Data Wrangling with Tidyverse

Estimated Time Needed: **40 min**

Welcome!

By the end of this notebook, you will have learned the basics of Data Wrangling! Data Wrangling is the process of converting data from the initial format to a format that may be better for analysis.

Table of Contents:

Load Libraries and Data

First, load the tidyverse library.

[ ]:
```
# Load tidyverse
library(tidyverse)
```

The original Airline dataset is hosted on IBM Data Asset eXchange. This sample dataset can be found here. We will be using a subset of the original dataset, which contains just LAX to JFK flights, throughout this course.

Now using the subset dataset link, you can load it and store as a dataframe `sub_airline`:

[ ]:
```
# url where the data is located
url <- "https://dax-cdn.cdn.appdomain.cloud/dax-airline/1.0.1/lax_to_jfk.tar.gz"

# download the file
download.file(url, destfile = "lax_to_jfk.tar.gz")

# untar the file so we can get the csv only
# if you run this on your local machine, then can remove tar = "internal"
untar("lax_to_jfk.tar.gz", tar = "internal")

# read_csv only
sub_airline <- read_csv("lax_to_jfk/lax_to_jfk.csv",
                        col_types = cols('DivDistance' = col_number(),
                                         'DivArrDelay' = col_number()))
```

1. Missing Values and Formatting

Now that we have the data loaded, let's first take a look at the data using the method `head()`.

[ ]:
```
head(sub_airline)
```

As we can see, several NA (not available) appeared in the dataframe; those are missing values which may hinder our further analysis. So, how do we identify all those missing values and deal with them?

In the following sections, we will go over the steps for working with missing data:

- Identify missing data
- Deal with missing data
- Correct data format

1.1 Identify Missing Malues

In R, there are some special symbols to represent special cases in data:

- `NA`: missing values are represented by the symbol `NA` (not available), it is a special symbol in R. Note, that `"NA"` (a string) is not the same as `NA`.
- `NaN`: Impossible values (e.g., dividing by zero) are represented by the symbol `NaN` (not a number).

The missing values in airline dataset are already represented with R's `NA` symbol. We use R's built-in (also called **base R**) functions to identify these missing values. There are two methods to detect missing data:

1. `is.na(x)` : x can be a vector or list, this method returns a vector of TRUE or FALSE depending if the according element in x is `NA` or not. For example `is.na(c(1, NA))` returns `FALSE TRUE`
2. `anyNA(x, recursive = FALSE)` : x can be a vector or list, this method returns TRUE if x contains any NAs and FALSE otherwise. For example `is.na(c(1, NA))` returns `TRUE`

[ ]:
```
is.na(c(1, NA))        #> FALSE  TRUE
is.na(paste(c(1, NA))) #> FALSE FALSE
```

Again, the output for `is.na()` is a vector of logical values where `TRUE` stands for missing value, while `FALSE` stands for not missing value.

[ ]:
```
anyNA(c(1, NA))
```

The output for `anyNA()` is a vector of logical values where `TRUE` stands for at least one missing value in the vector, while `FALSE` stands for no missing values in the vector.

Counting Missing Values

We can quickly figure out the number of missing values in each column. As mentioned above, when using the function `is.na()` , `TRUE` represents a missing value while `FALSE` is otherwise. The method `sum()` counts the number of `TRUE` values.

Let's check how many missing values in `CarrierDelay` column and also check how many missing values in each column:

[ ]:
```
# counting missing values
sub_airline %>%
  summarize(count = sum(is.na(CarrierDelay)))
```

We can use `purrr::map()` to count missing values in each of the columns.

`map()` essentially maps (applies) a function or formula to each given element. In the code below, it is mapping a *formula*, `~sum(is.na(.))` , that sums the NAs to every column in `sub_airline` . Since it is using a *formula*, you will also notice two special operators dot `.` and tilde `~` :

- The tilde `~` separates the left side of a formula with the right side. Normally formulas are two-sided like `y ~ x` , in this case in `map()` , the formula gets converted to a function so it only needs the right side.
- The dot `.` refers to each column in the dataset. If you view the documentation with `?map` , you can see that you use `.` when the function takes in just one parameter (a column in this case).

