Extras

R für empirische Wissenschaften v1.0.2

Jan Philipp Nolte

Inhaltsverzeichnis

1 Einführung **2**

2 Verschiedene Datenstrukturen **2**
2.1 tibble und data.frame 2
2.2 Vektor 3
2.2.1 Eckige Klammern 4
2.3 Matrix 6
2.4 Liste 7
2.5 Umwandlungen von Datenstrukturen 9

3 Funktionales Programmieren **9**
3.1 Funktionen erstellen 9
3.1.1 Non-standard evaluation 10
3.2 Mappen 10
3.3 Nesten 12

4 Transformation einer Ergebnismatrix **13**

1 Einführung

In den letzten fünf Kapiteln hast Du bereits alles dringend Nötige zum Verarbeiten von Daten kennengelernt. Allerdings wurden einige Konzepte bisher aus didaktischen Gründen ohne Erklärung verwendet. Hier werden diese Konzepte sowie andere für das Verständnis von R wichtigen Dinge genauer betrachtet. Außerdem werden mit funktionalem Programmieren forgeschrittene Konzepte vorgestellt. Zum Schluss wird noch eine Funktion erklärt, die eine Ergenismatrix zu einer binären Antwortmatrix formatiert. Also wenn man zum Beispiel in einem Online Fragebogen Aufgaben abfragt, die eine richtige Lösung haben, sind diese im rohen Datensatz in einem unbrauchbaren Format. Wenn in Aufgabe 1 die Zahl 42 herauskommt und 3 Personen den Fragebogen beantwortet haben, mit 42, 34 und 5, enthält der Rohdatensatz keinerlei Information darüber, dass nur die erste Person richtig geantwortet hat. Die Funktion zur Transformation würde daraus 1, 0, 0 machen. Ein Beispiel wird später gegeben.

2 Verschiedene Datenstrukturen

Wie bei den bereits kennengelernten Datentypen gibt es auch verschiedene Strukturen, die jeweils andere Eigenschaften haben. In den vorherigen Kapiteln ist oft das Wort tibble gefallen. Wenn Du bereits die eine oder andere Erfahrung mit R gemacht hast, kennst Du wahrscheinlich eher data.frames. Was diese Datenstruktur ausmacht und wie man verschiedene Datenstrukturen innerhalb von R direkt erstellen kann (ohne Einlesen eines bereits bestehenden Datensatzes), wird im Folgenden erklärt.

2.1 tibble und data.frame

Die vielleicht nützlichste Datenstruktur für den Wissenschaftler ist der tibble. Warum? Weil, obwohl innerhalb einer Spalte der selbe Datentyp benutzt werden muss, verschiedene Datentypen in verschiedenen Spalten sein dürfen. Man sollte nach Möglichkeit immer tibbles anstelle der veralteten data.frames benutzen, allerdings gibt es einige wenige alte Funktionen, die mit tibbles nicht kompatibel sind. Wie man Datenstrukturen umformt, sehen wir später. Die genauen Unterschiede und Vorteile von tibbles können hier nachgelesen werden.

Einen tibble kann man mit der gleichnamigen Funktion (nach laden des `tidyverse`) erstellen.

```
tibble(
  a = 1:3,
  b = 4:6
)
```

```
## # A tibble: 3 x 2
##       a     b
##   <int> <int>
## 1     1     4
```

```
## 2     2     5
## 3     3     6
```

2.2 Vektor

Ein Vektor ist eine eindimensionale Datenstruktur, die nur einen Datentypen enthalten darf. Also entweder Numeric, Character oder Logical. Jede Spalte in einem Datensatz ist nichts anderes als ein Vektor. Vektoren können auf verschiedene Art und Weise erstellt werden.

