Vignette zum Paket OlfaBA

ULR

2020-03-03

Das Paket BTBA1

Diese Paket soll als Langzeitalternative die Studenten im Studiengang Biotechnologie vor der Verwendung von Office-Programmen lösen. Laden könnt ihr das Paket über:

```
load_all(path = "~/Bachelor/Paket_BHT/BTBA1/")
#> Loading Biotech
```

Grundlegendes

Was macht R

Ich stelle mir R als einen mächtigen Taschenrechner vor, so mächtig, dass er wohl (fast) alles kann, warum? :

- R kann ableiten
- R kann aufleiten
- R kann mit komplexen Klassen arbeiten
- R kann Texte formatieren
- R ist beliebig erweiterbar
- eben wie ein virtueller, programmierbarer und schneller Taschenrechner!

Allerdings hängt es vom Anwender ab, wie gut all diese Möglichkeiten genutzt und angewandt werden. Um das zu können, sollte man die Sprache R können, also mit dem Programm R interagieren. Ein einfaches Beispiel:

```
1 + 1 # das ist ein Kommenta (alles nach dem #)
#> [1] 2
1 + 2 # das sollte selbstverständlich sein
#> [1] 3
4 / 2 # auch das sollte selbstverständlich sei
#> [1] 2
2 ^ 2 # zwei im quadrat
#> [1] 4
pi    # das hier ist pi
#> [1] 3.141593
cos( pi ) # die trigonometrische Funktion cos von pi
#> [1] -1
sin( pi ) # die trigonometrische Funktion sin von pi
#> [1] 1.224647e-16
```

```
e  <- exp(1) # hier wird e zur eulerschen Zahl
print( e ) # print() kann uns die Variable e printen
#> [1] 2.718282
log( e ) # was ist der ln von e?
#> [1] 1
```

Am Beispiel von log() sehen wir, dass *log* etwas macht, diese Information was es verarbeitet ist dann in den Klammern *()* hinterlegt. `log()` ist eine Funktion, eine Funktion um den natürlichen Logarithmus einer Zahl zu berechnen. Funktionen wie log() sind das Rüstzeug um komplexeren Rechnungen zu trotzen, sie sind ein essentieller Teil und werden später genauer betrachtet. Wir können beliebige Objekete erschaffen und diese Werte zuordnen um damit zu rechnen. Beispielhaft:

```
    # einen Vektor
Vektor.1  <- c( 1, 2, 3, 4 ) # das c macht aus dem eingeklammerten
                              # eine Vektor
Vektor.2  <- c( 5, 6, 7, 8 )

Vektor.3  <- Vektor.1 * Vektor.1 # man kann viel mit Vektoren anstellen
    # ein Tibble ist eine Tabelle
Vektoren <- tibble (
                    Vektor.1,
                    Vektor.2,
                    Vektor.3
                    )
    # eine Liste bietet Platz für viel Information:
Liste  <- list( "Buchstaben" = c('a', 'b', 'c'),
                "Zahlen" = c(1, 2, 3),
                "Ein Satz" = "Ick kann janich so viel fressen, wie ick kotzen möchte!
                , Max Liebermann")
    # Wenn wir nur den Namen eines Objekt schreiben
    # bekommen wir das Objekt ausgegeben
Vektor.1
#> [1] 1 2 3 4
Vektor.2
#> [1] 5 6 7 8
Vektor.3
#> [1]  1  4  9 16
Vektoren
#> # A tibble: 4 x 3
#>   Vektor.1 Vektor.2 Vektor.3
#>      <dbl>    <dbl>    <dbl>
#> 1        1        5        1
#> 2        2        6        4
#> 3        3        7        9
#> 4        4        8       16
Liste
#> $Buchstaben
#> [1] "a" "b" "c"
#>
#> $Zahlen
#> [1] 1 2 3
#>
#> $`Ein Satz`
```

```
#> [1] "Ick kann janich so viel fressen, wie ick kotzen möchte!\n                , Max Lie
```

Eine besondere Wichtigkeit bekommt das Pfeilchen zugesprochen, es ist innerhalb der Sprache R in der Lage einen Wert zuzuordnen Diese Zuordnung beschränkt nicht nur auf einfache Werte sondern gilt auch für Funktionen, Vektoren, Listen und so weiter. Vektoren, Listen, Tibbles usw. werden als Klasse bezeichnet, durch Zuordnung von Werten werden aus den Klassen Objekten. Man kann sich eine Klasse also ein wenig wie das Fließschema eines Bioreaktor vorstellen, wenn das Fließschema aus Metall gebaut wird und nicht mehr nur theoretisch existiert wird es zum Objekt, "mit Leben gefüllt", so wie wenn eben einer Klasse Werte zugeordnet werden. Der Vektor entsteht also durch Zuordnung von Daten zu einer leeren Klasse und bekommt dann einen Namen und einen klaren Sinn. Bei der Benennung eines Objektes sollte man sich Gedanken machen, nicht alle Namen sind sinnvoll, ein Leerzeichen kann da schon zu viel Problemen führen. Möchte man ein Objekt benennen sollte man vermeiden:

- - ist eben ein Minus-Operator
- + ist der Plus-Operator
- _*_ ist der Mal-Operator
- */* ist ist zum teilen
- *?* beschafft Hilfe :-), probier aus!
- *:* ein Sequenz-Operator
- = , <, >, & weitere Operatoren, später mehr!

Punkte und Unterstriche sind innerhalb der R-Umgebung ideal um Namen von Objekten zu strukturieren, Beispiele gab's bereits. Mehr dazu findet man zum Beispiel in R Core Team (2018) oder Wickham and Grolemund (2017). Um Vektoren durch bekannte mathematische Operationen zu generieren stehen uns verschiedene Funktionen zur Verfügung:

```
# von eins bis 10 zählen:
x  <- 1:10 %>% # eine Pipe, eine 'Röhre' zwischen den Funktinen
    print()
#>  [1]  1  2  3  4  5  6  7  8  9 10
# Von 1 bis 10 in 0.5-er Schritten:
x  <- seq(1, 10, 0.5) %>%
    print()
#>  [1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0  7.5  8.0
#> [16]  8.5  9.0  9.5 10.0
# einen Vektor der Länge 5 generieren:
x <- seq(1, 10, len = 5) %>%
    print()
#> [1]  1.00  3.25  5.50  7.75 10.00
```

seq() eignet sich gut für Sequenzen, rep() ist der Counterpart und wiederholt. Es ist auch möglich mit Buchstaben zu arbeiten, die heißen dann "character" aber interessieren uns erst mal nicht. Wichtig zu beachten ist, ist dass es einen Unterschied macht, wenn wir einen Buchstaben ohne Anführungszeichen verwenden oder mit!

```
print("Albatross")
#> [1] "Albatross"

Albatross <- 'Legehenne'

print(Albatross)
#> [1] "Legehenne"

Satz <- "Die Legehenne kennt's Eierlegen gut, sie macht's die ganze Zeit"
print(Satz)
```

```
#> [1] "Die Legehenne kennt's Eierlegen gut, sie macht's die ganze Zeit"
```

Am obigen Beispiel kann man schön erkennen, dass es Sinn macht, die Verwendung der Anführungszeichen nach einem Schema vorzunehmen. Es ist ratsam sich für eine Art von Anführungszeichen zu entscheiden, es hält alles übersichtlich.

Datenimport

Der Datenimport stellt häufig eine Hürde dar, die Messdaten liegen ja teilweise gar nicht als tibble oder Vektor oder Liste in der R-Umgebung rum sondern häufig als Excel-Tabelle im Ordner Downloads oder auf dem Desktop, weit weg von unserer R-Umgebung, die R-Umgebung ist übrigens der virtuelle Taschenrechner. Um nun Messdaten für uns nutzbar zu machen, müssen wir sie importieren wobei es nun wichtig ist zu wissen:

- wo die Daten sind
- in welchem Format sie vorliegen
- Was wir mit ihnen machen wollen

Der letze Punkt bezieht sich auf den Arbeitsaufwand. Beispielhaft hat man vier bereits vorbearbeitete Messwerte, diese können auch skrupellos abgetippt oder kopiert werden. Sollte es sich lohnen die Daten zu importieren stellt sich die Frage was eigentlich importiert wird, zum Beispiel eine Exceldatei oder eine CSV-Datei (comma separated value). Das Dateiformat ist Maßgeblich relevant bei der Wahl einer Funktion zum Import. Das “Wo” bezieht sich auf einen Dateipfad. Dieser gibt uns an wo auf unserem System die Datei auffindbar ist. Eine solche Datei muss aber nicht zwingend auf unserem Computer sein, sie kann sich auch im www befinden oder auf einem USB-Stick. Hier ist die Verwendung von R Studio sehr bequem, es ermöglicht uns innerhalb unseres Computers Dateien zu Suchen und generiert dann einen Code um den Datensatz zu importieren. Ein solcher Code besteht immer aus einer Import-Funktion, dem Dateipfad und optionalen Informationen:

```
 # Laden der benötigten Funktion
library(readxl)
 # Anwenden der Funktion
Rohdaten <- read_excel("~/Bachelor/Paket_BHT/Methoden_generell/aerober_abbau_detergenzien.
print(Rohdaten)
#> # A tibble: 8 x 3
#>   Gruppe `O2 [mg/L] davor` `O2 [mg/L] danach`
#>    <dbl>             <dbl>              <dbl>
#> 1      1               8.3                6.4
#> 2      2               8.2                6.1
#> 3      3               0                  0.1
#> 4      4               8.2                2.8
#> 5      5               8.3                4.3
#> 6      6               8.4                8.4
#> 7      7               8.1                6.4
#> 8      8               8.3                5.8
```

Der Importierte Datensatz heißt aerober_abbau_detergenzien.xlsx und wurde über readxl() importiert, in der R Umgebung heißt er nun Rohdaten. Der Dateipfad ist in Anführungszeichen. Nun liegt der Datensatz als “Rohdaten” vor und man kann damit arbeiten.