See `?formula` and `?map` for more information.

[ ]:
```
map(sub_airline, ~sum(is.na(.)))
```

[ ]:
```
# Check dimensions of the dataset
dim(sub_airline)
```

Based on the summary above, "CarrierDelay", "WeatherDelay", "NASDelay", "SecurityDelay" and "LateAircraftDelay" columns have 2486 rows of missing data, while "DivDistance" and "DivArrDelay" columns have 2855 rows of missing data. All other columns do not have missing data.

1. "CarrierDelay": 2486 missing data
2. "WeatherDelay": 2486 missing data
3. "NASDelay": 2486 missing data
4. "SecurityDelay" : 2486 missing data
5. "LateAircraftDelay": 2486 missing data
6. "DivDistance": 2855 missing data
7. "DivArrDelay": 2855 missing data

1.2 Handle Missing Data

How to deal with missing data?

1. Drop data
   a. Drop the whole column
   b. Drop the whole row
2. Replace data
   a. Replace it by mean
   b. Replace it by frequency
   c. Replace it based on other functions

Generally, you should not blindly drop NAs. However if an entire column or almost the entire column contains NAs, then it may be a good idea to leave it out. In our dataset, columns `DivDistance` and `DivArrDelay` are nearly all empty so we will drop them entirely.

Drop the whole column:

- "DivDistance": 2855 missing data
- "DivArrDelay": 2855 missing data

[ ]:
```
drop_na_cols <- sub_airline %>% select(-DivDistance, -DivArrDelay)
dim(drop_na_cols)
head(drop_na_cols)
```

Drop the whole row:

- "CarrierDelay": 2486 missing data
- "WeatherDelay": 2486 missing data
- "NASDelay": 2486 missing data
- "SecurityDelay" : 2486 missing data
- "LateAircraftDelay": 2486 missing data

We see `CarrierDelay` , `WeatherDelay` , `NASDelay` , `SecurityDelay` , `LateAircraftDelay` have the same amount of missing values from the summary. By dropping the missing values in one column will also solve the missing value issues in the others.

[ ]:
```
# Drop the missing values
drop_na_rows <- drop_na_cols %>% drop_na(CarrierDelay)
dim(drop_na_rows)
head(drop_na_rows)
```

We have some freedom in choosing which method to replace data; however, some methods may seem more reasonable than others. In this scenario, we would like to replace missing values with 0.

Convert NA to 0

In the airline dataset, missing data for the different types of delay corresponds to no delay. So, we can replace these NAs with 0 in this case. To do this we use the function:

```
tidyr::replace_na(data, replace, ...)
```

The columns that corresponds with types of delays are CarrierDelay, WeatherDelay, NASDelay, SecurityDelay, and LateAircraftDelay. For example if `CarrierDelay = NA` then this means there

is no delay in Carrier, so the delay in minutes can be changed to 0 or `CarrierDelay = 0`. Let's transform these columns and see the result:

[ ]:
```
# Replace the missing values in five columns
replace_na <- drop_na_rows %>% replace_na(list(CarrierDelay = 0,
                                               WeatherDelay = 0,
                                               NASDelay = 0,
                                               SecurityDelay = 0,
                                               LateAircraftDelay = 0))
head(replace_na)
```

Question #1:

According to the example above, let's try to replace NA in "CarrierDelay" column by the mean value.

[ ]:
```
# Write your code below and press Shift+Enter to execute
```

▼ Click here for the solution.

```
# Calculate the mean vaule for "CarrierDelay" column
carrier_mean <- mean(drop_na_rows$CarrierDelay)

# Replace NA by mean value in "CarrierDelay" column
sub_airline %>% replace_na(list(CarrierDelay = carrier_mean))
```

Good! Now, we obtain the dataset with no missing values.