Einen leeren Vektor mit einer bestimmten Anzahl an Elementen kann man mit `vector()` erstellen.

```
vector("numeric", 4)
```

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

Man kann mehrere Werte mit `c()` (combine) kombinieren.

```
c(1, 2, 3, 4)
```

```
## [1] 1 2 3 4
```

Bereits kennengelernt haben wir den Doppelpunkt. Werte von 1 bis 4 erhalten wir also durch

```
1:4
```

```
## [1] 1 2 3 4
```

Auch mit `seq()` erreichen wir das selbe Ergebnis. Vorteil hierbei ist der, dass man die Abstände zwischen den Zahlen verändern kann.

```
seq(from = 1, to = 4, by = 1)
```

```
## [1] 1 2 3 4
```

Außerdem gibt es nützliche `seq`-Funktionen für häufige Anwengungsfälle, wie `seq_len()` für eben genau das vorherige Beispiel.

```
seq_len(4)
```

```
## [1] 1 2 3 4
```

Und `seq_along()` für eine Sequenz von 1 bis zur Länge des Vektors, was häufig beim Loopen hilfreich ist.

```
seq_along(c("a", "b", "c"))
```

```
## [1] 1 2 3
```

2.2.1 Eckige Klammern

Wenn man auf Elemente innerhalb eines Vektor zugreifen möchte, erzielt man dies mit eckigen Klammern. Das dritte Element des Vektors `vec`

```
vec <- c(1, 2, 3, 4)
```

erhält man mit

```
vec[3]
```

```
## [1] 3
```

Eckigen Klammern können auch bei tibbles verwendet werden. Wählen wir zuerst erneut die Personenspalte aus.

```
tipp_wm[ ,"Person"]
```

```
## # A tibble: 384 x 1
##    Person
##    <chr>
##  1 Thomas_Bayes
##  2 Thomas_Bayes
##  3 Thomas_Bayes
##  4 Thomas_Bayes
##  5 Thomas_Bayes
##  6 Thomas_Bayes
##  7 Thomas_Bayes
##  8 Thomas_Bayes
##  9 Thomas_Bayes
## 10 Thomas_Bayes
## # ... with 374 more rows
```

Was ändert sich im Vegleich zu Vektoren? Innerhalb der eckigen Klammer befindet sich nun ein Komma. Links vom Komma kann man 1 bis alle **Zeilen** auswählen, rechts vom Komma die gewünschten **Spalten**. Wenn man es frei lässt, werden alle ausgewählt. Man kann alternativ auch numerisch auswählen.

```
tipp_wm[ ,2]
```

```
## # A tibble: 384 x 1
##    Person
##    <chr>
##  1 Thomas_Bayes
##  2 Thomas_Bayes
##  3 Thomas_Bayes
##  4 Thomas_Bayes
##  5 Thomas_Bayes
##  6 Thomas_Bayes
```

```
##  7 Thomas_Bayes
##  8 Thomas_Bayes
##  9 Thomas_Bayes
## 10 Thomas_Bayes
## # ... with 374 more rows
```

Aber man könnte genauso gut mehrere Spalten auswählen. Spalten 1 bis 3 beispielweise mit

```
tipp_wm[ ,1:3]
```

```
## # A tibble: 384 x 3
##    Spieltag Person       Tipp
##       <dbl> <chr>        <chr>
##  1        1 Thomas_Bayes 2:0
##  2        2 Thomas_Bayes 1:3
##  3        2 Thomas_Bayes 0:0
##  4        2 Thomas_Bayes 1:2
##  5        3 Thomas_Bayes 2:1
##  6        3 Thomas_Bayes 3:0
##  7        3 Thomas_Bayes 0:2
##  8        3 Thomas_Bayes 4:0
##  9        4 Thomas_Bayes 2:1
## 10        4 Thomas_Bayes 3:0
## # ... with 374 more rows
```

oder Spalten 2 und 4 mit

```
tipp_wm[ ,c(2, 4)]
```

```
## # A tibble: 384 x 2
##    Person       Ergebnis
##    <chr>        <chr>
##  1 Thomas_Bayes 5:0
##  2 Thomas_Bayes 0:1
##  3 Thomas_Bayes 0:1
##  4 Thomas_Bayes 3:3
##  5 Thomas_Bayes 2:1
##  6 Thomas_Bayes 1:1
##  7 Thomas_Bayes 0:1
##  8 Thomas_Bayes 2:0
##  9 Thomas_Bayes 0:1
## 10 Thomas_Bayes 0:1
## # ... with 374 more rows
```

