Accessing the SEC's EDGAR Database of Financial Information

One of the first steps in planning and analytical review is a review of current and prior year filings with the SEC. These will include annual and quarterly financial statements, restatements, proxy statements, lawsuits, and numerous other documents, where their acquisition and incorporation into workpapers is an essential prerequisite of audit planning. Fortunately, complete information is available on the SEC's website at sec.gov. Many of the most relevant documents to an audit are maintained by the SEC in XBRL format (as .XML files) which can be downloaded into the working papers from the Internet. XBRL is *eXtensible Business Reporting Language*, a freely available, global markup language for exchanging business information. XBRL allows the expression of semantic meaning, which lends to an unambiguous definition of accounts and other financial information. XBRL representations of financial reports are more reliable and less subject to misinterpretation than are disseminations in other formats. XBRL also allows for automated parsing of information, which can greatly improve the efficiency of audit ratio and statistical analysis.

The following code chunk accesses the SEC's XBRL databases to acquire current and prior year filings for any listed company, and read it as a dataset that can be manipulated by R. For this example, we extract General Motors' 2016 and 2017 financials from the EDGAR database at sec.gov. I use the `finstr` package to access EDGAR files.

```
library(devtools)
install_github("bergant/finstr")
library(finstr)
library(XBRL)

# Locate the XBRL format 10-K reports for years 2016 and 2017
# Search sec.gov for General Motors, choose the 10-K for the year

# Note that .xml is the XBRL file indicator)

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"

# Get EDGAR data in XBRL format from the sec.gov site
# parse XBRL (GM 10-K reports)
old_o <- options(stringsAsFactors = FALSE)
xbrl_data_2016 <- xbrlDoAll(xbrl_url2016)
xbrl_data_2017 <- xbrlDoAll(xbrl_url2017)
options(old_o)

## With xbrl_get_statements convert sec.gov's XBRL data to a list of lists

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

```
st2017
```

The 10-K XBRL file is a list of four lists -- balance sheet, income statement, cash flow and comprehensive income. Content and names used will vary widely from company to company, and from particular filing or statement. Thus it is important to inspect the files that are retrieved to determine the correct variable names, data formats and structure of each XBRL file prior to any subsequent analysis. Once the financial statements are loaded into the R session, the `finstr` package contains several commands to check consistency of the reports and to display data in a format suitable for auditing. The following code chunks provide examples of some of the most useful commands in the `finstr` package.

```
library(tidyverse)
library(kableExtra)

## To get a single statement, assign one of the four lists, e.g.

balance_sheet2017 <- st2017$ConsolidatedBalanceSheets
balance_sheet2016 <- st2016$ConsolidatedBalanceSheets
income2017 <- st2017$ConsolidatedIncomeStatements
income2016 <- st2016$ConsolidatedIncomeStatements

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

capture.output(
  bs_table <-
    print(
      balance_sheet2017,
      html = FALSE,
      big.mark = ",",
      dateFormat = "%Y"),
  file='NUL')

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

Planning review looks for changes from prior years, or trends that may be important in the current year's audit. The `merge()` command consolidate the information from different .XML files into single files.

```
library(tidyverse)
library(kableExtra)

## Use merge function to create single financial statement data from two
statements.

balance_sheet <- merge(balance_sheet2017, balance_sheet2016)

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

capture.output(bs_table <-
                 print(balance_sheet2017,
                       html = FALSE,
                       big.mark = ",",
                       dateFormat = "%Y"),
               file='NUL')

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

The `check_statement()` command in `finstr` will automatically validate internal consistency of transaction lines and summary lines in the EDGAR filings.

```
library(tidyverse)

##  Recalculate higher order concepts from basic values and check for
errors.

check <- check_statement(balance_sheet2017)
check

## In case of error the numbers with errors will be presented along with
elements:

check_statement(
  within(balance_sheet2017, InventoryNet <- InventoryNet * 2)
)
```

```
## validation returns all calculation results in a readable data frame.
## e.g., operating income from income statement:

check <- check_statement(income2017, element_id = "OperatingIncomeLoss")
check

check$expression[1]

check$calculated / 10^6
```

Rearranging statements is often a useful step before actual calculations. Rearrangements can offer several advantages in ad hoc analyses such as analytical review:

- We can avoid errors in formulas with many variables,
- Accounting taxonomies do change and using many formulas on original statement is harder to support than using custom hierarchy for analysis starting point,
- When sharing analysis it is easier to print fewer values.

To rearrange the statement to simple two-level hierarchy use the `expose` function.