1.3 Correct Data Format

We are almost there!

The last step in data cleaning is checking and making sure that all data is in the correct format.

Scoped dplyr Verbs

In dplyr, you can add `_all` and `_if` to the end of it's main functions like `mutate`, `filter`, `group_by`, and `summarize`.

dply function	_all	_if
mutate	`mutate_all()`	`mutate_if()`
filter	`filter_all()`	`filter_if()`
group_by	`group_by_all()`	`group_by_if()`
summarize	`summarize_all()`	`summarize_if()`

These are called **scoped** dplyr verbs. The `_all` variant applies an operation on *all* variaibles and the `_if` variant applies an operation to a variable if the given function is `TRUE`.

Now that we understand these different versions of dplyr functions, let's list the data types for each column by using `summarize_all()` and `gather()`:

[ ]:
```
sub_airline %>%
    summarize_all(class) %>%
    gather(variable, class)
```

It is important for later analysis to explore the feature's data type and convert them to the correct data types; otherwise, the developed models later on may behave strangely, and totally valid data may end up being treated like missing data.

Let's re-format the `FlightDate` field into three separate fields (year, month, day) using `separate()`.

[ ]:
```
date_airline <- replace_na %>%
    separate(FlightDate, sep = "-", into = c("year", "month", "day"))

head(date_airline)
```

For a number of reasons, including when you import a dataset into R or process a variable, the data type may be incorrectly established. For example, here we notice that the assigned data type to for `year`, `month`, and `day` is "character" although the expected data type should really be numeric. You can change the type using `mutate_all()` and `mutate_if()`.

In the below code, it is mutating `year`, `month`, and `day` values to numeric only if it is a character. Note that in `mutate_if()`, the first parameter is the function `is.character` that checks a *condition* while the second parameters is the function `as.numeric` that *modifies* the data *if* the condition is true.

[ ]:
```
date_airline %>%
    select(year, month, day) %>%
    mutate_all(type.convert) %>%
    mutate_if(is.character, as.numeric)
```

Wonderful!

Now, we finally obtain the cleaned dataset with no missing values and all data in its proper format.

2. Data Normalization

Why normalization?

Normalization is the process of transforming values of several features (variables) into a similar range. An example of why this could be important is if you have a variable for income and another variable for age. Income is likely much bigger values than age, so in a model, income would naturally influence the model more. Thus, normalization helps make comparisons between different variables more *fair*.

There are different types of ways to normalize:

- **Simple scaling**: divides each value by the maximum value in a feature. The new range is between 0 and 1.
$$x_{new} = \frac{x_{old}}{x_{max}}$$
- **Min-max**: subtracts the minimum value from the original and divides by the maximum minus the minimum. The minimum becomes 0 and the maximum becomes 1.
$$x_{new} = \frac{x_{old} - x_{min}}{x_{max} - x_{min}}$$
- **Standardization (Z-score)**: subtract the mean (μ) of the feature and divide by the standard deviation (σ).
$$x_{new} = \frac{x_{old} - \mu}{\sigma}$$

From these methods, you can see that typically normalizations include scaling the variable so that the range is from 0 to 1.

2.1 Simple scaling

Simple scaling divides each value by the maximum value in a feature. The new range is between 0 and 1.

$$x_{new} = \frac{x_{old}}{x_{max}}$$

Example

To demonstrate simpling scaling, let's say we want to scale the column "ArrDelay".

- **Target:** Would like to Normalize those variables so their value ranges from 0 to 1.
- **Approach:** Replace the original value by (original value)/(maximum value)

[ ]:
```
simple_scale <- sub_airline$ArrDelay / max(sub_airline$ArrDelay)
head(simple_scale)
```

Question #2:

According to the example above, normalize the column "DepDelay" using the simple scaling technique.

