Analytical Review

Analytical procedures are evaluations of financial information made by a study of plausible relationships between both financial and non-financial data. Analytical procedures are used in all stages of the audit including planning, substantive testing and final review stage. Auditing standards require the use of analytical review procedures in the planning and final review stages. It serves as a vital planning function in the entirety of the audit procedures.

Common Technical Metrics: Accessing Financial Information from EDGAR (https://www.sec.gov/edgar/)

The EDGAR database at https://www.sec.gov/edgar.shtml www.sec.gov/edgar.shtml offers a wealth of financial information on industry competitors, but only if they are publicly traded companies. Use package `edgar` to extract this information.

```
library(edgar)
library(tidyverse)
library(kableExtra)


  cik.no = 0001318605   # Tesla
  form.type = '10-K'
  filing.year = 2018
  quarter = c(1,2,3,4)


# getFilings function takes CIKs, form type, filing year, and quarter of
the filing as input. It creates
# new directory ’~/Downloads/Edgar filings_full text’ to store all
downloaded filings.

  getFilings(
  cik.no,
  form.type,
  filing.year,
  quarter,
  downl.permit="y")


# getFilingsHTML function takes CIKs, form type, filing year, and quarter
of the filing as input. The
# function imports edgar filings downloaded via getFilings function;
otherwise, it downloads the filings which are not already been downloaded.
It then reads # the downloaded filing, scraps main body
# the filing, and save the filing content in '~/Downloads/Edgar
filings_HTML view’ directory in HTML format.
```

```
getFilingsHTML(
  cik.no = cik.no,
  form.type = form.type,
  filing.year = filing.year,
  quarter = quarter
  )


# This function creates a new directory '~/Downloads/Master Indexes' into
current working directory to save these Rda Master Index.

getMasterIndex(filing.year)


# Management Discussion creates a new directory with name "~/Downloads/MD&A
section text"


getMgmtDisc(
  cik.no = cik.no,
  filing.year = filing.year)


# getSentiment function takes CIK(s), form type(s), and year(s) as input
parameters. The function first
# imports available downloaded filings in local woking directory 'Edgar
filings' created by getFilings
# function; otherwise, it downloads the filings which is not already been
downloaded. It then reads the
# filings, cleans the filings, and computes the sentiment measures. The
function returns a dataframe
# with filing information, and sentiment measures.



sentiment_analysis <-
  getSentiment(
  cik.no,
  form.type,
  filing.year) %>%
  t()

  d <- sentiment_analysis
  names <- rownames(d)
  rownames(d) <- NULL
  sentiment_analysis <- cbind(names,d)

  colnames(sentiment_analysis) <- c("sentiment","n")

  sentiment_analysis <-
as.data.frame(sentiment_analysis[10:nrow(sentiment_analysis),])

ggplot(sentiment_analysis, aes(sentiment, n)) +
  geom_col() +
  theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
  xlab("Sentiment") +
  ylab("Frequency expressed in 10-k")
```

Accessing Financial Information from EDGAR (https://www.sec.gov/edgar/) with the `finreportr` Package

```
library(finreportr)

# The following commands will directly load EDGAR information into the R
workspace for analysis

tesla_co <- CompanyInfo("TSLA")
tesla_ann <- AnnualReports("TSLA")
tesla_inc <- GetIncome("TSLA", 2018)
tesla_bs <- GetBalanceSheet("TSLA", 2018)
tesla_cf <- GetCashFlow("TSLA", 2018)

head(tesla_inc)
```

The `finreportr` package returns a data.frame in 'long form'. Because analysis typically benefits from datasets in 'short form' with one row per account, `finreportr` data.frames need to be reshaped. Hadley Wickham has created a comprehensive package called `reshape2` that uses metaphors of `melt` and `cast`. You `melt` data so that each row is a unique id-variable combination (i.e., is in 'long form') and then you `cast` the melted data into any shape you would like. There are specific commands for casting data.frames `dcast`, arrays `acast`, and so forth. In the next example, we take the Tesla income statement in 'long form' that we acquired with the `finreportr` package and `dcast` it into a more usable form.

```
library(tidyverse)
library(lubridate)
library(finreportr)
library(reshape2) # uses `melt` and `cast` to with between long and wide
formats

tesla_inc <- GetIncome("TSLA", 2018)  %>%
    rbind(GetIncome("TSLA", 2017)) %>%
    rbind(GetIncome("TSLA", 2016)) %>%
    rbind(GetIncome("TSLA", 2015))

head(tesla_inc)
```

```
tesla_inc <-tesla_inc %>%
  filter(month(startDate) == 01 & month(endDate) == 12) %>%
  mutate(Year = year(endDate)) %>%
  group_by(Metric, Year) %>%
  slice(1L) %>%
  dcast(Metric ~ Year, value.var = 'Amount')

head(tesla_inc)
```