Bei großen Datensätzen kann man die gewünschte Spaltennummer beispielsweise mit `str_which()` herausfinden.

```r
str_which(names(tipp_wm), "Person")
```

```
## [1] 2
```

Eine gewünschte Anzahl an Zeilen kann man nach dem gleichen Schema wie mit den Spalten auswählen. Möchte man zum Beispiel die ersten 20 Zeilen der Spalten 1 und 5 ausgegeben haben:

```r
tipp_wm[1:20, c(1, 5)]
```

```
## # A tibble: 20 x 2
##    Spieltag Tipp_Richtung
##       <dbl> <chr>
##  1        1 S
##  2        2 N
##  3        2 U
##  4        2 N
##  5        3 S
##  6        3 S
##  7        3 N
##  8        3 S
##  9        4 S
## 10        4 S
## 11        4 S
## 12        5 U
## 13        5 S
## 14        5 N
## 15        6 S
## 16        6 S
## 17        6 S
## 18        7 S
## 19        7 S
## 20        7 N
```

2.3 Matrix

Zwei zumsammengebundene Vektoren ergeben nicht zwingend einen tibble. Wenn die beiden Vektoren den selben Datentyp haben, kann man auch eine Matrix daraus machen. In der wissenschaftlichen Praxis wird das allerdings nur selten benötigt. Wir haben das lediglich bei der MANOVA verwendet. Wir schauen uns trotzdem drei verschiedene Möglichkeiten an, eine Matrix zu erstellen. Der Funktion `matrix()` übergibt man als erstes Argument alle gewünschten Werte. Mit `ncol` respektive `nrow` kann man dann die Anzahl der Spalten beziehungsweise Zeilen festlegen.

```r
matrix <- matrix(1:8, ncol = 2)
```

```
##      [,1] [,2]
## [1,]    1    5
## [2,]    2    6
## [3,]    3    7
## [4,]    4    8
```

Alternativ kann man auch zwei Vektoren mit `cbind()` (column bind) Spaltenweise aneinander hängen.

```r
cbind(1:4,
      5:8)
```

```
##      [,1] [,2]
## [1,]    1    5
## [2,]    2    6
## [3,]    3    7
## [4,]    4    8
```

Nach dem gleichen Prinzip kann `rbind()` (row bind) für zeilenweises Verbinden verwendet werden.

```r
rbind(c("a", "b", "c"),
      c("d", "e", "f"))
```

```
##      [,1] [,2] [,3]
## [1,] "a"  "b"  "c"
## [2,] "d"  "e"  "f"
```

Wie wir bereits beim Modellvergleich im Rahmen der hierarchischen Regression beobachtet haben, können mit `cbind()` und `rbind()` auch tibbles der gleichen Grösse aneinander gehängt werden. Der Zugriff auf Zeilen und Spalten innerhalb der Matrix funktioniert nur mit eckigen Klammern. Der Dollar-Operator funktioniert hier nicht, weil eine Matrix keine Spaltennamen hat. Auf die Ausführung von drei dimensionalen Matrizen – den Arrays – sei an dieser Stelle verzichtet.

2.4 Liste

Die allgemeinste Datenstruktur ist die Liste. In einer Liste kann man alles verstauen – egal ob Vektoren, Matrizen oder tibbles.

```r
liste <- list(vec, matrix, big_five)
```

```
## [[1]]
## [1] 1 2 3 4
##
```

```
## [[2]]
##      [,1] [,2]
## [1,]    1    5
## [2,]    2    6
## [3,]    3    7
## [4,]    4    8
##
## [[3]]
## # A tibble: 200 x 8
##    Alter Geschlecht Herkunft Extraversion Neurotizismus Vertraeglichkeit
##    <dbl> <chr>      <chr>           <dbl>         <dbl>             <dbl>
##  1    36 m          DE                3             1.9               3.4
##  2    30 f          US                3.1           3.4               3.1
##  3    23 m          US                3.4           2.4               3.6
##  4    54 m          US                3.3           4.2               3.6
##  5    24 f          US                3             2.8               3
##  6    14 f          US                2.8           3.5               3.2
##  7    32 m          HK                3.5           3.1               4.2
##  8    20 m          IN                3.5           2.6               3.5
##  9    29 f          IN                3             3.7               3.2
## 10    17 m          US                3.1           3.6               3
## # ... with 190 more rows, and 2 more variables: Gewissenhaftigkeit <dbl>,
## #   Offenheit <dbl>
```