[ ]:
```
# Write your code below and press Shift+Enter to execute
```

▼ Click here for the solution.

```
simple_scale2 <- sub_airline@@@@DepDelay)
head(simple_scale2)
```

2.2 Min-max

Min-max subtracts the minimum value from the original and divides by the maximum minus the minimum. The minimum becomes 0 and the maximum becomes 1.

$$x_{new} = \frac{x_{old} - x_{min}}{x_{max} - x_{min}}$$

Example

Using "ArrDelay" as an example again, you can transform this column using the min-max technique:

[ ]:
```
minmax_scale <- (sub_airline$ArrDelay - min(sub_airline$ArrDelay)) /
                (max(sub_airline$ArrDelay) - min(sub_airline$ArrDelay))
head(minmax_scale)
```

2.3 Data Standardization (Z-score)

Standardization (Z-score) subtracts the mean (μ) of the feature and divides by the standard deviation (σ).

$$x_{new} = \frac{x_{old} - \mu}{\sigma}$$

Example

Let's use "ArrDelay" again. We can use `mean()` to find the mean of the feature and we can use `sd()` to find the standard deviation of the feature.

[ ]:
```
z_scale <- (sub_airline$ArrDelay - mean(sub_airline$ArrDelay)) / sd(sub_airline$ArrDelay)
head(z_scale)
```

Question #3:

According to the example above, standardize the "DepDelay" column.

[ ]:
```
# Write your code below and press Shift+Enter to execute
```

▼ Click here for the solution.

```
z_scale2 <- (sub_airline@@@@DepDelay)) / sd(sub_airline$DepDelay)
head(z_scale2)
```

4. Binning

Why use binning?

Binning is a process of transforming continuous numerical variables into discrete categorical 'bins', for grouped analysis.

Example:

Using binning, we categorize arrival delays into four bins by quartiles. Quartiles refer to the boundaries that divide observations into four defined intervals. They are often determined based on the values of data points and how they are compared with the rest of the dataset. In the actual airline dataset, "ArrDelay" is a numerical variable ranging from -73 to 682, it has 2855 unique values. We can categorize them into 4 bins. Can we rearrange them into four 'bins' to simplify analysis?

We will use the tidyverse method `mutate(quantile_rank = ntile())` to segment the "ArrDelay" column into 4 bins

Example of Binning Data In Tidyverse

Lets plot the histogram of flight arrival delays, to see what the distribution of "ArrDelay" looks like.

[ ]:
```
ggplot(data = sub_airline, mapping = aes(x = ArrDelay)) +
  geom_histogram(bins = 100, color = "white", fill = "red") +
  coord_cartesian(xlim = c(-73, 682))
```

First we use the dplyr function `ntile` to break "ArrDelay" into 4 buckets, which have equal amount of observations of flight arrival delays. We then create a list "quantile_rank" that contains 4 bins, which are respectively labeled "1", "2", "3", "4". So bin 1 would contain the first 25% of data, bin 2 the next 25% of data and so on.

[ ]:
```
binning <- sub_airline %>%
      mutate(quantile_rank = ntile(sub_airline$ArrDelay,4))

head(binning)
```

The observations are put into different bins based on the flights' delay minutes. The larger the bin label is, the longer the flight was delayed.

Now if we look at a histogram of the bins, you can see that all bins are equal.

[ ]:
```
ggplot(data = binning, mapping = aes(x = quantile_rank)) +
  geom_histogram(bins = 4, color = "white", fill = "red")
```

5. Indicator variable

What is an indicator variable?

An indicator variable (or dummy variable) is a **numerical variable** used to **label categories**. They are called 'dummies' because the numbers themselves don't have inherent meaning.

Why we use indicator variables?

Regression models need numerical variables, however categorical variables in their original forms are strings. Using indicator variables allows us to be able to use these categorical variables in regression models.

Example

In the airline dataset, the "Reporting_Airline" feature is a categorical variable that has nine values, "AA", "AS", "B6", "DL", "HP", "PA (1)", "TW", "UA" or "VX", which are in character type. For further analysis, we need to convert these variables into some form of numeric format.

