Data Cleaning

CITS4009 Computational Data Analysis

A/Prof Wei Liu

Department of Computer Science and Software Engineering
The University of Western Australia

Semester 2, 2023

Domain-Specific Data Cleaning

Take the custdata (Version 2) for example,

```
custdata_v2 <- readRDS('../../data_v2/Custdata/custdata.RDS')
```

- The variable gas_usage mixes numeric and symbolic data: values greater than 3 are monthly gas_bills, but values from 1 to 3 are special codes. In addition, gas_usage has some missing values.
- The variable age has the problematic value 0, which probably means that the age is unknown. In addition, there are a few customers with ages older than 100, which may also be an error.
- The variable income has negative values. We'll assume for this discussion that those values are invalid.

Section 1

Dealing with invalid values

Converting Invalid Values to NA

A quick way to treat the problem in the age and income variables is to **convert the invalid values to NA**, as if they were missing values.

We can then treat the NAs using the automatic missing-value treatment to be discussed a bit later.

We can use the `mutate()` and `na_if()` functions from the `dplyr` package:

- `mutate()` adds columns to a data frame or modifies existing columns.
- `na_if()` turns a specific problematic value into `NA`.

Example Code

```r
library(dplyr)

customer_data <-
  mutate(custdata_v2,
         age = na_if(age, 0),
         income = ifelse(income < 0, NA, income))
```

The code above creates a new data frame `customer_data` from `custdata_v2`, with

- zero values in the `age` column converted to `NA`s; and
- negative values in the `income` column converted to `NA`s.

Section 2

Dealing with sentinel values

Sentinel Values in `gas_usage`

The values 1, 2, and 3 of the `gas_usage` variable are not numeric values, but *sentinel values*, i.e., *special codes*, where:

- the value 1 means "Gas bill included in rent or condo fee."
- the value 2 means "Gas bill included in electricity payment."
- the value 3 means "No charge or gas not used."

One way to treat the `gas_usage` variable is to

- convert all the special codes (1, 2, 3) to NA, and
- add three new indicator variables, one for each code. The three new indicator variables can be `gas_with_rent`, `gas_with_electricity`, and `no_gas_bill`.

as done in the code on the next slide.

Example Code

```
customer_data <- customer_data %>%
  mutate(
    gas_with_rent = (gas_usage ==1),
    gas_with_electricity = (gas_usage == 2),
    no_gas_bill = (gas_usage == 3)
  ) %>%
  mutate(
    gas_usage = ifelse(gas_usage < 4,
                       NA,
                       gas_usage
                       )
  )
```

`%>%` is like the Unix pipe `|`: The data frame `customer_data` is passed to the first `mutate()` function, whose output is passed to the second `mutate()` function. The final output is stored back to `customer_data`.

Section 3

Dealing with Outliers

Outliers

If we suspect that there are outliers in a variable (column) in our dataset, we should consider dealing with them in the data cleaning process. For example, in the smaller customer dataset in `custdata.tsv`, the `income` variable has a negative value and some zero values.

```
count.zero <- sum(custdata$income == 0)
count.neg <- sum(custdata$income < 0)
cat("Number of customers having 0 incomes: ", count.zero)
## Number of customers having 0 incomes:  78
cat("Number of customers having negative incomes: ", count.neg)
## Number of customers having negative incomes:  1
```

We could treat

- `income = 0` as an indication that the customer was not working when the data was collected and
- negative incomes as outliers.

Outliers (cont.)

We could use `boxplot.stats()` to help us identify the outliers.

```
summary(custdata$income)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   -8700   14600   35000   53505   67000  615000
stat <- boxplot.stats(custdata$income)
stat$n           # number of non-NAs
## [1] 1000
stat$stats       # quartiles
## [1]  -8700  14600  35000  67000 145000
length(stat$out) # number of outliers
## [1] 69
cat("min, max outlying incomes are: ", min(stat$out), max(stat$out))
## min, max outlying incomes are:  148000 615000
```

We can see that `boxplot.stats()` picks up many high income values as outliers. Unfortunately, it is not perfect. it doesn't recognize that these are income values and should not be negative.