Eine leere Liste der Länge 5 erstellt man mit

```
vector("list", 5)
```

```
## [[1]]
## NULL
##
## [[2]]
## NULL
##
## [[3]]
## NULL
##
## [[4]]
## NULL
##
## [[5]]
## NULL
```

Auf die Listenelemente erhält man mit doppelten eckigen Klammern Zugriff. Das zweite Listenelement (hier die Matrix) erhält man mit

```
liste[[2]]
```

```
##      [,1] [,2]
## [1,]    1    5
## [2,]    2    6
## [3,]    3    7
## [4,]    4    8
```

Weitergehend kann man direkt auf Elemente der Matrix im zweiten Listenelement zugreifen. Möchte man beispielsweise den Wert in der dritten Zeile und zweiten Spalte der Matrix ausgegeben haben, schreibt man

```
liste[[2]][3, 2]
```

```
## [1] 7
```

Listen sind extrem wichtig beim funktionalen Programmieren und nützlich beim Importieren mehrerer Datensätzen auf einmal.

2.5 Umwandlungen von Datenstrukturen

Datenstrukturen können grundsätzlich nur ineinander umgewandelt werden, wenn sämtliche notwendigen Eigenschaften erfüllt sind. Ein tibble, welcher in 2 Spalten verschiedene Datentypen beinhaltet, kann zum Beispiel **nicht** zu einer Matrix konvertiert werden. Beziehungsweise geht das schon, allerdings werden dann alle Werte zu einem Datentyp konvertiert.

```
as.matrix()
as.data_frame()
as.tibble()
as.numeric()
as.character()
```

3 Funktionales Programmieren

Die Möglichkeiten des funktionalen Programmierens werden nur exemplarisch angeschnitten, um eine Idee von den fortgeschrittenen Konzepten zu erhalten.

3.1 Funktionen erstellen

Manchmal bietet es sich an, eigene Funktionen zu schreiben. Dabei schreibt man das Argument, was der Funktion weitergegeben werden soll, in die Klammern von `function()`. Erst einmal gespeichert, kann man die Funktion beliebig oft anwenden. Sie verhält sich dabei genau wie sämtliche anderen Funktionen in R, wie z.B. `mean(x)`. Angenommen, man möchte eine

Spalte logarithmieren, die Nullen beinhaltet: Um sinnvolle Werte zu erhalten, sollte man die gesamte Skala anheben – beispielsweise um 2. Unsere erste eigene Funktion `log_add_2()` (willkürlich gewählter Name) nimmt eine Zahl `x`, addiert sie mit 2 und logarithmiert diesen Wert anschließend.

```
log_add_2 <- function(x) {
  log(x + 2)
}
```

Nun können wir die Funktion wie gewohnt anwenden.

```
log_add_2(0)
```

```
## [1] 0.6931472
```

3.1.1 Non-standard evaluation

Funktionen aus dem `tidyverse` stellen dabei eine Ausnahme dar. An dieser Stelle wird es leider sehr schnell komplex und kryptisch. Es sei nur erwähnt, damit bei Versuchen mit `tidyverse` Funktionen eigene Funktionen zu schreiben, kein Frust aufkommt. Haben wir zum Beispiel die einfache Aufgabe, den Mittelwert und die Standardabweichung einer Variablen in einem Datensatz zu berechnen, müssen wir so genannte quosures (`enquo()`) und doppelte Bang (`!!`) Operatoren benutzen, um die Aufgabe in Form einer Funktion zu automatisieren.

```
mean_var <- function(data, variable){
  variable <- enquo(variable)

  data %>%
    summarise(Mittelwert = mean(!!variable),
              SD = sd(!!variable))
}
```