Computing Technical Metrics

In the prior section, I showed you how to acquire financial information from the SEC's repositories. This section provides general guidelines for computing technical metrics such as ratios from that statement data. Consider the calculation of the current ratio which is defined as:

$$ Current Ratio = \frac{Current Assets}{Current Liabilities} $$ Here is how to calculate the ratio from our merged balance sheet obtained from EDGAR data using the `finstr` package.

```


library(devtools)
install_github("bergant/finstr")
library(finstr)
library(tidyverse)
library(kableExtra)
library(XBRL)

# Get EDGAR data in XBRL format from the sec.gov site
# parse XBRL (GM 10-K reports)

xbrl_url2016 <-
"https://www.sec.gov/Archives/edgar/data/1467858/000146785817000028/gm-
20161231.xml"

xbrl_url2017 <-
"https://www.sec.gov/Archives/edgar/data/1467858/000146785818000022/gm-
20171231.xml"

old_o <- options(stringsAsFactors = FALSE)
xbrl_data_2016 <- xbrlDoAll(xbrl_url2016)
xbrl_data_2017 <- xbrlDoAll(xbrl_url2017)
options(old_o)

st2017 <- xbrl_get_statements(xbrl_data_2017)
st2016 <- xbrl_get_statements(xbrl_data_2016)

balance_sheet2017 <- st2017$ConsolidatedBalanceSheets
balance_sheet2016 <- st2016$ConsolidatedBalanceSheets
balance_sheet <- merge(balance_sheet2017, balance_sheet2016)

##  calculate current ratio
```

```
balance_sheet %>% transmute(
  date = endDate,
  CurrentRatio = AssetsCurrent / LiabilitiesCurrent
   )
```

Other ratios may be calculated in a similar straightforward manner using the `dplyr` package in the `tidyverse` library. Note that several other packages (e.g., `edgar`, `finreportr`) are available to extract EDGAR filings. The `finstr` package can only process links to XBRL files, but `finreportr` can accesse data from both HTML and XBRL files.

Visualization of Technical Metrics

Visualizations are compact, yet can reveal patterns that wouldn't be readily identified in the raw data. This is because the human brain is much more attuned to analyzing visual scenes, than to analyzing lists of numbers and characters. Visualizing financial statements exposes a limited number of key values, and emphasizes their relationships and trends. In the following code chunk, I aggregate a balance sheet by selected concepts.

```
library(htmlTable)
library(ggplot2)
library(kableExtra)

bs_simple <- expose(balance_sheet,

  # Assets
  `Current Assets` = "AssetsCurrent",
  `Noncurrent Assets` = other("Assets"),
  # Liabilites and equity
  `Current Liabilities` = "LiabilitiesCurrent",
  `Noncurrent Liabilities` = other(c("Liabilities",
"CommitmentsAndContingencies")),
  `Stockholders Equity` = "StockholdersEquity"
)


## Print the balance sheet;
## capture the output to a NULL file, and
## reformat with the kableExtra package

capture.output(bs_table <-
                 print(
                   bs_simple,
                   html = FALSE,
                   big.mark = ",",
                   dateFormat = "%Y"),
               file='NUL')
```

```
bs_table %>%
  kable(longtable=T,
        caption="Abbreviated Balance Sheet",
        "latex",
        booktabs = T) %>%
  kable_styling(bootstrap_options =
                  c("striped", "hover", "condensed"),
                full_width = F, font_size=10)

## plot the balance sheet

finstr::plot_double_stacked_bar(bs_simple)

## facet balance sheet DR and CR accounts

finstr::plot_double_stacked_bar(bs_simple, by_date = FALSE)

## use proportional form to highlight changes in balance sheet structure

bs_simple_prop <- proportional(bs_simple)
finstr::plot_double_stacked_bar(bs_simple_prop)
```

U.S. Census Data

Where location and demographics are important in a business, the US Census provides extensive and reliable data. R provides several packages to access an use those repositories. Data from the U.S. Census Bureau is stored in tables, and to find the table for a particular metric you can use the function `acs.lookup` in the `acs` package. Note that to run this code you will need to get and install a census API key which you can request at `https://api.census.gov/data/key_signup.html`. Install the key with `api.key.install`.