```
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"
)
```

Here, the balance sheet stays divided by assets, liabilities and equity. For the second level we are `exposing` current assets from noncurrent and similarly for the liabilities. We choose to separate equity.

Function `expose` expects a list of vectors with element names. Function `other` helps us identify elements without enumerating every single element. Using `other` reduces potential errors, as the function knows which elements are not specified and keeps the balance sheet complete.

Sometimes it is easier to define a complement than a list of elements. In this case we can use the`%without%` operator. Lets expose, for example, tangible and then intangible assets:

```
library(tidyverse)
library(kableExtra)

expose( balance_sheet,

  # Assets
  `Tangible Assets` =
    "Assets" %without%
c("AssetsOfDisposalGroupIncludingDiscontinuedOperationCurrent",
"NotesAndLoansReceivableNetCurrent","gm_AssetsSubjecttoorAvailableforOperat
ingLeaseNetCurrent"),
  `Intangible Assets` = other("Assets"),

  # Liabilites and equity
  `Liabilities` = c("Liabilities", "CommitmentsAndContingencies"),
  `Stockholders Equity` = "StockholdersEquity"
)

## To calculate lagged difference for entire statement use diff function.
## The result is statement of changes between successive years

diff_bs <- diff(balance_sheet)


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

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

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

These are the basic tools that you need to access the information on sec.gov. Note that there are numerous reports on EDGAR; `finstr` will be able to access and format any financial statements in XBRL format on the EDGAR database. Almost all of the EDGAR information is maintained in HTML format, and I will provide code later in this chapter to access and parse HTML files in EDGAR.

Audit Staffing and Budgets

The audit program lays out in advance of mid-year and year-end tests, the procedures that will be used to collect evidence and to analyze it with the objective of reporting the ‘correct’ audit opinion, while keeping costs within the contracted audit budget. This section provides an example of an audit program that might be created after the risk assessment. In addition to auditing steps on samples drawn from specific computer files, the example demonstrates the sort of results that the audit would produce, and describes the corrective steps or reporting that would accompany the audit results.

Audit budgeting, which primarily is determined by the allocation of audit staff, is too often made in an *ad hoc* manner. Prior years' budgets and assignments influence staffing; so does availability of knowledgeable staff. Human resource problems, especially in specialized, knowledge intensive industries such as auditing, will never be an exact science. Nonetheless, management should attempt to instill a reasonable level of cost-benefit discipline in staffing decisions.

At the planning stage, audit managers will determine the scope of each audit test. From a statistical perspective, this can be estimated as a cost proportional to the sample sizes that are decided on for the tests. From a staffing perspective, this is proportional to the number of auditors assigned to the audit tasks. Similarly, the benefit derived from that expenditure can be perceived in terms of the monetary error that could be detected; typically a percentage of the value of the account or years transaction stream.

Audit programs are collections of audit tests that test different critical transaction and systems processing features in the client's accounting systems. Their exact nature is considered elsewhere in this book. For this section, I assume that audit planning requires for each of the individual audit tests is set to a scope which overall maximizes the audit risk reduction for a given cost of conducting the audit with this program.

Staffing is 'lumpy' in the sense that you typically get whole auditors assigned to an audit. The number of auditors assigned to an audit will be commensurate with their potential for detecting monetary error -- the benefit received from an audit test. We would hope to see the following sort of relation between the potential benefit derived from performing a set of audit tasks, and the person-months of audit staff assigned assigned to a specific audit program (and thus the cost).

```
library(ggplot2)

benefit <- seq(10, 10000, 10)
staff_allocated <- data.frame(benefit, floor(10*(log(benefit^.06))))

ggplot(staff_allocated,
       aes(staff_allocated[,1],staff_allocated[,2])  ) +
  geom_line() +
  labs(x ="Audit Risk Reduction",
       y ="Staff Auditor Person-Months")
```

The Risk Assessment Matrix

A Risk Assessment Matrix is constructed prior to and during the analytical review phase of the audit. During this phase, the auditor will scan any business intelligence from media and internet sources that would be relevant to potential risks to be encountered in the audit of the client. Prior years' working papers will also be perused for ascertain experiences, client specific risks and application of the "rule of 3's" to adjust any anticipated risks and expectations during the current year audit.

