been accessible to auditors in a cost-effective way.

Accessing General Web Content through Web Scraping

Where relevant intelligence is not available through APIs, but is presented on websites, it is possible to *web scrape* data. This can be difficult and messy, but R provides a number of very effective helper tools to scrape and organize data from websites. I provide here a brief introduction to the concept and practices of web scraping in R using the `rvest` package. Tools like `rvest` and *Beautiful Soup* (Python) inject structure into web scraping, which has become important because so few companies are willing to part with their proprietary customer data sets. They have no choice but to expose some of this proprietary data via the web, though, and this is where auditors have an opportunity to accumulate valuable information germane to audit risk. The process of scraping data from the web exemplifies the computer-plus-human model of computing. It is also a nice introduction to building custom software for scraping a specific website.

The basic functions in `rvest` are powerful, and you should try to utilize the following functions when starting out a new webscrapping project.

- `html_nodes()`: identifies HTML wrappers.
- `html_nodes(".class")`: calls node based on css class
- `html_nodes("#id")`: calls node based on id
- `html_nodes(xpath="xpath")`: calls node based on xpath
- `html_attrs()`: identifies attributes (useful for debugging)
- `html_table()`: turns HTML tables into data frames
- `html_text()`: strips the HTML tags and extracts only the text

Note on plurals: `html_node()` returns metadata; but `html_nodes()` iterates over the matching nodes. The `html_nodes()` function turns each HTML tag into a row in an R dataframe.

SelectorGadget

SelectorGadget is a *javascript bookmarklet* that allows you to interactively figure out what *css selector* you need to extract desired components from a page. To install it, go to the page:

```
https://cran.r-project.org/web/packages/rvest/vignettes/selectorgadget.html
```

Install *selectorgadget* on the Chrome Browser (only at the time of this writing) from https://selectorgadget.com/. SelectorGadget is an open source tool that simplifies CSS selector generation and discovery on complicated sites. Install the Chrome Extension or drag the bookmarklet to your bookmark bar, then go to any page and launch it. A box will open in the bottom right of the website. Click on a page element that you would like your selector to match (it will turn green). *SelectorGadget* will then generate a minimal CSS selector for that element, and will highlight (yellow) everything that is matched by the selector. Now click on a highlighted element to remove it from the selector (red), or click on an unhighlighted element to add it to the selector. Through this process of selection and rejection, *SelectorGadget* helps you come up with the perfect CSS selector for your needs.

To use it, open the page:

1. Click on the element you want to select. *Selectorgadget* will make a first guess at what css selector you want. It's likely to be bad since it only has one example to learn from, but it's a start. Elements that match the selector will be highlighted in yellow.

2. Click on elements that should not't be selected. They will turn red. Click on elements that should be selected. They will turn green.

3. Iterate until only the elements you want are selected. Selectorgadget is not perfect and sometimes won't be able to find a useful css selector. Sometimes starting from a different element helps.

Other important functions:

1. If you prefer, you can use xpath selectors instead of css: `html_nodes(doc, xpath = "//table//td"))`.

2. Extract the tag names with `html_tag()`, text with `html_text()`, a single attribute with `html_attr()` or all attributes with `html_attrs()`.

3. Detect and repair text encoding problems with `guess_encoding()` and `repair_encoding()`.

4. Navigate around a website as if you're in a browser with `html_session()`, `jump_to()`, `follow_link()`, `back()`, and `forward()`. Extract, modify and submit forms with `html_form()`, `set_values()` and `submit_form()`.

Example: Simple sentiment analysis

Here is an example of a simple sentiment analysis for customers comments on restaurants in Hanoi Vietnam. Start by pointing your browser to https://www.tripadvisor.com and searching for "Asian" cuisine in "Hanoi" (as homework, consider other cities or services based on specific audit needs). Click on the "Asian" menu, which brings you to web page https://www.tripadvisor.com/Restaurants-g293924-Hanoi.html. Turn on *selectorgadget* in Chrome browser and highlight all of the reviews. In the menu at the bottom of your screen, this will give you an index `".is-9"` which is the designator for the CSS code that you have outlined

(you can verify this in Chrome by clicking the three dot menu at the upper right-hand corner of the screen, clicking "More Tools" = "Developer Tools" and checking the webpage HTML; or right-click and inspect for a quick look)

```
library(rvest)
library(RColorBrewer)
library(wordcloud)

                ## Copy the URL of the page you are scraping
                url <- "https://www.tripadvisor.com/Restaurants-g293924-
Hanoi.html"

## Extract the reviews in the CSS ".is-9" selector
reviews <- url %>%
    read_html() %>%
    html_nodes(".is-9")

## Pull the text out of the reviews
quote <- reviews %>%  html_text()

                ## Turn the character string "quote" into a data.frame and
View
 data.frame(quote, stringsAsFactors = FALSE) %>% View()

pal2 <- brewer.pal(8,"Dark2") ## from RColorBrewer

wordcloud(quote, colors=pal2)
```

In TripAdvisor, you can use the same methods, in various geographical regions, for: Hotels, Things to do, Restaurants, Flights, Vacation Rentals,Cruises and other things. Similar methods work for other review and aggregation sites.

