Normalisation and Scaling

CITS4009 Computational Data Analysis

Dr. Mubashar Hassan

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

Semester 2, 2024

Section 1

Data Transformation

Basic Data Transformation actions

- Recoding variables (previous lecture)
- Renaming variables (previous lecture)
- Dealing with Missing values (previous lecture)
- Dealing with Dates (previous lecture)
- Type Conversions
- Sorting data
- Merging datasets
- Subsetting datasets
- Use SQL statements to manipulate data frames
- Use pipes
- Sampling

Section 2

Normalisation and Scaling

Normalise by mean

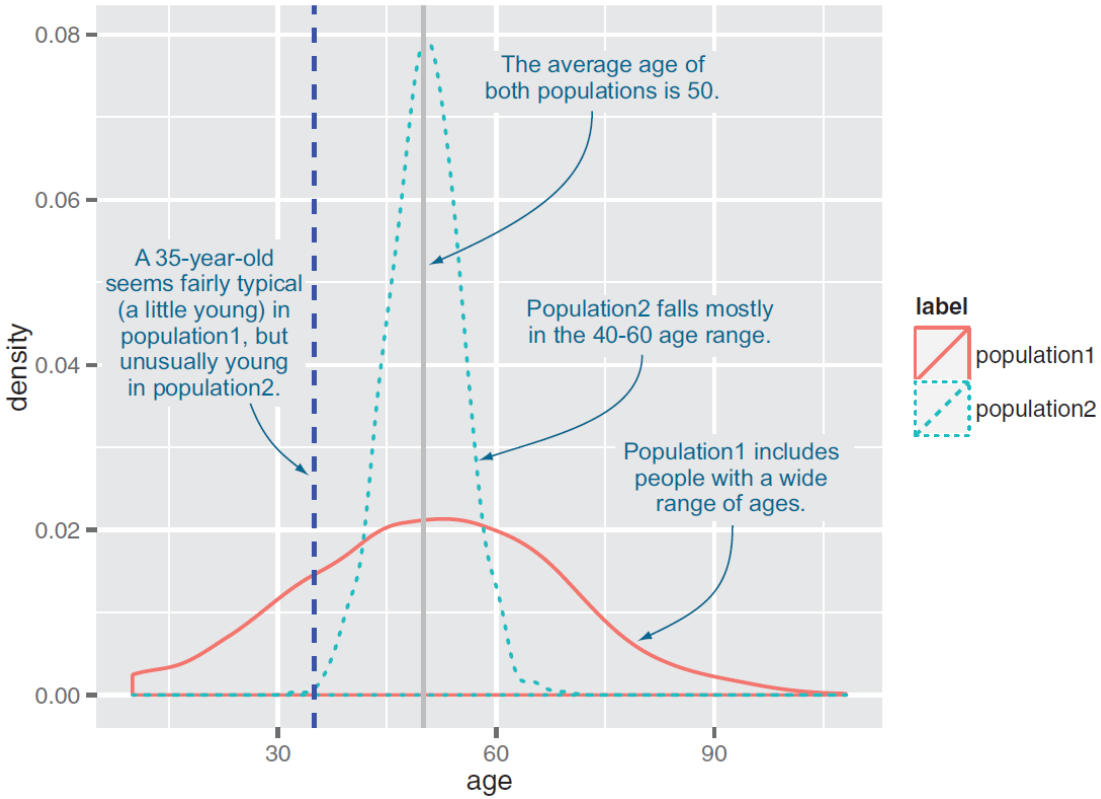
Normalisation is useful when absolute quantities are less meaningful than relative ones.

```
mean.age <- mean(custdata$age)
custdata$age.normalised <- custdata$age/mean.age
```

The age much less than 1 signifies an unusually young customer; much greater than 1 signifies an unusually old customer.

But what constitutes "much less" or "much greater" than 1? — that depends on the age spread of your customers.

Z-normalisation




```
mean.age <- mean(custdata$age)
std.age <- sd(custdata$age)
custdata$age.normalised <- (custdata$age-mean.age)/std.age
```

Z-normalisation (cont.)

```
cat("mean.age =", mean.age, "std.age =", std.age)
## mean.age = 51.69981 std.age = 18.86343
```

```
# ages and normalised ages of 6 random customers
indices <- sample(1:nrow(custdata), 6, replace=F)
custdata[indices, c("age", "age.normalised")]
##     age age.normalised
## 610  56     0.22796409
## 92   53     0.06892623
## 659  37    -0.77927566
## 413  34    -0.93831352
## 6    40    -0.62023781
## 348  24    -1.46843970
```

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

Standard Deviation

The common interpretation of standard deviation as a unit of distance implicitly assumes that the data is distributed normally.

For a normal distribution,

- roughly two-thirds of the data (about 68%) is within ± 1 standard deviation from the mean.
- About 95% of the data is within ± 2 standard deviations from the mean.