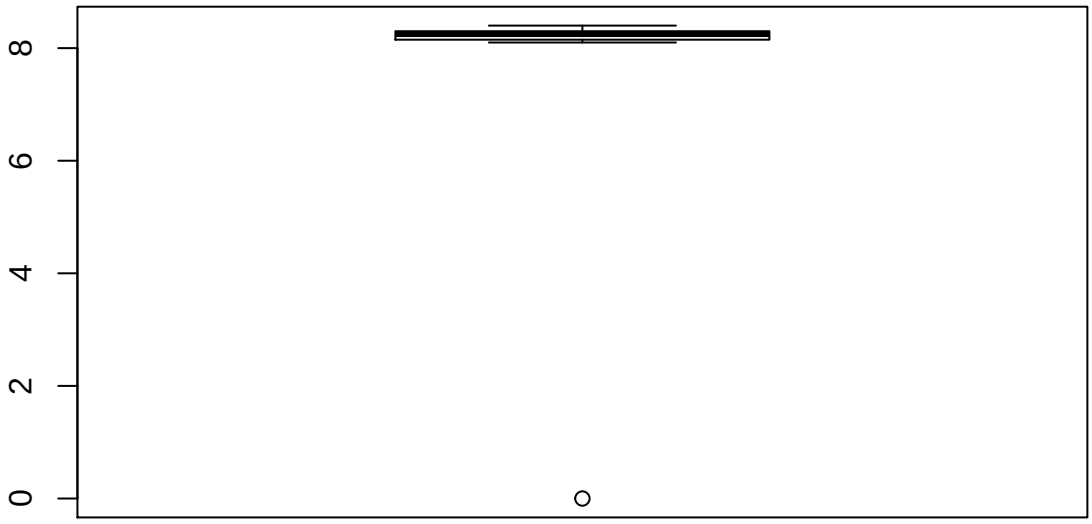
> es ist empfehlenswert die Messwerterfassung so einfach wie möglich durchzuführen, komplexe Tabellenstrukturen schaden da dem Datenimport und sind auch fürs Verständnis nicht unbedingt hilfreich.

Wenn beim Datenimport Probleme entstehen wird ein kleiner Text auf der Konsole erscheinen, dieser meistens sehr hilfreich um das Problem zu lösen, bei Härtefällen sollte eine Internetrecherche aber auch gute Lösungsansätze bringen.

Einfache Operationen

Die eben geladenen Messdaten sind nun innerhalb der R Umgebung für uns zugänglich, wir können sie nun untersuchen. Im Beispieldatensatz wurde die O_2-Konzentration in Wasser vor und nach der Behandlung mit Mikroorganismen gemessen. Das Wasser selbst wurde mit organischen Verschmutzungen belastet welche im Laufe der Zeit von Mikroorganismen abgebaut wurden. Während des Abbaus wurde Sauerstoff verbraucht. Die O_2-Anfangskonzentration sollte bei allen Gruppen gleich sein, wir können ja mal den Mittelwert berechnen:

```
# das Arithmetische Mittel:
mean(Rohdaten$`O2 [mg/L] davor`)
#> [1] 7.225
# und einen Boxplot:
boxplot(Rohdaten$`O2 [mg/L] davor`)
```



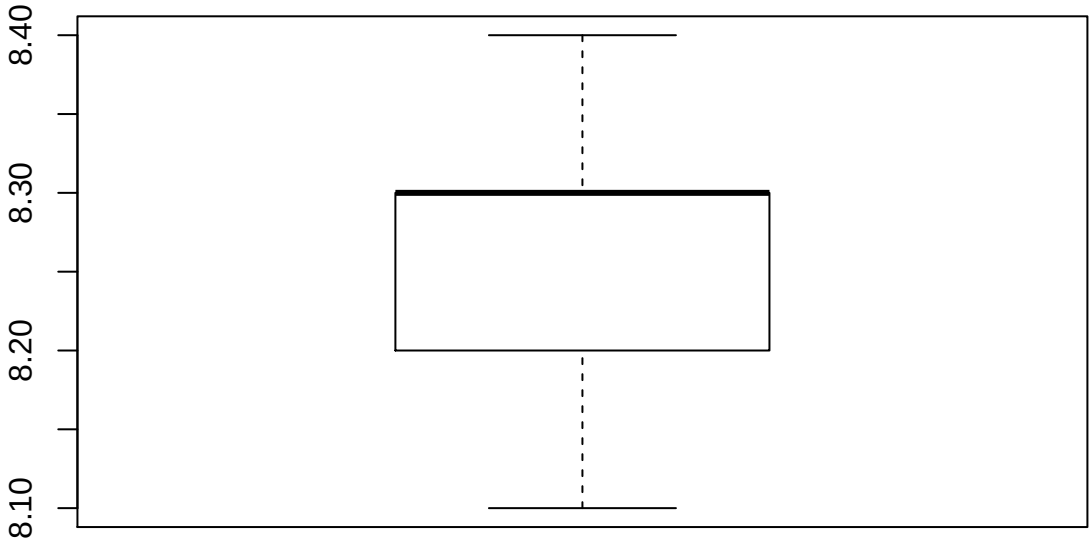
Bei genauerer Betrachtung fällt beim Boxplot ein Kreis auf Höhe der Null auf, es handelt sich um einen Datenpunkt welcher doch merkwürdig scheint. Wir möchten diesen Wert genauer untersuchen, dazu können wir ein Zusammenfassung erstellen:

```
summary(Rohdaten$`O2 [mg/L] davor`)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>   0.000   8.175   8.250   7.225   8.300   8.400
```

Wir sehen, dass das Minimum Null beträgt, eine recht unwahrscheinliches Ergebnis da im Versuch für alle sechs Durchläufe immer das Selbe Wasser aus dem selben Ansatz verwendet wurde. Da keine Information zum Messwert vorliegt, welche explizit auf einen Messfehler verweist, sollten nun Überlegungen angestellt werden, ob es legitim ist den Wert als Ausreißer zu verwerfen. Es gibt eine umfassendes Paket `outliers`:

```
library(outliers) # laden des Paketes

# der Dixon-Test:

dixon.test(Rohdaten$`O2 [mg/L] davor`)
```

```
#>
#>  Dixon test for outliers
#>
#> data:  Rohdaten$`O2 [mg/L] davor`
#> Q = 0.9759, p-value < 2.2e-16
#> alternative hypothesis: lowest value 0 is an outlier

# Grubbs-Test:

grubbs.test(Rohdaten$`O2 [mg/L] davor`)
#>
#>  Grubbs test for one outlier
#>
#> data:  Rohdaten$`O2 [mg/L] davor`
#> G = 2.47370000, U = 0.00095693, p-value = 1.096e-09
#> alternative hypothesis: lowest value 0 is an outlier

# Chi-im-quadrat-Test:

chisq.out.test(Rohdaten$`O2 [mg/L] davor`)
#>
#>  chi-squared test for outlier
#>
#> data:  Rohdaten$`O2 [mg/L] davor`
#> X-squared = 6.1191, p-value = 0.01337
#> alternative hypothesis: lowest value 0 is an outlier

# entfernen des Ausreißer:

rm.outlier(Rohdaten$`O2 [mg/L] davor`) %>%
    boxplot()
```



Die Ausreißer-Tests geben immer eine kleine Zusammenfassung und eine Begründung zu ihrer Wahl, das erleichtert den Umgang mit ihnen. Abhängig von der Fragestellung wird empfohlen sich mit der Arbeitsweise der Tests auseinander zu setzen, dann kann auch besser beurteilt werden, ob das gewählte Vorgehen zulässig ist.