Outliers (cont.)

We could fix the problem by turning all the negative incomes to NA and run `boxplot.stats()` again:

```
# create a new variable called income.mod with negative incomes set to NA
custdata$income.mod <- ifelse(custdata$income < 0, NA, custdata$income)
stat <- boxplot.stats(custdata$income.mod)

stat$n           # number of non-NAs
## [1] 999
q <- stat$stats  # quartiles
q
## [1]      0  14700  35000  67000 145000
length(stat$out) # number of outliers
## [1] 69
cat("min, max outlying incomes are: ", min(stat$out), max(stat$out))
## min, max outlying incomes are:  148000 615000
```

Outliers (cont.)

Like the missing values issue that we will look at later on, we need to justify what to do with the high income values that are identified as outliers.

- How many outliers are there?
- If only a few, then we can omit them, e.g., set them to NA. In our case, there are `69` customers (or `6.9%`)
- However, there are `78` customers having zero income. They should perhaps be treated differently and excluded.

The `income` column in our data has a skew distribution. We will keep the original values for this column and take an alternative outlier treatment if needed in the future.

Note that outliers are not necessarily bad/incorrect values that must be removed.

(See **Practical Data Science with R**, Section 3.1.1, page 56)

Section 4

Dealing with Missing Values

Missing Values

- An important feature of R is that it allows for NA (“not available”).
- NA represents an **unknown** value.
- Missing values are “contagious”: almost any operation involving an unknown value will also be unknown.

```
NA > 5
## [1] NA
```

```
10 == NA
## [1] NA
```

Missing Values (cont.)

This one below is a bit hard to understand, but imagine if the first NA represent customer John's income, the second is Mary's income, both are unknown. Then certainly we don't know whether John and Mary have the same income or not.

```
NA == NA
## [1] NA
```

Use `is.na()` to check if a value is NA or not.

Treating missing values (NAs)

Strategies for treating missing values vary depending on the answers to the following two questions:

- How many?
- Why they are missing?

Fundamentally, there are two things you can do with these variables:

- Drop the rows with missing values, or
- Convert the missing values to a meaningful value.

Counting the missing values in each variable

```r
count_missing <- function(df) {
  sapply(df, FUN = function(col) sum(is.na(col)) )
}

nacounts <- count_missing(customer_data)
hasNA = which(nacounts > 0)
nacounts[hasNA]
##          is_employed               income          housing_type
##                25774                   45                  1720
##          recent_move         num_vehicles                   age
##                 1721                 1720                    77
##            gas_usage        gas_with_rent gas_with_electricity
##                35702                 1720                  1720
##          no_gas_bill
##                 1720
```

Recall that `customer_data` was created on slide 5 and three extra variables (columns) were added on slide 8. It has 73,262 observations and 15 variables. Ten of these variables (shown above) have NAs.

When it is safe to drop rows?

```
count_missing = function(df) {
   sapply(df, FUN=function(col) sum(is.na(col)) )
}
```

Defines a function that counts the number of NAs in each column of a data frame

```
nacounts <- count_missing(customer_data)
hasNA = which(nacounts > 0)
nacounts[hasNA]
```

Applies the function to customer_data, identifies which columns have missing values, and prints the columns and counts

```
##         is_employed            income        housing_type
##               25774                45                1720
##         recent_move      num_vehicles                 age
##                1721              1720                  77
##           gas_usage     gas_with_rent gas_with_electricity
##               35702              1720                1720
##         no_gas_bill
##                1720
```

`customer_data` has 73,262 rows, safe to drop rows with NAs in the `income` and `age` variables, but not the `is_employed` or `gas_usage` variable.

Checking locations of missing data

In this example, let's look at the smaller `custdata.tsv` dataset:

```
custdata <- read.table('../../data/custdata.tsv',
               header=T, sep='\t')
nacounts <- count_missing(custdata)
hasNA = which(nacounts > 0)
nacounts[hasNA]
```

```
##  is.employed housing.type  recent.move num.vehicles
##          328           56           56           56
```

Checking locations of missing data (cont.)