(`pnorm(1) - pnorm(-1) = 0.6826895`;
`pnorm(2) - pnorm(-2) = 0.9544997`)

You can still use this transformation even if the data isn't normally distributed, but the standard deviation is most meaningful as a unit of distance if the data is unimodal and roughly symmetric around the mean.

Take home messages

- Appropriate data transformations can make the data easier to understand and easier to model.
- Normalisation and re-scaling are important when relative changes are more important than absolute ones.

References

- **Practical Data Science with R**, *Nina Zumel, John Mount*, Manning, 2nd Ed., 2020 (Chapter 4)

Selecting and Filtering — Subsetting

CITS4009 Computational Data Analysis

Dr. Mubashar Hassan

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

Semester 2, 2024

Section 1

Type conversion

Type conversion in R

R provides a set of functions to identify an object's data type and convert it to a different data type.

- **Implicit type coercion** — similar to other weakly typed programming languages, R automatically determines the data type of an operation.
  - For example, adding a character string to a numeric vector converts all the elements in the vector to character values.
- **Explicit type conversion using functions**.

Functions to check and convert to different types

Test	Convert
`is.numeric()`	`as.numeric()`
`is.character()`	`as.character()`
`is.vector()`	`as.vector()`
`is.matrix()`	`as.matrix()`
`is.data.frame()`	`as.data.frame()`
`is.factor()`	`as.factor()`
`is.logical()`	`as.logical()`

- Functions in the 1st column are for type checking.
- Functions in the 2nd column are for type conversion. Depending on the argument passed to the function, the conversion may not be always successful/possible.

We can also use the `str()` function to inspect a variable's type and do conversion if needed.

Functions to check and convert to different types (cont.)

A few examples:

```
x <- c(7,8);  y <- "Hello"
z <- list(name=c("Rose","Jon"), height=c(1.6,1.7))
cat(is.vector(x), is.matrix(x), is.character(y), is.list(z))
## TRUE FALSE TRUE TRUE
```

```
str(y)
##  chr "Hello"
```

```
x <- as.character(x)   # convert x to vector of character type
cat(is.vector(x), is.numeric(x), is.character(x))
## TRUE FALSE TRUE
```

```
as.logical("hello")  # can't convert this
## [1] NA
```

```
as.numeric("hello")  # can't convert this
## [1] NA
```

Section 2

Sorting Data

Sorting data in R using the `order()` function

- By default, the sorting order is ascending.
- Prepend the sorting variable with a minus sign to indicate descending order.

E.g., create a new data frame with the customers firstly sorted by `sex` in ascending order (female then male), then by `income` in descending order:

```
attach(custdata)
newdata <- custdata[order(sex, -income),]
detach(custdata)
```

The `attach()` function is used so that we do not have to prefix each variable name by its data frame name.

Section 3

Subsetting datasets

Selecting or dropping variables (columns)

- Selecting variables

```
myvars <- c("custid", "is.employed", "income",
            "marital.stat", "health.ins", "age")
newdata <- custdata[myvars] # newdata has 6 columns
```

- Excluding (dropping) variables

```
myvars <- names(custdata) %in% c("sex", "state.of.res")
myvars
##  [1] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
newdata <- custdata[!myvars] # newdata has 11 - 2 = 9 columns
```

Selecting or dropping observations (rows)

```
newdata <- custdata[1:3,]
newdata <- custdata[which(custdata$sex=="M" &
                                  custdata$age < 30),]
cat(nrow(custdata), nrow(newdata))
## 1000 59
```

```
attach(custdata)
newdata <- custdata[which(sex=="M" & age > 30),]
detach(custdata)
cat(nrow(custdata), nrow(newdata))
## 1000 560
```

In each of these examples, we provide the row indices and leave the column indices blank (therefore choosing all columns).

Using the subset() function for both

Calling format:

```
subset(<x>, <subset>, <select>, ...)
   <x>      - object to subset
   <subset> - logical expression indicating rows to keep
   <select> - expression, indicating columns to select
```

Example 1. Select rows using logical expression and columns using their names:

```
newdata <- subset(custdata, age >= 65 | age < 24,
                  select=c("custid", "marital.stat"))
```

Example 2. Select columns explicitly using col_start:col_end:

```
# custid is column 1; age is column 10
newdata <- subset(custdata, sex=="M" & age < 25,
                  select=custid:age)
```

Section 4

Using SQL to manipulate data frames

The SQL Data Frame Library - sqldf

For those who like the convenience of **Structured Query Language** (**SQL**), the sqldf package provides querying of data frames using SQL.

```
library(sqldf)
# sql is not case sensitive, so "income" and "Income"
# are considered the same
newdf <- sqldf("select * from custdata where
               income <= 1000 order by age",
               row.names=TRUE)
# kable() requires the knitr library
kable(newdf[1:5, c("custid","sex","is.employed","income","age","health.ins"
```

	custid	sex	is.employed	income	age	health.ins
178	220142	F	NA	0	18	TRUE
326	456859	M	TRUE	80	18	TRUE
352	489718	M	NA	0	21	TRUE
949	1356541	F	NA	0	21	TRUE
861	1219346	M	NA	0	23	FALSE

