Unit 1-Part 2

Data Science with R

- Staging and Curating the data, Exploring data, sing summary statistics to spot
- problems, Managing data, Cleaning data, Sampling for modeling and validation, Training and test set split, Sample group column,
- Record grouping, Data provenance,

Data staging

- Data staging is a crucial step in managing and preparing data for analysis
- Data staging involves storing data temporarily, allowing for programmatic processing and short-term data recovery
- It serves as an intermediate step between data sources and the target data warehouse or analytics platform.
- Benefits of data staging include testing source data, mitigating pipeline failures, creating an audit trail, and performing complex transformations.
- **External Staging:**
  1. In traditional data staging, data resides outside the warehouse (e.g., in cloud storage like Google Cloud Storage or AWS S3).
  2. Data engineers load data from external sources, perform simple transformations, and clean the data before loading it into the warehouse.
  3. Staged data is often stored in raw formats (e.g., JSON or Parquet) for further processing.

Data staging and curating

- **Internal Staging:**
  1. Modern data staging can also occur within the warehouse itself.
  2. Depending on the transformation process, staging may take place before or after loading data into the warehouse.
  3. Having a single source of staged data reduces data sprawl (uncontrolled growth of data) and ensures a consistent source of truth.
- **Curating the Data:**
  1. Curating data involves defining protocols for cleansing and organizing the data.
  2. It ensures that data is accurate, consistent, and ready for analysis.
  3. Curated data is essential for reliable reporting and decision-making.
  4. Remember, effective data staging and curation are critical for maintaining data quality and enabling successful analytics.

Key Aspects of Data Curation

1. **Collection and Integration:** Curators collect data from diverse sources (such as databases, APIs, or files) and integrate it into organized repositories.
2. **Annotation and Metadata:** Metadata (descriptive information) is added to the data, making it easier to understand and discover. Annotations provide context and enhance data quality.
3. **Quality Assurance:** Curators validate data quality, ensuring it is accurate, consistent, and reliable.
4. **Archiving and Preservation:** Data is stored securely and preserved for long-term use.
5. **Representation:** Data is transformed into formats suitable for analysis, visualization, or reporting.

- **Examples of Data Curation**
- **Biological Databases**: Curators extract relevant biological information from research articles and organize it in specialized databases.
- **Historical Archives**: Cultural and scholarly data from digital humanities projects require expert curation.

Using summary statistics to spot problems

- In R, you'll typically use the summary() command to take your first look at the data.
- The goal is to understand for example whether you have the kind of customer information that can potentially help you predict health insurance coverage, and whether the data is of good enough quality to be informative
- Typical problems revealed by data summaries
  - **Missing values**
  - **Invalid values and outliers**
  - **Data ranges that are too wide or too narrow**
  - **The units of the data**

Change this to your actual path to the directory where you unpacked PDSwR2

The variable is_employed is missing for about a third of the data. The variable income has negative values, which are potentially invalid.

```
setwd("PDSwR2/Custdata")
customer_data = readRDS("custdata.RDS")
summary(customer_data)
##     custid               sex          is_employed        income
##  Length:73262       Female:37837   FALSE: 2351     Min.   :  -6900
##  Class :character   Male  :35425   TRUE :45137     1st Qu.:  10700
##  Mode  :character                  NA's :25774     Median :  26200
##                                                    Mean   :  41764
##                                                    3rd Qu.:  51700
##                                                    Max.   :1257000

##              marital_status  health_ins
##  Divorced/Separated:10693   Mode :logical
##  Married           :38400   FALSE:7307
##  Never married     :19407   TRUE :65955
##  Widowed           : 4762
##
##
##
##                        housing_type   recent_move      num_vehicles
##  Homeowner free and clear    :16763   Mode :logical    Min.   :0.000
##  Homeowner with mortgage/loan:31387   FALSE:62418      1st Qu.:1.000
##  Occupied with no rent       : 1138   TRUE :9123       Median :2.000
##  Rented                      :22254   NA's :1721       Mean   :2.066
##  NA's                        : 1720                    3rd Qu.:3.000
##                                                        Max.   :6.000
##                                                        NA's   :1720
##       age              state_of_res     gas_usage
##  Min.   :  0.00   California  : 8962   Min.   :  1.00
##  1st Qu.: 34.00   Texas       : 6026   1st Qu.:  3.00
##  Median : 48.00   Florida     : 4979   Median : 10.00
##  Mean   : 49.16   New York    : 4431   Mean   : 41.17
##  3rd Qu.: 62.00   Pennsylvania: 2997   3rd Qu.: 60.00
##  Max.   :120.00   Illinois    : 2925   Max.   :570.00
##                   (Other)     :42942   NA's   :1720
```

About 90% of the customers have health insurance.

The variables housing_type, recent_move, num_vehicles, and gas_usage are each missing 1720 or 1721 values.

The average value of the variable age seems plausible, but the minimum and maximum values seem unlikely. The variable state_of_res is a categorical variable; summary() reports how many customers are in each state (for the first few states).

MISSING VALUES

- A few missing values may not really be a problem, but if a particular data field is largely unpopulated, it shouldn’t be used as an input without some repair
- In R, for example, many modeling algorithms will, by default, quietly drop rows with missing values.
- As you see in the following listing, all the missing values in the is_employed variable could cause R to quietly ignore more than a third of the data
- If a particular data field is largely unpopulated, it’s worth trying to determine why; sometimes the fact that a value is missing is informative in and of itself.
- For example, why is the is_employed variable missing for so many values?