Example: Movie Reviews

The next example scrapes information about The Lego Movie from IMDB. We start by downloading and parsing the file with `html()`. To extract the rating, we start with *Selectorgadget* to figure out which css selector matches the data we want. We use `html_node()` to find the first node that matches that selector, extract its contents with `html_text()`, and convert it to numeric with `as.numeric()`.

```
library(rvest)
lego_movie <- html("http://www.imdb.com/title/tt1490017/")

lego_movie %>%
  html_node("strong span") %>%
  html_text() %>%
```

```
  as.numeric()

## We use a similar process to extract the cast,
## using html_nodes() to find all nodes that match the selector:

lego_movie %>%
  html_nodes("#titleCast .itemprop span") %>%
  html_text()
```

Next find the actors listed on "The Lego Movie" IMDB movie page:

1. Navigate to the page and scroll to the actors list.
2. Click on the selectorgagdget link in the bookmarks. The selectorgadget console will appear at the bottom of the screen, and element currently under the mouse will be highlighted in orange.
3. Click on the element you want to select (the name of an actor). The element you selected will be highlighted in green. Selectorgadget guesses which css selector you want (.itemprop in this case), and highlights all matches in yellow.
4. Scroll around the document to find elements that you don't want to match and click on them. For example, we don't to match the title of the movie, so we click on it and it turns red. The css selector updates to `#titleCast .itemprop`.

```
library(rvest)

html <- read_html("http://www.imdb.com/title/tt1490017/")
cast <- html_nodes(html, "#titleCast .itemprop")
length(cast)

cast[1:2]

## Looking carefully at this output, we see twice as many matches as we
expected.
## That's because we've selected both the table cell and the text inside
the cell.
## We can experiment with selectorgadget to find a better match or look at
the html directly.

cast <- html_nodes(html, "#titleCast span.itemprop")
length(cast)


html_text(cast)
```

Example: Tabular Data

Some websites publish their data in an easy-to-read table without offering the option to download the data. Package `rvest` uses `html_table()` for tabular data. Using the functions listed above, isolate the

table on the page. Then pass the HTML table to html_table(). In the following case, you can go to https://www.nis.gov.kh/cpi/ and inspect the html.

```
library(rvest)
library(tidyverse)

accounts <- read_html("https://www.nis.gov.kh/cpi/Apr14.html")

table <- accounts %>%
  html_nodes("table") %>%
  html_table(header=T)

# lean up the table
# table[[1]]
dict <- table[[1]][,1:2]
accounts_df <- table[[1]][6:18,-1]

names <- c('id', 'weight.pct', 'jan.2013', 'dec.2013', 'jan.2014',
'mo.pctch', 'yr.pctch', 'mo.cont', 'yr.cont')
colnames(accounts_df) <- names

glimpse(accounts_df)
```

Example: XPaths

Xpaths are content hierarchies in a website. Sometimes you can get more comprehensive retrieval with an xPath. You can get the xpath that includes some content with the Chrome *xPath Finder* extension (it's like *SelectorGadget* but for xPaths)

```
# example of scraping a table with an XPath
library(rvest)
library(tidyverse)

h <- read_html(

"https://en.wikipedia.org/wiki/Current_members_of_the_United_States_House_o
f_Representatives")

reps <- h %>% html_node(xpath = '//*[@id="votingmembers"]') %>%
 html_table(fill=T)
reps <- reps[,c(1:2,4:9)] %>% as_tibble()
```

Example: Extracting Intelligence from Product User Forums

Product user forums are excellent sources of informed consumer and retailer information. This example provides a number of methods that can be used for general web scraping. Applying this to our goal of web scraping for intelligence on Roland's products, we can glean consumer sentiment on Roland's pianos as conveyed by discussions on the Piano World Forum website and display it with `wordcloud`.

```
#install.packages(c("tm", "SnowballC", "wordcloud", "RColorBrewer",
"RCurl", "XML"))

library(tm)
library(SnowballC)
library(RCurl)
library(tidyverse)
library(rvest)
library(RColorBrewer)
library(wordcloud)
library(stringr)
library(httr)
library(XML)

handle <- handle("http://forum.pianoworld.com//")
path   <- "ubbthreads.php/ubb/login.html?
ocu=http%3A%2F%2Fforum.pianoworld.com%2F"

# fields found in the login form.
login <- list(
  amember_login = "westland"
 ,amember_pass  = "powerpcc"
 ,amember_redirect_url =
   "http://forum.pianoworld.com//ubbthreads.php/forum_summary.html"
)

response <- POST(handle = handle, path = path, body = login)



# Copy the URL of the page you are scraping
url <- "http://forum.pianoworld.com/"

# Extract the reviews in the CSS selector
reviews <- url %>%
           read_html() %>%
           html_nodes("#cat2 div")

# Pull the selected text out of the reviews
quote <- reviews %>%
                  html_text() %>%
                   as.tibble()

quote <-  filter(quote[str_detect(quote, "Roland")])

pal2 <- brewer.pal(8,"Dark2") # from RColorBrewer
```

```
wordcloud(unlist(quote), colors=pal2)
```