These variables are only missing a few values. It's probably safe to just drop the rows that are missing values—especially if the missing values are all in the same 56 rows.

```
                        housing.type recent.move     num.vehicles
Homeowner free and clear      :157   Mode :logical  Min.   :0.000
Homeowner with mortgage/loan:412     FALSE:820      1st Qu.:1.000
Occupied with no rent         : 11   TRUE :124      Median :2.000
Rented                        :364   NA's :56       Mean   :1.916
NA's                          : 56                  3rd Qu.:2.000
                                                    Max.   :6.000
                                                    NA's   :56
```

```
is.employed
Mode :logical
FALSE:73
TRUE :599
NA's :328
```

The is.employed variable is missing many values. Why? Is employment status unknown? Did the company only start collecting employment information recently? Does NA mean "not in the active workforce" (for example, students or stay-at-home parents)?

Missing values – are they from the same rows?

In an extremely unlikely case, all missing data may come from the same rows. In `custdata`, some missing data indeed come from the same rows!

```
summary(custdata[is.na(custdata$housing.type),
                c("recent.move","num.vehicles")])
##  recent.move     num.vehicles
##  Mode:logical   Min.   : NA
##  NA's:56        1st Qu.: NA
##                 Median : NA
##                 Mean   :NaN
##                 3rd Qu.: NA
##                 Max.   : NA
##                 NA's   :56
```

The summary above shows that the 56 customers whose `housing.type` values are missing also have their `recent.move` and `num.vehicles` missing — they are the same 56 customers!

Missing Values – removing all rows with NAs

Use `na.omit()` to remove incomplete observations

```
newdata <- na.omit(custdata)
nrow(custdata)
```

```
## [1] 1000
```

```
nrow(newdata)
```

```
## [1] 664
```

This removes all the rows that contain `NA`s.

Missing Values – removing selective rows

Another option is to use *subsetting* to remove rows that have `NA`s for certain variables only:

```
newdata <-
  custdata[!is.na(custdata$housing.type),]
nrow(custdata)
```

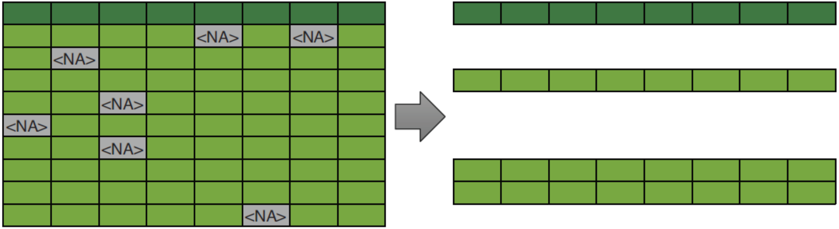
```
## [1] 1000
```

```
nrow(newdata)
```

```
## [1] 944
```

Should I drop rows if only a few values are missing?

Even for a few missing values, you can lose almost all your data!



Missing values – *To Drop* or *Not to Drop*?

- If only a small proportion of values are missing and they tend to be for the same data points, then consider dropping those rows from your analysis, this is called **listwise deletion**.
- If you are missing data for a particular variable from a large portion of the observations or NAs spread throughout the data, then consider:
  1. If the variable is categorical, then create a new category (e.g., *missing*) for the variable.
  2. If the variable is numerical,
     - when values are missing randomly, replace them with the mean value or an appropriate estimate, a.k.a. **imputing** missing values;
     - when values are missing systematically, convert them to categorical and add a new category, or replace them with zero and add a *masking variable*.

Missing values – categorical variables

Create a new category for the missing values, e.g., `missing` or `_invalid_`.

Example Code

In the code below, a masking variable `is.employed.fix` is created. It is the same as `is.employed` except that the NAs are mapped to a new category `missing`.

```
custdata$is.employed.fix <-
  ifelse(is.na(custdata$is.employed),
         "missing", ifelse(custdata$is.employed==T,
                           "employed", "not-employed"))
summary(as.factor(custdata$is.employed.fix))
##      employed      missing not-employed
##           599          328           73
```

Why having a masking variable?

- Better to have the original variable on hand, in case we second-guess our data cleaning and want to redo it.

