**Analytics mindset**

**Extract, Transform and Load (ETL) in R**

**Background**

Many business analysts claim that 80% of their job is extracting data from various systems, transforming it so that it can be analyzed and loading it into the analysis tool that will be used for analysis. These procedures are typically called extract, transform and load (ETL) procedures, but they can also be referred to as data wrangling, data munging or just cleaning data, among others. Getting data ready to analyze takes significant time, resources and expertise. This case will give you hands-on practice with the ETL process.

In this case, you will be performing ETL procedures on transactional data for Swift Arrow convenience stores (Swift). Swift operates in over 160 cities in the US and has over 180 stores.Note that while Swift is a fictional company, the data is based on a real company. Modifications have been made only to store addresses (e.g., store address information is oriented to a post office rather than the actual store address).

# **Tools**

The programming language R is commonly used for data analytics, and particularly for statistical analysis. It is generally seen as a relatively “learnable” programming language. There are many ways to use R. We recommend the integrated development environment (IDE) RStudio. This case assumes basic knowledge of R and RStudio. Your instructor will give you further guidance regarding your tools to use for this case.

# **Data**

Your data set for this case is provided in two CSV files:

1. **Swift\_Transactions.csv**
2. **States.csv**

The Swift file includes a random selection of customer transactions from Swift’s convenience stores for the years 2017 through 2019. These transactions include things like purchases of fuel, lottery tickets, soda and candy. Though the provided data set is relatively large — including over a million transactions — it only includes a very small sample of the transactions during that period. A full data set would be over 145 times larger than this and would consist of hundreds of millions of transactions. A smaller data set is provided to teach basic principles that could be applied to the full data set without requiring you to purchase a very fast computer to process the data in a timely manner.

The following table describes each column contained in the Swift data set.

| **Data field** | **Description** |
| --- | --- |
| unique\_id | The unique identifier for each row that has no duplicates and no nulls/omissions. Each row represents a single product purchased as part of a single transaction. Thus, the data is organized at the product by transaction level. For example, if you were to enter a store and purchase a bag of chips, a hot dog and an air freshener in one transaction, it would be recorded as three unique lines in the data set — one line for the purchase of each product. Example: 1161545 |
| transaction\_id | An identifier that is unique to each transaction and, therefore, has duplicates in  the data set because some transactions have more than one product purchased as referenced by the unique\_ID. Example: 20180417|433|2|1|4888093 |
| unformatted\_date | The unformatted date of the transaction in the years 2017 through 2019. Example: 2018-07-11 |
| customer\_id | A customer that participates in the customer loyalty program is assigned a loyalty card with a number as a unique identifier. When the customer uses this card at purchase, the number is populated for the transaction. Transactions with a null value represent customers that are not loyalty customers or those that made their purchase without providing their loyalty card. Example: 5362 |
| product\_name | The name of the product sold. There are over 9,000 different products. Example: DORITO SMPLY WHT CHDDR 2.5OZ |
| category\_name | The name of the product category. There are over 200 different categories of products. Example: Salty Snacks -tort/corn Chips (152) |
| parent\_name | The name identifier for over 80 parent categories for the products. The parent category is the top of the hierarchy under which is the category name and then the product name. Example: Salty Snacks |
| site\_id | A numeric unique identifier for each store that pertains to the site\_name. There are over 180 different stores. Example: 197 |
| site\_name | The name of the store for the site\_id. Example: 433 Bay Minette |
| address | The street address of the store. Example: 701 Mcmeans Ave |
| city | The city location for the store. Stores are spread across more than 160 different cities within the US. Example: Bay Minette |
| zip | The postal ZIP code for the store. Example: 36507 |
| latitude | The latitude coordinate for the store. Example: 40.1 |
| longitude | The longitude coordinate for the store. Example: -92.4 |
| revenue | The revenue earned for each product. This is typically how much the customer paid for the product (a positive number), but it can also represent how much a customer received for a product (negative number), i.e., a winning lottery ticket. Example: 1.69 |
| costs | The direct cost of the product. Example: 0.51\* |
| gross\_profit | The gross profit of the product calculated by subtracting the costs from the revenue. Example: 1.18\* |
| units | The number of units of the product in the transaction. This is generally one unit with the major exception being fuel, for which units represent the gallons of fuel sold. Example: 1 |
| gp\_margin | The gross profit margin for the product calculated as gross\_profit divided by the revenue. Gross profit margin represents the percentage of profit for each dollar of revenue. It is an important metric for measuring the profitability of the overall business and individual product categories. Example: 0.302\* |
| *\* Note that costs, gross\_profit and gp\_margin have several thousand NAs — missing values. These are generally for abnormal and fairly rare transactions, like prepaid fuel, money orders, coupons, etc.* | |

The States file includes a select list of US states by postal ZIP code. The following table describes each column contained in the data set.