Dafür könnte man diese Funktion schön in einer Pipe verwenden.

```
indonesisch %>%
  mean_var(Alter)
```

```
## # A tibble: 1 x 2
##   Mittelwert    SD
##        <dbl> <dbl>
## 1       21.6  5.93
```

3.2 Mappen

Der Kern von funktionalem Programmieren ist Automatisierung. Erinnern wir uns zurück an die Inferenzstatistik. Genauer gesagt an das Berechnen der Informationskriterien der

Modelle der hierarchischen Regression mithilfe von `glance()`. Wir haben dort 3 mal den Befehl ausführen müssen. Für jedes der 3 Modelle einmal. Aber was, wenn wir 10 Prädiktoren sukzessive hinzufügen wollen würden? 10 mal Copy & Paste? Eine fehleranfällige und mühsame Vorgehensweise. Mit `map()` aus dem `purrr` Package (enhalten im `tidyverse`) kann man mit 3 Zeilen Code das selbe erreichen. Zuerst speichern wir dafür die 3 Modelle in einer Liste.

```
modelle <- list(model1, model2, model3)
```

Nun greifen wir in gewohnter `tidyverse` Manier auf die Liste zu. Die Funktion `map()` wendet nun auf jedes Listenelement die Funktion `glance()` an. Nicht vergessen `broom` vorher zu laden.

```
modelle %>%
  map(glance)
```

```
## [[1]]
## # A tibble: 1 x 11
##   r.squared adj.r.squared sigma statistic p.value    df logLik   AIC   BIC
## *     <dbl>         <dbl> <dbl>     <dbl>   <dbl> <int>  <dbl> <dbl> <dbl>
## 1   0.00486     -0.000164 0.347     0.967   0.327     2  -70.9  148.  158.
## # ... with 2 more variables: deviance <dbl>, df.residual <int>
##
## [[2]]
## # A tibble: 1 x 11
##   r.squared adj.r.squared sigma statistic p.value    df logLik   AIC   BIC
## *     <dbl>         <dbl> <dbl>     <dbl>   <dbl> <int>  <dbl> <dbl> <dbl>
## 1    0.0163       0.00630 0.345      1.63   0.198     3  -69.7  147.  161.
## # ... with 2 more variables: deviance <dbl>, df.residual <int>
##
## [[3]]
## # A tibble: 1 x 11
##   r.squared adj.r.squared sigma statistic p.value    df logLik   AIC   BIC
## *     <dbl>         <dbl> <dbl>     <dbl>   <dbl> <int>  <dbl> <dbl> <dbl>
## 1    0.0364        0.0216 0.343      2.47  0.0634     4  -67.7  145.  162.
## # ... with 2 more variables: deviance <dbl>, df.residual <int>
```

Wenn wir nun wie in unserer Copy & Paste Lösung einen tibble und keine Liste zurück haben wollen, ändern wir `map()` einfach zu `map_df()`.

```
modelle %>%
  map_df(glance)
```

```
## # A tibble: 3 x 11
##   r.squared adj.r.squared sigma statistic p.value    df logLik   AIC   BIC
##       <dbl>         <dbl> <dbl>     <dbl>   <dbl> <int>  <dbl> <dbl> <dbl>
## 1   0.00486     -0.000164 0.347     0.967  0.327      2  -70.9  148.  158.
## 2   0.0163       0.00630  0.345     1.63   0.198      3  -69.7  147.  161.
```

```
## 3   0.0364       0.0216   0.343     2.47   0.0634     4  -67.7  145.  162.
## # ... with 2 more variables: deviance <dbl>, df.residual <int>
```

Alternativ könnte man natürlich auch die apply-Familie aus `base R` oder klassische for oder while Schleifen verwenden.

