Unit 03: Remus Luping and the msleep dataset

Applied AI with R

Ferdinand Ferber and Wolfgang Trutschnig

Paris Lodron Universität Salzburg

3/4/24

Table of contents I

Remus Lupin and the msleep dataset

AI generated image for the prompt "Remus Lupin sleeping in front of a computer in his office at Hogwards with a full moon shining through the window."

Remus Lupin and the msleep dataset

- Remus Lupin messed up his sleep-cycle after the birth of his son. As half-werewolf, half-human, how much sleep does he need every night in order to stay functional?
- To answer this question, Lupin analyses the `msleep` dataset, containing information like average sleep time, REM sleep time, brain weight, etc. for a wide range of animals.
- Will it help him hitting the right balance between his beauty sleep and the fight against the Dark Lord?

Section 1

More on programming

Default arguments

- We can specify default arguments to a function. The caller can overwrite those arguments.

```
myfun <- function(x, y = 1) {

    c(x, y)
}

myfun(3)
```

```
[1] 3 1
```

```
myfun(4, 5)
```

```
[1] 4 5
```

Dot-dot-dot

- Special syntax ... (pronounced dot-dot-dot) to capture any number of additional arguments and to redirect them.

```
myfun <- function(type, vec, ...) {
    if (type == "mean") {
        mean(vec, ...)
    } else {
        sum(vec, ...)
    }
}
myfun("mean", c(1, 2, 3, NA), na.rm = T)
```

```
[1] 2
```

```
myfun("sum", c(1, 2, 3, NA))
```

```
[1] NA
```

Dot-dot-dot

One can also use `list(...)` to capture the additional arguments as a named list.

```
myfun <- function(type, ...) {
    vec <- list(...) |> as.numeric()
    if (type == "mean") {
        mean(vec)
    } else {
        sum(vec)
    }
}
myfun("mean", 1, 2, 3)
```

```
[1] 2
```

Closures

- *Closures* are one of the most important concepts in functional programming: a function returns another function that has *free variables* (variables not defined locally).

```
plus <- function(a) {
    inner_fun <- function(b) {
        a + b   # inner_fun closes over `a`
    }
    return(inner_fun)
}

plus_two <- plus(2)
plus_three <- plus(3)
c(plus_two(10), plus_three(20))
```

```
[1] 12 23
```

Exercise

- Create a function `pick()` that takes as an argument an index `i` and returns a function that maps a vector `x` to `x[[i]]`.
- So

```
msleep |> map(pick(5))
```

- should be equivalent to

```
msleep |> map(function(x) x[[5]])
```

Section 2

More on lists

map2[1]

The `map2` verb traverses two lists at the same time, applying a function for every pair of elements.

[1] Image taken from the *purrr cheat sheet*, Posit Software, PBC

map2

- Sometimes we want to traverse two lists at the same time and apply functions to both of them (classical `do.call(fun,lst)` only works for one function).
- This is what `map2(lst1, lst2, fun)` allows us to do:

```
by_cyl <- mtcars |> split(mtcars$cyl)
mods <- by_cyl |> map(\(df) lm(mpg ~ wt, data = df))
Pred <- map2(mods, by_cyl, predict)
```

- The chunks above first splits the data by cyl, fits a linear model to each group, and then applies the model to the data.

map2

- Here's a (more or less) base R version doing the same.
- Easier to understand but more tedious to code:

```
by_cyl <- mtcars |> split(mtcars$cyl)
models <- vector("list",length=length(by_cyl))
predictions <- models

for(i in 1:length(by_cyl)){
  models[[i]] <- lm(mpg ~ wt, data = by_cyl[[i]])
}
for(i in 1:length(by_cyl)){
  predictions[[i]] <- predict(models[[i]],
                              newdata = by_cyl[[i]])
}
```

map2

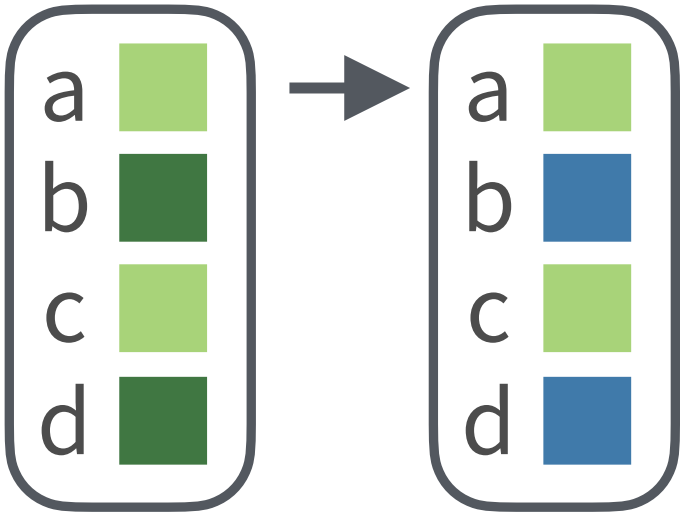
- A second example illustrating `map2(lst1, lst2, fun)`

```
myargs <- list(c(1,2,3), c(9,8,7))
myops <- list("sum", "mean")

myargs |>
    map(\(vec) as.list(vec)) |>
    map2_int(myops, \(arg, op) do.call(op, arg))
```

```
[1] 6 9
```

map_if[2]



- The `map_if` verb applies a transformation only to elements that satisfy a given predicate. All other elements remain untouched.