Take home messages

- Misc: type conversion & sorting
- Subsetting using `subset()` or `sqldf()`

References

- **Practical Data Science with R**, *Nina Zumel, John Mount*, Manning, 2nd Ed., 2020 (Chapter 5: Sections 5.1-5.4)
- **R for Data Science**, *Hadley Wickham, Garrett Grolemund*, https://r4ds.had.co.nz/ (Chapters 11,12,13,18)

Merging Datasets

CITS4009 Computational Data Analysis

Dr. Mubashar Hassan

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

Semester 2, 2024

Section 1

Combining datasets

Adding observations (rows) to a data frame: `rbind()`

Vertical concatenation is typically used to add observations to a data frame.

```
total <- rbind(dataframeA, dataframeB)
```

The two data frames must have the same variables, but they don't have to be in the same order.

If `dataframeA` has variables that `dataframeB` doesn't, then before joining them do one of the following:

- Delete the extra variables in `dataframeA`
- Create the additional variables in `dataframeB` and set them to NA (missing)

Combining variables (columns) of two data frames: cbind()

The two data frames must have the same number of rows.

Example:

```
df1 <- custdata[, c("custid","age")]
df2 <- custdata["income"]
# suppose that we now want to merge these two data frames
newdata <- cbind(df1, df2)
cat(nrow(newdata), ncol(newdata))
```

```
## 1000 3
```

```
cat(names(newdata))
```

```
## custid age income
```

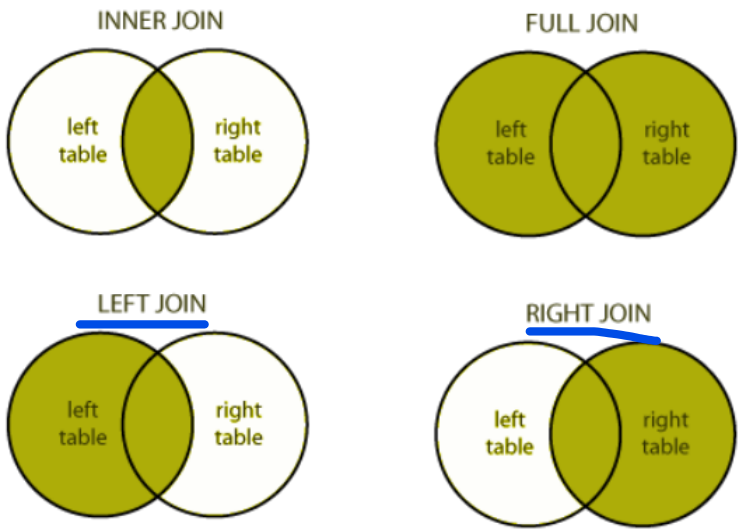
Section 2

Dealing with Relational Data using merge()

merge() – dealing with relational data using *join*

When working with a pair of tables, merge() will add new variables to one of the data frames using one of the following criteria:

- **Inner join** — only matching observations will be kept.
- **Left (or right) outer join** — matching observations as well as unmatched ones from left (or right) will be kept.
- **Full outer join** — matching observations as well as all unmatched ones from both tables are kept.



An author-book example - authors

```
authors <- data.frame(
    surname = c("Tukey", "Venables", "Ripley",
                "Tierney", "Winton"),
    nationality = c("US", "Australia", "NZ", "US", "UK"),
    deceased = c("yes", "yes", rep("no", 3)))
# kable() requires the knitr library
kable(authors)
```

surname	nationality	deceased
Tukey	US	yes
Venables	Australia	yes
Ripley	NZ	no
Tierney	US	no
Winton	UK	no

Our first data frame authors has 5 rows and 3 columns.

An author-book example - books

```
books <- data.frame(
    name = I(c("Tukey", "Venables", "Tierney",
               "Ripley", "Ripley", "McNeil", "R Core")),
    title = c("Exploratory Data Analysis", "Modern Applied Statistics",
              "LISP-STAT", "Spatial Statistics", "Stochastic Simulation",
              "Interactive Data Analysis", "An Introduction to R"),
    other.author = c(NA, "Ripley", NA, NA, NA, NA, "Venables & Smith"))
kable(books)
```

name	title	other.author
Tukey	Exploratory Data Analysis	NA
Venables	Modern Applied Statistics	Ripley
Tierney	LISP-STAT	NA
Ripley	Spatial Statistics	NA
Ripley	Stochastic Simulation	NA
McNeil	Interactive Data Analysis	NA
R Core	An Introduction to R	Venables & Smith

Inner join

authors and books do not have a *key* column (a common column) of the same name for merging. We can rename the *key* column in one of the data frames.

```
# rename "surname" to "name" in the authors data frame
authorN <- within(authors, { name <- surname; rm(surname) })
kable(authorN)
```

nationality	deceased	name
US	yes	Tukey
Australia	yes	Venables
NZ	no	Ripley
US	no	Tierney
UK	no	Winton

Inner Join (cont.)

```
# R finds columns of matching names
m0 <- merge(authorN, books)
kable(m0)
```

name	nationality	deceased	title	other.author
Ripley	NZ	no	Spatial Statistics	NA
Ripley	NZ	no	Stochastic Simulation	NA
Tierney	US	no	LISP-STAT	NA
Tukey	US	yes	Exploratory Data Analysis	NA
Venables	Australia	yes	Modern Applied Statistics	Ripley

- Only rows having matching values in the common column are retained in the output m0.