Missing values – Investigate a bit further

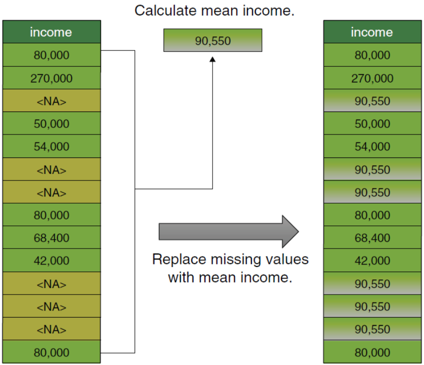
The fix will get the job done, but as a data scientist, one ought to be interested in why so many records are missing certain information.

In the case of the above example, the NAs might actually encode that the customer is not in the active workforce: they are a homemaker, a student, retired, or otherwise not seeking paid employment.

So the category "missing" can be better named as "not in active workforce".

Missing values – Numerical variables

- One might believe that the data collection failed at random so the missing values are independent of other variables. In this case, the missing values can be replaced by the mean or an appropriate estimate (e.g., the median).



Missing values – imputing with better estimate

The estimate can be improved (potentially better than *mean*) if other variables that relate to it are used for prediction.

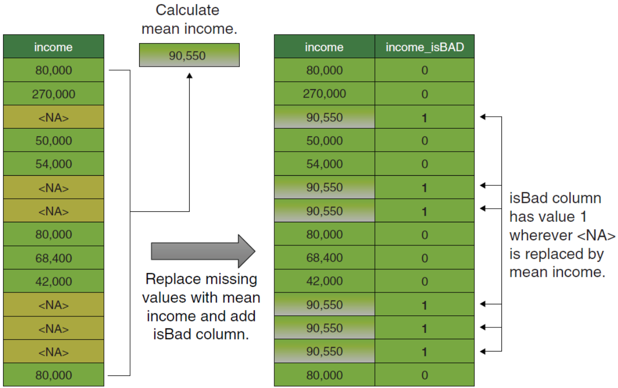
- For instance, from data exploration we know *income* is related to *age*, *state of residence* and *marital status*.
- We can use a *regression model* to predict the missing income if other variables for that data point are available.
- Other models such as *clustering* are also applicable.

How to best impute missing values is still under active research.

Note: imputing a missing value of an input variable based on the other input variables can be applied to categorical data as well.

Missingness Indicator

A trick that has worked well is to not only replace the `NA`s with the mean, but also add an additional indicator variable (e.g., `isBAD`) to keep track of which data points have been altered.



income
80,000
270,000
<NA>
50,000
54,000
<NA>
<NA>
80,000
68,400
42,000
<NA>
<NA>
<NA>
80,000

income	income_isBAD
80,000	0
270,000	0
90,550	1
50,000	0
54,000	0
90,550	1
90,550	1
80,000	0
68,400	0
42,000	0
90,550	1
90,550	1
90,550	1
80,000	0

Why *Missingness Indicators* can be useful

The idea is that at the modelling step, you give all the variables – `income`, `income_isBAD`, `gas_usage`, `no_gas_bill`, and so on – to the modelling algorithm, and it can determine how to best use the information to make predictions.

- If the missing values really are missing randomly, then the indicator variables are uninformative, and the model should ignore them.
- If the missing values are missing systematically, then the indicator variables provide useful additional information to the modelling algorithm.
- In many situations, the `isBAD` variables are sometimes even *more* informative and useful than the original variables!

Take home messages

- Strategies for dealing with missing values depend on how many missing values there are, and whether they are missing randomly or systematically.
- When in doubt, assume that missing values are missing systematically.