[2] Image taken from the *purrr cheat sheet*, Posit Software, PBC

map_if

- It is always possible to use `if...else` in the function that gets mapped over a container. But for simple cases there is a special case `map_if(cond, fn)`.
- For all elements not satisfying the condition, the identity transformation is applied instead.

```
# Apply the function `as.factor`
# only for elements satisfying
# the condition `is.character`
msleep |> map_if(is.character,
                 as.factor) |>
  map_chr(class)
```

	Class
name	factor
genus	factor
vore	factor
order	factor
conservation	factor
sleep_total	numeric
sleep_rem	numeric
sleep_cycle	numeric
awake	numeric
brainwt	numeric
bodywt	numeric

map_at

- The function `map_at(cond, fn)` is similar, but tests on the indices/names and not on the elements.

```
msleep |>
    map_at(\(col) col |>
             startsWith("sleep"),
           as.integer) |>
    map_chr(class)
```

	Class
name	character
genus	character
vore	character
order	character
conservation	character
sleep_total	integer
sleep_rem	integer
sleep_cycle	integer
awake	numeric
brainwt	numeric
bodywt	numeric

keep_at

- We already know `keep()` and `discard()` for filtering lists.
- The analogous function `keep_at(lst, pred)` keeps all elements of `lst` whose name satisfies `pred`. And `discard_at(lst, pred)` discards elements.

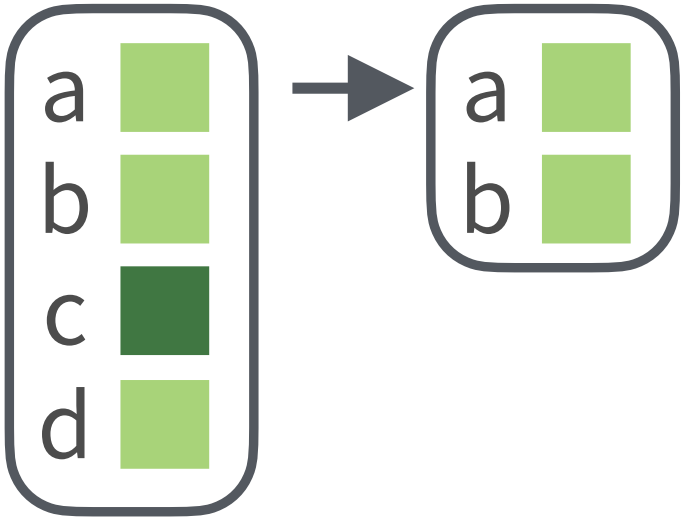
```
list(cat = 1, dog = 2, elephant = 3) |>
    keep_at(\(name) nchar(name) <= 3)
```

```
$cat
[1] 1

$dog
[1] 2
```

```
L <- list(cat =1,dog =2,elephant =3) #old school version
L1 <- L[nchar(names(L))<=3]
```

head_while and tail_while[3]



- The `head_while` verb traverses a list from the beginning and returns elements as long as they are satisfying a given predicate.
- After the first non-conforming element the process ends.

[3] Image taken from the *purrr cheat sheet*, Posit Software, PBC

head_while and tail_while

- The function `head_while(lst, pred)` returns elements starting from the beginning of `lst` until one element didn't pass `pred`.
- The function `tail_while(lst, pred)` does the same, but starts from the end.

```
# Throw a dice 50 times. What is the longest streak
# (from the beginning) of having only 3's or more?
x <- sample(1:6, size = 50, replace = T)
x[1:10]
```

```
[1] 3 2 4 1 4 6 6 6 3 2
```

```
x |> head_while(\(x) x >= 3)
```

```
[1] 3
```

Predicates on the whole list

- Instead of testing single elements for filtering, we can also test the whole list:

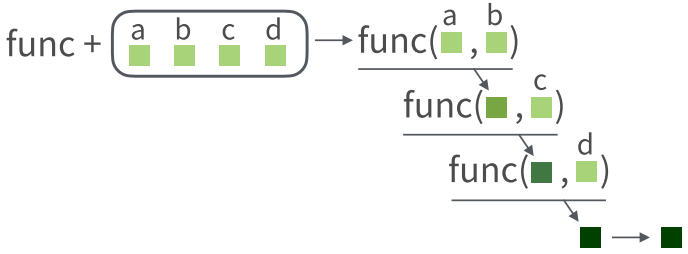
```
is_even <- function(x) x %% 2 == 0

3:10 |> every(is_even)
#> [1] FALSE

3:10 |> some(is_even)
#> [1] TRUE

3:10 |> none(is_even)
#> [1] FALSE
```

Reduce[4]



- The `reduce` verb traverses a list and recursively applies a function on the current element and the result of the last iteration.