Datentransformation

Zu Beginn kann es oft schwer sein sich innerhalb von Datensätzen zurecht zu finden und jene Werte, welche von Interesse scheinen zugänglich zu machen. Im obigen Beispiel hatten wir es leicht, es gab nur einen Ausreißer welcher sich leicht entfernen lies. Um mit dem Datensatz nun weiter arbeiten zu können, möchten wir den Ausreißer aber ganz entfernen. Das Paket `dplyr` bietet dazu die passenden Funktionen.

```
# Filtern der sinvollen Werte
library(dplyr) # laden des benötigten Paket
Daten <- filter(Rohdaten
                , Rohdaten$"O2 [mg/L] davor" > 1) %>%
    print()
#> # A tibble: 7 x 3
#>   Gruppe `O2 [mg/L] davor` `O2 [mg/L] danach`
#>    <dbl>             <dbl>               <dbl>
#> 1      1               8.3                 6.4
#> 2      2               8.2                 6.1
#> 3      4               8.2                 2.8
#> 4      5               8.3                 4.3
#> 5      6               8.4                 8.4
#> 6      7               8.1                 6.4
#> 7      8               8.3                 5.8
```

Wir sehen, dass wir nach der Datentransformation nur noch 7 Messungen in der Tabelle finden, wobei der Wert von Gruppe 3 fehlt, der Ausreißer Das ging super mit `filter()`, zu Erst steht in den Klammern der Name des Datensatz, welcher bearbeitet werden soll und dann ein logisches Argument welches vorgibt, was geschehen soll. In unserem Fall ein $ > 1$, also “Filter alle Werte, welche in der Spalte `O2 [mg/L] davor` einen Wert größer als eins aufweisen”.

Die logischen Operatoren können sein:

<, > größer oder kleiner

<=, >= größer gleich oder kleiner gleich

is.na() also ist ein NA, "nicht auswertbar", Werte die fehlen

! logische Verneinung

!is.na() ist KEIN NA

& und

| oder

%in% erkennt ein definiertes Element

Operatoren können auch Außerhalb von Funktionen verwendet werden:

```
Vektor.1 <- c(1, 2, 3, 4, 5)
# Wollen wir mal sehen, was %in% kann:

3 %in% Vektor.1
#> [1] TRUE

# man kann auch Vektoren in Vektoren suchen
c( 3, 4) %in% Vektor.1
#> [1] TRUE TRUE
```

```r
# Was passiert, wenn ein Element nich vorkommt?
c(0, 22, 3) %in% Vektor.1
#> [1] FALSE FALSE  TRUE
```

Mit `dplyr` besteht auch die Möglichkeit innerhalb des Datensatzes eine Funktion anzuwenden und mit ihr eine neue Spalte zu füllen, diese Funktion heißt `mutate`:

```r
# Wir berechnen die Differenz aus "davor" und "danach"
Daten <- mutate(Daten, Differenz = (`O2 [mg/L] davor` - `O2 [mg/L] danach` ))
print(Daten)
#> # A tibble: 7 x 4
#>   Gruppe `O2 [mg/L] davor` `O2 [mg/L] danach` Differenz
#>    <dbl>             <dbl>              <dbl>     <dbl>
#> 1      1               8.3                6.4      1.9
#> 2      2               8.2                6.1      2.10
#> 3      4               8.2                2.8      5.40
#> 4      5               8.3                4.3      4.
#> 5      6               8.4                8.4      0
#> 6      7               8.1                6.4      1.70
#> 7      8               8.3                5.8      2.5
```

Merkmale

Bei Studenten wird zur Qualitätsbewertung von Dozenten eigentlich immer nur zwischen “gut” oder “schlecht” unterscieden, so wie wir Farben als “grün” oder “rot” usw. bezeichnen, es gibt keine zwischengelagerten Zustände. Sollte eine solche Einteilung vorgenommen werden, so liegt eine Nominalskala vor. Das Gegenteil ist die Ordinalskala, sie kennt beliebig viele diskrete Zustände wie wir sie zum Beispiel am *pH*-Meter während einer Titration ablesen können. Alternativ kann man natürlich auch beim Dozente-Ranking Punkte und sogar Punkte mit Nachkommastellen vergeben. Abängig von diesen Merkmalen stehen uns verschiedene Möglichkeiten zur Verfügung, was im Anschluss mit unseren Messdaten geschieht und wie sie zu behandeln sind. Die Intervallskala hingegen erlaubt es die Differenzen auf der Ordinalskala (Intervalle) miteinzubeziehen, also eine Merkmalsausprägung. Zuletzt bleibt die Verhälnisskala, welche wie ihr Name verrät, Verhälnisse miteinbezieht und somit einen Vergleich unterschiedlicher Messungen erlaubt (*Covarianz* als Beispiel).

Klassen und die Klassenbreite

In der Biotechnologie ist meist bekannt welcher Klasse eine Stichprobe angehört, die Klassen sind dabei die übergeordneten Gruppen welche wiederum Merkmalswerte enthalten welche in den einzelnen Stichproben wiederzufinden sind. So als Beispiel eine Mehrfachbestimmung aus welcher dann eine Vielzahl technischer Replikate entsteht. Bei einer Absorptionsspektroskopischen Untersuchung zum Beispiel mehrfach Messungen mit gleichen Konzentrationen durchgeführt wobei immer Absorptionen gemessen werden, welche sich im Idealfall, kaum unterscheiden. Die Klasse ist somit schon im Voraus klar. Eine populäre Methode zur Berechnung der Klassenbreite b ist Sturges-Regel (Sturges 1926) unter Berücksichtigung des Stichproben-Umfangs n und der Spannweite v:

$$b = \frac{v}{1 + 3.32 \cdot \log n} \approx \frac{v}{5 \log n}$$

oder nach Scott (Scott 1979) unter Berücksichtigung der Standardabweichung σ :

$$b = \frac{3{,}49 \cdot \sigma}{\sqrt[3]{n}}$$

Mittelwert

Innerhalb einer solchen Klasse wird dann im eben beschriebenen Beispiel der Mittelwert errechnet. In R geschieht dies über die Funktion *mean()*. Eine Funktion kann aufgefasst werden als "Miniprogramm". Damit dieses Miniprogramm läuft braucht es Informationen, diese bekommt es vom Anwender. Diese Information ist, bezogen auf die Berechnung des arithmetischen Mittels, eine gewisse Anzahl von Werten, wobei für die Funktion *mean()* dabei schon alle wichtigen Information vorhanden sind, nämlich den Stichprobenumfang *n* sowie die gemessenen Werte x_i:

$$\overline{n} = \frac{1}{n} \sum x_i$$

Hierzu verwenden wir die Funktion *c()*, sie fügt die einzelnen Werte zu einem Vektor zusammen.

```
mean(c(1, 2, 3, 4))
#> [1] 2.5
# oder auch:
a <- c(1, 2.44, .56, 0.33, 1234)
mean(a)
#> [1] 247.666
```

Wichtig ist zu beachten dass die Zahlen durch ein Komma getrennt werden und Nachkommastellen durch eine Punkt! Der Pfeil (< −) dient als Zeichen zur Zuordnung eines Objektes, wie unserem Vektor zu einer Variablen, unser a. Eine Zuweisung in eine Andere Richtung ist unzulässig (− >) So können wir auch gleich eine Zuweisung des Mittelwert zu vornehmen:

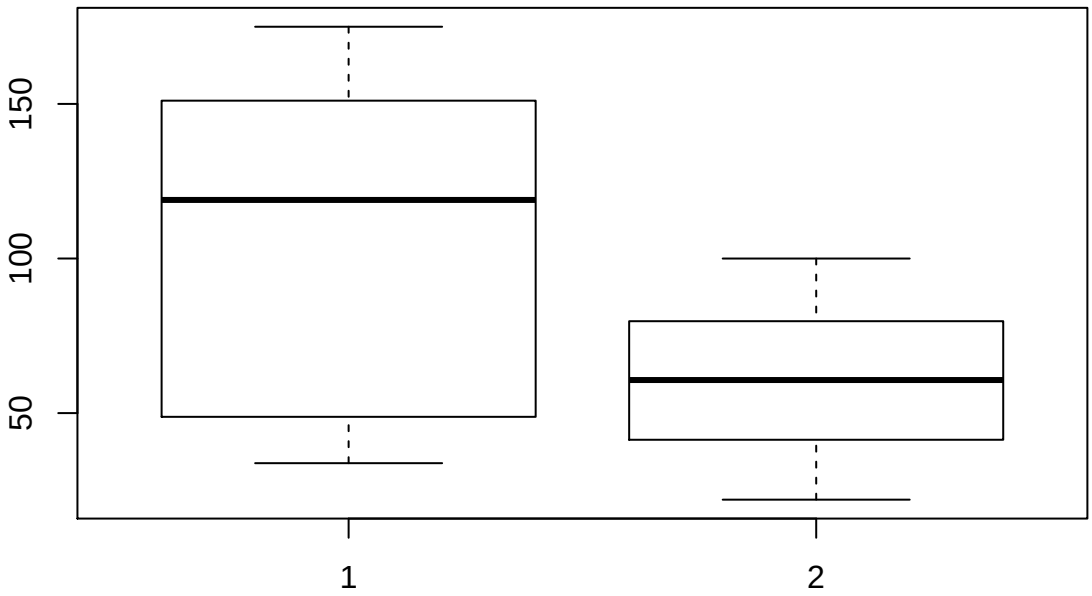
```
Mittelwert <- mean(a)
print(Mittelwert)
#> [1] 247.666
(Mittelwert)
#> [1] 247.666
```

Hier verwenden wir die Funktion *print()* um gleich den Mittelwert auf die Konsole gepromtet zu bekommen, das geschieht auch wenn wir die Variable *Mittelwert* umklammern. So auch wenn wir eine ganze Rechenoperation umklammern:

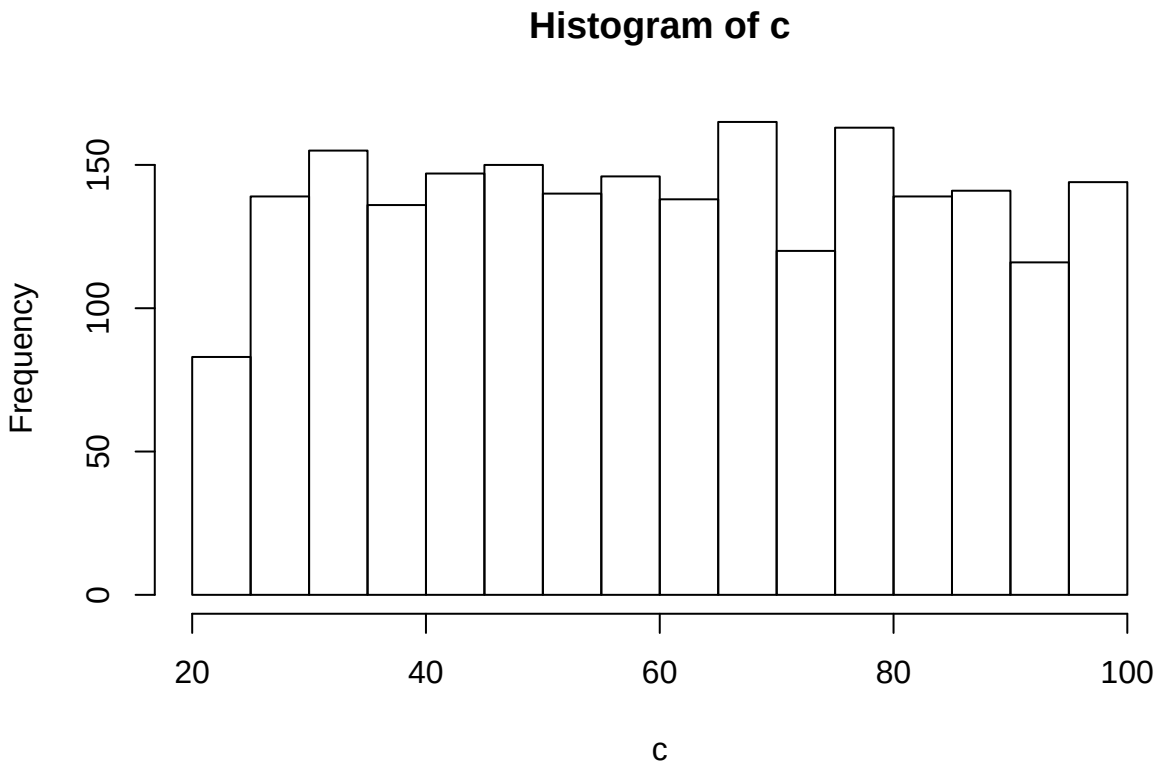
```
# Der Hashtag markiert Zeilen als Kommentar, alles was in der selben
# Zeile auf ihn folgt wird nicht mitinterpretiert
# runif verteitlt nun zehn Werte zufällig zwischen 20 und 200
(b <- runif(10, min = 20, max = 200))
#>  [1] 151.04713 106.92160 174.71970 137.01824  48.79186  33.80215  64.11797
#>  [8]  38.82527 174.95403 130.94529
(mean(b))
#> [1] 106.1143
b <- sort(b) # Überschreiben des alten b zu einem neuen sortierten b
```

Die bisher gezeigten Funktionen gehören zu den Paketen {base} und {stats} und sind, im Gebrauch von R.Studio immer vorhanden, anders als Pakete welche explizit geladen werden müssen. Auch zugänglich ist immer das Paket {graphics}:

```r
c <- runif(2222, 22, 100) # min und max kann man auch weglassen, ;-)
boxplot(b, c) # Ein einfacher Boxplot aus den Objekten a und b
```

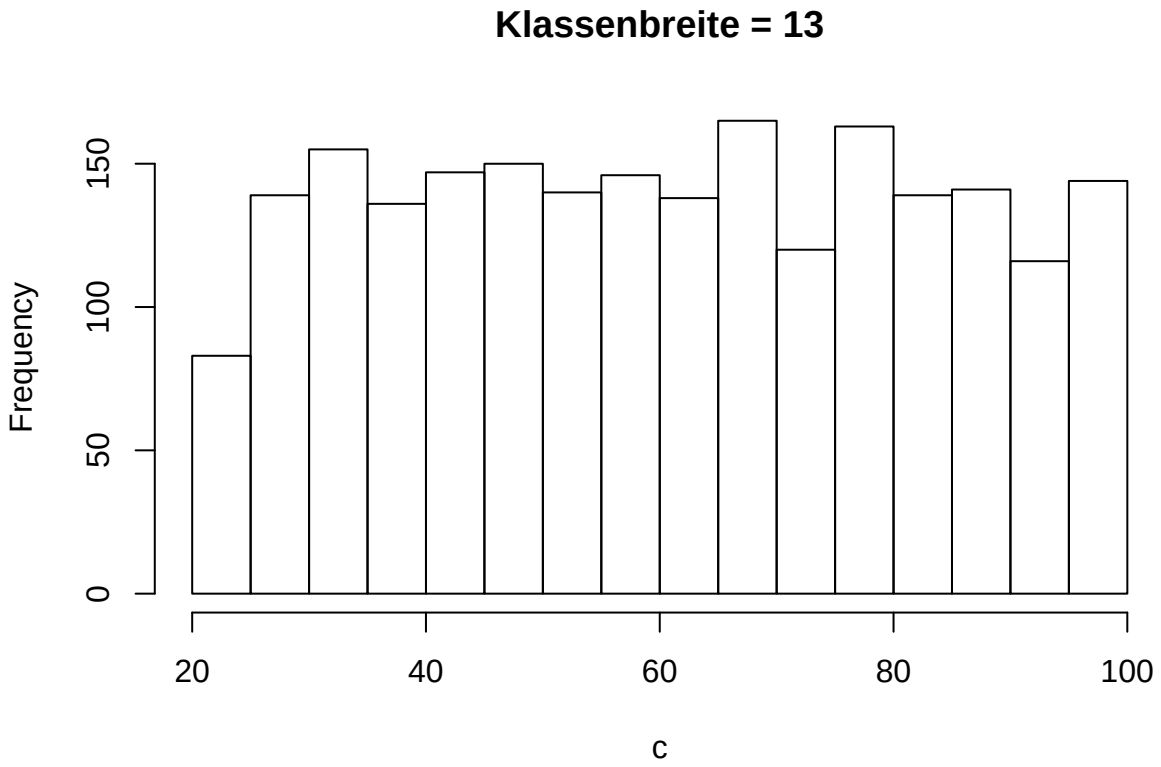



```r
hist(c) # Histogramm
```




```r
nclass.Sturges(c) # Berechnung der Klassenbreite nach Sturges
#> [1] 13
nclass.scott(c) # Berechnung der Klassenbreite nach Scott
#> [1] 14
hist(c,
```

```
     nclass = 13, # Wählen der Klassenbreite
     main = "Klassenbreite = 13" # einfügen einer Überschrift
     )
```



Das Paket {graphics} enthält eine umfassenden Sammlung an einfachen Plottingfunktionen, wer sich dafür interessiert ist unter https://www.rdocumentation.org/packages/graphics/versions/3.6.2 an der richtigen Stelle.

Steuerungsparameter

Um beurteilen zu können ob ein Experiment Informationen liefert, die im Kontext zur Durchführung sinnvoll scheinen, können wir untersuchen wie die einzelnen Messwerte mit einander korrelieren. Im Fall einer vierfach-Bestimmung ist das einfach nachvollziehbar:

```
mean( c(1,2,3,4) )
#> [1] 2.5
mean( c(2,2.5,3,2.5))
#> [1] 2.5
```

Die technischen Replikate der ersten Messung liegen weit auseinander aber haben das selbe arithmetische Mittel wie die der zweiten Messung welche aber doch recht ähnliche Ergebnisse liefert!

