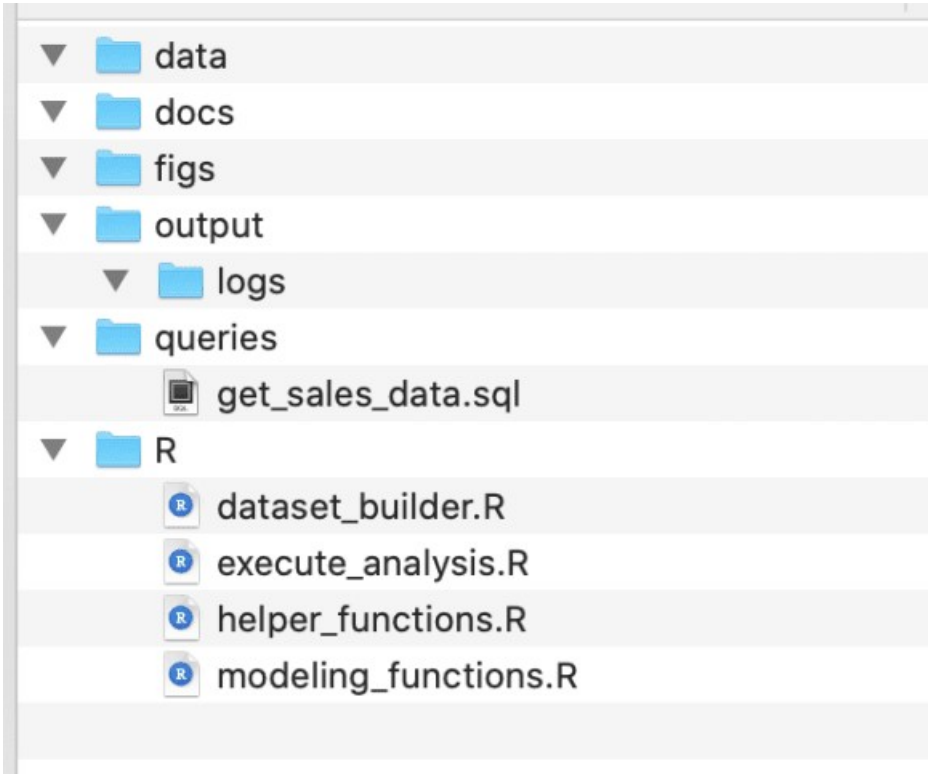
**A. Introduction**

Having completed some sort of data analysis, we often want to automate that process so that it will be executed at regular intervals. What that means is that code must to generated so that data acquisition, data cleaning, model development, document creation, and other components are fully executed from start to finish without any intervention from a human. In this post, I will share a very simple pipeline example that shows how we can use the R programming language to automate an analysis.

The underlying analysis that we are trying to automate in this post involves extracting data from a SQL database, running a couple time series forecasting models, and exporting the results as a csv file. For this task, there are several components that need to be created. The following R files will split the pipeline into very specific components that will execute particular parts of the process.

- helper_functions.R: This file would contain a number of functions for extracting the raw data, cleaning data, modifying strings, and so forth.
- modeling_functions.R: This file would contain a number of functions for each time series forecasting model that we plan on using.
- dataset_builder.R: This would contain the process for extracting the raw data from SQL and turn it into the processed data that we can feed into our forecasting models. Within this file, we would source in the contents of the helper functions file to execute particular tasks.
- execute_analysis.R: This would contain the main code to execute all the code. It would import the desired packages, construct the data set, perform modeling, and export the results to a directory. Within this file, we would use the helper functions, modeling functions, and dataset builder code to complete the task.

The file structure for such an endeavor would look as follows. Note that there is a queries sub directory that will contain a sql file that will be used in the dataset_builder file to pull the raw data from the database.

- 


**B. Code Overview**

To better understand what goes into our simple pipeline, let us go over every file and its contents.