[4] Image taken from the *purrr cheat sheet*, Posit Software, PBC

Reduce (with init)

- With `reduce(lst, fn, .init, .dir)` one can recursively apply `fn` to each element of `lst` and the previous result.
- At the first iteration the previous result doesn't exist and `.init` is used instead.
- The container is traversed in direction `dir` (default: forward).

```
# A complicated way to write sum(1:3)
1:3 |> reduce(\(acc, nxt) acc + nxt, .init = 0)
```

```
[1] 6
```

Reduce (without init)

- When the `.init` argument is not provided, the recursion starts with `fn(x[[1]], x[[2]])` instead of `fn(.init, x[[1]])`.
- The `done()` function can be used to stop the recursion.

```
limited_paste <- function(acc, nxt) {
    if (nchar(acc) > 4) {
        done(acc)
    } else {
        paste(acc, nxt, sep = ".")
    }
}
letters |> reduce(limited_paste)
```

```
[1] "a.b.c"
```

Exercises

- Ex1: Implement a function that uses `reduce` to calculate the factorial of a natural number.
- Ex2: Use `reduce` to check if every element of a logical vector is true.
- Ex3: Implement a function called `compose` that has a list of functions as input and returns their composition, e.g.

```
f1 <- function(x) {x + 1}
f2 <- function(x) {2 * x}
f3 <- function(x) {2 * (x + 1)}

f4 <- compose(list(f1, f2))
# Then f3 == f4
```

Section 3

More on ggplot

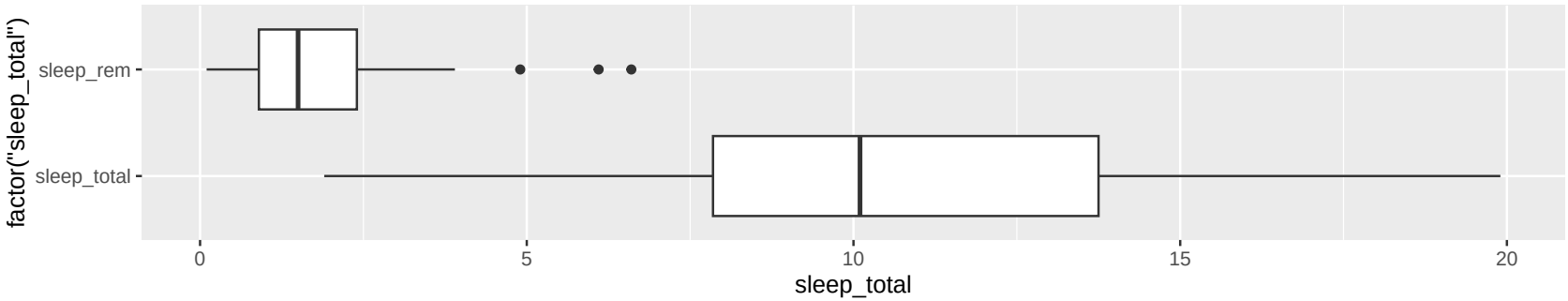
Long vs. wide dataframes

- Recall that in {ggplot2} every row of the input dataframe is mapped to one geometrical object.
- The object's visual properties are determined by the columns of the input dataframe, as specified by the aethetical mapping.

Long vs. wide dataframes

- When a row contains more than one observation, this doesn't play well with {ggplot2}.

```
msleep |> ggplot() +
  geom_boxplot(aes(x = sleep_total,
                   y = factor("sleep_total"))) +
  geom_boxplot(aes(x = sleep_rem,
                   y = factor("sleep_rem")))
```



- If we have a lot of observations, this gets very annoying. Also notice, that the axis labels are not correct.

pivot_longer[5]

country	1999	2000
A	0.7K	2K
B	37K	80K
C	212K	213K

→

country	year	cases
A	1999	0.7K
B	1999	37K
C	1999	212K
A	2000	2K
B	2000	80K
C	2000	213K

- `{ggplot2}` builds upon long (as opposed to wide) format.
- The `pivot_longer` verb collapses several columns into two columns, thus lengthening the dataframe.
- Column names go into the first column and values into the second.

