Notes for the BAN400 Exam

Table of contents

1 Functions

1.1 Basic functions

Function	Package	Description
mean()	base	Calculates the mean of a vector of numbers
median()	base	Calculates the median of a vector of numbers
sd()	base	Calculates the standard deviation of a vector of numbers
var()	base	Calculates the variance of a vector of numbers
sum()	base	Calculates the sum of a vector of numbers
c()	base	Creates vector
length()	base	The number of elements in a vector or list
ncol()	base	Number of columns of data frame or matrix
nrow()	base	Number of rows of data frame or matrix
min()	base	The smallest value in a set
max()	base	The largest value in a set
seq()	base	Create an individual sequence
rep()	base	Repeat a vector or elements of a vector
vector()	base	Creates an empty vector

1.2 Math

Function	Package	Description
sqrt()	base	Calculates square root of number or vector of numbers
abs()	base	Calculates absolute value of number or vector of numbers
sign()	base	Returns the sign of x
round()	base	Rounds x to n decimal places
ceiling()	base	Rounds up to the nearest integer
floor()	base	Rounds down to the nearest integer
cumsum()	base	Cumulative sum
cor()	base	Correlation

1.3 Reading data

Function	Package	Description
read_csv()	readr	Read csv-file (comma separated values)
read_csv2()	readr	Uses semicolons as separators and keeps commas
read_delim()	readr	Read file with columns separated by any delimiter
read_excel()	readxl	Read data from excel files
read_fwf()	readr	Reads fixed width data
read_tsv()	readr	Reads tab separated values
readLines	base	Read some or all text lines from a connection
list.files()	base	lists files in a directory
list_rbind()	purrr	combines data from a list into a single data frame

1.4 Data wrangling

Function	Package	Description
head()	base	Returns the first few rows of a data frame or vector
tail()	base	Returns the first few rows of a data frame or vector
filter()	dplyr	Returns elements that satisfy conditions
select()	dplyr	Choose specific columns from a data frame
arrange()	dplyr	Sorts rows of a data frame by specified columns
sort()	base	Sorts a vector in ascending or descending order
mutate()	dplyr	Adds or modifies columns in a data frame
transmutate()	dplyr	Creates a new data frame containing only the specified computations
summarise()	dplyr	Summary statistics for columns in a data frame (typically used with grouped data)
group_by()	dplyr	Group data by one or more variables
ungroup()	dplyr	Ungroup data such that subsequent operations to apply to the entire dataset
left_join()	dplyr	Returns all values from the first data frame with all columns and values from the second data frame where there is a match
inner_join()	dplyr	Joins two data frames by keeping only records that match in both data sets
right_join()	dplyr	Returns all values from the second data frame with matching columns and values from the first data frame where there is a match
full_join()	dplyr	Returns all values and columns from both data frames and filling in NA where there is no match
semi_join()	dplyr	Filters the first data frame keeping only rows with matching keys in the second data frame
anti_join()	dplyr	Filters the first data frame to keep only rows with no match in the second data frame

1.5 Machine learning

Function	Package	Description
logistic_reg()	tidymodels	Specifies a logistic regression model
set_engine()	tidymodels	Specifies the computational engine for a model
set_mode()	tidymodels	Sets the mode (e.g. classification or regression) for a model
fit()	tidymodels	Fits the model to data
nearest_neighbor()	tidymodels	Specifies a k-nearest neighbors model
tune()	tidymodels	Marks a parameter for tuning in a model
finalize_workflow()	tidymodels	Finalizes the workflow with specific parameters
workflow()	tidymodels	Creates a workflow object
add_model()	tidymodels	Adds a model to a workflow
add_recipe()	tidymodels	Adds a recipe to a workflow
tune_grid()	tidymodels	Tunes hyperparameters across a grid of values
select_best()	tidymodels	Selects the best tuning parameter combination based on a metric
predict()	stats	predicts outcome variable according to a specified model-e.g Predict.lm- predict.glm
initial_split()	rsample	splits dataset in test and training and test data
vfold_cv()	rsample	randomly splits data into different folds (for cross validation)
recipe()	recipes	creates a recipe for processing data
grid_space_filling()	dials	for making a search grid
roc_auc()	yardstick	calculated the area under the AUC curve

