PREPARING DATA

Introducción a la Ciencia de Datos

EDA steps

Stage 1
- Frame the problem
- Ask interesting questions. What would you like to model?

Stage 2
- Acquire the data, resources?
- Which are the relevant data?

Stage 3
- Explore the data
- Plot the data, find trends and correlation patterns

Stage 4
- Manipulate the data, solve problems missing values, outliers
- Imputation and transformation of data
- Feature engineering

Prepare the ingredients: your data

- Reading data from CSV files
- Reading XML data
- Reading JSON data
- Reading data from fixed-width formatted files
- Reading data from R files and R libraries
- Removing cases with missing values
- Replacing missing values with the mean
- Removing duplicate cases
- Rescaling a variable to specified min-max range
- Normalizing or standardizing data in a data frame
- Binning numerical data
- Creating dummies for categorical variables
- Handling missing data
- Correcting data
- Imputing data
- Detecting outliers

Handling missing data

- In most real-world problems, data is likely to be incomplete because of:
  - incorrect data entry,
  - faulty equipment,
  - improperly coded data.
- In R, missing values are represented by the symbol NA (not available) and are considered to be the first obstacle in predictive modeling.
- So, it's always a good idea to check for missing data in a dataset before proceeding for further predictive analysis.

Why my data has missing values?

- **Data Extraction**: It is possible that there are problems with extraction process. In such cases, we should double-check for correct data
- **Data collection**: These errors occur at time of data collection and are harder to correct:
  - **Missing completely at random:** probability of missing variable is same for all observations.
  - **Missing at random:** missing ratio varies for different values / level of other input variables. (age data collection)
  - **Missing that depends on unobserved predictors:** missing values are not random and are related to the unobserved input variable. (discomfort)
  - **Missing that depends on the missing value itself:** This is a case when the probability of missing value is directly correlated with missing value itself (higher & lower income)

Handling missing data

- R provides three simple ways to handle missing values:

1. Deleting the observations.
2. Deleting the variables.
3. Transforming and binning values
4. Replacing the values with mean, median, or mode.
5. Prediction Model **
6. KNN Imputation **

Removing cases with missing values

- To get a data frame with no missing values for any variable, use the na.omit() function:
  - `dat.cleaned <- na.omit(dat)`
- We might sometimes want to selectively eliminate cases that have NA only for a specific variable. The example data frame has two missing values for Income. To get a data frame with only these two cases removed, use:

```
> dat.income.cleaned <- dat[!is.na(dat$Income),]
> nrow(dat.income.cleaned)
[1] 25
```

Finding cases with no missing values

- The `complete.cases()` function takes a data frame or table as its argument and returns a Boolean vector with TRUE for rows that have no missing values, and FALSE otherwise:

```
> complete.cases(dat)

 [1]  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE
[10]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE
[19]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Rows 4, 6, 13, and 17 have at least one missing value.
Instead of using the `na.omit()` function, we can do the following as well:

```
> dat.cleaned <- dat[complete.cases(dat),]
> nrow(dat.cleaned)
[1] 23
```

Converting specific values to NA

- Sometimes, a specific value in a data frame means that the data was not available.
- For example, in the dat data frame, a value of 0 for Income probably means that the data is missing. We can convert these to NA by a simple assignment

```
> dat$Income[dat$Income==0] <- NA
```

Handling missing data

4. Replacing the values with mean, median, or mode

- **Generalized Imputation:** we calculate the mean or median for all non missing values of that variable then replace missing value with mean or median.

- **Similar case Imputation:** In this case, we calculate average for gender “**Male”** (29.75) and “**Female**” (25) individually of non missing values then replace the missing value based on gender

Replacing missing values with the mean

- Disregarding cases with any missing variables implies loosing useful information.
- You may sometimes want to impute reasonable values (those that will not skew the results of analysis very much) for the missing values.
- replace missing values with the mean