Audit firms tend to enforce firm specific procedures in auditing. Each of the “Big 4” audit firms (which audit almost all of the listed firms on US exchanges) displayed unique biases in rendering adverse attestations: Ernst & Young focused on accounts receivables, revenue recognition, taxes and fixed assets; PricewaterhouseCoopers focused on accounts receivables, revenue recognition, taxes and payables; KPMG focused on accounts receivables, revenue recognition, taxes and inventory; and Deloitte & Touche focused on revenue recognition, taxes, liabilities, inventory and executive compensation. These biases are likely to reflect signature audit methods, internal forms and checklists, and audit histories that are unique to individual firms. These firms will consequently allocate larger portions of the audit budget to certain accounts at the expense of others. Additionally, auditors tend to allocate more time to auditing debit balance accounts, assuming double entry will assure the accuracy of the credit accounts. The specific accounts selected for audit depend on firm policy, procedures and managing partners.

In this section, I will show how to construct a Risk Assessment Matrix on a client-server dashboard. Dashboards are well suited to auditing -- they accommodate the information needs of auditors (and their laptops) in the field, while assuring the security, integrity, completeness and privacy of client and audit records behind firewalls is well suited to auditing. The Risk Assessment Matrix will assume we are planning the audit of the simulated system presented in chapter 13.

```
library(knitr)
#library(kableExtra)

knitr::include_graphics(

"/home/westland/audit_analytics_book/aaa_chapters/pictures/risk_asst_matrix
.png",
    dpi = 300)
```

Using Shiny to create a Risk Assessment Matrix Dashboard

Auditors face a particular problem in the field, in that much of the information they need may be in prior years' workpapers, in proprietary client files, in central locations in the audit firm, behind fire walls and on powerful servers or cloud platforms. To secure the clients records and address privacy concerns, it is important that only the necessary information be maintained on mobile platforms such as laptops that are used in the field.

The standard solution to such problems in client-server systems, that place high-performance, secure systems on a centralized server, and provide the field auditor with light, client software that runs on a laptop, communicating with the server over the internet. Shiny is the client-server extension of the R language, and as with the rest of the R language, is uniquely suited to handling the *ad hoc* nature of audits, where each audit often represents an entirely new set of analyses.

Shiny is a tool for fast prototyping of digital dashboards, giving you a large number of HTMLWidgets at your disposal which lend themselves really well to building general purpose web applications. Shiny is particularly suited for fast prototyping and is fairly easy to use for someone who's not a programmer. Dashboards locally display some data (such as in a database or a file) providing a variety of metrics in an interactive way.

Reactive programming starts with reactive values that change in response to user input (such as positioning the confidence and cost sliders) and builds on top of them with reactive expressions that access reactive values and execute other reactive expressions. Reactivity based code for the Risk Assessment Matrix dashboard appears below. This has two parts: (1) the user interface 'ui' which conceivably would operate be on the field auditor's laptop, and (2) the server-side 'server' operations that would take place at the audit firms headquarters behind a firewall, and with access to firm and client files. First I will look at the user interface.

```


# Define the User Interface (UI)
ui <- fluidPage(


  titlePanel("Risk Assessment Matrix"),

  sidebarLayout(

    sidebarPanel(

      # Input: statistical confidence level of the audit tests
      sliderInput("confidence", "Confidence:",
                  min = .7, max = .999,
                  value = .95),

      # Input: cost of auditing per transaction sampled
      sliderInput("cost", "Audit $ / transaction:",
                  min = 0, max = 500,
                  value = 100),


      # Input: Text for providing a caption for the RAM
      textInput(inputId = "caption",
                label = "Client:",
                value = "XYZ Corp.")

    ),


    # Main panel for displaying outputs
    mainPanel(

      # Output: slider values entered
      tableOutput("values"),

        # Output: Formatted text for caption
        h3(textOutput("caption", container = span)),

        # Output: total cost of the audit
        textOutput("view"),

       # Output: RAM summary with sample sizes (scope) and cost
        verbatimTextOutput("summary"),



      h6("Risk choices are:  1 = Low,  2=Medium, 3=High"),
      h6("Risk_intel = the risk level indicated by business intelligence
scanning"),
      h6("Risk_prior = the risk level indicated by audits in prior years"),
      h6("Account Amount and the Ave. Transaction size are in $ without
decimals or 000 dividers"),
      h6("Scope = estimated discovery sample size that will be needed in
the audit of this account"),
      h6("Audit cost = audit labor dollars per sampled transaction"),
      h6("Confidence = statistical confidence")


    )
  )
)
```