Um einen Wert für die Abweichung der Messpunkte einer Messung zu einander zu können wir die Summe der Abweichungsquadrate SQ ermitteln:

$$SQ := \sum_{i=1}^{n} (x_i - \overline{x})^2 = (n-1)s_x^2$$

Wobei nun immer nur ein einzelner Punkt betrachtet wird, für die Messung selbst ist die Varianz s_x^2 von Interesse, sie beschreibt die Summe aller *SQ*:

$$s_x^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

Die dazugehörige Funktion in R heißt *var()*:

```
var( c(1, 2, 3, 4))
#> [1] 1.666667
var( c(2,2.5,3,2.5))
#> [1] 0.1666667
var( rep(2.5, 4)) # rep() lässt einen Vektor aus vier mal 2,5 entstehen!
#> [1] 0
```

Die Standardabweichung $s = \sqrt{s_x^2}$ wiederum ist die positive Wurzel der Varianz und bekommt die den Namen *sd()*:

```
sqrt(var(c(1, 2, 3, 4))) # sqrt() ist die Quadratwurzel
#> [1] 1.290994
var(c(1, 2, 3, 4)) ^ (1/2) # Wobei es auch so geht
#> [1] 1.290994
sd( c(1, 2, 3, 4))
#> [1] 1.290994
sd( c(2,2.5,3,2.5))
#> [1] 0.4082483
sd( rep(2.5, 4))
#> [1] 0
```

> R eignet sich im Labor super als Taschenrechner-Ersatz, Alle mathematischen Funktionen sind vorhanden, die Rechnungen können auch leicht modifiziert werden und wenn man sie als Skript schreibt kann super nachvollzogen werden was gerechnet wurde! Dies hilft auch sich an die Sprache R zu gewöhnen, sollte aber kein allzu aufwendiger Prozess werden :-) (Ich hab's ja auch geschafft)

Möchten wir etwas mehr über unseren Mittelwert erfahren dividieren wir die Standard-Abweichung *s* durch die Wurzel des Stichprobenumfang, also den Standardfehler $s_{\bar{x}} = \frac{s}{\sqrt{n}}$. Wir haben nun alle Werkzeuge um die Funktion für den Standardfehler selbst zu definieren. Dazu verwenden wir den Befehl *function()*:

```
sde <- function(x) { # die Runde Klammer definiert das Argument der Funktion
  # Die geschweifte Klammer enthält die Definition der Funktion
  sd(x)/
    sqrt(length(x)) # length() ist die länge des Vektors von x
}
sde(c(1, 2, 3, 4))
#> [1] 0.6454972
sde(c(2,2.5,3,2.5))
#> [1] 0.2041241
sde(rep(2.5, 4))
#> [1] 0
```

Der Variationskoeffizient drückt prozentual die Abweichung der Standardabweichung zum arithmetischen Mittel aus $\left(cv = \frac{s}{\bar{x}} \cdot 100\%\right)$.

```r
cv <- function(x){
return( (sd(x)/mean(x))*100 ) # return() lässt, wie die doppelte Umklammerung das Ergebnis
}
cv(c(1, 2, 3, 4))
#> [1] 51.63978
cv(c(2,2.5,3,2.5))
#> [1] 16.32993
cv(rep(2.5, 4))
#> [1] 0
```

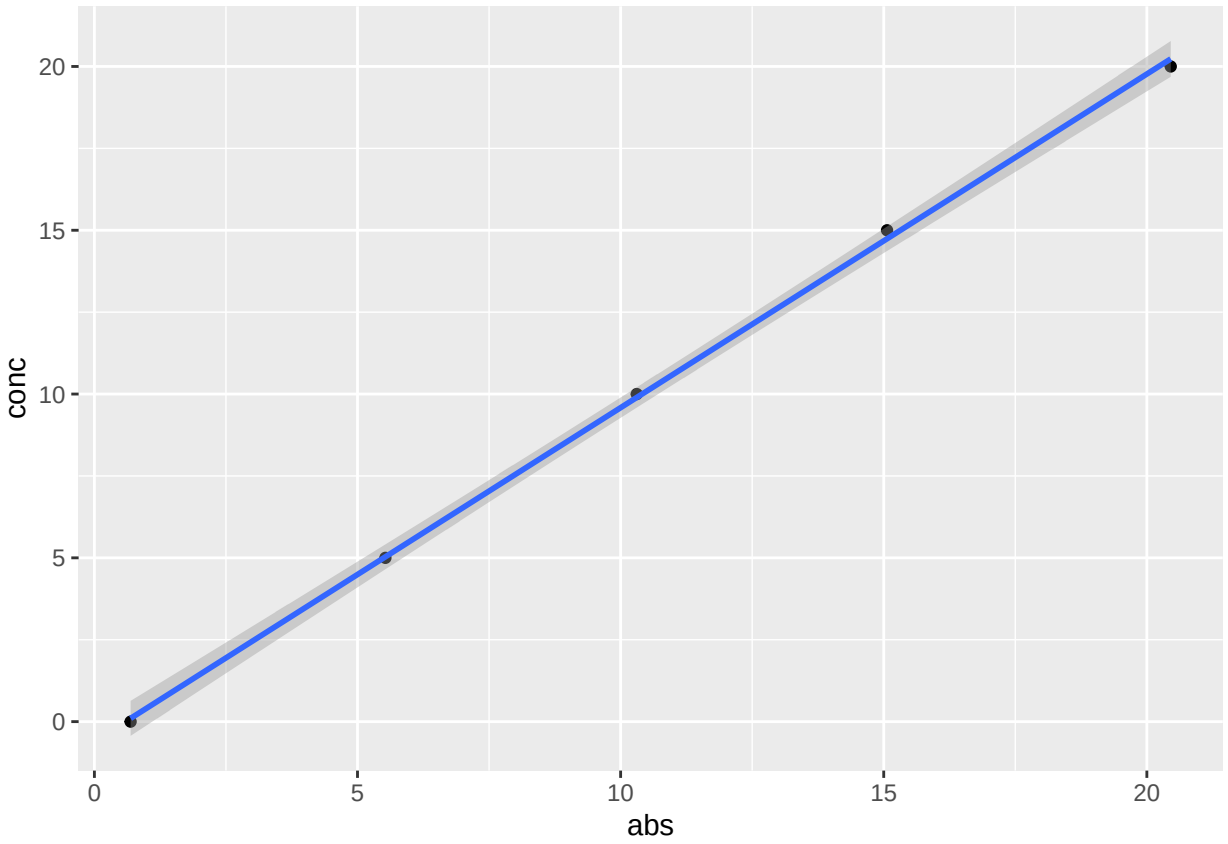
Der Variationskoeffizient lässt zu, dass wir verschiedene Stichproben mit unterschiedlichen Mitteln vergleichen:

```r
a <- c(500, 350, 400, 330, 370)
b <- c(50, 35, 40, 33, 37)
c <- c(234, 3, 44, 577, 9)
cv(a)
#> [1] 17.1047
cv(b)
#> [1] 17.1047
cv(c)
#> [1] 141.0778
```

Lineare Regression

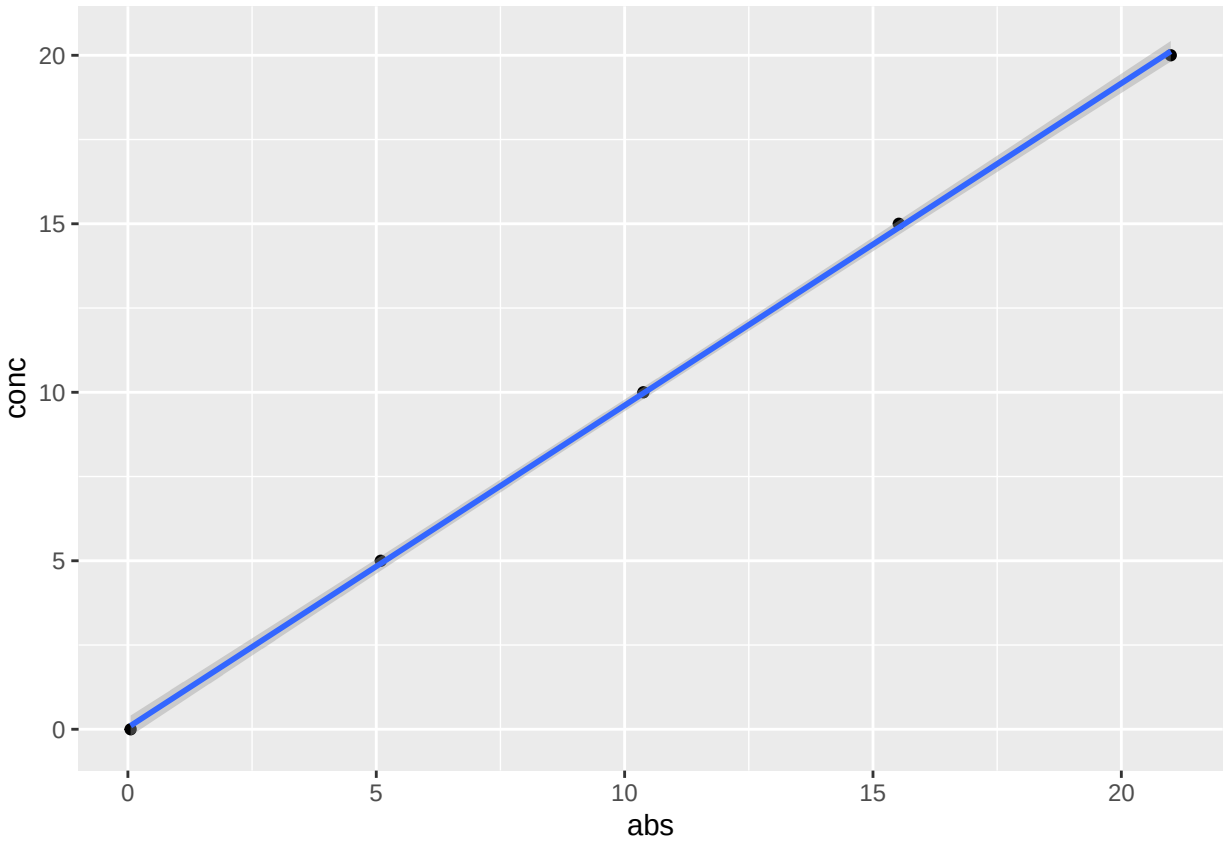
Die lineare Regression ist eine recht häufig verwendete Methode in der BT, als klassisches Beispiel ist die Konzentrationsbestimmung über eine Regrssion (“Eichgerade”, “Kalibrationsgerade” etc.) zu nennen. Das Paket OlFaBA enthält die Funktion *plot_regression()* um schnell eine solche Regressionsgerade zu plotten:

```r
conc <- seq(0, 20, 5)
abs <- runif(5, 0, 1) + conc # Simmulierte Absorptionsmessung
plot_regression(abs, conc)  # Plotten den Absorption gegen Konzentration
```