[5] Image taken from the *tidyr cheat sheet*, Posit Software, PBC

pivot_wider[6]

country	year	type	count
A	1999	cases	0.7K
A	1999	pop	19M
A	2000	cases	2K
A	2000	pop	20M
B	1999	cases	37K
B	1999	pop	172M
B	2000	cases	80K
B	2000	pop	174M
C	1999	cases	212K
C	1999	pop	1T
C	2000	cases	213K
C	2000	pop	1T

country	year	cases	pop
A	1999	0.7K	19M
A	2000	2K	20M
B	1999	37K	172M
B	2000	80K	174M
C	1999	212K	1T
C	2000	213K	1T

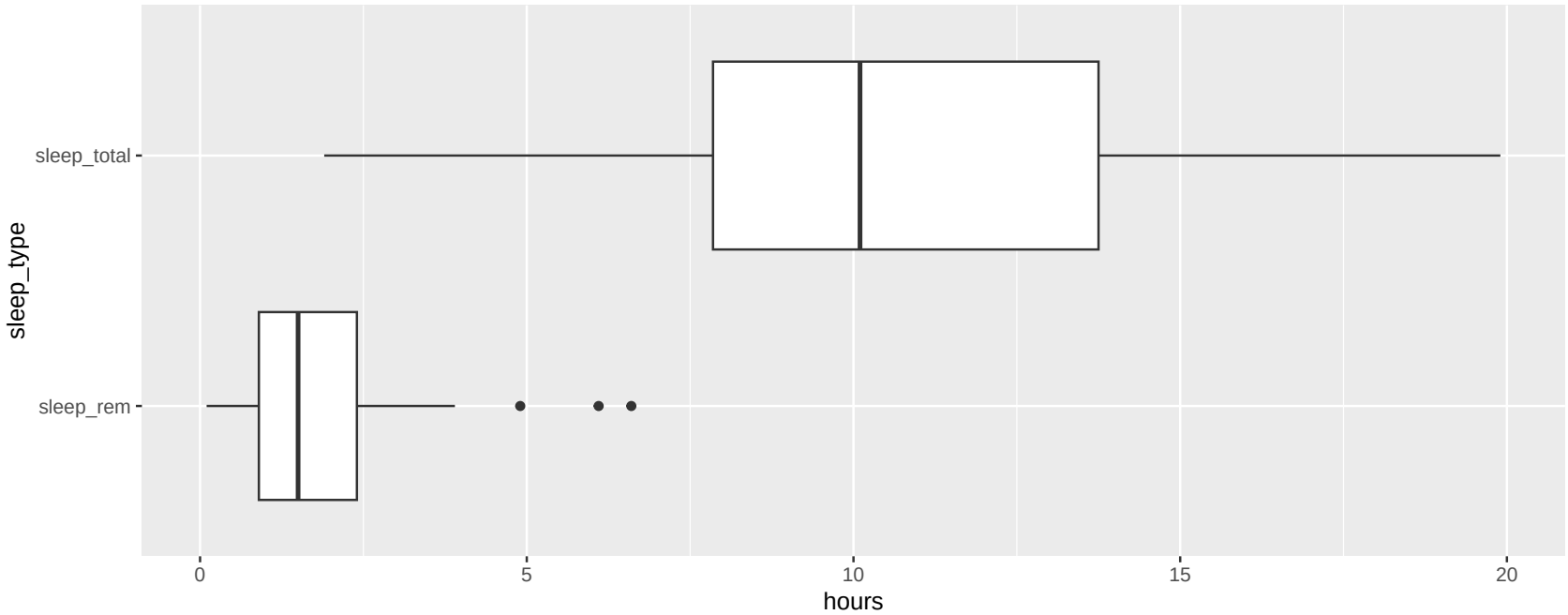
- The `pivot_longer` verb reverses the effect of `pivot_longer`.
- One column gives the new column names and the other column provides the values.

[6] Image taken from the *tidyr cheat sheet*, Posit Software, PBC

Long vs. wide dataframes

Now we can plot the sleep times:

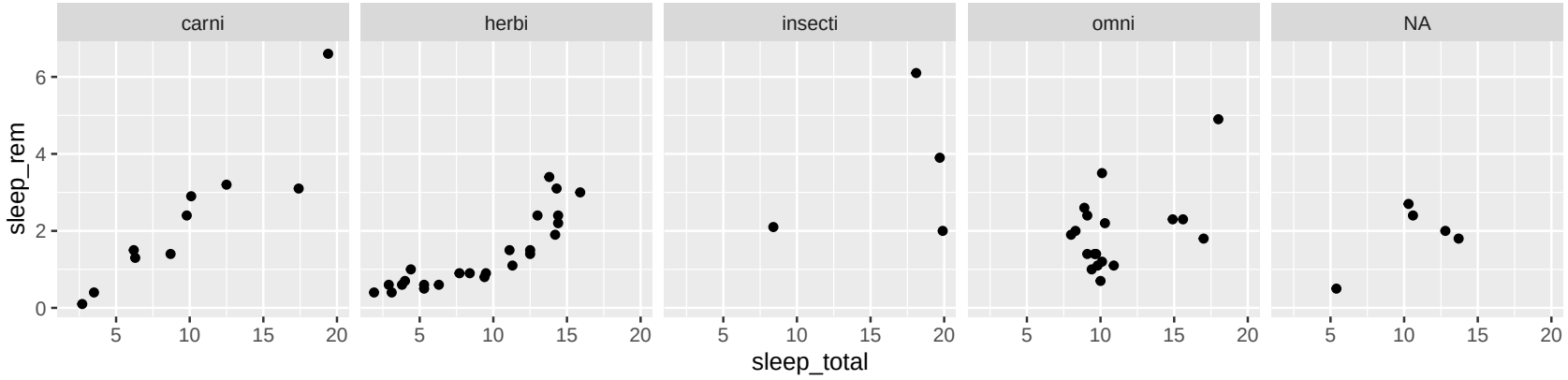
```
G <- msleep |> pivot_longer(c(sleep_total, sleep_rem),
                            names_to = "sleep_type",
                            values_to = "hours")
ggplot(data=G,aes(x = hours, y = sleep_type)) +
geom_boxplot()
```