We will use the tidyverse's method `spread()` method to convert categorical variables to dummy variables. The parameters to use in the function are:

- `key` : the column to convert into categorical values
- `value` : the value you want to set the key to
- `fill` : fills the missing values with this value

Also keep in mind that the method will drop the "Reporting_Airline" column by default. So if you used "ArrDelay" as the key value instead, that column would get dropped.

The below code does the following:

1. `mutate` - creates a column `dummy` with all 1's then
2. `spread` - creates new column for every `Reporting_Arilline` and sets the value to 1 from `dummy`. But replaces the 1 with a 0 if the corresponding value is `NA` in `Reporting_Airline` then
3. `slice` - looks at the specified rows

[ ]:
```
sub_airline %>%
  mutate(dummy = 1) %>% # column with single value
  spread(
    key = Reporting_Airline, # column to spread
    value = dummy,
    fill = 0) %>%
  slice(1:5)
```

When a value occurs in the original feature, we set the corresponding value to one in the new feature; the rest of the features are set to zero.

So in the output above, for the first row, the reporting airline is "UA". Therefore, the feature "UA" is set to one and the other features to zero. Similarly, for the second row, the reporting airline value is "AS". Therefore, the feature "AS" is set to one and the other features to zero.

Alternatively, instead of assigning dummy values 0 or 1, we can assign flight delay values to each feature. So this will also create new columns, one for each `Reporting_Airline`. Taking the first row for example, the column "UA" is now set to 2 because that is the value from ArrDelay.

[ ]:
```
sub_airline %>%
  spread(Reporting_Airline, ArrDelay) %>%
  slice(1:5)
```

Visualize Airline Category

Let's visualize how many data points in each airline category.

[ ]:
```
sub_airline %>% # start with data
   mutate(Reporting_Airline = factor(Reporting_Airline,
                                     labels = c("AA", "AS", "DL", "UA", "B6", "PA (1)", "HP", "TW", "VX")))%>%
  ggplot(aes(Reporting_Airline)) +
  stat_count(width = 0.5) +
  labs(x = "Number of data points in each airline")
```

Question #4:

As above, create indicator variable to the column of "Month".

[ ]:
```
# Write your code below and press Shift+Enter to execute
```

▼Click here for the solution.

```
sub_airline %>%
  mutate(dummy = 1) %>% # column with single value
  spread(
    key = Month, # column to spread
    value = dummy,
    fill = 0) %>%
  slice(1:5)
```

Question #5:

Now, create indicator variable to the column of "Month" by applying departure delay values

[ ]:
```
# Write your code below and press Shift+Enter to execute
```

▼Click here for the solution.

```
sub_airline %>%
  spread(Month, DepDelay) %>%
  slice(1:5) # Show only the first 5 rows
```

Thank you for completing this notebook

About the Authors:

This notebook was written by Yiwen Li and Gabriela de Queiroz.

Yiwen Li has approximately three year experiences in big tech industry. Currently, she is a developer advocate, a data scientist, a product manager at IBM, where she designs and develops data science solutions and Machine Learning models to solve real world problems. She has delivered talks this year in JupyterCon, PyCon, Pyjamas, CrowdCast.ai, Global AI on Tour 2020 and Belpy 2021 with hundreds of attendants per talk. Gabriela de Queiroz is a Sr. Engineering & Data Science Manager at IBM where she manages and leads a team of developers working on Data & AI Open Source projects. She works to democratize AI by building tools and launching new open source projects. She is the founder of AI Inclusive, a global organization that is helping increase the representation and participation of gender minorities in Artificial Intelligence. She is also the founder of R-Ladies, a worldwide organization for promoting diversity in the R community with more than 190 chapters in 50+ countries. She has worked in several startups and where she built teams, developed statistical models, and employed a variety of techniques to derive insights and drive data-centric decisions