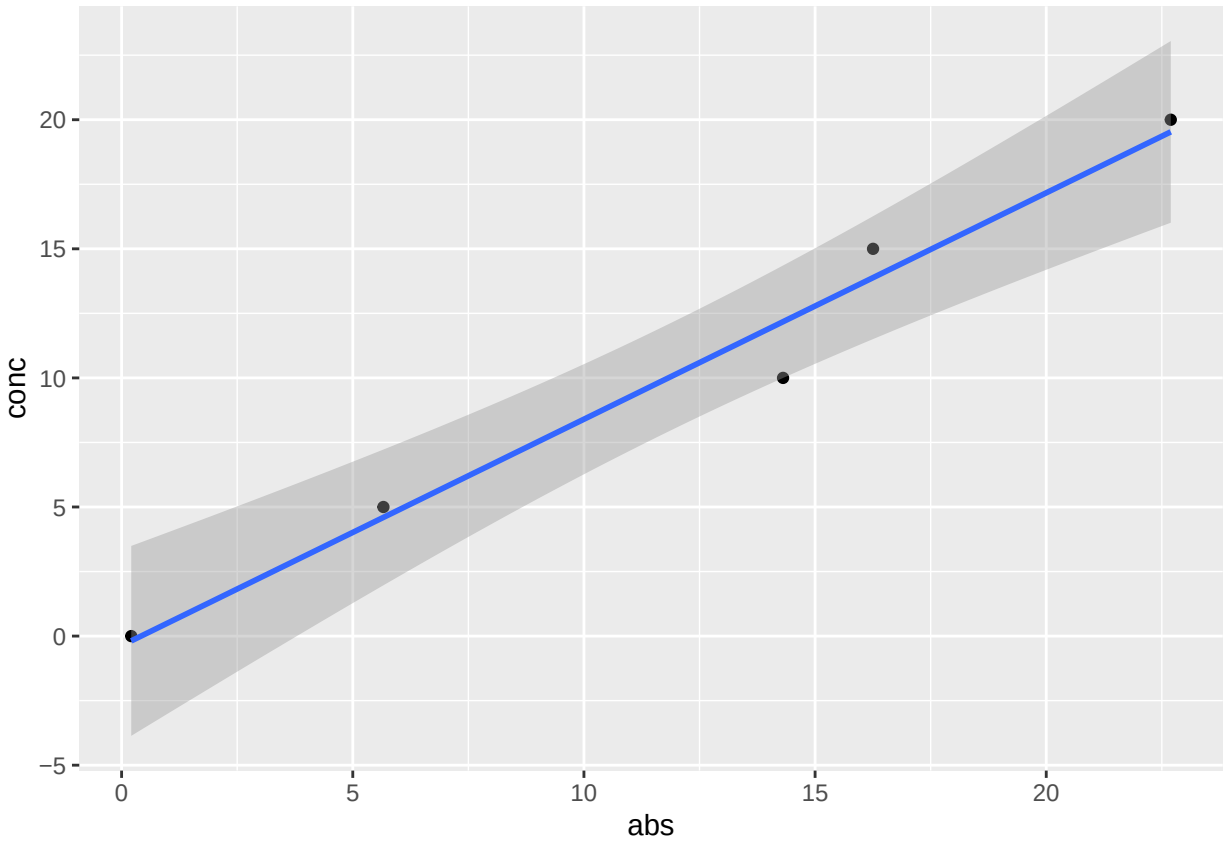
Dieser Plott zeigt uns, in grau hinterlegt die gleitenden durchnittliche Abweichung, auch ein Maß für die Zuverlässigkeit unserer Arbeitsweise, es könnt auch anders aussehen:

```
conc <- seq(0, 20, 5)
abs <- sort(runif(5, 0, 1)) + conc # Simmulierte Absorptionsmessung
plot_regression(abs, conc)  # Plotten den Absorption gegen Konzentration
```



Aber schlimmer geht ja bekanntlich immer:

```
# Erstellen einer Funktion:
LinMod_CEv <- function(abs, conc) {
  return(
  stats::lm(
    conc ~ abs
  )
)
  return(summary(LinMod_CEv))
}


conc <- seq(0, 20, 5)
abs <- runif(5, 0, 5) + conc # Simmulierte Absorptionsmessung
plot_regression(abs, conc)  # Plotten den Absorption gegen Konzentration
```




```
Modell <- LinMod_CEv(abs, conc)
summary(Modell)
#>
#> Call:
#> stats::lm(formula = conc ~ abs)
#>
#> Residuals:
#>       1       2       3       4       5
#>  0.1861  0.4037 -2.1764  1.1167  0.4698
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept) -0.36896    1.16991  -0.315  0.77314
#> abs          0.87686    0.08207  10.684  0.00175 **
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 1.461 on 3 degrees of freedom
#> Multiple R-squared:  0.9744, Adjusted R-squared:  0.9659
#> F-statistic: 114.2 on 1 and 3 DF,  p-value: 0.001753
```

Das R^2 gibt uns bekanntlich die Genauigkeit unserer Regressionsgerade an, es wird als Bestimmtheitsmaß bezeichnet. Alternativ wird im Laborjargon häufig auch der Begriff “Quadratfehler” verwendet. Um an diese Information zu kommen verwenden wir hierbei explizit eine Funktion aus dem Paket {base}, nämlich *summary()*, diese Funktion liefert uns eine Menge an Information, die wirklich relevanten sind hier zusammengestellt:

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	-0.369	1.17	-0.3154	0.7731
abs	0.8769	0.08207	10.68	0.001753

Table 2: Fitting linear model: conc ~ abs

Observations	Residual Std. Error	R^2	Adjusted R^2
5	1.461	0.9744	0.9659

Intercept ist der Schnittpunkt mit der *y*-Achse also *b* und der Steigung *m* als **conc** für $y = mx + b$. Damit kann jetzt auch weitergearbeitet werden, nun kann die Konzentration einer unbekannten Probe über ihre Absorption bestimmt werden. Dafür steht in {OlFaBA} die Funktion *conc_eval()* zur Verfügung, bevor wir diese jedoch verwenden muss *LinMod_CEv()* für die Standardreihe durchgeführt worden sein, im Beispiel mit realen Messdaten:

```
conc <- seq(0, 100, length.out = 6) # Ein Vektor der Länge 6, die Konzentrationen den Stan
abs <- c(.374, .5, .617, .78, .874, .985) # Die Real gemessenen Konzentrationen
conc_eval(abs_P = abs, abs_std = abs, conc_std = conc) #Berechnung der Konzentrationen der
#>
#> Call:
#> stats::lm(formula = conc_std ~ abs_std)
#>
#> Residuals:
#>       1       2       3       4       5       6
#>  0.4796  0.2449  1.4556 -4.7210  0.1833  2.3576
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)  -60.541      3.813  -15.88 9.19e-05 ***
#> abs_std      160.592      5.293   30.34 7.03e-06 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 2.752 on 4 degrees of freedom
#> Multiple R-squared:  0.9957, Adjusted R-squared:  0.9946
#> F-statistic: 920.5 on 1 and 4 DF,  p-value: 7.031e-06
#>
#> [1] -0.4795571 19.7550904 38.5444059 64.7209737 79.8166631 97.6424240
```

Teil der Funktion conc_eval() ist es, die errechneten Daten in einer Tabelle darzustellen, diese kann besonders gut bei der Verwendung von R Markdown zur Geltung kommen.

```
#>
#> Call:
#> stats::lm(formula = conc_std ~ abs_std)
#>
#> Residuals:
#>       1       2       3       4       5       6
#>  0.4796  0.2449  1.4556 -4.7210  0.1833  2.3576
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
```

```
#> (Intercept)  -60.541      3.813  -15.88 9.19e-05 ***
#> abs_std      160.592      5.293   30.34 7.03e-06 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 2.752 on 4 degrees of freedom
#> Multiple R-squared:  0.9957, Adjusted R-squared:  0.9946
#> F-statistic: 920.5 on 1 and 4 DF,  p-value: 7.031e-06
#>
#> [1] -0.4795571 19.7550904 38.5444059 64.7209737 79.8166631 97.6424240
```

Absorption	Ist-Konzentration	Berechnete Konzentration
0.374	0	-0.4796
0.5	20	19.76
0.617	40	38.54
0.78	60	64.72
0.874	80	79.82
0.985	100	97.64