Faceting

- Faceting is a tool to show different subsets of data in the same plot. Every group will be displayed in its own facet, but all facets share the same axes.
- Use the `facet_wrap()` function to add a faceting specification to the plot. The ~ is part of the syntax (can not be avoided).

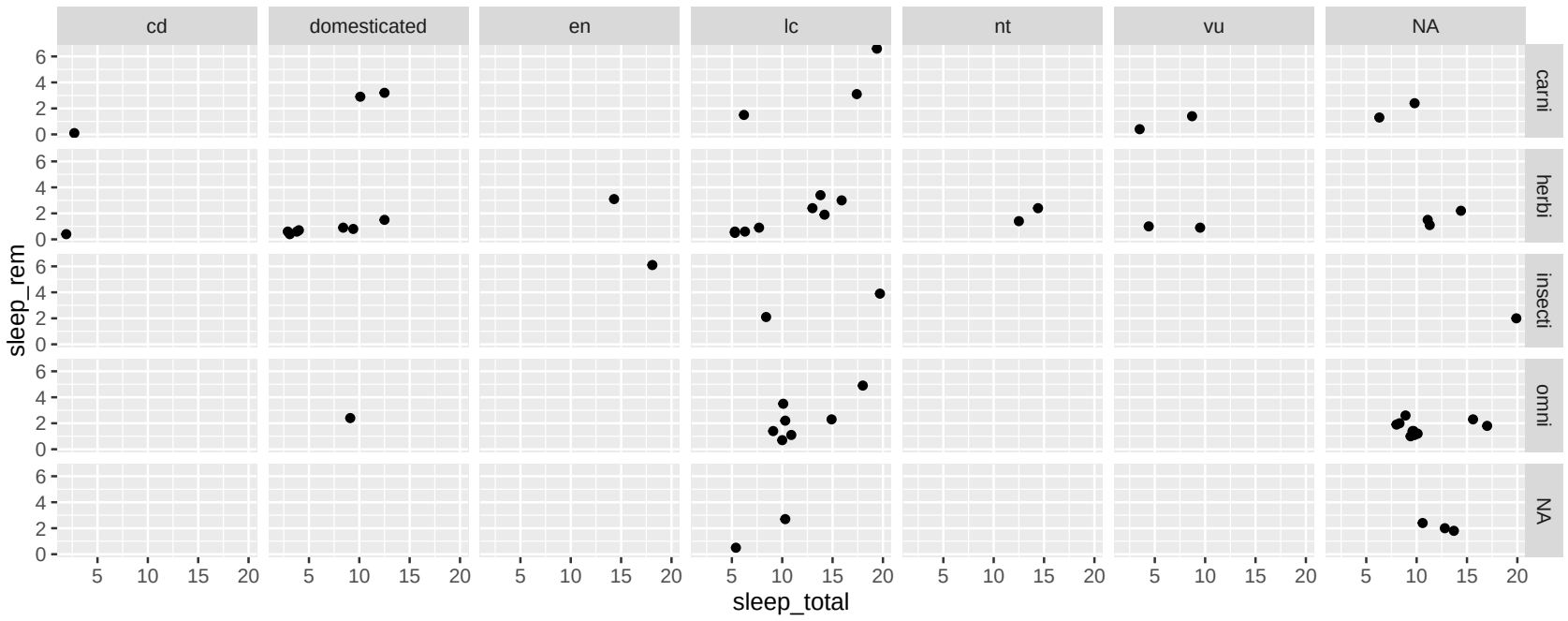
```
msleep |> ggplot(aes(x = sleep_total, y = sleep_rem)) +
  geom_point() + facet_wrap(~vore, nrow = 1)
```



Faceting

- Faceting also works with two variables:

```
msleep |> ggplot(aes(x = sleep_total, y = sleep_rem)) +
  geom_point() +
  facet_grid(vore ~ conservation)
```