1.6 Many models

Function	Package	Description
add_predictions()	modelr	Adds model predictions to a data frame
add_residuals()	modelr	Adds residuals from a model to a data frame
group_by()	dplyr	Groups data by one or more variables
ungroup()	dplyr	Removes grouping structure from a data frame
nest()	tidyr	Creates a nested data frame by collapsing rows into list-columns
unnest()	tidyr	Expands list-columns back into regular columns
select()	dplyr	Selects specific columns from a data frame
pull()	dplyr	Extracts a single column as a vector
pluck()	purrr	Extracts an element from a list or vector by index or name
map()	purrr	Applies a function to each element of a list or vector
map2()	purrr	Applies a function to pairs of elements from two lists
glance()	broom	Generates a summary of model diagnostics in a tidy format

1.7 Parsing

Function	Package	Description
class()	base	Returns the class of an object
typeof()	base	Determines the type or storage mode of any object
mode()	base	Indicates the mode of storage
as.numeric()	base	Converts data to numeric format
as.character()	base	Converts data to character format
as.logical()	base	Converts data to a logical format (TRUE or FALSE)
as.factor()	base	Converts data to factor format
gsub()	base	Find and replace
reshape()	base	Reshape datasets between wide and long formats
pivot_longer()	tidyr	Convert wide data to long format
pivot_wider()	tidyr	Convert long data to wide format.
na.omit()	base	Remove missing values
is.na()	base	Check for missing values
ymd()	lubridate	Parse dates
hms()	lubridate	Parse times
parse_datetime()	readr	Parse datetimes
separate()	tidyr	Split one column into multiple columns

1.8 Selecting

Function	Package	Description
any_of()	dplyr	Selects columns that match any of the given names in a vector
all_of()	dplyr	Selects columns that match all the given names in a vector
starts_with()	dplyr	Selects columns whose names start with a specified prefix
ends_with()	dplyr	Selects columns whose names end with a specified suffix
contains()	dplyr	Selects columns whose names contain a specified string
matches()	dplyr	Selects columns whose names match a specified regular expression
select()	dplyr	Selects specified columns from a data frame
slice_min()	dplyr	Selects rows with the smallest values of a variable
slice_max()	dplyr	Selects rows with the largest values of a variable
everything()	tidyselect	Selects all variables
where()	tidyselect	Selects the variables for which the inserted function returns TRUE
across()	tidyselect	Apply the same transformation to multiple columns

1.9 Parallel computing

Function	Package	Description
tic()	tictoc	Initialize time taking of a function
toc()	tictoc	Stop time taking of a function
detectCores()	doParallel	Detect cores in CPU
makeCluster()	doParallel	Initialize cores
registerDoParallel	doParallel	Register the cluster
stopCluster()	doParallel	Close off clusters
foreach() %dopar%	doParallel	Designate tasks for parallelization
plan()	future	used it to define how many Cores to use (together with future_map() from furrr)
future_map()	furrr	works as normal map function but you can do things parralel-also many alternatives (e.g. future_map2_dbl()

1.10 Making maps

Function	Package	Description
distm()	geosphere	Calculate distance between 2 points
st_as_sf()	sf	Converting data frame to geometric object
geom_sf()	sf & ggplot2	Allows mapping of geometric objects
st_crop()	sf	Cropping maps (x and y chords)
st_layers()	sf	Check layers of files
st_read()	sf	Read map data
st_transform()	sf	Transform map projection (we used crs = 4326)
st_cast()	sf	cast geometry to another type
ne_countries()	rnaturalearth	getting data for the map
auto_merge()	countries	easy merging for country data (especially different spelling of same name)
st_intersection()	sf	set operations with geometry collections
coord_sf()	ggplot2	just display the area we want

2 Topics

2.1 Empty vectors (and tibbles) for further use

To set up an empty vector, one can for instance use the following chunk of code. While doing so, the seq-function may be useful. `NA_integer_` serves as some kind of "placeholder" for figures/outcomes of formulas calculated later on.

```
vector <-
  tibble(
    number = seq(
      from = 1,
      to = 10,
      by = 1
    ),
    placeholder =
      NA_integer_
  )
```