Aufgaben

Bereche für eine realtive Absorption von $A_{595} = 0.33$ die Konzentration unter Berücksichtigung der Standardreihe:

Absoption (bei 595 nm)	Konzentration in µg/ml
0.13	0
0.22	28.57
0.3	57.14
0.37	85.71
0.44	114.3
0.49	142.9
0.54	171.4
0.57	200

Nicht-lineare Regressionen

Nicht-lineare Regression für die Michaelis-Menten-Funktion

Die Michaelis-Menten Gleichung kann mathematisch beschrieben werden als:

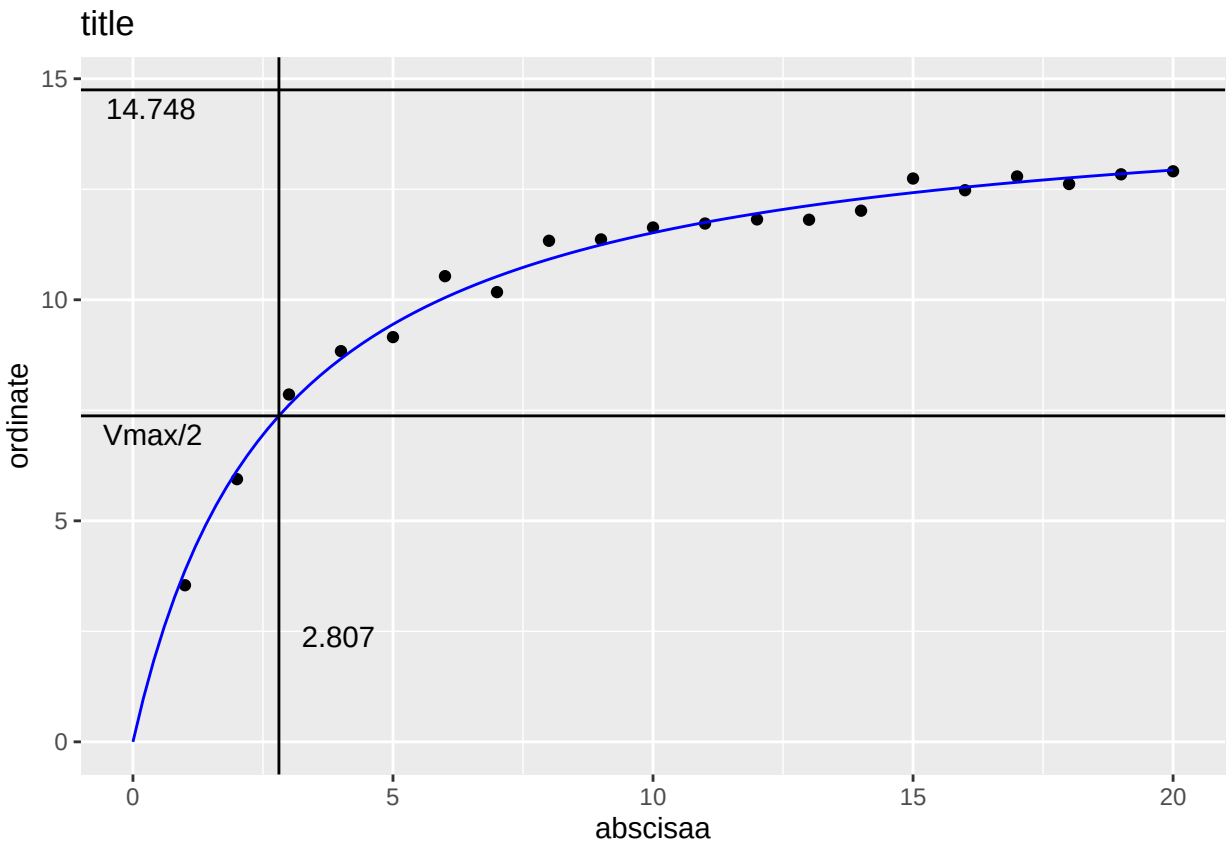
$$v_0 = \frac{v_{max} \cdot [S]}{K_m + [S]}$$

Wir können uns als nächstes wieder Test-Daten simulieren:

```r
sub <-seq(1,20,1)
velo <-(
  (runif(1,14.7,15)*sub
  )/( # Könnt ihr die MM-Kinetik wiedererkennen? ;-)
    runif(1,2.5,3)+sub))+rnorm(20,0,.3)
```

Im nächsten Schritt soll ein Fitting für die Substratvariation gegen die Enzymaktivität vorgenommen werden:

```
plot_MM_direct(sub, velo)
#> Nonlinear regression model
#>   model: velo ~ SSmicmen(sub, Vm, K)
#>    data: parent.frame()
#>     Vm      K
#> 14.748  2.807
#>  residual sum-of-squares: 1.22
#>
#> Number of iterations to convergence: 0
#> Achieved convergence tolerance: 9.731e-07
```



Dose-Response-Modelle

Zu erst sollten wir uns noch ein mal an die Logit-Funkion erinnern. Die Logit-Funktion ist der Logarithmus einer Chance $\left(\frac{p}{1-p}\right)$, mit p einer Wahrscheinlichkeit:

$$\begin{aligned} L &= \text{Logit}(p) = \ln \tfrac{p}{1-p} \\ &= 2 \cdot \text{artanh}(2 \cdot p - 1) \end{aligned} \tag{1}$$

und deren Umkehrung:

$$\begin{aligned} p &= \tfrac{e^L}{1+e^L} = \tfrac{1}{1+e^{-L}} \\ &= \tfrac{1}{2} \cdot \left(1 + \tanh \tfrac{L}{2}\right) \end{aligned} \tag{2}$$

Die 4-Parameter-logistische Funktion kann auch Logit-transformiert werden:

Dise Funktion wurde zu erst von Robertson [**?**] 1908 in dieser Form als Modell für Biochemische Prozesse vorgeschlagen:

$$\frac{dg}{dz} = \frac{\theta_3}{\theta_1} g(z)(\theta_1 - g(z)) \tag{3}$$

mit:

z einer Konzentration

$g(z)$ der Antwort des Systems auf die Konzentration

$\theta_1\, \theta_3 \in \mathbb{R}$ zusätzliche Parameter

Die Lösung der Gleichung führt zu:

$$g(z|\theta_1, \theta_2, \theta_3) = \theta_1 - \frac{\theta_1}{1 + \left(\frac{z}{\theta_2}\right)^{\theta_3}} \tag{4}$$

Wobei nun noch der vierte Parameter $\theta_4 \in \mathbb{R}$ eingeführt wird was uns zur 4PLM (4-Parameter logistische Modell):

$$f(x|\theta) = \theta_1 + \frac{\theta_4 - \theta 1}{1 + \left(\frac{z(x)}{\theta_2}\right)^{\theta_3}} = \theta_1 + \frac{\theta_4 - \theta 1}{1 + 10^{\theta_3(x - \log \theta_2)}} \tag{5}$$

Nun wird eher ersichtlich was die einzelnen Parameter zu bedeuten haben, θ_1 ist die obere Asymptote und θ_4 die untere Asymptote. θ_2 ist der Wendepunkt und θ_3 mit der Steigung assoziert. Für $\theta_3 < 0$ steigt die Funktion, für $\theta_3 > 0$ sinkt sie. Unter Verwendung des Paketes `nl4pl()` wird ein Modell entsprechend 5 auf einen Datensatz $\{(x_i, y_i)|i = 1, 2, \ldots, n\}$ mit x_i als dekadischen Logarithmus von z der i-ten Messung und y_i der gemessenen Reaktion darauf, gefittet. Die beim Fitting angewandte Verlustfunktion:

$$\mathcal{L}(\theta) = \frac{1}{n} \sum_{i=1}^{n} (y_i - f(x_i|\theta))^2 \tag{6}$$

Gewöhnlich ist die Funktion durch die Parameter θ nichtlinear, deswegen ist die Verringerung von $\mathcal{L}$ zur Konvergenz ein Problem der nichtlinearen Regression. Um den in 5 enthaltenen Term $(x - \log \theta)$ zu linearisieren kann er in die Hill-Gleichung überführt werden:

$$\log\left(\frac{f(x|\theta) - \theta_4}{\theta_1 - f(x|\theta)}\right) = \theta_3 x - \theta_3 \log(\theta_2) \tag{7}$$

Die Logit-Transformation links der durchschnittlischen Anwort des Modells ist äquivalent zur linearen Funktion des $\log x$. Es erfolgt eine Ersetzung der Werte für die Asymptoten θ_4, θ_1:

$$\log\left(\frac{y_i - y_{min}}{y_{max} - y_i}\right) \approx -\theta_3 \log(\theta_2) + \theta_3 x_i \tag{8}$$

Die Logit-Transformation links der durchschnittlischen Anwort des Modells ist äquivalent zur linearen Funktion des $\log x$. Der Rechte Teil aus 8 kann in ein lineares Regressionsproblem überführt werden, das erlaubt uns die Einführung von Grenzen für $\theta_3 \& \theta_2$:

$$\log\left(\frac{y_i - y_{min}}{y_{max} - y_i}\right) = \beta_2 + \beta_3 x_i + \varepsilon_i \tag{9}$$

mit:

$\beta_2 = -\theta_3 \log(\theta_2)$

$\beta_3 = \theta_3$

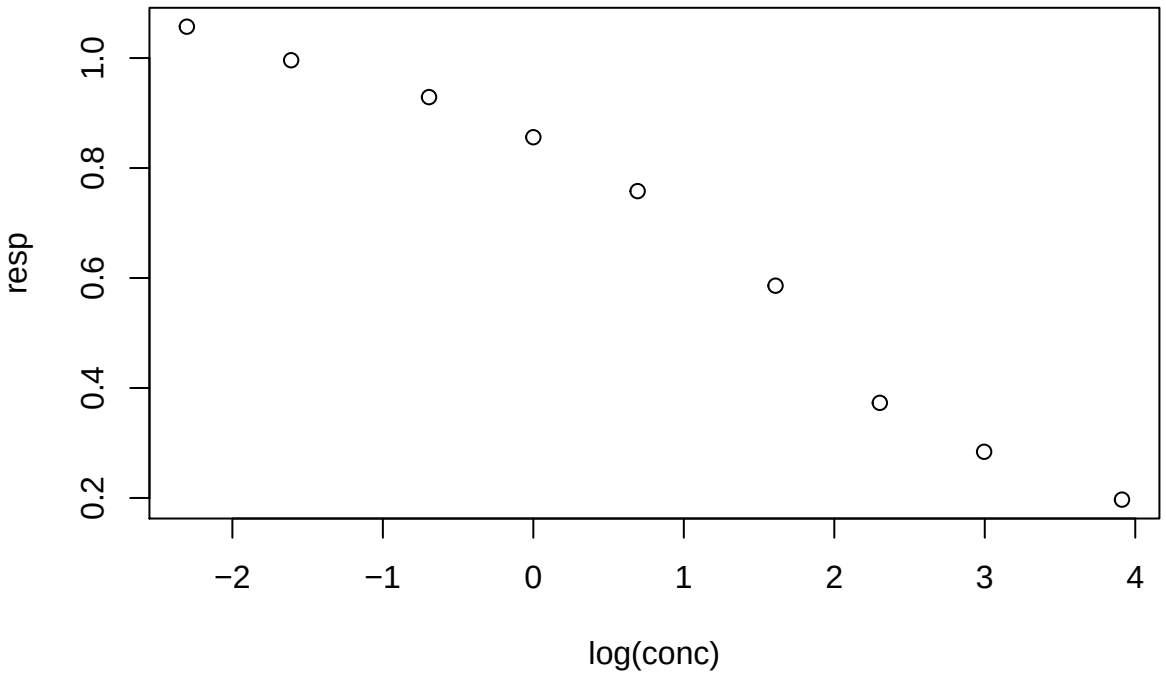
$\varepsilon_i \sim \mathcal{N}(O, \sigma^2)$

Diese Grenzen sind charakteristisch für verschiedene Sachverhalte, einige Beispiele:

- kompetitive Immunoessays
- LD-50 Bestimmung
- Ermittlung der wahrnehmbaren Konzentration eines Riechstoffes in der Umgebungsluft

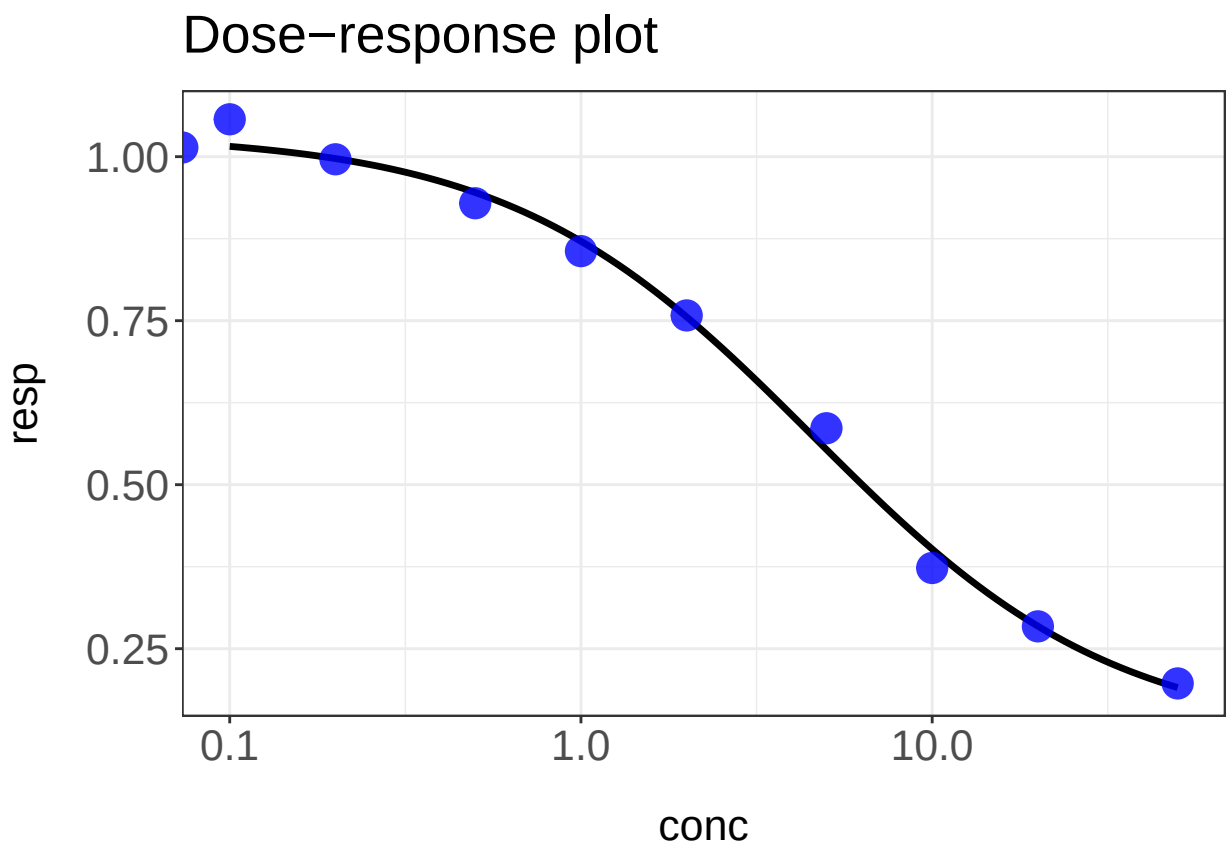
Anhand von Daten eines kompetitiven ELISA's aus dem IC-Praktikum wirst du eine Anwendung sehen:

```
conc <- c(0, .1, .2, .5, 1, 2, 5, 10, 20, 50) # Die Konzentration
resp <- c(1.014, 1.057, .996, .929, .856, .758, .586, .373, .284, .197) # die gemessene Ab
plot(resp~log(conc)) # Ein einfacher Plot der Daten
```




```
dose_response_plot(conc, resp) # das eigentliche Modell
#> $convergence
#> [1] TRUE
#>
#> $data
#>    Dose Response
#> 1   0.0    1.014
#> 2   0.1    1.057
#> 3   0.2    0.996
#> 4   0.5    0.929
#> 5   1.0    0.856
#> 6   2.0    0.758
#> 7   5.0    0.586
#> 8  10.0    0.373
#> 9  20.0    0.284
#> 10 50.0    0.197
```

```
#>
#> $hessian
#>            UpperLimit    Log(IC50)        Slope LowerLimit
#> UpperLimit  1.0438577  0.266445465 -0.101915799 0.23883920
#> Log(IC50)   0.2664455  0.193414412 -0.000518044 0.24363225
#> Slope      -0.1019158 -0.000518044  0.042544346 0.06563857
#> LowerLimit  0.2388392  0.243632254  0.065638569 0.47846390
#>
#> $loss.value
#> [1] 0.0004567097
#>
#> $method.robust
#> [1] "squared"
#>
#> $parameters
#> UpperLimit       IC50      Slope LowerLimit
#>  1.0348369  4.5580193 -1.0080644  0.1151731
#>
#> $sample.size
#> [1] 10
#>
#> $call
#> dr4pl.formula(formula = resp ~ conc)
#>
#> $formula
#> resp ~ conc
#> <environment: 0x55bea89ad8f8>
#>
#> attr(,"class")
#> [1] "dr4pl"
#> Warning: Transformation introduced infinite values in continuous x-axis

#> Warning: Transformation introduced infinite values in continuous x-axis
```



R Core Team. 2018. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. https://www.R-project.org/.

Scott, David W. 1979. “On optimal and data-based histograms.” *Biometrika* 66 (3): 605–10. https://doi.org/10.1093/biomet/66.3.605.

Sturges, Herbert A. 1926. “The Choice of a Class Interval.” *Journal of the American Statistical Association* 21 (153). Taylor & Francis: 65–66. https://doi.org/10.1080/01621459.1926.10502161.

Wickham, Hadley, and Garrett Grolemund. 2017. *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. 1st ed. O’Reilly Media, Inc.