```
> dat <- read.csv("missing-data.csv", na.strings = "")
> dat$Income.imp.mean <- ifelse(is.na(dat$Income),
mean(dat$Income, na.rm=TRUE), dat$Income)
```

Understanding missing data

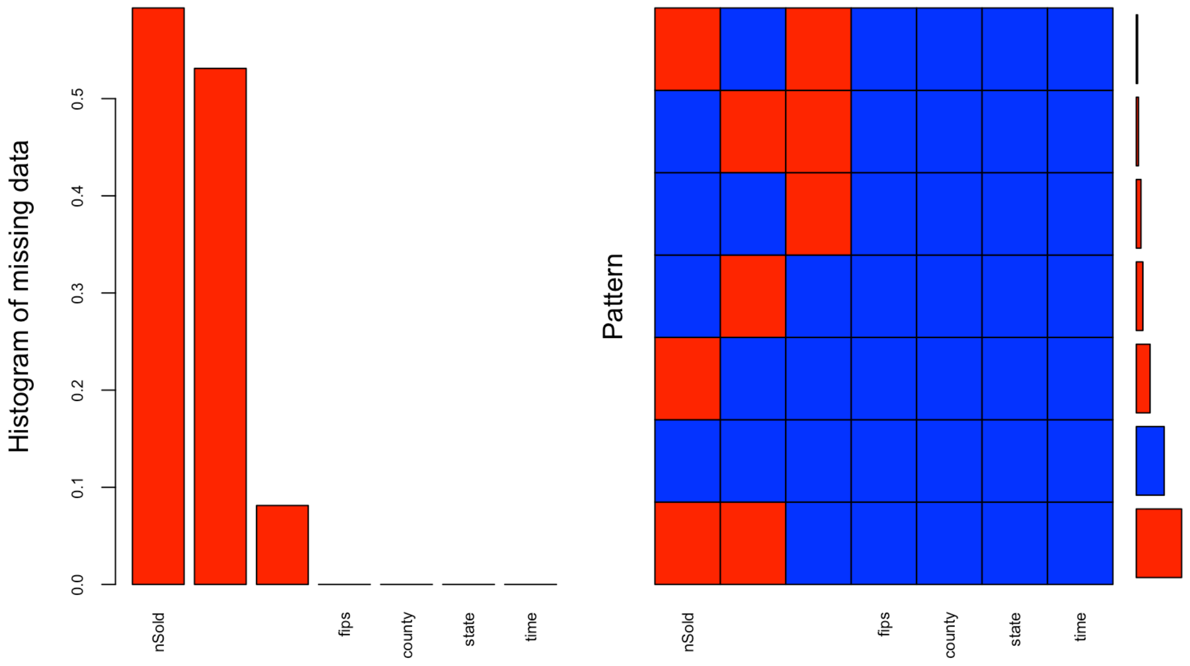
- use the md.pattern() function from the mice package to get a better understanding of the pattern of missing data.

```
> library(mice)
> install.packages("housingData")
> housing.dat<-housingData::housing
> str(housing.dat)
> md.pattern(housing.dat)
```

```
 fips county state time medListPriceSqft medSoldPriceSqft  nSold
 67740    1      1     1    1                1                1      1      0
 33453    1      1     1    1                1                1      0      1
 11403    1      1     1    1                0                1      1      1
 15929    1      1     1    1                1                0      1      1
  3259    1      1     1    1                0                1      0      2
109883    1      1     1    1                1                0      0      2
  5415    1      1     1    1                0                0      1      2
          0      0     0    0            20077           131227 146595 297899
```

The package VIM (visualization and imputation of missing values)

```
library(VIM)
aggr_plot <- aggr(housing.dat, col=c('blue','red'), numbers=TRUE,
        sortVars=TRUE, labels=names(housing.dat), cex.axis=.7, gap=3,
         ylab=c("Histogram of missing data","Pattern"))
```



Next we will visualize the housing data to understand missing information