**B1. helper_functions.R**

The main contents of this file are two user defined functions that will be used to parse through a sql file and then use it to export data from a database. The first of these user defined functions is get_sql_query, which parses through a .sql file and collects its contents as a string variable. That parsed query will be used as an argument in get_sql_data, which connects to a sql server database and extracts the desired data.

```
##############################################################################
### FUNCTION TO EXTRACT QUERY FROM A SQL FILE
```

```
get_sql_query <- function(filepath){

  con = file(filepath, "r")
  sql.string <- ""

  while (TRUE){
    line <- readLines(con, n = 1)

    if ( length(line) == 0 ){
       break
    }

    line <- gsub("\\t", " ", line)

    if(grepl("--",line) == TRUE){
        line <- paste(sub("--","/*",line),"*/")
    }

    sql.string <- paste(sql.string, line)
  }

  close(con)
  return(sql.string)

}

################################################################################
### FUNCTION TO EXTRACT DATA FROM SQL

get_sql_data <- function(driver = "SQL Server",
                         server = "sdl02-vm161",
                         db_name = "OpsDW",
                         query_text = which_query){

    con = dbConnect(odbc(),
                    Driver = driver,
                    Server = server,
                    Database = db_name,
                    Trusted_Connection = "True")

    temp <- dbSendQuery(con, which_query)
    dat <- dbFetch(temp)

    return(dat)

}

################################################################################
```

**B2. modeling_functions.R**

The main contents of this file are user defined functions that will be used to train the univariate forecasting models on our data. For each algorithm that we plan to use, I have created separate functions that take a variety of arguments. The functions require the user to specify the training and test data, how many values to forecast, the columns of the date values, and so forth.

```
################################################################################
### SIMPLE EXPONENTIAL SMOOTHING

SES_Forecast <- function(full_data = dat,
                         train_data = train_dat,
                         test_data = test_dat,
                         forecast_test_for = nrow(test_dat),
                         forecast_for = 12,
                         use_seed = TRUE,
```

```
                          mod_name = "forc_ses",
                          dim_name = dim_name,
                          dim_value = dim_value,
                          date_column = "date",
                          data_column = "value",
                          train_and_test = TRUE,
                          run_full_model = TRUE
                          ){

    message(paste0("Executing forecast: ", mod_name))

    if (use_seed) set.seed(master_seed)

    if(train_and_test){

        # train and test
        fit_ses <- ses(train_data[, get(data_column)], h=forecast_test_for,
initial="simple")
        ses_test_fcast <- forecast(fit_ses)

        forecast_test_results[[mod_name]] <<- data.table(
                          forc_test_dates = seq.Date(
                            DescTools::AddMonths(train_data[.N, get(date_column)], 1),
                            DescTools::AddMonths(train_data[.N, get(date_column)],
forecast_test_for),
                            by = "month"),
                            forc_ses = as.numeric(ses_test_fcast$mean),
                          model = mod_name,
                          dim_name = dim_name,
                          dim_value = dim_value)

        forecast_test_accuracy_results[[mod_name]] <<- data.table(
                            accuracy(ses_test_fcast$mean[1:forecast_test_for], test_data[,
get(data_column)]),
                            model = mod_name,
                            dim_name = dim_name,
                            dim_value = dim_value)
    }

    if(run_full_model){

        # run full forecast
        fit_ses <- ses(full_data[, get(data_column)], forecast_for, initial="simple")
        ses_full_forecast <- forecast(fit_ses, h=forecast_for)
        dat[.N, get(date_column)]
        forecast_full_results[[mod_name]] <<- data.table(
                            forc_dates = seq.Date(
                                  DescTools::AddMonths(full_data[.N, get(date_column)], 1),
                                  DescTools::AddMonths(full_data[.N, get(date_column)],
forecast_for),
                                  by = "month"),
                            forc_ses = as.numeric(ses_full_forecast$mean),
                            model = mod_name,
                            dim_name = dim_name,
                            dim_value = dim_value)
    }
}

##############################################################################
### AUTO ARIMA

AA_Forecast <- function(full_data = dat,
                          train_data = train_dat,
                          test_data = test_dat,
                          forecast_test_for = nrow(test_dat),
```

```
                          forecast_for = 12,
                          use_seed = TRUE,
                          mod_name = "forc_aa",
                          dim_name = dim_name,
                          dim_value = dim_value,
                          date_column = "date",
                          data_column = "value",
                          train_and_test = TRUE,
                          run_full_model = TRUE
){

  message(paste0("Executing forecast: ", mod_name))

  if (use_seed) set.seed(master_seed)

  if(train_and_test){
      # train and test
      fit_aa <- auto.arima(train_data[, get(data_column)])
      aa_test_fcast <- forecast(fit_aa, h=forecast_test_for)

      forecast_test_results[[mod_name]] <<- data.table(
        forc_test_dates = seq.Date(
          DescTools::AddMonths(train_data[.N, get(date_column)], 1),
          DescTools::AddMonths(train_data[.N, get(date_column)], forecast_test_for),
          by = "month"),
        forc_aa = as.numeric(aa_test_fcast$mean),
        model = mod_name,
        dim_name = dim_name,
        dim_value = dim_value)

      forecast_test_accuracy_results[[mod_name]] <<- data.table(
        accuracy(aa_test_fcast$mean[1:forecast_test_for], test_data[, get(data_column)]),
        model = mod_name,
        dim_name = dim_name,
        dim_value = dim_value)
  }

  if(run_full_model){
      # run full forecast
      fit_aa <- auto.arima(full_data[, get(data_column)], forecast_for)
      aa_full_forecast <- forecast(fit_aa, h=forecast_for)
      forecast_full_results[[mod_name]] <<- data.table(
        forc_dates = seq.Date(
        DescTools::AddMonths(full_data[.N, get(date_column)], 1),
        DescTools::AddMonths(full_data[.N, get(date_column)], forecast_for),
        by = "month"),
        forc_aa = as.numeric(aa_full_forecast$mean),
        model = mod_name,
        dim_name = dim_name,
        dim_value = dim_value)
  }
}

##############################################################################
```