2.2 Filtering

The symbol | works as an "or" operator, meaning it will return items that satisfy either one or both of the conditions before and after the operator.

```
filter(flights, dest == "IAH" | dest == "HOU")
```

The `&` is an "and" operator, meaning both conditions need to hold. This is useful in conjunction with the `|`-operator, since otherwise you can simply just add filters after each other like this:

```
filter(!is.na(flight2), !is.na(flight1))
```

Another operator `%in%` returns `TRUE` if an item exists inside a vector:

```
flights %>%
  filter(dest %in% c("IAH", "HOU")
```

To create the opposite logic, i.e. "not in c()", you can use `!`:

```
flights %>%
  filter(!(dest %in% c("IAH", "HOU")))
```

2.3 Selecting

Some basic selection functions from `dplyr` are `starts_with`, `ends_with` and `contains`.

```
select(flights, starts_with("dep_"), starts_with("arr_"))
```

2.3.1 Regex (regular expressions)

The function `matches` uses *regular expressions*. These can be used to match precise patterns.

```
select(flights, matches("^(dep|arr)_(time|delay)$"))
```

Some common Regex meta characters are:

- `.` Matches any character except newline.
- `^` Matches the beginning of a string.
- `$` Matches the end of a string.
- `[]` Matches any one of the characters inside the brackets.
- `|` Logical OR.
- `*` Matches 0 or more repetitions of the preceding character.
- `+` Matches 1 or more repetitions.
- `?` Matches 0 or 1 repetition (optional match).
- `{n,m}` Matches between n and m repetitions.

2.4 If-Else and case_when

2.4.1 Basic syntax of an if-else statement

In the following, you can see the basic syntax of an if-else statement

```
if (condition) {
# What should happen in case condition is TRUE?
} else {
  # What should happen in any other case?
}
```

2.4.2 Applying an if-else statement within a data frame

If we want to apply an if-else statement to a specific column of a data frame, there are several ways to do so. Two possible ways are either to use the aforementioned syntax in combination with a for-loop or make use of the case_when function. The syntax is as following:

```
case_when(
  column1 == "Norway" ~ "Norway",
  !column1 == "Norway" ~ "Rest of the world"
)
```

It depends on the given case what way is more convenient to use.

2.5 Loops and iterations

2.5.1 Standard for-loop

```
for(i in 1:n) {
  ... do something with i...
}
```

Note that we can iterate over any type of vector, not just numbers, and we can give the iteration variable any name we want. In the example above it is `i`.

2.5.2 While loop

Repeat until a certain condition is met. For example

```
i <- 1
while(i < 10) {
  print(i)
  i <- i + 1
}
```

2.6 Machine Learning

2.6.1 Basic structure

We dealt with Machine learning. Here are the basic steps you have to do. Steps are for the process of cross validation.

1. Get train & testdata
2. Make folds
3. Define model and recipe
4. Set up workflow
5. Search grid
6. Get cross validation AUC
7. Determine "optimal" tune parameters and put them in the workflow
8. Fit the model
9. Predict test data
10. Calculate overall AUC

2.7 Math

The Integer Division Operator %/% performs integer division. It divides two numbers and returns the whole number part of the quotient, discarding any remainder. Example:

```
10 %/% 3  # Returns 3 (quotient without remainder)
```

The Modulo Operator %% operator returns the remainder from the division of two numbers. Example:

```
10 %% 3  # Returns 1 (remainder)
```

2.8 Creating own functions

In R, we have the opportunity to create own functions. In order to do so, we have to stick to the following syntax.

2.8.1 Basic syntax

```
function_name <-
  function(input1, input2, input3){
    # In the following, function is defined/written
    example = (input1 + input2) * input3
    return(example) # Using return, we can exactly define function's return
  }
```

2.8.2 Anonymous functions

We can also create functions for one use only.

```
{\(x) content of the function}
#x is the parameter variable here
#an example for an anonymous function:
{\(x) x^3}
#don't forget the brackets () when you use it in a pipe
```

2.9 Parallel Computing

Here are some general steps for parallel computing:

2.9.1 DoParallel:

1. Determine number of cores
2. Instantiate cores
3. Register the cluster
4. Start timer
5. Use foreach and %dopar%, do whatever you would have done in the regular for loop
6. Close the clusters
7. Stop timer