The mathematics of scope assessment takes place on the server. I used a very simple 'discovery sampling' inspired model (see chapter 7) to compute audit scope which I interpret as sample sizes for various transaction flows, computed as:

$$n \approx \frac{log(1-confidence)}{log(1-\frac{10-risk_{intelligence} \times risk_{prior}}{100})}$$

These are dynamically (reactively in the Shiny vernacular) updated for changes in confidence level and transaction auditing costs established by the auditor. A total audit cost of field tests is computed, to be incorporated into the overall budget of the audit.

In practice, the server side of the Risk Assessment Matrix will have access to prior years working papers (assuming they are digitized) and to client accounting files, as well as proprietary audit firm data and technologies.

```
# Define Server-size calculations
server <- function(input, output) {

  # auditors risk assessment matrix generated from prior years' workpapers,
etc.
  ram <-
```

```
read.csv("~/audit_analytics_book/aaa_chapters/tables/risk_asst_matrix.csv")


  # Reactive expression to create data frame of slider input values
  sliderValues <- reactive({

    data.frame(
      Audit_Parameter = c("confidence",
               "cost"),
      Value = as.character(c(input$confidence,
                             input$cost)),
      stringsAsFactors = FALSE)
      })


  # Show the values in an HTML table ----
  output$values <- renderTable({
    sliderValues()
  })


  output$caption <- renderText({
    input$caption
  })


  # Recompute scope and cost whenever input$confidence or input$cost change

  output$summary <- renderPrint({
    ram <- ram
    conf <- input$confidence
    cost <- input$cost
    risk <- (10 - (as.numeric(ram[,2]) * as.numeric(ram[,3])) )/100
    Scope <-  ceiling( log(1-conf) / log( 1- risk))
    ram <- cbind(ram[,1:5], Scope)
    Min_cost <- Scope * cost
    ram <- cbind(ram[,1:6], Min_cost)
    ram
  })

 # Recompute minimum audit cost whenever input$confidence or input$cost
change

  output$view <- renderText({
    ram <- ram
    conf <- input$confidence
    cost <- input$cost
    risk <- (10 - (as.numeric(ram[,2]) * as.numeric(ram[,3])) )/100
    Scope <-  ceiling( log(1-conf) / log( 1- risk))
    ram <- cbind(ram[,1:5], Scope)
    Min_cost <- Scope * cost
    minimum_audit_cost <- sum(Min_cost)
    c("Minimum estimated audit cost = ",minimum_audit_cost)
  })
  }
```

R Studio gives you various options for assembling Shiny apps, including apps with server side code resident on either an RStudio or a bespoke server, and stand-alone client side apps which can be constructed with the following code.

```
#
#  See http://shiny.rstudio.com/ for documentation
#

library(shiny)

# Define UI for application
 ** Insert UI Code Here **

# Define server logic required to draw a histogram
 ** Insert Server Code Here **

# Run the application
shinyApp(ui = ui, server = server)
```

Sample Sizes for Budgeting

There are two types of sampling in interim tests:

1. Discovery sampling for discovery of *out-of-control* transaction streams
2. Attribute sampling for estimating transaction error rate

Discovery sampling sets a sample size that is likely to discover at least one error in the sample if the actual transaction error rate exceeds the minimum acceptable error-rate (alternatively called the out-of-control* rate of error). Discovery tests helps the auditor decide whether the systems processing a particular transaction stream are in or out of control. Budgeted sample sizes in interim testing will depend on whether the RAM suggests that control risk is low or high. If it is low, then the discovery sample size plus a 'security' factor for cases where error is discovered will estimated the scope of auditing.

```
 confidence <- seq(.99,.7,-.01)
 n <- (log(1-confidence))/log(1-.05)
 plot(confidence,n, type="l")
```

So for a 5% intolerable error rate at 95% confidence we have:

```
confidence <- .95
n <- (log(1-confidence))/log(1-.05)
cat("\n Discovery sample size = ", ceiling(n))
```

Where the RAM assesses control risk to be anything higher, the auditor can assume that scope will be expanded to include attribute sampling. Attribute sampling estimates the error rate in the entire transaction population with some confidence (e.g., 95%) that the estimate is within the out-of-control error-rate cutoff for that transaction stream. If it is found that a particular transaction stream is out of control, then attribute estimation will help us decide on the actual error rate of the systems that process this transaction stream. Errors estimates from attribute samples may either be *rates* or *amounts* or both.