Will the variable with missing values be useful for modeling?

```
## is_employed
## FALSE: 2321
## TRUE :44887
## NA's :24333
```

The variable is_employed is missing for more than a third of the data. Why? Is employment status unknown? Did the company start collecting employment data only recently? Does NA mean “not in the active workforce” (for example, students or stay-at-home parents)?

```
##                         housing_type   recent_move
## Homeowner free and clear     :16763   Mode :logical
## Homeowner with mortgage/loan:31387   FALSE:62418
## Occupied with no rent        : 1138   TRUE :9123
## Rented                       :22254   NA's :1721
## NA's                         : 1720
##
##
##   num_vehicles      gas_usage
##  Min.   :0.000   Min.    :  1.00
##  1st Qu.:1.000   1st Qu.:  3.00
##  Median :2.000   Median : 10.00
##  Mean   :2.066   Mean   : 41.17
##  3rd Qu.:3.000   3rd Qu.: 60.00
##  Max.   :6.000   Max.   :570.00
##  NA's   :1720    NA's   :1720
```

The variables housing_type, recent_move, num_vehicles, and gas_usage are missing relatively few values—about 2% of the data. It’s probably safe to just drop the rows that are missing values, especially if the missing values are all in the same 1720 rows.

What do with variable having missing values

- Whatever the reason for missing data, you must decide on the most appropriate action.
- Do you include a variable with missing values in your model, or not?
- If you decide to include it, do you drop all the rows where this field is missing, or do you convert the missing values to 0 or to an additional category?
- In this example, you might decide to drop the data rows where you're missing data about housing or vehicles, since there aren't many of them
- You probably don't want to throw out the data where you're missing employment information, since employment status is probably highly predictive of having health insurance; you might instead treat the NAs as a third employment category

INVALID VALUES AND OUTLIERS

- Even when a column or variable isn't missing any values, you still want to check that the values that you do have make sense.
- Do you have any invalid values or outliers?
- Examples of invalid values include negative values in what should be a non-negative numeric data field (like age or income) or text where you expect numbers.
- Outliers are data points that fall well out of the range of where you expect the data to be.
- Can you spot the outliers and invalid values here in the summary stats

```
summary(customer_data$income)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   -6900   11200   27300   42522   52000 1257000

summary(customer_data$age)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.00   34.00   48.00   49.17   62.00  120.00
```

Negative values for income could indicate bad data. They might also have a special meaning, like "amount of debt." Either way, you should check how prevalent the issue is, and decide what to do. Do you drop the data with negative income? Do you convert negative values to zero?

Customers of age zero, or customers of an age greater than about 110, are outliers. They fall out of the range of expected customer values. Outliers could be data input errors. They could be special sentinel values: zero might mean "age unknown" or "refuse to state." And some of your customers might be especially long-lived.

Invalid Values

- Often, invalid values are simply bad data input
- A negative number in a field like `age`, however, could be a *sentinel value* to designate "unknown."
- Outliers might also be data errors or sentinel values, or they might be valid but unusual data points—people do occasionally live past 100.
- As with missing values, you must decide the most appropriate action: drop the data field, drop the data points where this field is bad, or convert the bad data to a useful value.
- For example, even if you feel certain outliers are valid data, you might still want to omit them from model construction, if the outliers interfere with the model-fitting process.
- ***Generally, the goal of modeling is to make good predictions on typical cases, and a model that is highly skewed to predict a rare case correctly may not always be the best model overall.***

DATA RANGE

- You also want to pay attention to how much the values in the data vary.
- If you believe that age or income helps to predict the probability of health insurance coverage, then you should make sure there is enough variation in the age and income of your customers for you to see the relationships.
- Let's look at income again, Is the data range wide? Is it narrow?
- Data that ranges over several orders of magnitude like this can be a problem for some modeling methods. (log transformation may be used to transform the data before using)
- Data can be too narrow, too. Suppose all your customers are between the ages of 50 and 55.
- It's a good bet that age range wouldn't be a very good predictor of the probability of health insurance coverage for that population, since it doesn't vary much at all.

```
summary(customer_data$income)
##     Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    -6900   10700   26200   41764   51700 1257000
```

Income ranges from zero to over a million dollars, a very wide range.

Units

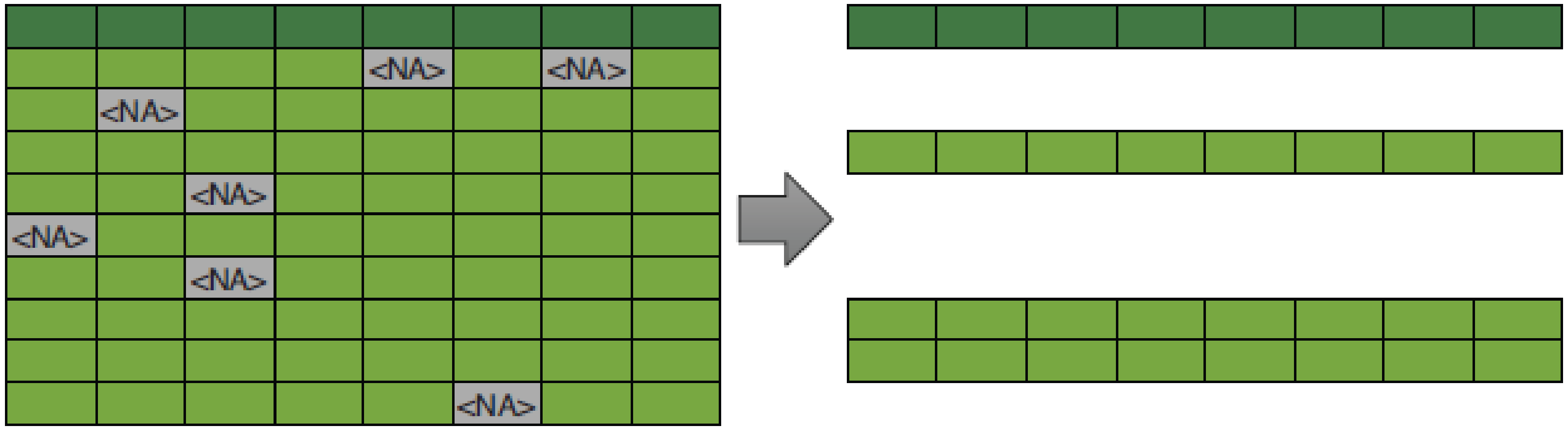
- Does the income data represent hourly wages, or yearly wages in units of $1000?
- As a matter of fact, it's yearly wages in units of $1000, but what if it were hourly wages?
- You might not notice the error during the modeling stage, but down the line someone will start inputting hourly wage data into the model and get back bad predictions in return.

```
IncomeK = customer_data$income/1000
summary(IncomeK)
 ##     Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    -6.90   10.70   26.20   41.76   51.70 1257.00
```