The `acs.fetch` function is used to download data from the US Census American Community Survey. The `acs.lookup` function provides a convenience function to use in advance to locate tables and variables that may be of interest.

`acs.lookup` takes arguments similar to `acs.fetch` — in particular, "table.number", "table.name", and "keyword", as well as "endyear","span", and "dataset" — and searches for matches in the meta-data of the Census tables. When multiple search terms are passed to a given argument (e.g., keyword=c("Female", "GED")), the tool returns matches where ALL of the terms are found; similarly, when more than one lookup argument is used (e.g., table.number="B01001", keyword="Female"), the tool searches for matches that include all of the terms (i.e., terms are combined with a logical "AND", not a logical "OR").

Results from `acs.lookup` — which are `acs.lookup` class objects — can then be inspected, subsetted (with [square brackets]), and combined (with c or +) to create custom `acs.lookup` objects to store and later pass to `acs.fetch` which has the following arguments:

- `endyear` an integer indicating the latest year of the data in the survey (e.g., for data from the 2007-2011 5-year ACS data, endyear would be 2011)

- `span` an integer indicating the span (in years) of the desired ACS data (should be 1, 3, or 5 for ACS datasets, and 0 for decennial census SF1 and SF3 datasets); defaults to 5, but ignored and reset to 0 if dataset="sf1" or "sf3".
- `geography` a `geo.set` object specifying the census geography or geographies to be fetched; can be created "on the fly" with a call to geo.make()
- `table.name` a string giving the search term(s) to find in the name of the ACS census table (for example, "Sex" or "Age"); accepts multiple words, which must all be found in the returned table names; always case-sensitive. (Note: when set, this variable is passed to an internal call to acs.lookup—see acs.lookup).
- `table.number` a string (not a number) indicating the table from the Census to fetch; examples: "B01003" or "B23013"; always case-sensitive. Used to fetch all variables for a given table number; if "table.number" is provided, other lookup variables ("table.name" or "keyword") will be ignored.
- `variable` an object of `acs.lookup` class, or a string or vector of strings indicating the exact variable number to fetch. Non-`acs.lookup` examples include "B01003_001" or "B23013_003" or c("B01003_001", "B23013_003").
- `keyword` a string or vector of strings giving the search term(s) to find in the name of the census variable (for example, "Male" or "Haiti"); always case-sensitive.
- `dataset` either "acs" (the default), "sf1", or "sf3", indicating whether to fetch data from in the American Community Survey or the SF1/SF3 datasets.

In the following example, we will compute the 2014-2019 female-to-male populations of the U.S. across age groups, and plot these as a bar graph.

```
library(acs)
library(tidyverse)


look <- acs.lookup(endyear = 2019,
                   keyword=c("Female"))
i_look <- look@results[1:24,c(1,4)] %>% t()
colnames(i_look) <- i_look[1,]

geo <- geo.make(state = "IL")
fet <- acs.fetch(endyear = 2014,
          span = 5,
          table.number="B01001",
          keyword=c("Female"),
          geography =  geo)

fet_tbl <-  fet@estimate
fet_tbl <- rbind(fet_tbl,i_look[2,]) %>% t()
colnames(fet_tbl) <- c("population","age_group")

fet_tbl <-as.data.frame(fet_tbl)
```

```
# make age_group an ordered factor and convert population to numeric
fet_tbl$age_group <- factor(fet_tbl$age_group, levels = fet_tbl$age_group)
fet_tbl$population <- as.numeric(as.character(fet_tbl$population))
fet_fem <-as.data.frame(fet_tbl)


fet <- acs.fetch(endyear = 2014,
          span = 5,
          table.number="B01001",
          keyword=c("Male"),
          geography =  geo)

fet_tbl <-  fet@estimate

fet_tbl <- rbind(fet_tbl,i_look[2,]) %>% t()
colnames(fet_tbl) <- c("population","age_group")
fet_tbl <-as.data.frame(fet_tbl)
fet_tbl$age_group <- factor(fet_tbl$age_group, levels = fet_tbl$age_group)
fet_tbl$population <- as.numeric(as.character(fet_tbl$population))
fet_male <-as.data.frame(fet_tbl)

# Compute the ratio of females to males by U.S. county
fet_ratio <- inner_join(fet_fem,fet_male, by = "age_group")
fet_ratio$fem_to_male <- fet_ratio$population.x / fet_ratio$population.y
fet_ratio$age_group <- sub("Female:", "", fet_ratio$age_group)
fet_ratio$age_group <- factor(fet_ratio$age_group, levels =
fet_ratio$age_group)

ggplot(fet_ratio[-1,], aes(age_group, fem_to_male)) +
  geom_col() +
  theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
  xlab("Age Group") +
  ylab("Female to Male Ratio")
```