The `vtreat` package

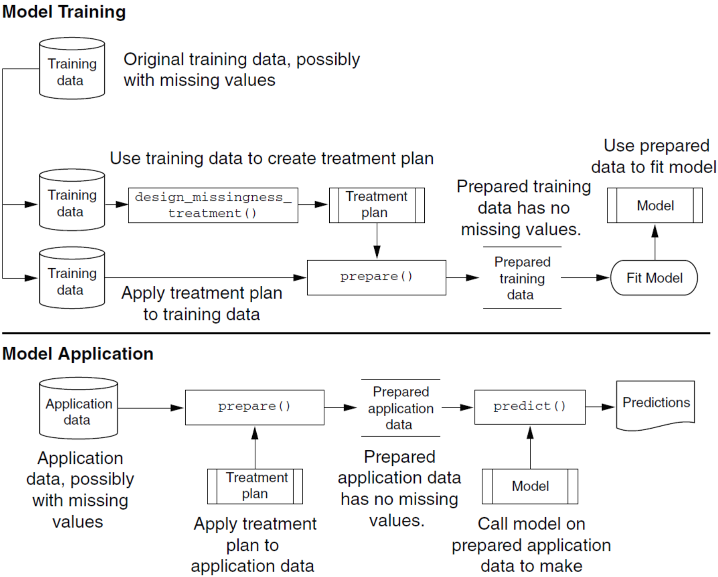
Missing values are such a common problem with data, it's useful to have an automatic and repeatable process for dealing with them.

`vtreat` is a package for automatically treating missing values. It creates a `treatment plan` that records all the information needed so that the data treatment process can be repeated. You then use this treatment plan

- to "prepare" or treat your training data before you fit a model, and
- then again to treat new data before feeding it into the model.

The idea is that treated data is "*safe*", with no missing or unexpected values, and should not ruin the model.

Creating and Applying a Simple Treatment Plan



Sample Code

```
library(vtreat)

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

treatment_plan <- design_missingness_treatment(
  customer_data, varlist = varlist)

training_prepared <- prepare(treatment_plan,
                             customer_data)

nacounts <- count_missing(training_prepared)
sum(nacounts)
```

```
## [1] 0
```

Examining the data treatment

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

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

customer_data[missing.ht, columns_to_look_at] %>% head()
##            custid is_employed num_vehicles housing_type hea
## 80  000082691_01        TRUE           NA         <NA>
## 100 000116191_01        TRUE           NA         <NA>
## 237 000269295_01          NA           NA         <NA>
## 299 000349708_01          NA           NA         <NA>
## 311 000362630_01          NA           NA         <NA>
## 413 000443953_01          NA           NA         <NA>
```

Examining the data treatment (cont.)

```
columns_to_look_at = c("custid",
   "is_employed", "is_employed_isBAD", "num_vehicles",
   "num_vehicles_isBAD", "housing_type", "health_ins")
training_prepared[missing.ht, columns_to_look_at] %>% head()
##            custid is_employed is_employed_isBAD num_vehicles num_vehicles_isBAD
## 80  000082691_01   1.0000000                 0       2.0655                  1
## 100 000116191_01   1.0000000                 0       2.0655                  1
## 237 000269295_01   0.9504928                 1       2.0655                  1
## 299 000349708_01   0.9504928                 1       2.0655                  1
## 311 000362630_01   0.9504928                 1       2.0655                  1
## 413 000443953_01   0.9504928                 1       2.0655                  1
##     housing_type health_ins
## 80     _invalid_      FALSE
## 100    _invalid_       TRUE
## 237    _invalid_      FALSE
## 299    _invalid_      FALSE
## 311    _invalid_       TRUE
## 413    _invalid_       TRUE
```

References

- **Practical Data Science with R**, *Nina Zumel, John Mount*, Manning, 2nd Ed., 2020 (Chapters 3 & 4)
- **R in Action**, *Robert I. Kabacoff*, Manning, 2011 (Chapter 4)

Converting Continuous Variables to Categorical

CITS4009 Computational Data Analysis

A/Prof Wei Liu

Department of Computer Science and Software Engineering
The University of Western Australia

Semester 2, 2023

Section 1

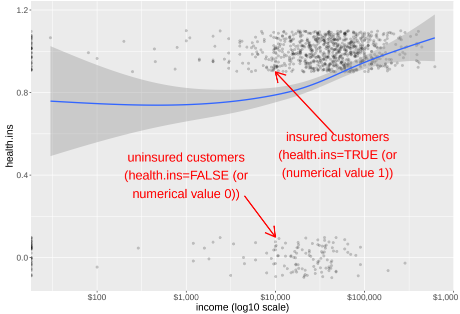
Recoding Variables

Recoding variables

- Change a continuous variable into a set of categories
- Create a pass/fail variable based on a set of cutoff scores
- Replace miscoded values with correct values

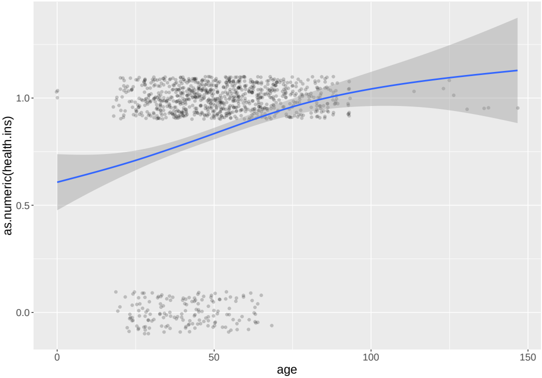
Discretizing continuous variables – Motivation