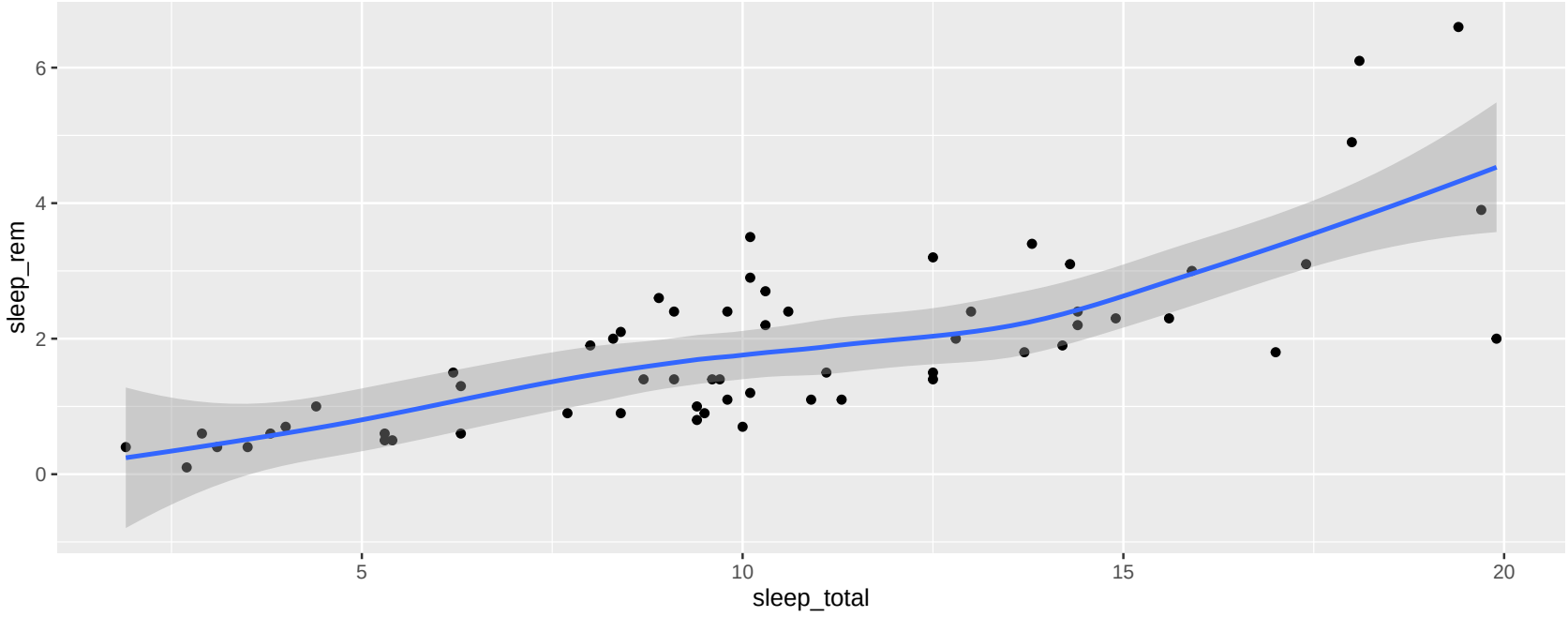
Statistical transformations

- In scientific plots, we might not just want to show the raw data, but also some statistical summaries.
- For example a regression line through a scatter plot or the marginal densities along the axes.
- The layered nature of `{ggplot2}` allows us to do this easily.

Statistical transformations

- Example with a smoother (regression):