| **Data field** | **Description** |
| --- | --- |
| postal\_code | The postal ZIP code. |
| state\_province | The full name of the state based on the ZIP code. |
| state\_province\_code | The two-letter abbreviated name of the state based on the ZIP code. |
| country\_name | The abbreviated code of the name of the country for the state based on the ZIP code. These are all USA. |

# **Required**

Complete the steps for the required deliverables below. Some deliverables require multiple steps. These deliverables should be completed in executed code and, where required, markdown text in an R Notebook file (a .Rmd file). You have been provided with a template R Notebook file to complete your work, **Analytics\_mindset\_case\_studies\_ETLinR.Rmd.**

This template includes the following for each deliverable or step of a deliverable:

* Text section for the title of each deliverable or step of a deliverable
* Text section to provide your response to any required questions
* Line for your code
* Area for the executed visible code

**Analytics mindset**

**Extract, Transform and Load (ETL) in R**

**Deliverable 1**

In order to start cleaning this data, you need to load the data into the R program you are using. But, before you do that, you need to load a package that you will use later on.

**Required**

* First, load the package “tidyverse,” for later use.
* Next, load the data from the CSV file “Analytics\_mindset\_case\_studies\_ETLinR\_Swift.csv.” Use the function `read.csv().`

**Analytics mindset**

**Extract, Transform and Load (ETL) in R**

**Deliverable 2**

You will now examine the Swift data to get a better understanding of it to help draw initial conclusions about the data in regard to its usefulness and necessary cleansing.

**Required**

Use the following specific functions to examine the Swift data:

* str():
  + This gives the overall structure of the data. This is a common function to use, but note that you could just see this in the environment tab in RStudio on the right side of the screen.
* summary():
  + This gives descriptive statistics of each column in the dataframe.
* head() – show 10 rows:
  + This prints the first few rows of the dataframe.
* tail() – show 10 rows:
  + This prints the last 10 rows of the dataframe.
* slice\_sample() – show 50 rows:
  + This prints a random selection of 50 rows from the dataframe.

After examining the data based on the results of the functions above, provide your answers to the following questions:

1. Can you identify three analyses that you might find beneficial to perform for Swift? Which Swift data columns (also known as features) would you utilize for each analysis?
2. Which aspects of the data do you think need to be cleansed? Explain why.

**Analytics mindset**

**Extract, Transform and Load (ETL) in R**

**Deliverable 3**

The next step is to clean the Swift data so that it is ready to analyze. Relatively speaking, the data provided is fairly clean. For example, each row already contains one unique observation, and each column contains only one item of data (e.g., no lists or vectors). However, there are a few things that still need cleansing.

The column `unformatted\_date` is a name that could be improved. It looks like a date, so why is it called unformatted? It also has the data format of character (chr). This means that `unformatted\_date` holds strings (or, in other words, text characters) and, therefore, the computer will not treat the contents as dates. To be able to analyze this data as a date, you need to convert this column to a date data type. That way, you can perform operations on it, like extracting the year or month. That is, you need to create a new column called `date` that copies the `unformatted\_data` column and converts it into a date object class or `POSIXct/POSIXt` object class so that you can perform operations on it.

To convert this column to a date format, you need to use a function from the “lubridate” package that specifies the current format of the data. That is, you have to know what the strings in the column look like so you can tell the function how to change those to dates.

**Required**

1. Load the lubridate package.
2. To see what these strings look like, use the `str()` function and the `head()` function to look at only the `unformatted\_date` column (not the whole dataframe, just that column).
3. Review your output. It should reveal that the data here is in the format YYYY-MM-DD. The lubridate package allows you to convert the date once you know the text layout, using the correct function that matches that text layout. In this case, the correct function is `ymd()` — which you might guess stands for year, month and day. Other examples would be `mdy()` for month-day-year, `dmy()` day-month-year, etc. Using this function, create a new column called `date` using the `mutate()` function. The `mutate()` function is a handy way to create new variables. Look up how it works within RStudio since you will use it again. Specifically, you need to use two functions, nested together, to create this new variable:

* The `ymd()` function
* The `mutate()` function, e.g., mutate(yourdataframe, newvariable = ymd())

**Analytics mindset**

**Extract, Transform and Load (ETL) in R**

**Deliverable 4**

Now that you have a column that RStudio recognizes as a date, you can easily create additional useful columns that break the date into additional fields, such as year and month.

**Required**

* Using the `mutate()` function and the appropriate functions from the lubridate package, create two new columns — a column called `year` that extracts the year from `date` and a column called `month` that extracts the month from `date.`
  + Make sure your month column returns the months by their abbreviated names (Jan, Feb, etc.) and not by their numbers.
* Finally, look at 10 random rows from the dataframe to make sure you created these two new columns correctly.