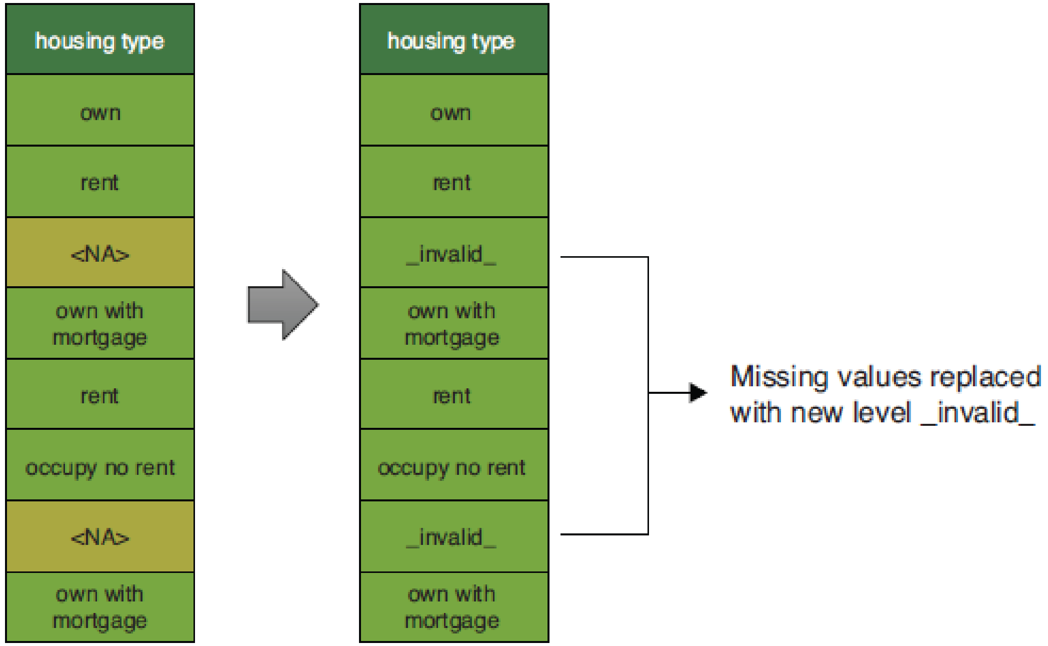
The variable IncomeK is defined as IncomeK = customer_data$income/1000. But suppose you didn't know that. Looking only at the summary, the values could plausibly be interpreted to mean either "hourly wage" or "yearly income in units of $1000."

Data Cleaning

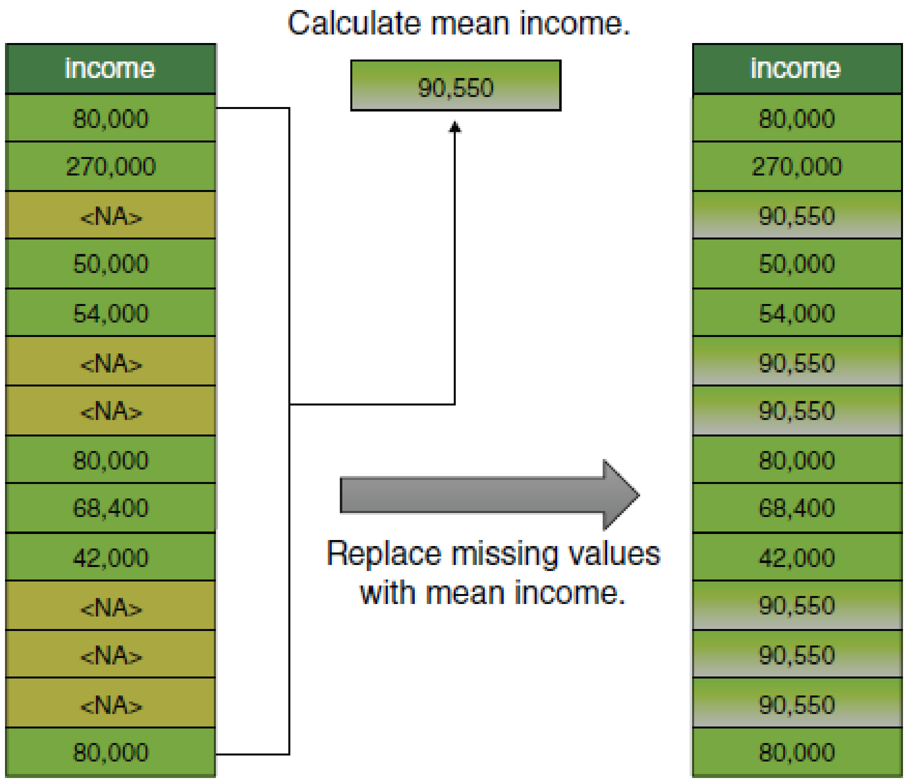
Even a few missing values can lose all your data.