```
library(scales)
ggplot(data = custdata,
       mapping = aes(x=income, y=as.numeric(health.ins))) +
  geom_jitter(alpha=1/5, height = 0.1) + geom_smooth() +
  scale_x_log10(breaks = c(100,1000,10000,100000,1000000), labels=dollar) +
  labs(x="income (log10 scale)", y="health.ins")
```



Discretizing continuous variables – Motivation (cont.)

```
ggplot(data = custdata,
       mapping = aes(x=age, y=as.numeric(health.ins))) +
  geom_jitter(alpha = 1/5, height = 0.1) + geom_smooth() +
  theme(text = element_text(size=16))
```



Discretizing continuous variables – Motivation (cont.)

For some continuous variables, their exact values matter less than whether they fall into a certain range. For example,

- Customers with incomes less than $20,000 have different health insurance patterns than customers with higher incomes.
- Customers younger than 25 and older than 65 have high probabilities of insurance coverage, because they tend to be on their parents' coverage or on a retirement plan, respectively, whereas customers between those ages have a different pattern.

In these cases, you might want to convert the continuous `age` and `income` variables into ranges, or discrete variables. Discretizing continuous variables is useful when the relationship between input and output isn't linear.

Converting income into range

```
custdata$income.lt.20K <- custdata$income < 20000
summary(custdata$income.lt.20K)
```

```
##    Mode   FALSE    TRUE
## logical     678     322
```

Converting age into range

Use the `cut()` function, which specifies the category names automatically.

```
brks <- c(0, 25, 65, Inf)
custdata$age.range <- cut(custdata$age,
                          breaks=brks, include.lowest=T)
summary(custdata$age.range)
```

```
##    [0,25]  (25,65] (65,Inf]
##        56      732      212
```

The code above creates a new categorical variable `age.range`, which has 3 categories.

Infinite age – immortal?

Age values extending to `Inf` is beyond reality for mortals. :-) In fact, age values over 120 might even be data entry errors. We can treat them as missing or unknown values.

```
# altering the original data
custdata$age[custdata$age > 120] <- NA
# or create a new variable
custdata$age.alt <- ifelse(custdata$age > 120,
                           NA, custdata$age)
# then convert custdata$age.alt into range
```

Explicit categorisation

We could also define the categorization explicitly.

```
custdata$agecat[custdata$age > 120] <- NA
custdata$agecat[custdata$age > 65
                & custdata$age <= 120] <- "Elder"
custdata$agecat[custdata$age > 25
                & custdata$age <= 65] <- "Middle Aged"
custdata$agecat[custdata$age <= 25] <- "Young"
```

More explicit categorisation

The code can be written more compactly using the `within()` function. We

- first create a variable `agecat`, and set to missing (`NA`) for each row of the data.
- then execute the remaining statements in the curly braces in order.

```
custdata <- within(custdata, {
  agecat <- NA
  agecat[age > 120] <- NA
  agecat[age > 65 & age <= 120] <- "Elder"
  agecat[age > 25 & age <= 65] <- "Middle Aged"
  agecat[age <= 25] <- "Young" })
```

Note: `agecat` is of string type. It needs to be converted to factor.