2.9.2 Furrr

1. Use a map function to write the for loop
2. Specify how many workers to use
3. Start timer
4. Use the sample loop as with the map-function but replace the map-function with the equivalent future_map-function (e.g. map2_dbl() -> future_map2_dbl())
5. Stop the timer

2.9.3 Syntax for foreach:

```
#a normal for loop:
for (i in 1:nrow(df)) {
  some calculations
}

#Foreach loop:
foreach (i = 1:nrow(df)) %dopar% {
  some calculations
}
```

```
#Good idea to specify how to combine results and which packages should be available to the foreach loop
```

2.10 Plotting

We use `ggplot2` as the standard package for plotting, and the main function is `ggplot`. We supply a data frame to the first argument and an aesthetic mapping to the second argument. We add *layers* of plotting components using the plus sign. A simple example:

```
ggplot(df, aes(x = x_variable, y = y_variable, colour = grouping_variable)) +
  geom_point()
```

Many types of layers may contain other data sets via the `data` argument and/or updated aesthetic mappings via the `mapping` argument. Data and mappings are typically inherited from the layer above if not specified in a new layer. There are many types of functions for making further adjustments to labels, titles, axes and other properties. A more complete example may look like this:

```
ggplot(df, aes(x = x_variable, y = y_variable)) +
  geom_point() +
  geom_point(mapping = aes(x = new_x_variable, y = new_y_variable, colour = grouping_var),
             data = new_df) +
  xlab("X label") +
  ylab("Y label") +
  ggtitle("Title of plot") +
  theme_minimal()
```

2.10.1 Grouping

You can visually group items in plots together using the `color` argument.

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy, color=drv))
```

Grouping can also be used in calculations. For example, if you want to create several regression lines based on classification, you can group with `group` argument:

```
geom_smooth(aes(group = drv)) # Creates separate regression lines.
```

Sometimes need to group by several variables at the same time. You can use `unite()` to achieve this.

```
whodata %>%
  unite(country_sex, country, sex, remove = FALSE) %>%
  ggplot()+
  geom_line(aes(x=year,y=cases, group=country_sex, color=sex))
```

2.10.2 Stacking plots

You can stack several plots on top of each other. If you stack plots of the same type , it may be useful to distinguish them using color. For example:

```
ggplot(diamonds) +
  geom_freqpoly(aes(x = x, color = "x"), binwidth = 0.1) +
  geom_freqpoly(aes(x = y, color = "y"), binwidth = 0.1)
```

2.10.3 One categorical and one continuous variable

To explore the relationship between a continuous variable and one categorical variable, some plotting options are:

- `geom_violin`
- `geom_freqpoly` with `colour`-argument
- `geom_histogram` with`facet_wrap(vars())`
- If the data set is large, you may want to use `geom_lv` from the library `lvplot`.

Example:

```
ggplot(diamonds, aes(x=price,colour=cut))+
  geom_freqpoly(bins=50)
```

2.10.4 Two categorical variables

To explore the relationship between two categorical variables, `geom_tile` is often useful.

```
diamonds %>% count(cut,color) %>% ggplot(aes(x=cut,y=color))+
  geom_tile(aes(fill=n))
```

If you want to normalize proportions within a given categorical variable, here's an example of that:

```
diamonds %>%
  count(color, cut) %>%
  group_by(color) %>%
  mutate(
    prop=n/sum(n)
  ) %>%
  ggplot(aes(x=color,y=cut))+
  geom_tile(aes(fill=prop))
```

2.10.5 Two continuous variables

There are many options to explore the relationship between two categorical variables, such as:

- `geom_point`
- `geom_smooth`
- `geom_boxplot` where a continuous variable is divided into bins.

```
ggplot(diamonds, aes(x = cut_width(price, 2000, boundary = 0), y = carat)) +
  geom_boxplot(varwidth = TRUE)
```

Tip: choosing binned plots versus such as scatterplots depends on the nature of data analysis. For example, binned plots can "hide" outliers that are unusual combinations of x and y, but where x or y individually are not extreme values.

2.10.6 Discrete variables

A small tip about plotting discrete variables: If there is overplotting (e.g. multiple "dots" overlapping in dotplots; which happens often with discrete variables), then use jitter functions.