MISSING DATA IN CATEGORICAL VARIABLES

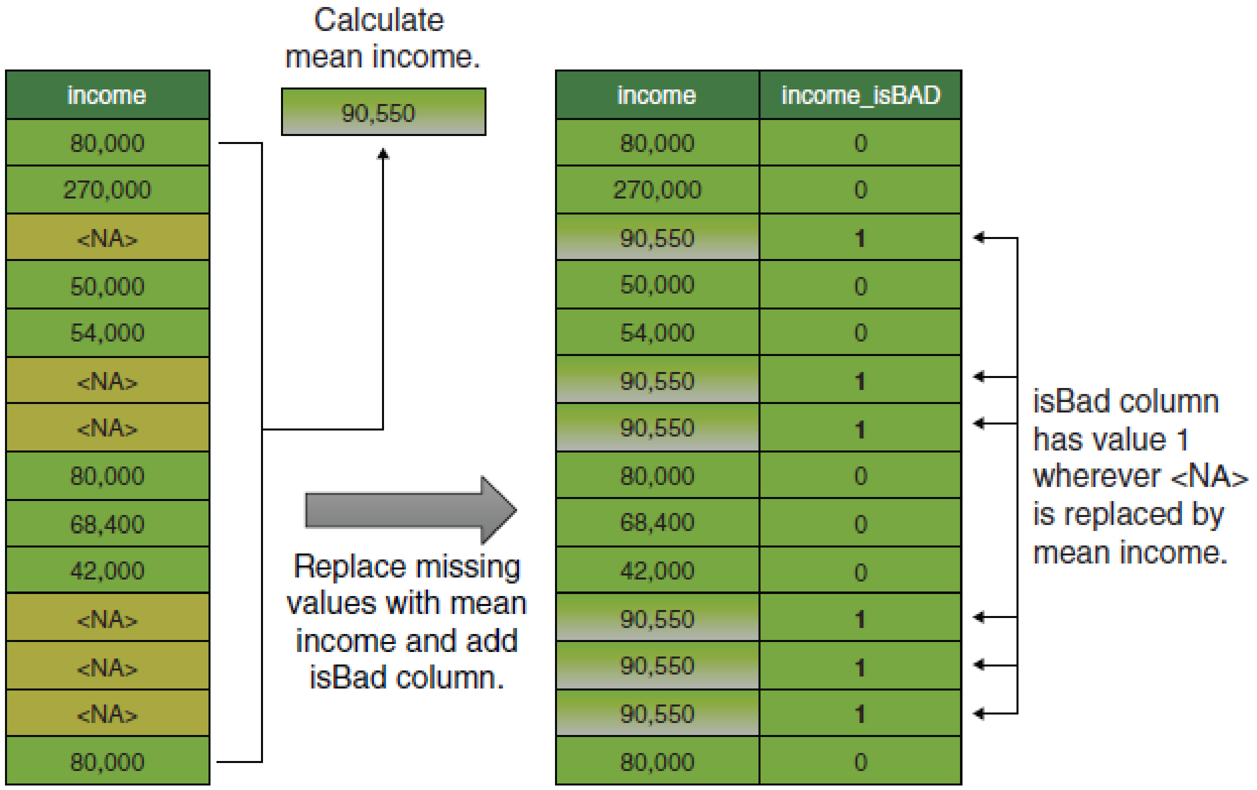


MISSING VALUES IN NUMERIC OR LOGICAL VARIABLES



Treating missing values as information

Replacing missing values with the mean and adding an indicator column to track the altered values



Example-Treating age and income data

```
library(dplyr)
customer_data = readRDS("custdata.RDS")
```

Loads the data

```
customer_data <- customer_data %>%
  mutate(age = na_if(age, 0),
         income = ifelse(income < 0, NA, income))
```

Converts negative incomes to NA

The function mutate() from the dplyr package adds columns to a data frame, or modifies existing columns. The function na_if (), also from dplyr, turns a specific problematic value (in this case, 0) to NA.

vtreat package for automatically treating missing variables

- Separate the variables on which to apply treatment

```
varlist1 <- setdiff(colnames(customer_data), c("custid", "health_ins"))
```

- Then, you create the treatment plan, and "prepare" the data

```
library(vtreat)
treatment_plan <-
design_missingness_treatment(customer_data, varlist = varlist1)
training_prepared <- prepare(treatment_plan, customer_data)
```

- The data frame `training_prepared` is the treated data that you would use to train a model

Comparing original and treated data

```
colnames(customer_data)
##  [1] "custid"           "sex"                   "is_employed"
##  [4] "income"           "marital_status"        "health_ins"
##  [7] "housing_type"     "recent_move"           "num_vehicles"
## [10] "age"              "state_of_res"          "gas_usage"
## [13] "gas_with_rent"    "gas_with_electricity"  "no_gas_bill"

colnames(training_prepared)
##  [1] "custid"                     "sex"
##  [3] "is_employed"                "income"
##  [5] "marital_status"             "health_ins"
##  [7] "housing_type"               "recent_move"
##  [9] "num_vehicles"               "age"
## [11] "state_of_res"               "gas_usage"
## [13] "gas_with_rent"              "gas_with_electricity"
## [15] "no_gas_bill"                "is_employed_isBAD"
## [17] "income_isBAD"               "recent_move_isBAD"
## [19] "num_vehicles_isBAD"         "age_isBAD"
## [21] "gas_usage_isBAD"            "gas_with_rent_isBAD"
## [23] "gas_with_electricity_isBAD" "no_gas_bill_isBAD"
```

The prepared data has additional columns that are not in the original data, most importantly those with the _isBAD designation.

```
nacounts <- sapply(training_prepared, FUN=function(col) sum(is.na(col)) )
sum(nacounts)
## [1] 0
```

The prepared data has no missing values.

Now examine a few columns that you know had missing values.

Finds the rows where housing_type was missing

Looks at a few columns from those rows in the original data

```
htmissing <- which(is.na(customer_data$housing_type))

columns_to_look_at <- c("custid", "is_employed", "num_vehicles",
                        "housing_type", "health_ins")

customer_data[htmissing, columns_to_look_at] %>% head()
##           custid is_employed num_vehicles housing_type health_ins
## 55  000082691_01        TRUE           NA         <NA>      FALSE
## 65  000116191_01        TRUE           NA         <NA>       TRUE
## 162 000269295_01          NA           NA         <NA>      FALSE
## 207 000349708_01          NA           NA         <NA>      FALSE
## 219 000362630_01          NA           NA         <NA>       TRUE
## 294 000443953_01          NA           NA         <NA>       TRUE

columns_to_look_at = c("custid", "is_employed", "is_employed_isBAD",
                       "num_vehicles","num_vehicles_isBAD",
                       "housing_type", "health_ins")

training_prepared[htmissing, columns_to_look_at] %>%  head()
##           custid is_employed is_employed_isBAD num_vehicles
## 55  000082691_01   1.0000000                 0       2.0655
## 65  000116191_01   1.0000000                 0       2.0655
## 162 000269295_01   0.9504928                 1       2.0655
## 207 000349708_01   0.9504928                 1       2.0655
## 219 000362630_01   0.9504928                 1       2.0655
## 294 000443953_01   0.9504928                 1       2.0655
##     num_vehicles_isBAD housing_type health_ins
## 55                   1    _invalid_      FALSE
## 65                   1    _invalid_       TRUE
## 162                  1    _invalid_      FALSE
## 207                  1    _invalid_      FALSE
## 219                  1    _invalid_       TRUE
## 294                  1    _invalid_       TRUE

customer_data %>%
    summarize(mean_vehicles = mean(num_vehicles, na.rm = TRUE),
    mean_employed = mean(as.numeric(is_employed), na.rm = TRUE))
##   mean_vehicles mean_employed
## 1        2.0655     0.9504928
```