```
 Variables sorted by
number of missings:
         Variable      Count
            nSold 0.59330506
 medSoldPriceSqft 0.53110708
 medListPriceSqft 0.08125642
             fips 0.00000000
           county 0.00000000
            state 0.00000000
             time 0.00000000
```

Identifying duplicate cases

- Sometimes we just want to identify the duplicated values without necessarily removing them for this, use the `duplicated()` function:

```
> duplicated(prospect)
[1] FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE
```

- From the data, we know that cases 2, 5, and 7 are duplicates. To list the duplicate cases, use the following code:

```
> prospect[duplicated(prospect), ]

  salary family.size     car
5  30000           3 Compact
7  30000           3 Compact
```

Removing duplicate cases

- Removing duplicates

```
> salary <- c(20000, 30000, 25000, 40000, 30000, 34000, 30000)
> family.size <- c(4,3,2,2,3,4,3)
> car <- c("Luxury", "Compact", "Midsize", "Luxury",     "Compact",
"Compact", "Compact")
>prospect <- data.frame(salary, family.size, car)
```

- The `unique()` function can do the job. It takes a vector or data frame as an argument and returns an object of the same type as its argument, but with duplicates removed.

```
>prospect
  salary family.size     car
1  20000           4  Luxury
2  30000           3 Compact
3  25000           2 Midsize
4  40000           2  Luxury
5  30000           3 Compact
6  34000           4 Compact
7  30000           3 Compact
```

```
> prospect.cleaned <- unique(prospect)
> nrow(prospect)
[1] 7
> nrow(prospect.cleaned)
[1] 5
```

Rescaling a variable to specified min-max range

- Distance computations play a big role in many data analytics techniques.
- We know that variables with higher values tend to dominate distance computations and you may want to rescale the values between [0,1]

```
> install.packages("scales")
➢ library(scales)
➢ library(mice)
> head(housing.dat)
> housing.dat$nSold.rescaled <- rescale(housing.dat$nSold)
```

```
 fips             county state       time  nSold medListPriceSqft medSoldPriceSqft nSold.rescaled
1 06037 Los Angeles County    CA 2008-01-31 505900               NA         360.1645      0.3162685
2 06037 Los Angeles County    CA 2008-02-29 497100               NA         353.9788      0.3103794
3 06037 Los Angeles County    CA 2008-03-31 487300               NA         349.7633      0.3038212
```

Rescaling a variable to specified min-max range

- To rescale use the function rescale() from the library "scales"

```
housing.dat

# rescale between [0,1]
housing.dat$nSold.rescaled <- rescale(housing.dat$nSold)
head(housing.dat)

# rescale to other ranges
housing.dat$nSold.rescaled
                  <- rescale(housing.dat$nSold, to = c(1, 100))
```

Rescaling many variables at once

- Here an example of function to rescale many variables:

```
rescale.many <- function(dat, column.nos) {
  nms <- names(dat)
  for(col in column.nos) {
    name <- paste(nms[col],".rescaled", sep = "")
    dat[name] <- rescale(dat[,col])
  }
  cat(paste("Rescaled ", length(column.nos),      " variable(s)n"))
  dat
}
```

- With the preceding function it is possible to rescale the fifth and seventh variables in the data frame:

```
> rescale.many(housing.dat, c(5,7))
```

Normalizing or standardizing data in a data frame

- The `scale()` function computes the standard z score for each value (ignoring NAs) of each variable. That is, from each value it subtracts the mean and divides the result by the standard deviation of the associated variable.
- The `scale()` function takes two optional arguments, center and scale:

Argument	Effect
`center = TRUE, scale = TRUE`	Default behavior described earlier
`center = TRUE, scale = FALSE`	From each value, subtract the mean of the concerned variable
`center = FALSE, scale = TRUE`	Divide each value by the root mean square of the associated variable, where root mean square is *sqrt(sum(x^2)/(n-1))*
`center = FALSE, scale = FALSE`	Return the original values unchanged

Binning numerical data

- Sometimes, we need to convert numerical data to categorical data or a factor:
  - *(e.g. Naive Bayes classification requires all variables to be categorical).*
  - In other situations, we may want to apply a classification method to a problem where the dependent variable is numeric but needs to be categorical

Binning numerical data

Income is a numeric variable, and you may want to create a categorical variable from it by creating bins.

- Suppose you want to label incomes of $10,000 or below as Low, incomes between $10,000 and $31,000 as Medium, and the rest as High