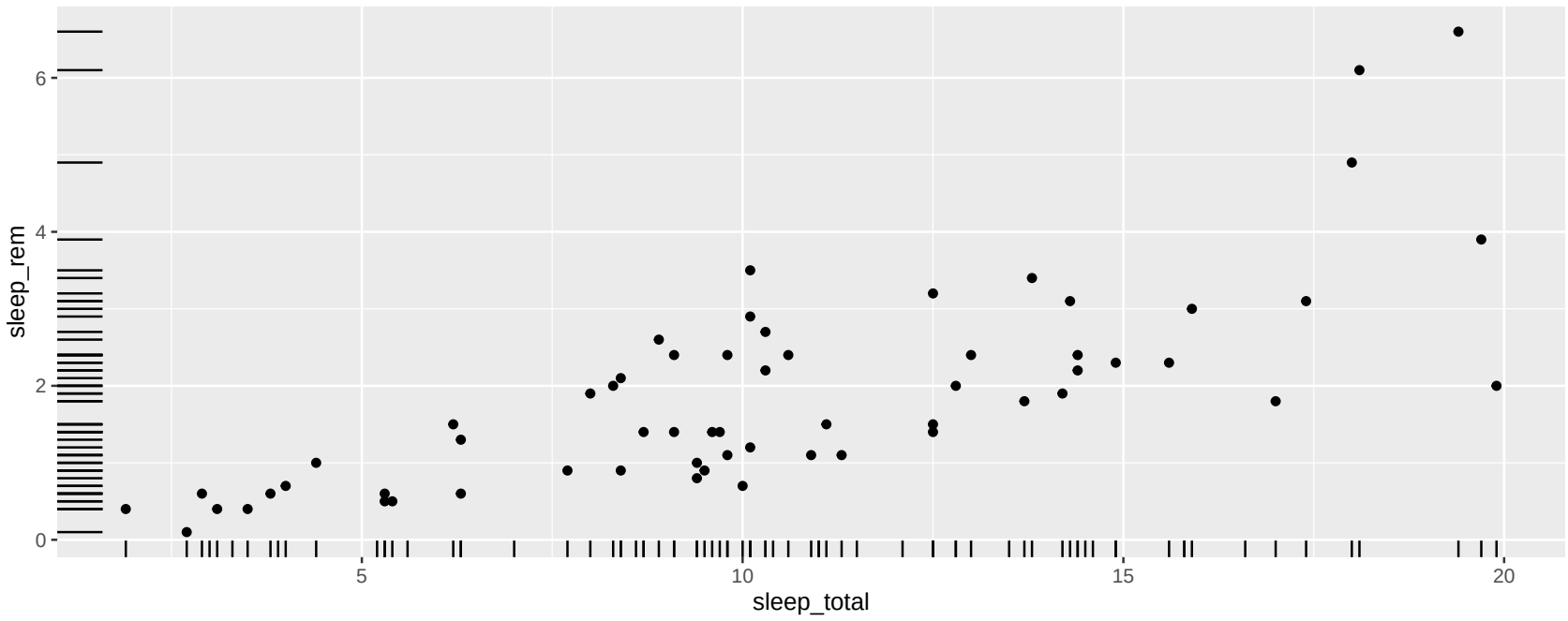
```
msleep |> ggplot(aes(x = sleep_total, y = sleep_rem)) +
  geom_point() +
  geom_smooth()
```



Statistical transformations

- Example visualizing the marginal densities:

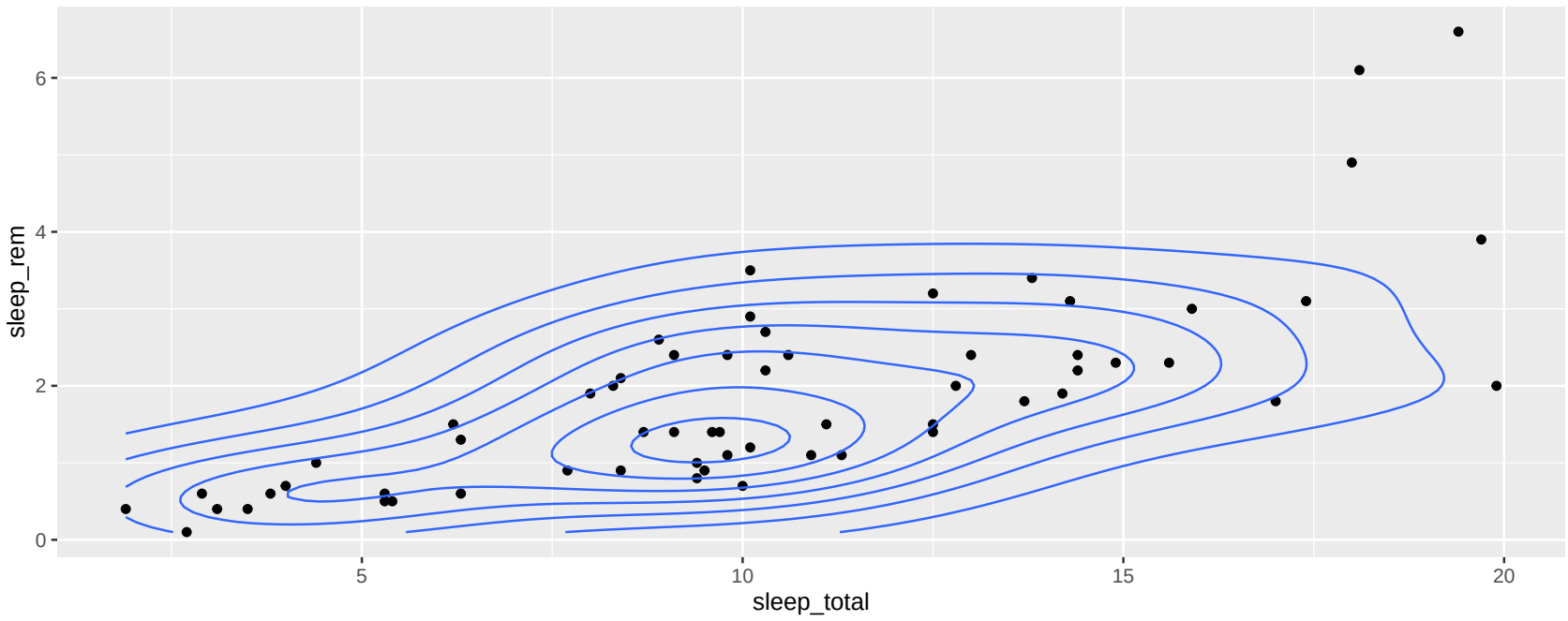
```
msleep |> ggplot(aes(x = sleep_total, y = sleep_rem)) +
  geom_point() +
  geom_rug()
```