If discovery sampling suggests that a particular transaction stream is out of control, then attribute estimation will help us decide on the actual error rate of the systems that process this transaction stream. Attribute sampling size is determined using Cohen's power analysis which is implemented in R's *pwr* package, We compute both in the following code chunk

```
library(readr)
library(pwr)


## Attribute sample for estimating 'rate' of errors

size <- 1000                                    ## number of transactions
Delta <- .05*size                               ## detect 5% occurence
error
sigma <- .3*size                                ## variability (guess ~1/3
rd)
effect <- Delta/sigma



sample <- pwr.t.test(
  d=effect,
  sig.level = 0.05,
  power = .8,
  type="one.sample",
  alternative="greater")                       ## look for overstatement of
earnings

cat("\n Attribute sample size for occurrence of error = ",
ceiling(sample$n))



## Attribute sample for estimating 'amount' of errors
```

```
size <- 100000                                    ## total amount of
transactions
mu <- 50                                          ## average value of
transaction
Delta <- .05*mu                                   ## detect 5% amount
intolerrable error
sigma <- 30                                       ## variability
effect <- Delta/sigma


sample <- pwr.t.test(
  d=effect,
  sig.level = 0.05,
  power = .8,
  type="one.sample",
  alternative="greater")   ## look for sales value too large

cat("\n Attribute sample size for amount of error = ", ceiling(sample$n))
```

The auditor faces different decisions in substantive testing. The particular type of account determines the impact of control weaknesses found in interim testing. For example, a 5% error rate in a $1 million sales account discovered in interim testing implies a $50,000 error in annual sales on the trial balance. In contrast, assume that accounts receivable turn over 10 times annually, then that 5% error rate implies only a $5000 misstatement in accounts receivable. Whether sales or accounts receivable are 'fairly stated' depends on the immateriality level set by the auditor -- a $10,000 materiality level would imply that sales is not fairly presented, while accounts receivable is fairly stated.

At year-end where there will be a complete set of transactions available for the year, and substantive samples are typically focused on acceptance sampling to determine of the account balance is 'fairly stated' (does not contain intolerable or material error). The approach is the same as attribute sampling of amounts, and is inherently more straightforward than interim control tests. Substantive tests estimate the error rate in an account balance with some confidence (e.g., 95%) that the estimate is within the 'materiality' or 'intolerable error' cutoff for that account balance

For example, consider sampling sales invoices from the accounts receivable aging report and comparing them to supporting documentation to see if they were billed in the correct amounts, to the correct customers, and on the correct dates. Additionally, auditors might trace invoices to shipping log, and match invoice dates to the shipment dates for those items in the shipping log, to see if sales are being recorded in the correct accounting period. This can include an examination of invoices issued after the period being audited, to see if they should have been included in a prior period.

Acceptance sampling size is determined using Cohen's power analysis which is implemented in R's *pwr* package, If discovery sampling suggests that a particular transaction stream is out of control, then attribute estimation will help us decide on the actual error rate of the systems that process this transaction stream. Errors estimates from attribute samples may either be *rates of erroneous transactions* or from a monetary unit sampling perspective, can be *rates of monetary error in the transaction stream.* We compute both in the following code chunk

```
library(readr)
library(pwr)


## Acceptance sample for estimating 'amount' of error in an account balance


size <- 100000                                         ## Account balance
mu <- 50                                               ## average value of account
transaction
Delta <- .05*mu                                        ## detect 5% amount
sigma <- 30                                            ## variability
effect <- Delta/sigma


sample <- pwr.t.test(
  d=effect,
  sig.level = 0.05,
  power = .8,
  type="one.sample",
  alternative="greater")   ## look for value too large

cat("\n Attribute sample size for amount of error = ", ceiling(sample$n))
```