```
custdata$agecat <- factor(custdata$agecat)
```

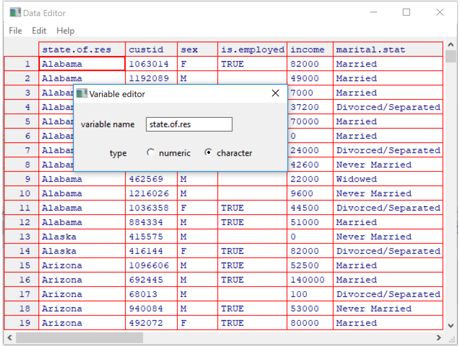
Section 2

Renaming Variables

Renaming variables Manually via the Interactive Editor

- Use `fix()` to invoke the interactive editor.

```r
fix(custdata)
```



	state.of.res	custid	sex	is.employed	income	marital.stat
1	Alabama	1063014	F	TRUE	82000	Married
2	Alabama	1192089	M		49000	Married
3	Alabar				7000	Married
4	Alabar				37200	Divorced/Separated
5	Alabar				70000	Married
6	Alabar				0	Married
7	Alabar				24000	Divorced/Separated
8	Alabar				42600	Never Married
9	Alabama	462569	M		22000	Widowed
10	Alabama	1216026	M		9600	Never Married
11	Alabama	1036358	F	TRUE	44500	Divorced/Separated
12	Alabama	884334	M	TRUE	51000	Married
13	Alaska	415575	M		0	Never Married
14	Alaska	416144	F	TRUE	82000	Divorced/Separated
15	Arizona	1096606	M	TRUE	52500	Married
16	Arizona	692445	M	TRUE	140000	Married
17	Arizona	68013	M		100	Divorced/Separated
18	Arizona	940084	M	TRUE	53000	Never Married
19	Arizona	492072	F	TRUE	80000	Married

Rename variables programmatically

The dplyr package has a rename() function that's useful for altering the names of variables.

```
rename(dataframe, newname=oldname, newname=oldname, ...)
```

For example,

```
library(dplyr)
custdata <- rename(custdata, age.cat=agecat, gender=sex)
names(custdata)
##  [1] "custid"          "gender"          "is.employed"
##  [5] "marital.stat"    "health.ins"      "housing.type"
##  [9] "num.vehicles"    "age"             "state.of.res"
## [13] "income.lt.20K"   "age.range"       "age.cat"
```

References

- **Practical Data Science with R**, *Nina Zumel, John Mount*, Manning, 2nd Ed., 2020 (Chapter 4)
- **R in Action**, *Robert I. Kabacoff*, Manning, 2011 (Chapter 4)

Dealing with Date and Time

CITS4009 Computational Data Analysis

A/Prof Wei Lu

Department of Computer Science and Software Engineering
The University of Western Australia

Semester 2, 2023

Date and time can get complicated

The more you learn about dates and times, the more complicated they seem to get.

- Does every year have 365 days?
- Does every day have 24 hours?
- Does every minute have 60 seconds?

Not every year has 365 days, but do you know the full rule for determining if a year is a leap year?

Many parts of the world use daylight savings time (DST), so that some days have 23 hours, and others have 25.

Some minutes have 61 seconds because every now and then leap seconds are added because the Earth's rotation is gradually slowing down.

The 100/400 exclusion rule of leap years

In the Gregorian calendar, a normal year consists of 365 days.

- The actual length of a sidereal year (the time required for the Earth to revolve once about the Sun) is actually 365.2425 days.
  - A "leap year" of 366 days is used once every four years to eliminate the error caused by three normal (but short) years.
- However, there is still a small error that must be accounted for.
  - To eliminate this error, the Gregorian calendar stipulates that a year that is evenly divisible by 100 (for example, 1900) is a leap year only if it is also evenly divisible by 400.

https://docs.microsoft.com/en-us/office/troubleshoot/excel/determine-a-leap-year

Number of days in a month or a year