Inner Join (cont.)

Alternatively, we can explicitly state the matching columns in the two data frames:

```
m1 <- merge(authors, books, by.x = "surname", by.y = "name")
kable(m1)
```

surname	nationality	deceased	title	other.author
Ripley	NZ	no	Spatial Statistics	NA
Ripley	NZ	no	Stochastic Simulation	NA
Tierney	US	no	LISP-STAT	NA
Tukey	US	yes	Exploratory Data Analysis	NA
Venables	Australia	yes	Modern Applied Statistics	Ripley

Left Outer Join

We can specify that we want a *left outer join*:, i.e., *all* the rows in the left table (authorN) to appear in the output data frame:

```
m2 <- merge(authorN, books, all.x = TRUE)
kable(m2)
```

name	nationality	deceased	title	other.author
Ripley	NZ	no	Spatial Statistics	NA
Ripley	NZ	no	Stochastic Simulation	NA
Tierney	US	no	LISP-STAT	NA
Tukey	US	yes	Exploratory Data Analysis	NA
Venables	Australia	yes	Modern Applied Statistics	Ripley
Winton	UK	no	NA	NA

(Default value for `all.x` is `FALSE`)

Right Outer Join

```
m3 <- merge(authorN, books, all.y = TRUE)
kable(m3)
```

name	nationality	deceased	title	other.author
McNeil	NA	NA	Interactive Data Analysis	NA
R Core	NA	NA	An Introduction to R	Venables & Smith
Ripley	NZ	no	Spatial Statistics	NA
Ripley	NZ	no	Stochastic Simulation	NA
Tierney	US	no	LISP-STAT	NA
Tukey	US	yes	Exploratory Data Analysis	NA
Venables	Australia	yes	Modern Applied Statistics	Ripley

(Default value for `all.y` is `FALSE`)

Full Outer Join

```
m4 <- merge(authorN, books, all = TRUE)
kable(m4)
```

name	nationality	deceased	title	other.author
McNeil	NA	NA	Interactive Data Analysis	NA
R Core	NA	NA	An Introduction to R	Venables & Smith
Ripley	NZ	no	Spatial Statistics	NA
Ripley	NZ	no	Stochastic Simulation	NA
Tierney	US	no	LISP-STAT	NA
Tukey	US	yes	Exploratory Data Analysis	NA
Venables	Australia	yes	Modern Applied Statistics	Ripley
Winton	UK	no	NA	NA

(same as specifying `all.x=T` and `all.y=T`)

Inner join – another example

Suppose that we want to create a new customer data frame having an extra `income.normalised` variable, where the income of each customer is normalised by the median income of state of residence of the customer.

Step 1: Calculate the state median income using the `aggregate()` function:

```
median.income <- aggregate(custdata[, 'income'],
                           list(custdata$state.of.res), median)
kable(head(median.income))
```

Group.1	x
Alabama	22000
Alaska	33600
Arizona	22700
Arkansas	64000
California	31600
Colorado	25800

Inner join – another example (cont.)

- Note the default variable names after aggregation are `Group.1` and `x`.

Step 2: merge the two data frames:

```
custdata <- merge(custdata, median.income,
                  by.x="state.of.res", by.y="Group.1")
```

Step 3: create a new column of normalised income:

```
custdata$income.normalised <- with(custdata, income/x)
kable(custdata[1:6, c("custid","state.of.res","income","x","income.normalis
```

custid	state.of.res	income	x	income.normalised
999444	Alabama	24000	22000	1.0909091
572341	Alabama	16000	22000	0.7272727
799074	Alabama	56500	22000	2.5681818
971768	Alabama	8000	22000	0.3636364
896869	Alabama	22000	22000	1.0000000
863391	Alabama	17000	22000	0.7727273

Section 3

Relational Data with join(x,y) in dplyr

Dealing with Realational Data using dplyr

Comparing dplyr's join functions with the base merge function

dplyr	merge
inner_join(x, y)	merge(x, y)
left_join(x, y)	merge(x, y, all.x = TRUE)
right_join(x, y)	merge(x, y, all.y = TRUE)
full_join(x, y)	merge(x, y, all.x = TRUE, all.y = TRUE)