```
#Create a vector of break points:
> b <- c(-Inf, 10000, 31000, Inf)
#Create a vector of names for break points:
> names <- c("Low", "Medium", "High")
#Cut the vector using the break points with the function cut():
> students$Income.cat <- cut(students$Income, breaks = b, labels = names)
> head(students)
   Age State Gender Height Income Income.cat
1   23    NJ      F     61   5000        Low
2   13    NY      M     55   1000        Low
3   58    NY      F     70  30000     Medium
4   29    TX      F     63  10000        Low
```

Binning numerical data

If we leave out names, cut() uses the numbers in the second argument to construct interval names

```
#Create a vector of break points:
> b <- c(-Inf, 10000, 31000, Inf)
#Create a vector of names for break points:
> names <- c("Low", "Medium", "High")
#Cut the vector using the break points:
> students$Income.cat <- cut(students$Income, breaks = b)
> head(students)
   Age State Gender Height Income Income.cat     Income.cat1
1   23     NJ      F     61   5000        Low    (-Inf,1e+04]
2   13     NY      M     55   1000        Low    (-Inf,1e+04]
3   36     NJ      M     66   3000        Low    (-Inf,1e+04]
4   31     VA      F     64   4000        Low    (-Inf,1e+04]
5   58     NY      F     70  30000     Medium (1e+04,3.1e+04]
6   29     TX      F     63  10000        Low    (-Inf,1e+04]
7   39     NJ      M     67  50000       High  (3.1e+04, Inf]
```

Creating a specified number of intervals automatically

- Rather than determining the breaks and hence the intervals manually, as mentioned earlier, we can specify the number of bins we want, say n, and let the cut() function handle the rest automatically.
- In this case, cut() creates n intervals of approximately equal width, as follows:

```
>students$Income.cat2 <- cut(students$Income,
breaks = 4, labels = c("Level1", "Level2", "Level3","Level4"))
```

Creating dummies for categorical variables

- This is needed in situations where we have categorical variables (factors) but need to use them in analytical methods that require numbers:
  - *K nearest neighbors (KNN)*
  - *Linear Regression*

```
> install.packages("dummies")
> library(dummies)
> attach(Salaries)
> head(Salaries)
 rank discipline yrs.since.phd yrs.service    sex salary
1        Prof          B            19          18   Male 139750
2        Prof          B            20          16   Male 173200
3    AsstProf          B             4           3   Male  79750
```

Creating dummies for categorical variables

```
> names(Salaries)
[1] "rank"          "discipline"    "yrs.since.phd" "yrs.service"
"sex”  "salary"
str(Salaries)
data.frame':    397 obs. of  6 variables:
$ rank         : Factor w/ 3 levels "AsstProf","AssocProf",..: 3 3    $
discipline   : Factor w/ 2 levels "A","B": 2 2 2 2 2 2 2 2 2
$ yrs.since.phd: int  19 20 4 45 40 6 30 45 21 18 ...
$ yrs.service  : int  18 16 3 39 41 6 23 45 20 18 ...
$ sex          : Factor w/ 2 levels "Female","Male": 2 2 2 2 2 2 2 $
salary       : int  139750 173200 79750 115000 141500 97000


Salaries.dummy<- dummy.data.frame(Salaries, sep = ".")
```

Creating dummies for categorical variables