```
ggplot(data, aes(x = category, y = value)) +
  geom_jitter(width = 0.2, height = 0.1)
```

2.10.7 Statistics

Most geoms come in pairs with complementary statistics arguments that are almost always used in concert. These functions can be used to retrieve the data that is used to generate the plot. For example, for these geoms:

```
geom_smooth()
geom_dotplot()
geom_point()
geom_bar()
```

We have these `stat` functions respectively:

```
stat_smooth()
stat_bindot()
stat_qq()
stat_count()
```

The corresponding stat functions can be found by reading the documentation with for example `?geom_smooth`.

2.11 Reading data

Sometimes data contains the delimiter. In that case, use quote argument to escape:

```
read_csv("x,y\n1,'a,b'", quote="'")
```

2.11.1 Reading multiple files

Instead of reading in every single file on its own, you can list all the files in a directory, read each of them into a list and combine them into a data frame:

```
paths <- list.files(path = "your_files", pattern = "*.csv")

list_of_data <- map(paths, read_csv)

single_df <- list_rbind(list_of_data)
```

2.11.2 Reading data from the files names

When information is given in the name of the file itself, we can extract it using the set_names function:

```
paths <- list.files(path = "your_files", pattern = "*.csv")

paths %>%
  set_names(basename) %>%
  map(read_csv)
```

2.11.3 Parsing data

There are various parsing functions from `lubdridate` library, such as `ymd()`. These functions can be used to convert objects into time and date formats.

2.11.4 Dates and time

Examples:

```
d1 <- "January 1, 2010"
d2 <- "2015-Mar-07"
d3 <- "06-Jun-2017"
d4 <- c("August 19 (2015)", "July 1 (2015)")
d5 <- "12/30/14" # Dec 30, 2014
t1 <- "1705"
t2 <- "11:15:10.12 PM"
```

Can be parsed respectively:

```
mdy(d1)     #> [1] "2010-01-01"
ymd(d2)     #> [1] "2015-03-07"
dmy(d3)     #> [1] "2017-06-06"
mdy(gsub("\\(", "", gsub("\\)", "", d4)))  #> [1] "2015-08-19" "2015-07-01"
mdy(d5)     #> [1] "2014-12-30"
hm(t1)      #> [1] "17:05:00"
hms(t2)     #> [1] "23:15:10.12"
```

2.11.5 Pivoting

The library `tidyr` provides various ways to pivot data. An example of a pivot is from

```
preg <- tribble(
  ~pregnant, ~male, ~female,
  "yes", NA, 10,
  "no", 20, 12
)
```

To a longer data format:

```
preg_tidy2 <- preg %>%
  pivot_longer(c(male, female), names_to = "sex", values_to = "count", values_drop_na = TRUE)
```

2.11.6 Splitting columns

You can split a column into multiple columns using `separate()` and `extract()` from `tidyr`:

```
tibble(x = c("a,b,c", "d,e,f,g", "h,i,j")) %>%
  separate(x, c("one", "two", "three"))
```

In the above case, the second item has four “values” `"d,e,f,g"`. In this case, the `g` is dropped. There are various arguments in the functions to determine how to deal with extra or missing items like this.

2.11.7 Basic find and replace

You can perform a basic find-and-replace operation on a vector like this:

```
y[y == "find"] <- "replace"
```

2.12 Regression

2.12.1 linear regression

```
linear_model <- lm(score ~ STR, data = df)
```

2.12.2 nonlinear regression with polynomials

```
#degree determines degree of polynom (degree = 2 for quadratic model)
cubic_model <- lm(score ~ poly(income, degree = 3, raw = TRUE), data = df)


#Alternative:
#use the I and define all the terms you want in the regression
quadratic_model <- lm(score ~ income + I(income^2), data = df)
```

2.12.3 Interaction terms

```
model_interaction <- lm(score ~ size + HiEL + size * HiEL, data = df)
```

2.12.4 Dealing with regression Output

```
model: some regression model

#simplest way to get regression results
summary(model)
#regression output with stargazer package
stargazer(model, type = "text")
#using the jtools package
summ(model)
#another function of the jtools package
plot_summs(model)
#adding the distributions
plot_summs(model, plot.distributions = TRUE)
#turn an object into a tidy tibble
tidy(model)
```