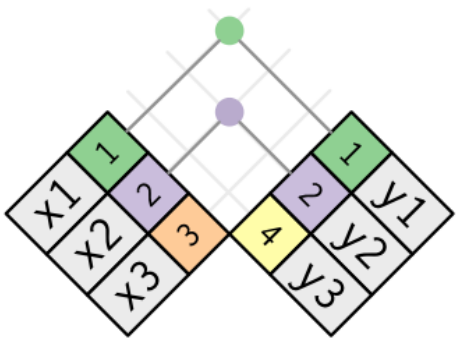
Comparing dplyr's join functions with the sql

dplyr	SQL
inner_join(x, y, by = "z")	SELECT * FROM x INNER JOIN y USING (z)
left_join(x, y, by = "z")	SELECT * FROM x LEFT OUTER JOIN y USING (z)
right_join(x, y, by = "z")	SELECT * FROM x RIGHT OUTER JOIN y USING (z)
full_join(x, y, by = "z")	SELECT * FROM x FULL OUTER JOIN y USING (z)

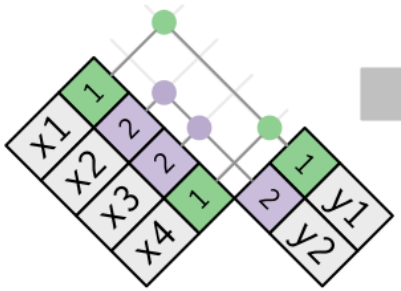
dplyr supports more joins

- **Mutating joins**, which add new variables to one data frame from the matching observations in another.
  - inner_join(x,y)
  - left_join(x, y), right_join(x, y), full_join(x, y)
- **Filtering joins**, which filter observations from one data frame based on whether or not they match an observation in the other table.
  - semi_join(x, y) keeps all observations in x that have a match in y.
  - anti_join(x, y) drops all observations in x that have a match in y.
- **Set operations**, which treat observations as if they were set elements.
  - intersect(x, y) returns only observations in both x and y.
  - union(x, y) returns unique observations in x and y.
  - setdiff(x, y) returns observations that are in x but not in y.