**B3. dataset_builder.R**

This file will be an 'analytics script' that extracts the raw data and produces the processed data set that will be used in our algorithms. To do that, we will source in the contents of the helper functions file and use
the sql file in the queries sub directory.

Let's say that there is a table in SQL Server that contains data on each sale at an automotive dealership. The raw data looks like the following.

| sale_id | sale_date | make | model | year | cost | units |
|---|---|---|---|---|---|---|
| 101 | 2020-01-01 | Honda | Civic | 2019 | 20000 | 1 |
| 102 | 2020-01-01 | Honda | CRV | 2019 | 25000 | 1 |
| 103 | 2020-01-01 | Honda | Pilot | 2019 | 15000 | 1 |
| ... | | | | | | |

The get_sales_data file contains the following SQL snippet. All it does is aggregates the total number of units sold by date for each make and model.

```
SELECT
        st.sale_date,
  st.make,
  st.model,
        SUM(st.units) AS sales_cnt
FROM sales_table as st
GROUP BY st.sale_date, st.make, st.model
```

To use this SQL code within out dataset_builder file, we will import the SQL script using the two user defined functions from the helper functions file.

```
source("./R/helper_functions.R")

which_query <- get_sql_query("./queries/get_sales_data.sql")

sdat <- get_sql_data(driver = "SQL Server",
                     server = "sdl02-vm161",
                     db_name = "OpsDW",
                     query_text = which_query)
```

This will generate a data frame that contains the following data.

| sale_date | make | model | sales_cnt |
|---|---|---|---|
| 2020-01-01 | Honda | Civic | 10 |
| 2020-01-02 | Honda | Civic | 15 |
| 2020-01-03 | Honda | Civic | 12 |
| 2020-01-04 | Honda | Civic | 18 |