Statistical transformations

- Example with the contour lines of the bivariate density:

```
msleep |> ggplot(aes(x = sleep_total, y = sleep_rem)) +
  geom_point() +
  geom_density2d()
```



Exercise

- How many omnivores are in the dataset?
- How many hours do domesticated animals sleep on average?
- Plot body weight against brain weight. Play around with different scales (e.g. sqrt-log or log-log). What do you observe?
- Can you find other patterns?

Section 4

More on tidymodels

Other models

- So far we only learned about the linear model.
- But the {tidymodels} framework provides interfaces to many other models:

Model	Name in {tidymodels}
Linear model	`linear_reg()`
Naive Bayes	`naive_Bayes()`
Decision Tree	`decision_tree()`
Random forest	`rand_forest()`
Support Vector Machine	`svm_linear()`
Neural Network	`mlp()`

Other models

```r
data_split <- msleep |> drop_na() |>
  initial_split(prop = 3/4)
model <- rand_forest() |> set_mode("regression")
fitted <- model |>
  fit(sleep_total ~ vore + sleep_rem + brainwt,
      data = data_split |> training())
fitted |> augment(data_split |> testing()) |>
  rmse(truth = sleep_total, .pred)
```

```
# A tibble: 1 x 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 rmse    standard        2.92
```

Preprocessors

- If we remove the `drop_na()` verb in the first line of the previous slide, we get an error, because the default engine for the `rand_forest()` model doesn't support `NA`s in the input data.
- Most ML models are picky when it comes to input data: Some models don't support factorial/discrete variables; others get unstable if numerical variables are not normalized and so on...
- This is where *preprocessors* come into play: They are part of the overall model and transform the raw input data to the required form for the actual model.
- In general, preprocessors need to be trained on the training data, e.g. for estimating a normalization transformation.

Preprocessors

- In the {tidymodels} framework, one starts with an empty *recipe* and then adds preprocessing steps to it:

```
rec <- recipe(sleep_total ~ vore + sleep_rem + brainwt,
              data = msleep) |>
  step_impute_mean(all_numeric_predictors()) |>
  step_impute_mode(all_string_predictors())
```

- The `recipe()` functions defines which variables are *predictors* and which variable is the *outcome*.
- All preprocessing steps can selectively applied, e.g. `all_predictors()` or `all_nominal_predictors()`.

Preprocessors

- To use the preprocessor, one defines a *workflow*, consisting of the preprocessor along with the model specification, and trains both of them:

```
wflow <- workflow() |>
  add_recipe(rec) |>
  add_model(model)
fitted <- wflow |> fit(data = data_split |> training())
fitted |> augment(data_split |> testing()) |>
  rmse(truth = sleep_total, .pred)
```

```
# A tibble: 1 x 3
  .metric .estimator .estimate
  <chr>   <chr>          <dbl>
1 rmse    standard        2.91
```

Preprocessors

Some common preprocessors:

- `step_dummy()` does one-hot-encoding of all selected variables
- `step_impute_bag` does imputation using bagged trees
- `step_YeoJohnson()` tries to normalize all selected variables (mean $= 0$, sd $= 1$)
- `step_nzv()` throws away all selected variables with near-zero variance
- `step_corr()` throws away all selected variables that strongly correlate

Different metrics

So far we only looked at the R-squared metric to assess the predictive power of our ML models. You already learned about other metrics and `{tidymodels}` supports a wide range of metrics:

- `sens()` and `spec()` measure sensitivity and specificity (binary classification)
- `precision()` and `recall()` measure precision and recall (binary classification)
- `accuracy()` measures the accuracy (binary classification)
- `kap()` measures *Kohen's kappa* (multiclass classification)
- `roc_auc()` measures the Area under the Receiver Operator Curve (binary class probability classification)
- `rmse()` and `mae()` measure the root-mean-squared error and the mean absolute error

Different metrics

- There are many more (and exotic) quality measures.
- A full list is available by typing `help(package = yardstick)` in the `R` console.

Exercise

- Remove the `name` column from the `msleep` dataset and transform string columns to factorial columns. Do a training-testing split.
- Define a preprocessor for the `msleep` dataset: One-hot-encoding of factorial variables and mean/mode-imputation for numerical variables.
- Train a single-layer neural network on the training data for predicting `sleep_rem`
- Plot true vs. estimated `sleep_rem` and report the rmse on the test data