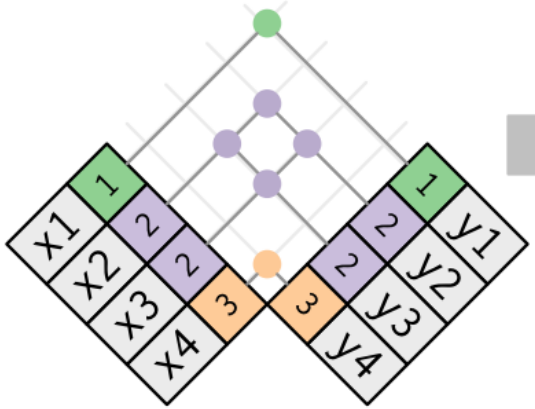
Mutating joins - inner



key	val_x	val_y
1	x1	y1
2	x2	y2

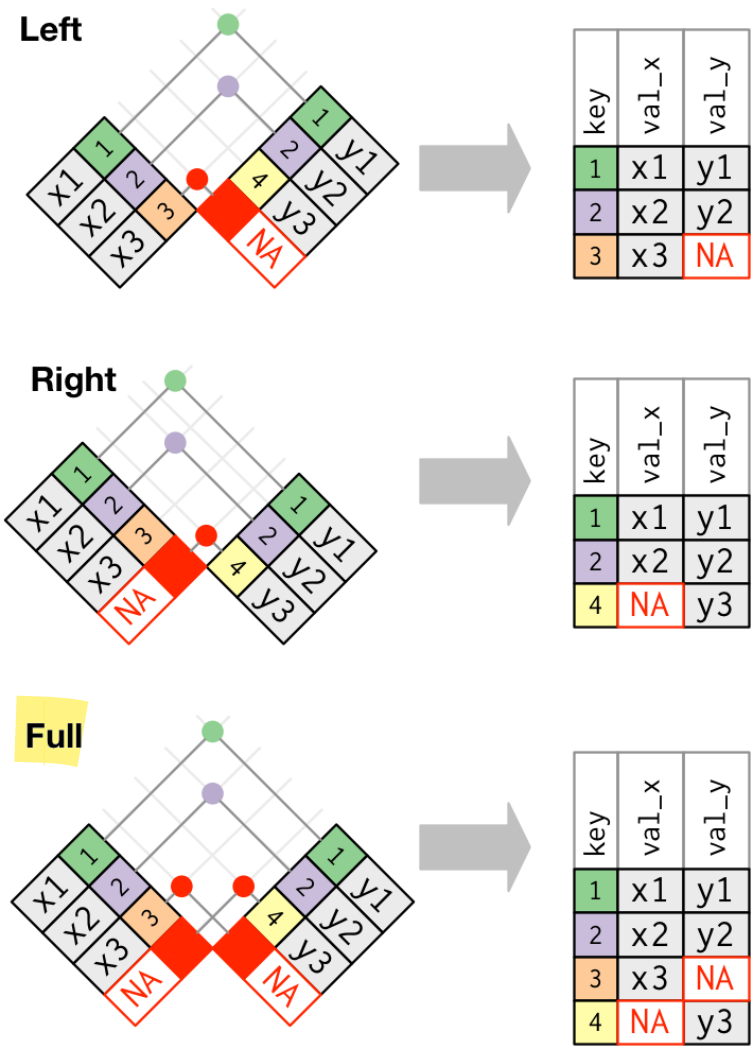


val_x	key	val_y
x1	1	y1
x2	2	y2
x3	2	y2
x4	1	y1



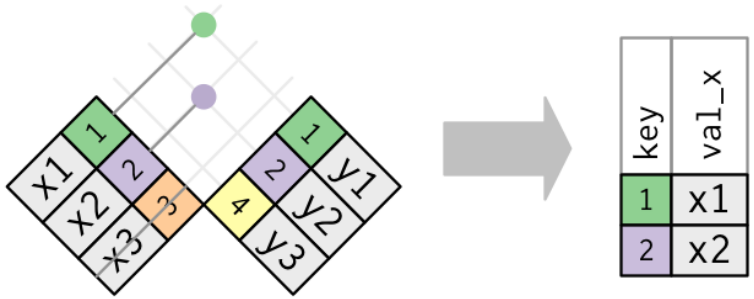
key	val_x	val_y
1	x1	y1
2	x2	y2
2	x2	y3
2	x3	y2
2	x3	y3
3	x4	y4

Mutating joins - outer



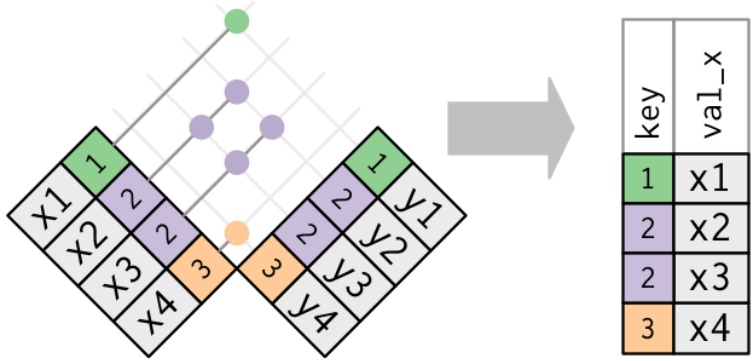
Filtering joins - semi_join(x,y)

semi_join(x, y) keeps all observations in x that have a match in y.



Only the existence of a match is important; it doesn't matter which observation is matched.

This means that filtering joins never duplicate rows like mutating joins do:



Filtering joins - semi (example)

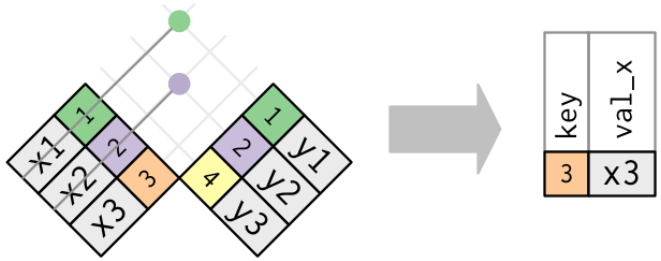
Filter the book data by keeping only books that have author information.

```
library(dplyr)
m5 <- semi_join(books, authorN)
kable(m5)
```

name	title	other.author
Tukey	Exploratory Data Analysis	NA
Venables	Modern Applied Statistics	Ripley
Tierney	LISP-STAT	NA
Ripley	Spatial Statistics	NA
Ripley	Stochastic Simulation	NA

Filtering joins - anti_join(x,y)

anti_join keeps all observations in x that don't have a match in y. Useful for diagnosing join mismatches.




```
m6 <- anti_join(books, authorN)
kable(m6)
```

name	title	other.author
McNeil	Interactive Data Analysis	NA
R Core	An Introduction to R	Venables & Smith

Set operations

Take the author and book data frame as an example,

```
x <- authorN$name
y <- books$name

x
## [1] "Tukey"    "Venables" "Ripley"   "Tierney"  "Winton"
```

```
y
## [1] "Tukey"    "Venables" "Tierney"  "Ripley"   "Ripley"
```

Set operations operates on a single variable

```
setdiff(x, y) # What are in x but not y?
## [1] "Winton"
```

```
union(x, y)   # Duplications counted once only
## [1] "Tukey"    "Venables" "Ripley"   "Tierney"  "Winton"
```

```
intersect(x, y)  # Common elements in x and y
## [1] "Tukey"    "Venables" "Ripley"   "Tierney"
```

Take Home Message

- Merging two data frames
- Merging relational data

References

- **Practical Data Science with R**, *Nina Zumel, John Mount*, Manning, 2nd Ed., 2020 (Chapter 5: Sections 5.1-5.4)
- **R for Data Science**, *Hadley Wickham, Garrett Grolemund*, https://r4ds.had.co.nz/ (Chapters 11,12,13,18)

Pipes (%>% or |>)

CITS4009 Exploratory Data Analysis

Dr. Mubashar Hassan

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

Semester 2, 2024

Piping

The pipe `%>%` comes from the `magrittr` package by Stefan Milton Bache, which comes with `tidyverse`.