The dataset builder file would also contain code that cleans values, aggregates by month, and other necessary actions. For example, the dataset builder may also contain the following code so that we only have monthly sales data for Subaru Forester sales.

```
mydat <- copy(sdat)

mydat[, sales_date_month := floor_date(sales_date, "month")]

mydat <- mydat[make == "Subaru" & model == "Forester",
                               .(sales = sum(sales_cnt)),
                               by = .(sales_date_month)]
```

**B4. execute_analysis.R**

This final file will execute the entire process from start to end. It should start with a number of parameters that will determine how the script will run. For example, we would want to specify the packages that need to be imported, a logical variable to determine whether the output should be saved, and so forth.

```
##############################################################################################
#######
### SET PARAMETERS

MAIN_PATH = "K:/Data Excellence/4 - Reducing Direct Clinical Costs/ATM/NP_Supply/NP_Supply_
Estimation_V3/"

USE_THESE_PACKAGES <- c("dplyr", "ggplot2", "data.table", "lubridate",
```

```
                        "stringr", "forecast")

GET_LATEST_DATA <- TRUE

SAVE_OUTPUT = TRUE
```

These type of parameters will dictate how our main script will execute. For example, the vector with packages will be fed into an if else statement whereby uninstalled packages will be installed and then all required packages will be imported. Once this parameter has been created with the abstracted code, things can be modified and the script will still run as desired.

```
#############################################################################################
#######
### IMPORT PACKAGES

new_packages <- USE_THESE_PACKAGES[!(USE_THESE_PACKAGES %in% installed.packages()[,"
Package"])]

if(length(new_packages)) {
  install.packages(new_packages)
  sapply(use_these_packages, require, character.only = TRUE)
} else {
  sapply(use_these_packages, require, character.only = TRUE)
}
```

Another parameter was get_latest_data, which is a logical variable that will be used to used to run the dataset_builder file when it is set to TRUE. When the get latest data parameter is set to true, the dataset_builder.R file will be sourced in and produce the final processed data that we plan to use in
our analysis. The resulting data set will then be saved as a csv file with the date within the file name.

```
#############################################################################################
#######
### IMPORT DATA

log_info("execute_analysis: Importing raw data....")

if(GET_LATEST_DATA){
  source("./R/dataset_builder.R")
  fwrite(mydat_base, paste0("./data/final_contact_data_NY_", TIMESTAMP_SAVE, ".csv",
sep=""))
} else {
  mydat_base <- fread("/Users/abrahammathew/Desktop/work_stuff/consent/
final_contact_data_NY_20200707.csv")
}

mydat_base[1:2]
dim(mydat_base) # 5823378

log_info("execute_analysis: The dataset includes {nrow(mydat_base)} rows")
log_info("execute_analysis: The dataset includes data on {length(unique(mydat_base$
MemberACESID))} members")

data_to_score <- copy(mydat_base)

#############################################################################
```

Later in the execute analysis script, we would run each of the forecasting models and export the results as a csv file. Since the goal is to execute the process and save the results, the script will conclude with the following lines.

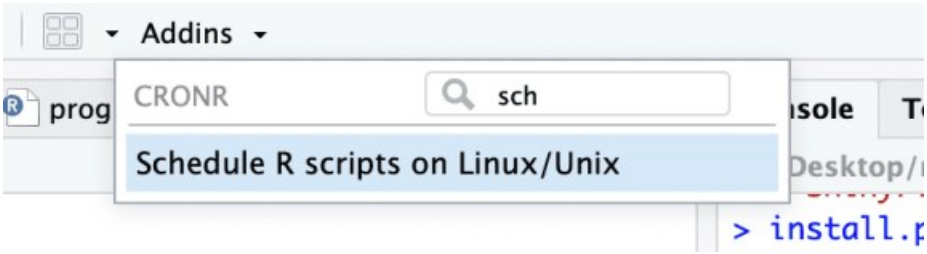
```
#############################################################################################
#######
### EXPORT RESULTS

if(SAVE_OUTPUT){

  log_info("Export final results")
```