**Analytics mindset**

**Extract, Transform and Load (ETL) in R**

**Deliverable 5**

When cleansing your data, it is important to make sure there are no issues with data format, no missing data, data outliers or other data issues that would be problematic. You will practice these skills by examining the revenue in the Swift data, since revenue is critical to any business. Specifically, below you will look for missing values, data format issues and outliers in the revenue column.

**Required**

1. Make certain that the values are all numbers in this column as you expect revenue to be a number. You could check this by looking in the Environment pane/tab in RStudio, but instead use the `str()` function to look at just the `revenue` feature/column. What format is the data in? Write a sentence about how you know that.
2. Make certain that revenue has no missing values and initially assess whether there might be really high or low values, that is, any outliers. You can check that with the `summary()` function. Use the `summary()` function to look at just the `revenue` feature/column.
3. Are there any missing values? Write a sentence about how you learned that answer.
4. What are the maximum and minimum values for the column? Are these in line with your expectations? What additional steps could you take to gain even more perspective about the reasonableness of the values?
5. Further analyze your data set for the products with the highest revenue to make certain the revenue meets your expectations and is not an outlier. Sort the data set in the revenue column and display the first 25 rows of the dataframe after it is sorted so that the highest revenue is at the top (sort the dataframe by the revenue column from highest to lowest). Is there any product that has a revenue value that does not appear reasonable to you? Explain why or why not.
6. When missing data or outliers are identified, you would typically investigate to find out why the revenue is misstated (the root cause of the problem) and then fix the problem in the data. However, for the purposes of this case, you will just eliminate these instead. There are several ways to do this, but for our purposes, just delete all of these rows. To do this, first, create a new dataframe called `df\_clean` and use the `filter()` function to remove this product. The `filter()` function is a useful function from tidyverse that easily subsets the data. Specifically, you can use it to select only a subset of rows based upon a condition. Look it up in the help section to better understand it. Then, use it to select only the rows of the dataframe that do not have this product in them. Finally, check the distribution of the revenue columns now using the `summary()` function again and discuss whether and why it seems more reasonable now.

**Analytics mindset**

**Extract, Transform and Load (ETL) in R**

**Deliverable 6**

Another important task in ETL is to join (or merge) data together. In the Swift data, there are no states in the data set. There are cities and ZIP codes. The states data does have this information and it would be ideal to have the data joined. The steps below will take you through the process of importing the states data set and joining it to the Swift data set so that you have a new column of states that matches the existing ZIP codes in each row in the Swift data set with the ZIP codes in the states data set.

**Required**

1. Use `read.csv()` to load the states data set into a new dataframe. Call the dataframe `df\_states,` Next, use the functions `str(),` `summary(),` `head()` and `tail()` to examine this new dataframe to identify potential problems you might encounter when you join the states data set with the Swift data set. Write a few sentences about any potential problem you identify and describe why it is a problem. Focus on problems you might have when you join this data to the main data set you have been working on, since that is the task you will perform next.
2. Your first step is to put the `postal\_code` column from `df\_states` into the same format as the `zip` column from `df\_clean,` which is `integer.` To do this, create a new column in `df\_states` using the `mutate()` function and the `as.interger()` function to convert the data type. First, tell R which column you are going to create. Then, apply the functions to that column. To be safe, do this to a new column called `postal\_code2.` Thus, use the `mutate()` function to create this column in `df\_states.`

Unfortunately, when you do this you get a warning message that NAs are introduced. If you were to use the `summary()` function to look at the three columns — `postal\_code,` `postal\_code2` and `zip` — you would see that, indeed, over 300 NAs are created. What is going on? If you were to scroll back up to your original view of the `df\_states` dataframe using the `tail()` function, you would see that the reason R made the column a character type in the first place is that some ZIP codes include a dash and four extra numbers, like this:

A screenshot of a computer

Description automatically generated with medium confidence

When you loaded the `df\_states` data, R must not have defined the dash and extra numbers as the number format, but rather the character format. If you look at `df\_clean` using `summary()` for the Swift data set, you don’t see the dash and extra numbers in `zip.` In fact, all numbers are only five digits. These extra four numbers are called “+four codes” and are used to make ZIP codes even more specific. You do not need these and `df\_clean` does not have them.