```
names(Salaries.dummy)
 [1] "rank.AsstProf"  "rank.AssocProf" "rank.Prof"      "discipline.A"
"discipline.B"  "yrs.since.phd"  "yrs.service"    "sex.Female"
"sex.Male"        "salary"
```

```
>head(Salaries.dummy)
  rank.AsstProf rank.AssocProf rank.Prof discipline.A discipline.B yrs.since.phd yrs.service
1             0              0         1            0            1            19          18
2             0              0         1            0            1            20          16
3             1              0         0            0            1             4           3
4             0              0         1            0            1            45          39
5             0              0         1            0            1            40          41
6             0              1         0            0            1             6           6
  sex.Female sex.Male salary
1          0        1 139750
2          0        1 173200
3          0        1  79750
4          0        1 115000
5          0        1 141500
6          0        1  97000
```

Detecting outliers

Outlier is an observation that appears far away and diverges from an overall pattern in a sample. Outlier can be of two types:

- Univariate
- Multivariate.

The ideal way to deal with them is to find out the reason of having these outliers. Causes of outliers can be classified in two broad categories:

- Artificial (Error) / Non-natural
- Natural.

Cleaning data

The five most common problems with messy datasets

- Column headers are values, not variable names.
- Multiple variables are stored in one column.
- Variables are stored in both rows and columns.
- Multiple types of observational units are stored in the same table.
- A single observational unit is stored in multiple tables.

Cleanning data packages in R

- **dplyr:** package for data manipulation
- **tidyr:** suitable for data tidying (splitting, joining variables) and it supports the dplyr package's data pipelines
- **reshape2:** it provides easy aggregations and restructuring of data
- **data.table:** allows fast aggregations and transformations of large data sets

Some packages specialize in addressing character variables, strings, and text or date and time information are:

- **stringi and stringr** t support work with strings and allow, for example, pattern searching, string generation, date and time formatting, parsing, ect…
- **lubridate:** helps R users with date and time stamps, their extraction and advanced manipulation.

tydir()

- `gather()` and `spread()` convert data between wide and long format
- `separate()` and `unite()` separate a single column into multiple columns and vice versa
- `complete()` turns implicit missing values in explicit missing values by completing missing data combinations

gather()

```
> messy
  id       trt    work.T1   home.T1   work.T2    home.T2
1  1 treatment 0.08513597 0.6158293 0.1135090 0.05190332
2  2   control 0.22543662 0.4296715 0.5959253 0.26417767
3  3 treatment 0.27453052 0.6516557 0.3580500 0.39879073
4  4   control 0.27230507 0.5677378 0.4288094 0.83613414
```

We want to gather all the columns, except for the id and trt ones, in two columns key and value

```
gathered.messy <-
gather(messy, key, value, -id, -trt)
```

```
  id       trt     key      value
1  1 treatment work.T1 0.08513597
2  2   control work.T1 0.22543662
3  3 treatment work.T1 0.27453052
4  4   control work.T1 0.27230507
5  1 treatment home.T1 0.61582931
6  2   control home.T1 0.42967153
```

spread()

Takes different levels of a factor and spreads them out into different columns. This means we can convert from long data to wide.

```
> stocksm
        time stock       price
1 2009-01-01     X -0.66184983
2 2009-01-02     X  1.71895416
3 2009-01-03     X  2.12166699
4 2009-01-04     Y  1.49715368
5 2009-01-05     Y -0.03614058
6 2009-01-06     Y  1.23194518
```

```
spread.stock <- spread(stocksm, stock, price)
head(spread.stock)
```

```
> spread.stock
        time           X          Y          Z
1 2009-01-01 -0.66184983 -0.7656438 -5.0672590
2 2009-01-02  1.71895416  0.5988432 -0.7943331
3 2009-01-03  2.12166699  1.3484795  0.5554631
```

Piping and chaining code

- dplyr allows you to use the pipe (%>%) operator to chain functions together.
- Chaining code allows you to streamline your workflow and make it easier to read.
- When using the %>% operator, first specify the data frame that all following functions will use.
- For the rest of the chain the data frame argument can be omitted from the remaining functions.

unite()

- unite(data, col, ..., sep = "_", remove = TRUE)