3.3 Nesten

Wirklich mächtig wird `map()` allerdings erst in Kombination mit `nest()`. Zu Beginn ist es durchaus abstrakt und gewöhnungsbedürftig. Aber durch das Nesten kann man gruppenweise mappen. Nehmen wir uns das Beispiel unseres `repeated` Datensatzes. Es gibt 6 verschiedene Gruppen - A bis F. Was, wenn wir nun innerhalb jeder dieser Gruppen einen F-Test zum Vergleich der Varianzen von `iq` und `kreativitaet` machen möchten? Normalerweise müsste man 6 mal filtern und anschließend die Funktion F-Test aufrufen. Mit `nest()` benötigt man bloss 4 Zeilen Code.

```
repeated %>%
  nest(-gruppe) %>%
  mutate(f_tests = map(data, ~ var.test(.$iq, .$kreativitaet)),
         ergebnisse = map(f_tests, tidy)) %>%
  unnest(ergebnisse)
```

```
## Multiple parameters; naming those columns num.df, denom.df
## Multiple parameters; naming those columns num.df, denom.df
## Multiple parameters; naming those columns num.df, denom.df
## Multiple parameters; naming those columns num.df, denom.df
## Multiple parameters; naming those columns num.df, denom.df
## Multiple parameters; naming those columns num.df, denom.df
```

```
## # A tibble: 6 x 12
##   gruppe data      f_tests  estimate `num df` `denom df` statistic p.value
##   <chr>  <list>    <list>      <dbl>    <int>      <int>     <dbl>   <dbl>
## 1 A      <tibble ~ <S3: ht~    1.28        14         14     1.28   0.651
## 2 B      <tibble ~ <S3: ht~    0.465       14         14     0.465  0.164
## 3 C      <tibble ~ <S3: ht~    0.672       14         14     0.672  0.466
## 4 D      <tibble ~ <S3: ht~    0.282       14         14     0.282  0.0242
## 5 E      <tibble ~ <S3: ht~    0.697       14         14     0.697  0.508
## 6 F      <tibble ~ <S3: ht~    0.397       14         14     0.397  0.0948
## # ... with 4 more variables: conf.low <dbl>, conf.high <dbl>,
## #   method <chr>, alternative <chr>
```

Auf die selbe Art könnte man auch mehrere hundert Regressionen ohne auch nur eine zusätzliche Zeile Code rechnen.

4 Transformation einer Ergebnismatrix

Angenommen wir machen eine Online-Umfrage mit einem kleinen Mathetest bestehend aus 3 Aufgaben. Die richtigen Antworten sind 3, 2 und 4. Es nehmen 4 Personen teil. Unser Datensatz `survey` würde in etwas so aussehen

```
## # A tibble: 4 x 3
##      I1    I2    I3
##   <dbl> <dbl> <dbl>
## 1     3     2     4
## 2     2     2     1
## 3     3     2     4
## 4     1     1     4
```

Das bringt uns allerdings herzlich wenig zur Auswertung. Was wir möchten, sind die Informationen pro Item und Person, ob die Aufgabe korrekt gelöst wurde. Mit der Funktion `transform_binary()` des `rBasics` Packages, kann man anhand eines Antwortvektors, der die richtigen Antworten enthält, die Matrix transformieren.

```
antwortvec <- c(3, 2, 4)

transform_binary(survey, antwortvec)
```

```
## # A tibble: 4 x 3
##      I1    I2    I3
##   <dbl> <dbl> <dbl>
## 1     1     1     1
## 2     0     1     0
## 3     1     1     1
## 4     0     0     1
```

Falls sowohl Zahlen als auch Characters als richtige Antworten in Frage kommen, kann man der Funktion ebenfalls einen einzeiligen tibble übergeben. Wenn wir also die Umfrage in der Form haben

```
## # A tibble: 4 x 3
##      I1 I2          I3
##   <dbl> <chr>    <dbl>
## 1     3 Apfel        4
## 2     2 Banane       1
## 3     3 Erdbeere     4
## 4     1 Apfel        4
```

und die richtigen Antworten 3, Apfel und 4 sind

```
antwortvec <- tibble(3, "Apfel", 4)
```

erhält man mit dem gleichen Befehl das Ergebnis.

```
transform_binary(survey, antwortvec)
```

```
## # A tibble: 4 x 3
##      I1    I2    I3
##   <dbl> <dbl> <dbl>
## 1     1     1     1
## 2     0     0     0
## 3     1     0     1
## 4     0     1     1
```