You can also use `library(magrittr)` or `library(dplyr)` to get the pipe symbol `%>%` or `|>` defined in your R environment.

Why piping? – A motivating example

```
library(crayon)
little_bunny <- function(name) {
  return("Little bunny " %+% name)
}

hop <- function(data, through) {
  return(data %+% "\nWent hopping through the " %+% through)
}

scoop <- function(data, up) {
  return(data %+% "\nScooping up the " %+% up)
}

bop <- function(data, on) {
  return(data %+% "\nAnd bopping them on the " %+% on)
}
```

Options to produce the popular Children's peom

- Save each intermediate step as a new object.
- Overwrite the original object many times.

```
s <- little_bunny("Foo Foo")
s <- hop(s, "forest")
s <- scoop(s, "field mice")
s <- bop(s, "head")
cat(s)
## Little bunny Foo Foo
## Went hopping through the forest
## Scooping up the field mice
## And bopping them on the head
```

- Compose functions:

```
s <-  bop(scoop(hop(little_bunny("Foo Foo"), "forest"), "field mice"), "head")
cat(s)
## Little bunny Foo Foo
## Went hopping through the forest
## Scooping up the field mice
## And bopping them on the head
```

- Use the pipe

The pipe version

```
library(dplyr)

little_bunny("Foo Foo") |>
hop(through = "forest") |>
scoop(up = "field mice") |>
bop(on = "head") |>
cat()
```

```
## Little bunny Foo Foo
## Went hopping through the forest
## Scooping up the field mice
## And bopping them on the head
```

Using %>% to connect a series of operations

You can use %>% to connect any chain of operations (i.e., function calls) so long as the output of the preceding function fits with the input required by the next function in the chain. Most R functions accept the input in the first argument. By default, %>% will pass the output from the preceding function to the first argument of the next function.

Example 1. Chaining two `mutate()` function calls using %>% (for data cleaning in the previous lecture):

```
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)
  )
```

Using %>% to connect a series of operations (cont.)

Example 2. Perform an inner join of the `authors` and `books` data frames (see earlier slides), subset to select authors from NZ, and print the output data frame.

```
# kable() requires the knitr library
merge(authors, books, by.x="surname", by.y="name") %>%
    subset(nationality=="NZ") %>%
    kable()
```

surname	nationality	deceased	title	other.author
Ripley	NZ	no	Spatial Statistics	NA
Ripley	NZ	no	Stochastic Simulation	NA

Note: we don't need to specify the input data frame when calling `subset()` as the data comes from the output of `merge()`. The condition `nationality="NZ"` which specifies the row selection becomes the 1st argument for `subset()`. Similarly, no argument is needed for `kable()`.

Using %>% to connect a series of operations (cont.)

Example 3. Consider the following data frames:

```
library(tibble)
students <- tribble (
   ~name, ~degree, ~start.year, ~mode,
   "John", "MDS", 2020, "part-time",
   "Jack", "MIT", 2019, "full-time",
   "Rose", "BSc", 2020, "full-time",
   "Mary", "MDS", 2018, "part-time",
   "Paul", "BPhil", 2020, "full-time"
   )

degrees <- tribble (
   ~degree, ~duration,
   "BPhil", 4,
   "BSc", 3,
   "MDS", 2,
   "MIT", 2,
   "MPE", 2
)
```

Using %>% to connect a series of operations (cont.)

Example 3. cont.

```
units <- tribble (
   ~unit, ~degree,
   "CITS1401", "BSc",
   "CITS4009", "MDS",
   "CITS4009", "MIT",
   "CITS4401", "MIT",
   "CITS4402", "BPhil",
   "CITS5508", "MDS",
   "CITS5508", "MIT"
)
```

Work out the R code, which includes using %>%, for

1. generating a data frame containing the units that Jack needs to complete for the degree he's enrolled in. Your data frame only needs to have a single variable (column).
2. finding out the year that Mary is expected to graduate, assuming that part-time enrolments take twice the number of years compared to full-time enrolments.

References

- **R for Data Science**, *Hadley Wickham, Garrett Grolemund*, https://r4ds.had.co.nz/
  - Chapter 18 has a very good introduction on %>%
  - Section 27.2 also has a small example of %>% written as R code embedded in R markdown. In the example there, try to replace `geom_freqpoly` by `geom_histogram` and compare the two plots side-by-side (using `grid.arrange` from the `gridExtra` library).

Sample solutions for the two problems for example 3 are on the next 3 slides.

Sample solutions for example 3

Problem 1:

```
Jack.units <- subset(students, name=="Jack", select="degree") %>%
   merge(units) %>% subset(select="unit")
cat("Jack's list of units:")
```

```
## Jack's list of units:
```

```
kable(Jack.units)
```

unit
CITS4009
CITS4401
CITS5508

The following will work also:

```
Jack.units <- students %>% subset(name=="Jack", select="degree") %>%
   merge(units) %>% subset(select="unit")
```

Sample solutions for example 3 (cont.)