Looks at those rows and columns in the treated data (along with the isBADs)

Verifies the expected number of vehicles and the expected unemployment rate in the dataset

Transform Data in R

- Common data transformation techniques using built-in functions
- **Reorder Data**:
- Sometimes, you need to reorder data for better analysis. Sorting data is a common example
- **Subset/Filter Data:**
  - **To extract specific subsets of your data, use functions like subset() or logical indexing.**

```
# Subset rows where Age is greater than 30
subset_data <- subset(my_data, Age > 30)
```

- **t()**: Returns the transpose of a matrix or data frame. For example:

```
# Create a sample matrix
my_matrix <- matrix(1:6, nrow = 2)
transposed_matrix <- t(my_matrix)
```

Transform Data in R

- Combine Data:
  - Merging or combining data from different sources is, essential.
- Functions like merge() can help
  - `# Merge two data frames by a common column`
  - `merged_data <- merge(df1, df2, by = "ID")`
- **Transform Data**:
  - Transformations include creating new variables, scaling, or applying mathematical operations
  - Adding a new column to a data frame in **R** is a common operation

Adding new column

```
# Create a data frame
df <- data.frame(a = c('A', 'B', 'C', 'D', 'E'), b = c(45, 56, 54, 57, 59))
# Define a new column
new_column <- c(3, 3, 6, 7, 8)
# Add the new column
df$new <- new_column
# 2nd method Add the new column
df['new1'] <- new_column
```

```
#3rd method define new column
new2 <- c(3, 3, 6, 7, 8)
#add column called 'new2' using cbind
df_new <- cbind(df, new2)
#view new data frame
df_new
```

Normalization

- Example Suppose you are considering the use of income as an input to your insurance model.
- The cost of living will vary from state to state, so what would be a high salary in one region could be barely enough to scrape by in another.
- Because of this, it might be more meaningful to normalize a customer's income by the typical income in the area where they live. This is an example of a relatively simple (and common) transformation.

```
library(dplyr)
median_income_table <-
      readRDS("median_income.RDS")
head(median_income_table)

##   state_of_res median_income
## 1      Alabama         21100
## 2       Alaska         32050
## 3      Arizona         26000
## 4     Arkansas         22900
## 5   California         25000
## 6     Colorado         32000

training_prepared <-  training_prepared %>%
  left_join(., median_income_table, by="state_of_res") %>%
   mutate(income_normalized = income/median_income)
```

If you have downloaded the PDSwR2 code example directory, then median_income.RDS is in the directory PDSwR2/Custdata. We assume that this is your working directory.

Joins median_income_table into the customer data, so you can normalize each person's income by the median income of their state

```
head(training_prepared[, c("income", "median_income", "income_normalized")])

##   income median_income income_normalized
## 1  22000         21100         1.0426540
## 2  23200         21100         1.0995261
## 3  21000         21100         0.9952607
## 4  37770         21100         1.7900474
## 5  39000         21100         1.8483412
## 6  11100         21100         0.5260664

summary(training_prepared$income_normalized)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.0000  0.4049  1.0000  1.5685  1.9627 46.5556
```

Compares the values of income and income_normalized

```
summary(training_prepared$age)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   21.00   34.00   48.00   49.22   62.00  120.00

mean_age <- mean(training_prepared$age)
age_normalized <- training_prepared$age/mean_age
summary(age_normalized)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.4267  0.6908  0.9753  1.0000  1.2597  2.4382
```

Centering and scaling for transformation

- You can rescale your data by using the standard deviation as a unit of distance.
- A customer who is within one standard deviation of the mean age is considered not much older or younger than typical.
- A customer who is more than one or two standard deviations from the mean can be considered much older, or much younger.
- To make the relative ages even easier to understand, you can also center the data by the mean, so a customer of “typical age” has a centered age of 0.

```
(mean_age <- mean(training_prepared$age))
 ## [1] 49.21647

(sd_age <- sd(training_prepared$age))
 ## [1] 18.0124

print(mean_age + c(-sd_age, sd_age))
 ## [1] 31.20407 67.22886

training_prepared$scaled_age <- (training_prepared$age -
      mean_age) / sd_age

training_prepared %>%
  filter(abs(age - mean_age) < sd_age) %>%
  select(age, scaled_age) %>%
  head()

##   age scaled_age
## 1  67  0.9872942
## 2  54  0.2655690
## 3  61  0.6541903
## 4  64  0.8207422
## 5  57  0.4321210
## 6  55  0.3210864
```

Takes the mean

Takes the standard deviation

The typical age range for this population is from about 31 to 67.

Uses the mean value as the origin (or reference point) and rescales the distance from the mean by the standard deviation

Customers in the typical age range have a scaled_age with magnitude less than 1.

```
##   age scaled_age
## 1  24  -1.399951
## 2  82   1.820054
## 3  31  -1.011329
## 4  93   2.430745
## 5  76   1.486950
## 6  26  -1.288916
```

Customers outside the typical age range have a scaled_age with magnitude greater than 1.

Now, values less than -1 signify customers younger than typical; values greater than 1 signify customers older than typical.