The first table on the Census Bureau list is "B01001". Look at `look_tbl = look@results` under `variable.code` for example, the "_009" at the end indicates the column of the table; each column tabulates a different age range for males or females.

If your end result is to create choropleth maps, using the `choroplethr` package it is more straightforward to use the function `get_acs_data` inside the `choroplethr` package instead of `acs.fetch` in the `acs` package. You will still need to load the `acs` package to use `get_acs_data`.

```
library(choroplethr)
library(acs)
library(tidyverse)


# map = one of "state", "county" or "zip"
# column_idx = 15 of table B01001 is "Sex by Age: Male: 45 to 49 years"
# column_idx = 39 of table B01001 is "Sex by Age: Female: 45 to 49 years"
```

```
M45 <-  get_acs_data(tableId = "B01001", map = "county", column_idx=15)
F45 <-  get_acs_data(tableId = "B01001", map = "county", column_idx=39)
M45 <- M45[[1]]
head(M45)
F45 <- F45[[1]]
head(F45)

R45 <- inner_join(M45, F45, by = "region")
R45$value <- R45$value.y / R45$value.x

county_choropleth(R45[,c(1,4)], title = "    Female / Male Ratio: 45 - 49
y.o. by County")
```

Technical Analysis of Product and Customer News Sources on the Web

Most text mining and natural language processing (NLP) modeling uses 'bag of words' or 'bag of n-grams' methods. Despite their simplicity, these models usually demonstrate good performance on text categorization and classification tasks. But in contrast to their theoretical simplicity and practical efficiency, building bag-of-words models involves technical challenges. This is especially the case in R because of its copy-on-modify semantics.

Let's briefly review some of the steps in a typical text analysis pipeline:

1. The auditor usually begins by constructing a document-term matrix (DTM) or term-co-occurrence matrix (TCM) from input documents. In other words, the first step is to vectorize text by creating a map from words or n-grams to a vector space.

2. The auditor fits a model to that DTM. These models might include text classification, topic modeling, similarity search, etc. Fitting the model will include tuning and validating the model.

3. Finally the auditor applies the model to new data.

Texts themselves can take up a lot of memory, but vectorized texts usually do not, because they are stored as sparse matrices. Because of R's copy-on-modify semantics, it is not easy to iteratively grow a DTM; constructing a DTM, even for a small collections of documents, can be a serious bottleneck. It involves reading the whole collection of text documents into memory and processing it as single vector, which can easily increase memory use by a factor of two to four. The `text2vec` package solves this problem by providing a better way of constructing a document-term matrix.