```
library(Hmisc)

monthDays(as.Date('2020-02-01'))
## [1] 29
monthDays(as.Date('2022-02-01'))
## [1] 28
monthDays(as.Date('2000-02-01'))
## [1] 29
monthDays(as.Date('1900-02-01'))
## [1] 28

yearDays(as.Date('1900-02-01'))
## [1] 365
```

Calendar dates and times

There are three types of date/time data that refer to an instant in time:

- A date.
- A time within a day.
- A date-time is a date plus a time: it uniquely identifies an instant in time (typically to the nearest second)

Section 1

Parsing Dates

Parsing dates

- Dates are typically entered into R as character strings and then translated into *date* variables that are stored numerically.
- The function `as.Date()` is used to make this translation.
- The syntax is `as.Date(x, "input_format")`, where `x` is the character string and `input_format` gives the appropriate format for reading the date.

Symbol	Meaning	Example
`%d`	Day as a number	(1-31) 01-31
`%a`	Abbreviated weekday	Mon
`%A`	Unabbreviated weekday	Monday
`%m`	Month	(1-12) 01-12
`%b`	Abbreviated month	Jan
`%B`	Unabbreviated month	January
`%y`	2-digit year	18
`%Y`	4-digit year	2018

Date Example

The default format for inputting dates is `yyyy-mm-dd`.

This statement converts the character string vector using default format.

```
mydates <- as.Date(c("2007-06-22", "2004-02-13"))
class(mydates)
## [1] "Date"
```

In contrast, the following reads the data using the `mm/dd/yyyy` format.

```
strDates <- c("01/05/1965", "08/16/1975")
dates <- as.Date(strDates, "%m/%d/%Y")
```

Current dates

`Sys.Date()` returns today's date, of class type `Date`.

```
Sys.Date()
## [1] "2023-08-18"
```

`date()` returns the current date and time, of class type `character`.

```
date()
## [1] "Fri Aug 18 16:49:48 2023"
```

`Sys.time()` contains timezone, of class `c("POSIXct" "POSIXt")`.

```
Sys.time()
## [1] "2023-08-18 16:49:48 AWST"
```

Section 2

Formatting dates

Formatting dates

We can use `format(x, format="output_format")` to format a date variable:

```
today <- Sys.Date()
format(today, format="%B %d %Y")
```

```
## [1] "August 18 2023"
```

```
format(today, format="%A")
```

```
## [1] "Friday"
```

Extracting information from date/time

`weekdays()` and `months()` return a character vector of names in the locale in use.

```
weekdays(Sys.time())
## [1] "Friday"
```

`quarters()` returns a character vector of "`Q1`" to "`Q4`".

```
quarters(Sys.time())
## [1] "Q3"
```

Section 3

Extracting information from dates

Extracting information from date/time

`julian()` returns the number of days (possibly fractional) since the *origin* day (1970 Jan 1st), which can be changed by the `origin` argument.

All of the time calculations in R are done ignoring leap-seconds.

```
julian(Sys.time())
## Time difference of 19587.37 days
```

```
julian(Sys.time(), origin = as.Date("2022-08-01"))
## Time difference of 382.3679 days
```

Section 4

Using dates in calculation

Dates used for calculation

When R stores dates internally, they're represented as the number of days since **January 1, 1970**, with negative values for earlier dates.

We can check the number of days using `as.double()`

```
dob <- as.Date("1956-10-12")
dob_num_days <- as.double(dob)
dob_num_days
## [1] -4829
```

We can also convert this number back to a date object:

```
as.Date(dob_num_days, origin=as.Date("1970-01-01"))
## [1] "1956-10-12"
```

Finding Time Difference

```
today <- Sys.Date()

d <- difftime(today, dob, units="weeks")
class(d)
## [1] "difftime"
d
## Time difference of 3488 weeks
```

Use Time Difference in Calculation (class `difftime`)

```
difftime(time1, time2, tz,
         units = c("auto", "secs", "mins", "hours",
                   "days", "weeks"))
```

To get the number weeks for calculation, we can use `as.double(x)`, where `x` is of class `difftime`.

```
age <- as.double(d)/52
age
## [1] 67.07692
```

Take home messages

- Parsing dates
- Formatting dates
- Extracting information out of date/time values
- Use date/time for calculation