Centering and scaling multiple numeric variables

```
dataf <- training_prepared[, c("age", "income", "num_vehicles", "gas_usage")]
summary(dataf)

##       age              income           num_vehicles      gas_usage
##  Min.   : 21.00   Min.   :      0   Min.   :0.000   Min.   :  4.00
##  1st Qu.: 34.00   1st Qu.:  10700   1st Qu.:1.000   1st Qu.: 50.00
##  Median : 48.00   Median :  26300   Median :2.000   Median : 76.01
##  Mean   : 49.22   Mean   :  41792   Mean   :2.066   Mean   : 76.01
##  3rd Qu.: 62.00   3rd Qu.:  51700   3rd Qu.:3.000   3rd Qu.: 76.01
##  Max.   :120.00   Max.   :1257000   Max.   :6.000   Max.   :570.00

dataf_scaled <- scale(dataf, center=TRUE, scale=TRUE)

summary(dataf_scaled)
##       age                income            num_vehicles          gas_usage
##  Min.   :-1.56650   Min.   :-0.7193   Min.   :-1.78631   Min.   :-1.4198
##  1st Qu.:-0.84478   1st Qu.:-0.5351   1st Qu.:-0.92148   1st Qu.:-0.5128
##  Median :-0.06753   Median :-0.2666   Median :-0.05665   Median : 0.0000
##  Mean   : 0.00000   Mean   : 0.0000   Mean   : 0.00000   Mean   : 0.0000
##  3rd Qu.: 0.70971   3rd Qu.: 0.1705   3rd Qu.: 0.80819   3rd Qu.: 0.0000
##  Max.   : 3.92971   Max.   :20.9149   Max.   : 3.40268   Max.   : 9.7400

(means <- attr(dataf_scaled, 'scaled:center'))
 ##         age       income num_vehicles    gas_usage
##     49.21647  41792.51062      2.06550     76.00745

(sds <- attr(dataf_scaled, 'scaled:scale'))
##          age       income num_vehicles    gas_usage
##    18.012397 58102.481410     1.156294    50.717778
```

Centers the data by its mean and scales it by its standard deviation

Gets the means and standard deviations of the original data, which are stored as attributes of dataf_scaled

Log Transformations

- Normalizing by mean and standard deviation, is most meaningful when the data distribution is roughly symmetric.
- Log transformations are transformations that can make some distributions more symmetric.
- For Example: Monetary amounts—incomes, customer value, account values, or purchase sizes— are some of the most commonly encountered sources of skewed distributions in data science applications.
- Monetary amounts are often lognormally distributed : the log of the data is normally distributed.
- This leads us to the idea that taking the log of monetary data can restore symmetry and scale to the data, by making it look “more normal.”
- It’s also generally a good idea to log transform data containing values that range over several orders of magnitude, for example, the population of towns and cities, which may range from a few hundred to several million.
- One reason for this is that modeling techniques often have a difficult time with very wide data ranges.
- Another reason is because such data often comes from multiplicative processes rather than from an

additive one, so log units are in some sense more natural.

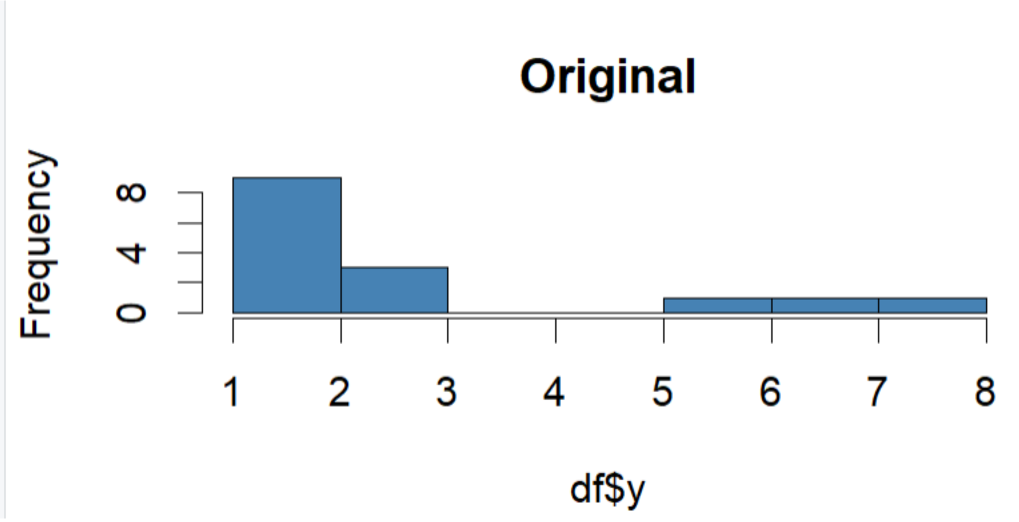
Log transformation in R

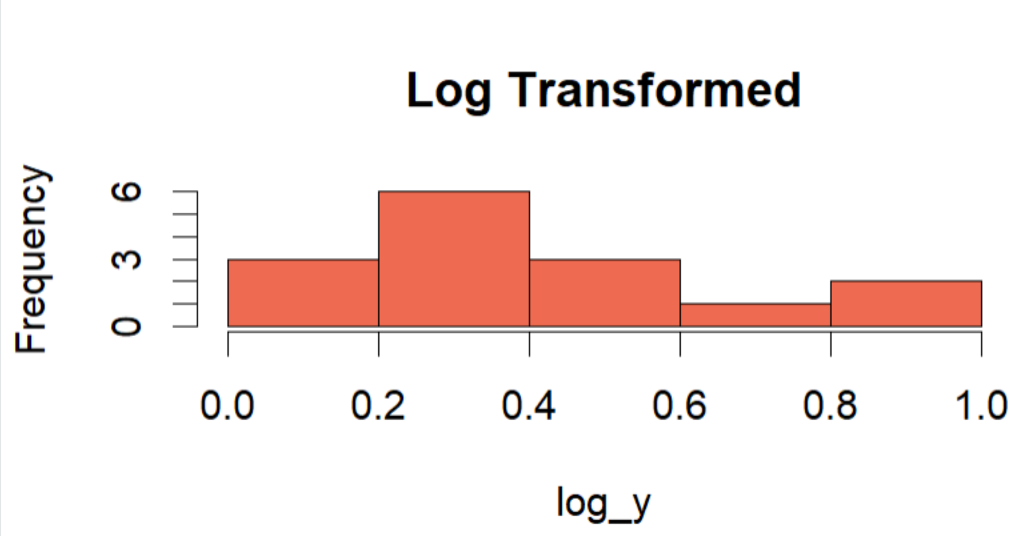
```
#create data frame
df <- data.frame(y=c(1, 1, 1, 2, 2, 2, 2, 2, 2, 3, 3, 3, 6, 7, 8),
                 x1=c(7, 7, 8, 3, 2, 4, 4, 6, 6, 7, 5, 3, 3, 5, 8),
                 x2=c(3, 3, 6, 6, 8, 9, 9, 8, 8, 7, 4, 3, 3, 2, 7))

#perform log transformation
log_y <- log10(df$y)
```