Another alternative solution is:

```
Jack.units <- merge(students, units) %>%
   subset(name=="Jack", select="unit")
cat("Jack's list of units:")
```

```
## Jack's list of units:
```

```
kable(Jack.units)
```

	unit
7	CITS4009
8	CITS4401
9	CITS5508

Although only one `subset()` call is required here, the `merge()` function has to merge two larger data frames and then discard most rows of the result.

Sample solutions for example 3 (cont.)

Problem 2:

```
df <- subset(students, name=="Mary") %>% inner_join(degrees, by="degree")
graduation.year <- df$start.year +
   df$duration * ifelse(df$mode == "part-time", 2, 1)
cat("Mary is expected to graduate in", graduation.year)
```

```
## Mary is expected to graduate in 2022
```

Your code does not need to be identical to the sample code above.

Sampling

CITS4009 Computational Data Analysis

Dr. Mubashar Hassan

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

Semester 2, 2024

Why Sampling?

With today's computer power, we can analyze very large datasets than before, but sampling is a necessary task for other reasons also.

- It is 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 can often obscure the patterns in a graph.
- To train a model, we often need to split the entire dataset into a test (or hold-out) set and a training set.
  - The training set is the data that you feed to your model-building algorithm (regression, decision trees, and so on).
  - The test set is the data that you feed into the resultant trained model, to verify that its predictions are accurate.

Random Samples

The `sample()` function allows us to take a random sample (with or without replacement) of size `n` from the elements in `x` (usually a vector).

```
rows <- sample(1:nrow(custdata), 3, replace=FALSE)
mysample <- custdata[rows, ]

# show the first 6 columns of mysample
mysample[, 1:6]
```

```
##     custid sex is.employed income marital.stat health.ins
## 625 874159   F        TRUE   7500      Married       TRUE
## 349 487124   M        TRUE   2000      Married      FALSE
## 421 578596   M        TRUE 143100      Married       TRUE
```

Sampling with or without replacement

- Sampling **with replacement** – one data point can be drawn multiple times, because the drawn data points are *replaced* or put back into the population again.
  - This means all the data points have the same probability of being sampled in each draw.
- Sampling **without replacement** – once the data point is drawn, it is no longer available for future selection.
  - This means the remaining data points have the same but a higher probability of being sampled in each later draw.
- When the population is large enough, sampling without replacement is not much different from sampling with replacement.

Test and training splits (reproducible sampling)

One way to manage random sampling is to add a *sample group* column.

- The *sample group* column contains numbers generated uniformly from zero to one, using the `runif` function.

```
# create a new column
custdata$gp <- runif(dim(custdata)[1])
```

- This makes the samples reproducible as compared to the built-in `sample()` function. That is, we use the `custdata$gp` column again and again whenever we want to extract a test set and a training set.

```
split.ratio <- 0.1
testSet <- subset(custdata, custdata$gp <= split.ratio)
trainingSet <- subset(custdata, custdata$gp > split.ratio)
cat("Test set size:", dim(testSet)[1])
## Test set size: 94
```

```
cat("Training set size:", dim(trainingSet)[1])
## Training set size: 906
```

Test and training splits (random sampling)

If we want to test how robust a model (e.g., a *decision trees classifier*) is for different subsets of observations in our dataset, then we want the test and training sets to be different in each trial. In such a case, we can modify the code above as follows:

```
s <- runif(dim(custdata)[1])
split.ratio <- 0.1
testSet <- subset(custdata, s <= split.ratio)
trainingSet <- subset(custdata, s > split.ratio)
cat("Test set size:", dim(testSet)[1])
## Test set size: 86
```

```
cat("Training set size:", dim(trainingSet)[1])
## Training set size: 914
```

That is, in each trial, we generate the random numbers from `runif()` again.

Random seed

Same as in other programming languages, random numbers in R are pseudorandom numbers generated by a *random number generator* (an algorithm). See the examples given in https://r-coder.com/set-seed-r/

We can set the *random seed* to initialize the random number generator.

- If we set the *seed* to a fixed number (any integer), e.g., `set.seed(5)`, then we get the same sequence of pseudorandom numbers generated each time.
- If we want a different sequence of pseudorandom numbers generated each time, the best solution is set the seed to the system time, e.g., `set.seed(Sys.time())` or simply `set.seed(NULL)`. Type `?set.seed` in R to see the details.

Take home messages

- Random Sampling
- Reproducible Sampling

References

Below are the sections for further reading for all the topics in this week's lecture:

- **Practical Data Science with R**, *Nina Zumel, John Mount*, Manning, 2nd Ed., 2020 (Chapter 4 - Sections 4.3.1-4.3.2; Chapter 5: Sections 5.1-5.4)
- **R for Data Science**, *Hadley Wickham, Garrett Grolemund*, https://r4ds.had.co.nz/ (Chapters 11,12,13,18; Section 27.2 (for the %>% pipe and R markdown))