```
> data
        date hour min second event
1 2016-01-01    7  30     29     u
2 2016-01-02    9  43     36     a
3 2016-01-03   13  58     60     l
4 2016-01-04   20  22     11     q
5 2016-01-05    5  44     47     p
```

```
> dataNew
             datetime event
1   2016-01-01 7:30:29     u
2   2016-01-02 9:43:36     a
3  2016-01-03 13:58:60     l
4  2016-01-04 20:22:11     q
```

```
# Now, let us combine the date, hour, min, and
# second columns into a new column called datetime.
dataNew <- data %>%
        unite(datehour, date, hour, sep = ' ') %>%
        unite(datetime, datehour, min, second, sep = ':')
dataNew
```

separate()

- We can get back the original data we created using separate as follows:

```
data1 <- dataNew %>%
  separate(datetime, c('date', 'time'), sep = ' ') %>%
  separate(time, c('hour', 'min', 'second'), sep = ':')
data1
```

dyplyr()

- `filter()` subset data based on logical criteria
- `select()` select certain columns
- `arrange()` order rows by value of a column
- `rename()` rename columns
- `group_by()` group data by common variables for performing calculations
- `mutate()` create a new variable/column
- `summarize()` summarize data into a single row of values

filter()

- The filter function will return all the rows that satisfy a following condition.

```
filter(airquality, Temp > 70)
filter(airquality, Temp > 80 & Month > 5)
```

mutate()

- Mutate is used to add new variables to the data. For example, let’s adds a new column that displays the temperature in Celsius

```
mutate(airquality, TempInC = (Temp - 32) * 5 / 9)
```

Summarise

- The summarise function is used to summarise multiple values into a single value. It is very powerful when used in conjunction with the other functions in the dplyr package, as demonstrated below. na.rm = TRUE will remove all NA values while calculating the mean, so that it doesn’t produce spurious results.

```
> summarise(airquality, mean(Temp, na.rm = TRUE))
  mean(Temp, na.rm = TRUE)
1                 77.88235
```

Group by

- The group_by function is used to group data by one or more variables.
- For example, we can group the data together based on the Month, and then use the summarise function to calculate and display the mean temperature for each month.

```
summarise(group_by(airquality, Month), mean(Temp, na.rm = TRUE))
# A tibble: 5 x 2
  Month `mean(Temp, na.rm = TRUE)`
  <int>                      <dbl>
1     5                   65.54839
2     6                   79.10000
3     7                   83.90323
4     8                   83.96774
5     9                   76.90000
```

arrange()

- The arrange function is used to arrange rows by variables. Currently, the airquality dataset is arranged based on Month, and then Day.
- We can arrange the rows in the descending order of Month, and then in the ascending order of Day.

```
arrange(airquality, desc(Month), Day)
```

All together

```
msleep %>%
    group_by(order) %>%
    summarise(avg_sleep = mean(sleep_total),
              min_sleep = min(sleep_total),
              max_sleep = max(sleep_total),
              total = n())
```

```
## Source: local data frame [19 x 5]
##
##             order avg_sleep min_sleep max_sleep total
## 1     Afrosoricida 15.600000      15.6      15.6     1
## 2     Artiodactyla  4.516667       1.9       9.1     6
## 3        Carnivora 10.116667       3.5      15.8    12
## 4          Cetacea  4.500000       2.7       5.6     3
## 5       Chiroptera 19.800000      19.7      19.9     2
## 6        Cingulata 17.750000      17.4      18.1     2
## 7  Didelphimorphia 18.700000      18.0      19.4     2
## 8    Diprotodontia 12.400000      11.1      13.7     2
```