1. Thus, you are going to need to take off the dash and the extra numbers. Do that with the `str\_sub()` function. That is, use `mutate()` to create a new column called `postal\_code2` `df\_states` (this will replace `postal\_code2` that you just created above). Nested within that `mutate()` function, use `str\_sub()` to keep only the five ZIP code characters (e.g., 80026). This function extracts a substring from each cell in a column. You need to use this function to extract only the ZIP code. For example, you need to enter arguments into the `str\_sub()` function to extract 80026 from 80026-9998 so that all you have left is the five-digit ZIP code 80026. Next, run the `tail()` function and compare `postal\_code` and `postal\_code2` to see that this worked. You should see that `postal\_code` has the extra numbers (e.g., 80026-9998) while `postal\_code2` does not (e.g., 80026).
2. Now that you have only five numbers in `postal\_code2,` you can reperform the `as.integer()` function from step b. Replace the `postal\_code2` column using `postal\_code2` and convert it to an integer. After doing that, run the `str()` and `summary()` functions for `postal\_code2` and make sure things worked. If it works, `postal\_code2` will be in the integer format and will not have NAs.
3. Next, change the name of the `postal\_code2` column to `zip` with the `rename()` function from tidyverse.
4. Now, clean up the dataframe before you join it by reordering and keeping only the columns from `df\_states` that you need. To do this, use another common function from tidyverse, the `select()` function. This function selects which columns you want and don’t want for a new dataframe and reorders them, if desired. Thus, use the `select()` function to overwrite your existing dataframe (to prevent the need to rerun code, only rewrite your dataframe when you are confident your code will do what you want). Keep the `zip` and `state\_province` columns and put `zip` first.

Finally, you are prepared to join your two data frames. Merging or joining (we use these words interchangeably) can be tricky. It is important to think about what you want before merging, and then to make sure you get what you want after merging. To do this, you should do the following:

* Think about how you want your data to look after you join (e.g., if you want all of the rows from both dataframes, only the rows that match up from both dataframes, only the rows from one dataframe).
* Know the level of aggregation in your dataframes.
* Check the number of rows of your dataframes before and after you join to make sure you did what you thought you did.

What does “level of aggregation” mean? By this, we just mean what each row of your data represents. For example, each row of `df\_clean` represents one or more of the same product from one transaction. Thus, the level of aggregation is product-transaction. On the other hand, each row in `df\_states` is a unique ZIP code, so the level of aggregation is ZIP code.

Once you know the level of aggregation of your two dataframes, you can ask the critical question: What level of aggregation do you want when you join these dataframes? You want to just add state names to rows that already exist in `df\_clean,` so your final dataframe should have exactly the number of rows that “df\_clean” has right now. Keep that in mind.

That does not sound so difficult, so what could go wrong? Well, a lot. Look at these potential problems:

* First, maybe `df\_states` has duplicate ZIP codes. If this is true, then you might get duplicate rows in `df\_clean.` You could check this with `n\_distinct(),` but we already have done this and all ZIP codes are unique.
* The next problem is that you use the wrong join method. There are multiple ways to join things, and using the wrong method will cause problems.
* Using the wrong method could add rows that you don’t want because they are in the “df\_states” dataframe but not in the `df\_clean` dataframe. Alternatively, using the wrong method could take away rows from the `df\_clean` dataframe because they are not in the state dataframe. Remember, you want to keep everything in the `df\_clean` dataframe and add just states from the state dataframe.

There are many ways to join two dataframes in R. Use the methods from tidyverse. Search the internet and find the documentation from the package “dplyr” in tidyverse about mutating joins. The mutating joins add columns from the second dataframe to the first dataframe, matching rows based on the keys supplied in the function. Here are the different types you can choose from:

* inner\_join(): includes all rows in the first dataframe and the second dataframe that match up using the variable you are matching on
* left\_join(): includes all rows in the first dataframe and only those rows from the second dataframe that match up with the first dataframe using the variable you are matching on
* right\_join(): includes all rows in the second dataframe and only those rows from the first dataframe that match up with the second dataframe using the variable you are matching on
* full\_join(): includes all rows in the first dataframe and all rows in the second dataframe

Take a minute and think about which one of these methods you want to use:

1. Use the wrong join type first, just to illustrate why it is important to use the correct join method. Use the `full\_join()` method to join `df\_clean` and `df\_states` and create a new dataframe called `df\_clean\_etl.` Match up these dataframes using the `zip` column. Finally, write a few sentences that discuss why this method of joining did not work correctly. Provide evidence about why it failed.
2. Now, join the data correctly. What join type do you actually want? You want the `left\_join()` method. This includes all rows from the first dataframe and **only** those rows from the second dataframe that match up with the first dataframe using the variable you are matching on. Use the `left\_join()` method to join `df\_clean` and `df\_states` and create a new dataframe called `df\_clean\_etl.` Match up these dataframes using the `zip` column. Finally, write a few sentences to discuss why this method of joining worked correctly.

**Analytics mindset**

**Extract, Transform and Load (ETL) in R**

**Final deliverables**

**Required**

* Submit the following for grading:
  + Save your executed Rmd file with your name included in the naming convention of the file (e.g., ETLinR\_Name.Rmd.).
  + Knit the Rmd file into an html file (Ctrl + Shift + K in RStudio) and submit this for grading.
    - You should include your first and last name in the naming convention of the file (e.g., ETLinR\_First Name Last Name.html.).