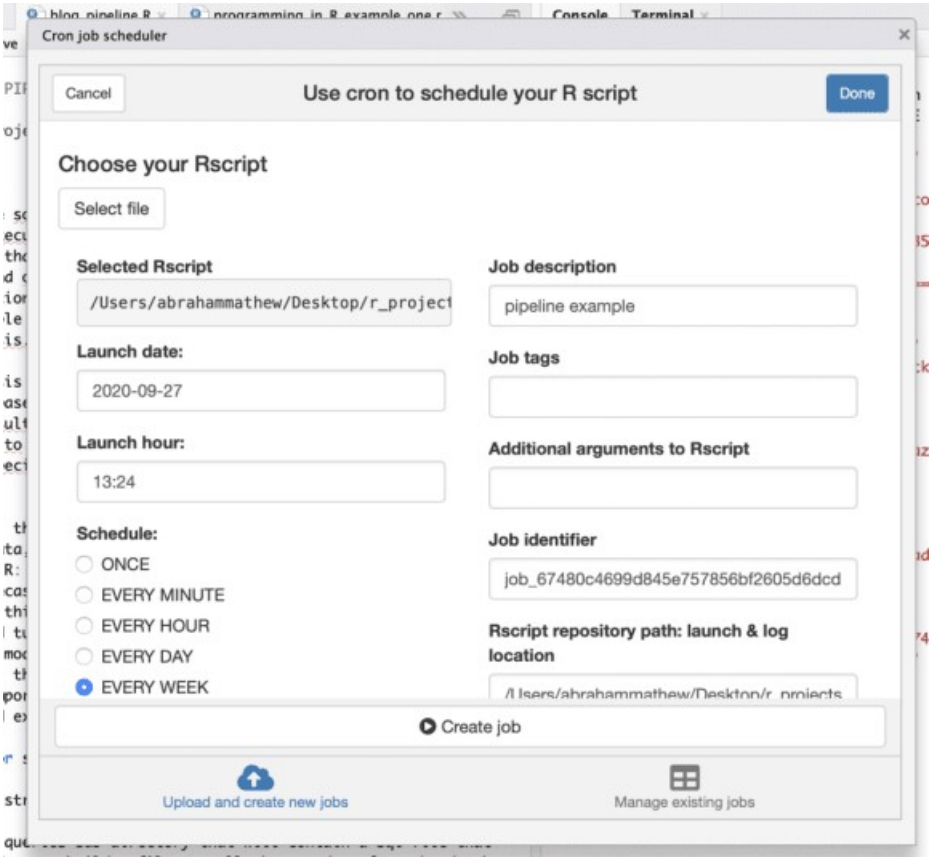
```
  fwrite(final_output,
         paste0("K:/Data Excellence/4 - Reducing Direct Clinical Costs/ATM/NP_Supply
/NP_Supply_Estimation_V3/output/",
                "Supply_Estimates_Using_Averages_by_GeoZone_",
                str_replace_all(Sys.Date(), "[^[:alnum:]]", ""),
                ".csv", sep=""))

}

log_info("Script completed...")

################################################################################
```

**C. Schedule Script**

Now that we have constructed the desired code and process, we need to schedule this code. Remember, the goal is to have this process run every week without having any intervention from a human. For those using Windows computers, the simplest approach would be to use the task scheduler addin that is available
via RStudio. Mac users can run cron jobs via the cronR package or 'cron job scheduler' addin in RStudio.

Here are screenshots of the cron job scheduler in RStudio and how a process can be scheduled.

- 

- 


This post will not dive deeper into CronR, but basic steps to use the package are available here .

**D. Conclusion**

This post provided an example of how to build a very basic pipeline using the R programming language. …