As an example of NLP using the `text2vec` package, I will parse a dataset that comes with the package -- the `movie_review` dataset. It consists of 5000 movie reviews, each of which is marked as positive or negative. First, split the dataset into two parts - train and test.

```


library(text2vec)
library(data.table)
library(magrittr)
data("movie_review")
setDT(movie_review)
```

```
setkey(movie_review, id)
set.seed(2017L)
all_ids = movie_review$id
train_ids = sample(all_ids, 4000)
test_ids = setdiff(all_ids, train_ids)
train = movie_review[J(train_ids)]
test = movie_review[J(test_ids)]

head(movie_review)
class(movie_review)
```

Vocabulary-based vectorization

To represent documents in vector space, we first have to create 'term' mappings. We call them terms instead of words because they can be, not just single words, but arbitrary n-grams -- contiguous sequence of n items, where items can be phonemes, syllables, letters, words or base pairs. We represent a set of documents as a sparse matrix, where each row corresponds to a document and each column corresponds to a term. Create a vocabulary-based DTM by collecting unique terms from all documents and mark each of them with a unique ID using the `create_vocabulary()` function, using an iterator to create the vocabulary.

The following code chunk:

1. creates an iterator over tokens with the `itoken()` function. All functions prefixed with `create_` work with these iterators. R users might find this idiom unusual, but the iterator abstraction allows us to hide most of details about input and to process data in memory-friendly chunks.
2. builds the vocabulary with the `create_vocabulary()` function.

```
# define preprocessing function and tokenization function
prep_fun = tolower
tok_fun = word_tokenizer

it_train = itoken(train$review,
             preprocessor = prep_fun,
             tokenizer = tok_fun,
             ids = train$id,
             progressbar = FALSE)
vocab = create_vocabulary(it_train)
```

Alternatively, we could create list of tokens and reuse it in further steps. Each element of the list should represent a document, and each element should be a character vector of tokens.

```
library(magrittr)
library(tidyverse)
library(text2vec)

train_tokens = train$review %>%
  prep_fun %>%
  tok_fun
it_train = itoken(train_tokens,
                  ids = train$id,
                  # turn off progressbar because it won't look nice in rmd
                  progressbar = FALSE)

vocab = create_vocabulary(it_train)
vocab

vectorizer = vocab_vectorizer(vocab)
t1 = Sys.time()
dtm_train = create_dtm(it_train, vectorizer)
print(difftime(Sys.time(), t1, units = 'sec'))

dim(dtm_train)

identical(rownames(dtm_train), train$id)
```

Once we have a vocabulary, we can construct a document-term matrix.

```
vectorizer = vocab_vectorizer(vocab)
t1 = Sys.time()
dtm_train = create_dtm(it_train, vectorizer)
print(difftime(Sys.time(), t1, units = 'sec'))
```

At this point, we are ready to fit our model. Here we will use the `glmnet` (Lasso and Elastic-Net Regularized Generalized Linear Models) package to fit a logistic regression model with an L1 penalty (LASSO = least absolute shrinkage and selection operator) and 4-fold cross-validation.

```
library(glmnet)
NFOLDS = 4
t1 = Sys.time()
glmnet_classifier =
  cv.glmnet(x = dtm_train,
            y = train[['sentiment']],
                              family = 'binomial',

##  for the movie_reviews this is "binomial",
## for the food reviews it is 1-5, so "multinomial"
```

```
                              # L1 penalty
                               alpha = 1,
                               # interested in the area under ROC curve
                               type.measure = "auc",
                               # 5-fold cross-validation
                               nfolds = NFOLDS,
                               # high value is less accurate, but has faster
training
                               thresh = 1e-3,
                               # again lower number of iterations for faster
training
                               maxit = 1e3)
print(difftime(Sys.time(), t1, units = 'sec'))

plot(glmnet_classifier)



print(paste("max AUC =", round(max(glmnet_classifier$cvm), 4)))
```

We have successfully fit a model to our DTM. Now we can check the model's performance on test data. Note that we use exactly the same functions from prepossessing and tokenization. Also we reuse/use the same vectorizer - function which maps terms to indices.

```
# Note that most text2vec functions are pipe friendly!
it_test = test$review %>%
  prep_fun %>% tok_fun %>%
  # turn off progressbar because it won't look nice in rmd
  itoken(ids = test$id, progressbar = FALSE)


dtm_test = create_dtm(it_test, vectorizer)

preds = predict(glmnet_classifier, dtm_test, type = 'response')[,1]
glmnet:::auc(test$sentiment, preds)
```