Log transformation in R

- The following code shows how to create histograms to view the distribution of y before and after performing a log transformation:

```
#create histogram for original distribution
hist(df$y, col='steelblue', main='Original')
#create histogram for log-transformed distribution
hist(log_y, col='coral2', main='Log Transformed')
```





Sampling for modelling and validation

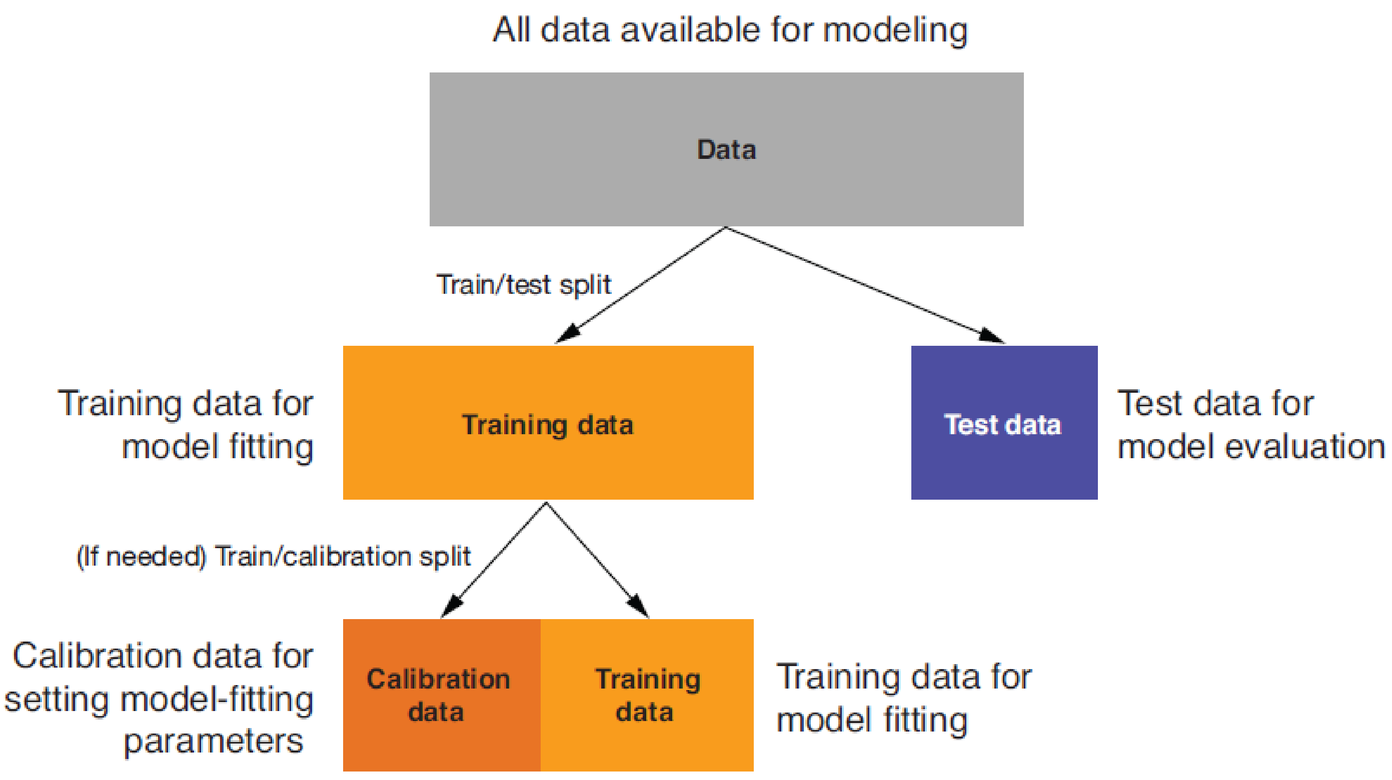
- Sampling is the process of selecting a subset of a population to represent the whole during analysis and modeling.
- In the current era of big datasets, some people argue that computational power and modern algorithms let us analyze the entire large dataset without the need to sample.
- But keep in mind even “big data” is usually itself a sample from a larger universe.
- So some understanding of sampling is always needed to work with data.
- We can certainly analyze larger datasets than we could before, but sampling is still a useful tool.
- When you’re in the middle of developing or refining a modeling procedure, it’s easier to test and debug the code on small subsamples before training the model on the entire dataset.
- Visualization can be easier with a subsample of the data; ggplot runs faster on smaller datasets, and too much data will often obscure the patterns in a graph

Sampling for modelling and validation

- It’s important that the dataset that you do use is an accurate representation of your population as a whole.
- For example, your customers might come from all over the United States.
- When you collect your customer data, it might be tempting to use all the customers from one state, say Connecticut, to train the model.
- But if you plan to use the model to make predictions about customers all over the country, it’s a good idea to pick customers randomly from all the states, because what predicts health insurance coverage for Texas customers might be different from what predicts health insurance coverage in Connecticut.
- Another reason to sample your data is to create test and training splits.

Test and training splits

- When you're building a model to make predictions, like our model to predict the probability of health insurance coverage, you need data to build the model
- You also need data to test whether the model makes correct predictions on new data.
- The first set is called the *training set*, and the second set is called the *test* (or *holdout*) set.
- The training set is the data that you feed to the model-building algorithm so that the algorithm can fit the correct structure to best predict the outcome variable.
- The test set is the data that you feed into the resulting model, to verify that the model's predictions will be accurate on new data.