The result shows that performance on the test data is roughly the same as we expected from cross-validation. Note though that the training time for the model was high. We can reduce it and also significantly improve accuracy by 'pruning' the vocabulary. For example, we can find words "a", "the", "in", "I", "you", "on", etc in almost all documents, but they do not provide much useful information. Usually such words are called 'stop words'. On the other hand, the corpus also contains very uncommon terms, which are contained in only a few documents. These terms are also useless, because we don't have sufficient statistics for them. Here we will remove pre-defined stopwords, very common and very unusual terms.

```
stop_words = c("i",
               "me",
               "my",
               "myself",
               "we",
               "our",
               "ours",
               "ourselves",
               "you",
               "your",
               "yours")

t1 = Sys.time()
vocab = create_vocabulary(it_train, stopwords = stop_words)
print(difftime(Sys.time(), t1, units = 'sec'))

pruned_vocab = prune_vocabulary(vocab,
                                 term_count_min = 10,
                                 doc_proportion_max = 0.5,
                                 doc_proportion_min = 0.001)
vectorizer = vocab_vectorizer(pruned_vocab)


# create dtm_train with new pruned vocabulary vectorizer

t1 = Sys.time()
dtm_train  = create_dtm(it_train, vectorizer)
print(difftime(Sys.time(), t1, units = 'sec'))


dim(dtm_train)

# create DTM for test data with the same vectorizer:

dtm_test = create_dtm(it_test, vectorizer)
dim(dtm_test)
```

This model can be improved by using n-grams instead of words -- in the following code-chunk, I use up to 2-grams:

```
t1 = Sys.time()
vocab = create_vocabulary(it_train, ngram = c(1L, 2L))
print(difftime(Sys.time(), t1, units = 'sec'))


vocab = prune_vocabulary(vocab, term_count_min = 10,
                         doc_proportion_max = 0.5)
```

```
bigram_vectorizer = vocab_vectorizer(vocab)

dtm_train = create_dtm(it_train, bigram_vectorizer)

t1 = Sys.time()
glmnet_classifier = cv.glmnet(x = dtm_train, y = train[['sentiment']],
                 family = 'binomial',
                 alpha = 1,
                 type.measure = "auc",
                 nfolds = NFOLDS,
                 thresh = 1e-3,
                 maxit = 1e3)
print(difftime(Sys.time(), t1, units = 'sec'))


plot(glmnet_classifier)


print(paste("max AUC =", round(max(glmnet_classifier$cvm), 4)))


# apply vectorizer
dtm_test = create_dtm(it_test, bigram_vectorizer)

preds =
  predict(glmnet_classifier,
          dtm_test,
          type = 'response')[,1]

glmnet:::auc(test$sentiment, preds)
```

To further improve performance, we can use 'feature hashing' which achieves greater speed by avoiding a lookup over an associative array. Another benefit is that it leads to a very low memory footprint, since we can map an arbitrary number of features into much more compact space, using `text2vec`. The method often makes AUC slightly worse in exchange for improved execution times, which on large collections of documents can provide a significant advantage.

```
h_vectorizer = hash_vectorizer(hash_size = 2 ^ 14, ngram = c(1L, 2L))

t1 = Sys.time()
dtm_train = create_dtm(it_train, h_vectorizer)
print(difftime(Sys.time(), t1, units = 'sec'))


t1 = Sys.time()
glmnet_classifier = cv.glmnet(x = dtm_train, y = train[['sentiment']],
                                 family = 'binomial',
                                 alpha = 1,
                                 type.measure = "auc",
                                 nfolds = 5,
                                 thresh = 1e-3,
                                 maxit = 1e3)
print(difftime(Sys.time(), t1, units = 'sec'))


plot(glmnet_classifier)


print(paste("max AUC =", round(max(glmnet_classifier$cvm), 4)))


dtm_test = create_dtm(it_test, h_vectorizer)

preds =
  predict(glmnet_classifier,
          dtm_test ,
          type = 'response')[, 1]

glmnet:::auc(test$sentiment, preds)
```

Before analysis it can be useful to transform DTM, using the `normalize` function, since lengths of the documents in collection can significantly vary. Normalization transforms the rows of DTM so we adjust values measured on different scales to a not to transform rows so that the sum of the row values equals 1. There is also a `TfIdf` function which not only normalizes DTM, but also increase the weight of terms which are specific to a single document or handful of documents and decrease the weight for terms used in most documents. Both can further improve execution time and accuracy, which may be of value in very large datasets.