Splitting data into training and test (or training, calibration, and test) sets

Creating a sample group column

- A convenient way to manage random sampling is to add a sample group column to the data frame.
- The sample group column contains a number generated uniformly from zero to one, using the `runif()` function.
- You can draw a random sample of arbitrary size from the data frame by using the appropriate threshold on the sample group column.
- For example, once you've labeled all the rows of your data frame with your sample group column (let's call it `gp`), then the set of all rows such that `gp` < 0.4 will be about four-tenths, or 40%, of the data.
- The set of all rows where `gp` is between 0.55 and 0.70 is about 15% of the data (0.7 – 0.55 = 0.15).
- So you can repeatably generate a random sample of the data of any size by using `gp`.

Splitting into test and training using a random group mark

```
set.seed(25643)
customer_data$gp <- runif(nrow(customer_data))
customer_test <- subset(customer_data, gp <= 0.1)
customer_train <- subset(customer_data, gp > 0.1)

dim(customer_test)
## [1] 7463   16

dim(customer_train)
## [1] 65799    16
```

Sets the random seed so this example is reproducible

Creates the grouping column

Here we generate a test set of about 10% of the data.

Here we generate a training set using the remaining data.

Using dplyr

- The dplyr package also has functions called sample_n() and sample_frac() that draw a random sample (a uniform random sample, by default) from a data frame.
- Why not just use one of these to draw training and test sets?
- You could, but you should make sure to set the random seed via the set.seed() command to guarantee that you'll draw the same sample group every time.
- Reproducible sampling is essential when you're debugging code.
- In many cases, code will crash because of a corner case that you forgot to guard against.
- This corner case might show up in your random sample.
- If you're using a different random input sample every time you run the code, you won't know if you will tickle the bug again.
- This makes it hard to track down and fix errors.
- We find that storing a sample group column with the data is a more reliable way to guarantee reproducible sampling during development and testing.

REPRODUCIBLE SAMPLING IS NOT JUST A TRICK FOR R

- If your data is in a database or other external store, and you only want to pull a subset of the data into R for analysis, you can draw a reproducible random sample by generating a sample group column in an appropriate table in the database, using the SQL command `RAND`.

Record grouping

- Supppose you're interested less in "*which customers don't have health insurance*", and more in "*which households have uninsured members?*"
- If you're modeling a question at the household level rather than the customer level, then every member of a household should be in the same group (test or training).
- In other words, the random sampling also has to be at the household level.
- Suppose your customers are marked both by a household ID and customer ID.
- We want to split the households into a training set and a test set.

	household_id	customer_id	age	income
household 1	000000004	000000004_01	65	940
household 2	000000023	000000023_01	43	29000
	000000023	000000023_02	61	42000
household 3	000000327	000000327_01	30	47000
	000000327	000000327_02	30	37400
household 4	000000328	000000328_01	62	42500
	000000328	000000328_02	62	31800
household 5	000000404	000000404_01	82	28600
household 6	000000424	000000424_01	45	160000
	000000424	000000424_02	38	250000

Ensuring test/train split doesn't split inside a household

If you have downloaded the PDSwR2 code example directory, then the household dataset is in the directory PDSwR2/Custdata. We assume that this is your working directory.

Gets the unique household IDs

Generates a unique sampling group ID per household, and puts in a column named gp

Joins the household IDs back into the original data

```
household_data <- readRDS("hhdata.RDS")
hh <- unique(household_data$household_id)

set.seed(243674)
households <- data.frame(household_id = hh,
                         gp = runif(length(hh)),
                         stringsAsFactors=FALSE)

household_data <- dplyr::left_join(household_data,
                                   households,
                                   by = "household_id")
```

Sampling the dataset by household rather than customer

- Everyone in a household has the same sampling group number.
- Now we can generate the test and training sets as before.
- This time, however, the threshold 0.1 doesn't represent 10% of the data rows, but 10% of the households, which may be more or less than 10% of the data, depending on the sizes of the households.

	household_id	customer_id	age	income	gp
household 1	000000004	000000004_01	65	940	0.20952116
household 2	000000023	000000023_01	43	29000	0.40896034
	000000023	000000023_02	61	42000	0.40896034
household 3	000000327	000000327_01	30	47000	0.55881933
	000000327	000000327_02	30	37400	0.55881933
household 4	000000328	000000328_01	62	42500	0.55739973
	000000328	000000328_02	62	31800	0.55739973
household 5	000000404	000000404_01	82	28600	0.54620515
household 6	000000424	000000424_01	45	160000	0.09107758
	000000424	000000424_02	38	250000	0.09107758

Notice that each member of a household has the same group number.

Data provenance

- You'll also want to add a column (or columns) to record data provenance: when your dataset was collected, perhaps what version of your data-cleaning procedure was used on the data before modeling, and so on.
- This metadata is akin to version control for data.
- It's handy information to have, to make sure that you're comparing apples to apples when you're in the process of improving your model, or comparing different models or different versions of a model.
- Recording the data source, collection date, and treatment date with data
- If, for example, the treatment date on the data is earlier than the most recent version of your data treatment procedures, then you know that this treated data is possibly obsolete.
- Thanks to the metadata, you can go back to the original data source and treat it again.

data_source_id	data_collection_date	data_treatment_date	custid	health_ins	income	is_employed
data_pull 8/2/18	2018-08-02	2018-08-03	000006646_03	TRUE	22000	TRUE
data_pull 8/2/18	2018-08-02	2018-08-03	000007827_01	TRUE	23200	NA
data_pull 8/2/18	2018-08-02	2018-08-03	000008359_04	TRUE	21000	TRUE
data_pull 8/2/18	2018-08-02	2018-08-03	000008529_01	TRUE	37770	NA
data_pull 8/2/18	2018-08-02	2018-08-03	000008744_02	TRUE	39000	TRUE
data_pull 8/2/18	2018-08-02	2018-08-03	000011466_01	TRUE	11100	NA