Basics und mehr

Schmidt, Sophie C.
22 Mai, 2019

Das hier ist ein Code Chunk, der am Anfang immer als setup steht. Hier gebe ich vor, dass aller Code, den ich in Chunks schreibe, nicht nur umgewandelt werden soll, sondern auch abgedruckt werden soll (echo = TRUE).

```
knitr::opts_chunk$set(
  collapse = TRUE,
  warning = FALSE,
  message = FALSE,
  echo = TRUE,
  comment = "#>",
  fig.path = "../figures/",
  fig.width=2,
  fig.height=2
)
```

R Markdown basics

Hier das gewünschte recap der basics, mit denen wir ursprünglich begonnen hatten, später geht es dann um den Umgang mit Daten, Liniendiagramme und Dichte-Darstellungen, sowie tolle Möglichkeiten *noch mehr Variablen gleichzeitig* darzustellen.

Wir beginnen noch einmal mit rrtools, denn inzwischen sollte das wieder gehen. rrtools ist von Ben Marwick et al entwickelt worden, um es einfach zu machen ein R-Paket zu erstellen, in dem man alle seine Daten, seine Analyse und sein “paper” zusammen ablegt. Dieses Paket kann man dann auch einfach anderen Menschen zur Verfügung stellen, damit diese die eigenen Analysen nachvollziehen können.

Das Bereitstellen von Code und Daten (open access) ist wichtig für die Wissenschaftlichkeit einer Analyse, da sie so reproduzierbar wird (wenn jemand weder Code noch Daten hergibt, kann niemand jemals überprüfen, ob er/sie sich nicht doch verrechnet hat), aber auch für die Replizierbarkeit. Replizieren heißt, ich nehme die Methode der anderen und wende sie auf meine eigenen Daten an. Viele Artikel sind nicht sonderlich replizierbar, weil die Methode kaum im Detail beschrieben wird. Das ist frustrierend, wenn man die Methode und das Ergebnistoll fand und selber ausprobieren möchte.

Deswegen ist Code teilen toll und wichtig.

Also. Wir installieren rrtools. Weitere Infos dazu gibt es hier:
https://github.com/benmarwick/rrtools

```
devtools::install_github("benmarwick/rrtools") #devtools:: ruft das Paket devtools.
```

```
ich brauch es nur einmal, deswegen habe ich es nicht mit library ins Boot geholt.

library(rrtools) # wir laden das Paket und alles was zum Thema git kommt, ignorieren wir.
# Wer Lust hat git kennen zu lernen, findet hier ein paar hilfreiche Links zu dem Thema: http://archaeoinformatics.net/git-how-not-to-learn-it/

#rrtools::use_compendium("C:/Hier/kommt/ein/Pfadname/pkgname") # wir definieren wo unser Paket liegt und wie es heißen wird
```

Jetzt muss sich einfach so ein neues Projekt / Paket in Rstudio geöffnet haben. Oben rechts seht ihr den Namen eures Pakets.

Wenn ihr jetzt rechts unter “Files” schaut, seht ihr eine DESCRIPTION-Datei. Hier kann man jetzt ein paar Metadaten zu dem Paket editieren.

Auf jeden Fall sollte man die Autoreninformation ändern und einen Satz dazu schreiben, was das Paket soll (Title). Sollte man seinen Code tatsächlich eines Tages veröffentlichen wollen, ist außerdem unbedingt über eine Lizenz nachzudenken. Lizenzen regeln, wer wie den Code weiterbenutzen darf und wie ihr als Autoren zitiert werden möchtet. Neben Encoding: empfehle ich UTF-8 zu schreiben, falls da was anderes stehen sollte.

Wenn wir jetzt mit rrtools weiter arbeiten wollen, müssen wir es noch einmal laden, weil wir in einem neuen Projekt unterwegs sind als ganz am Anfang. Deswegen:

```
#library(rrtools)
#use_analysis()
```

Dinge passieren! Netterweise erklärt rrtools in der Konsole, was es gerade getan hat. Wir haben jetzt eine sinnvolle Ordnerstruktur und schon das erste Rmd-Dokument. Was insgesamt also entstanden ist, ist ein “ganz normaler” Ordner, der den Namen eures Pakets trägt, in dem weitere Ordner angelegt wurden und eine Vorlage für ein Paper.

Bitte arbeitet für eure Analysen der eigenen Daten in einem eigenen Projekt nur dafür.

Ein Markdown-Dokument ist eigentlich nur ein txt-Dokument, also das simpelste an Textdatei was man sich vorstellen kann. In dem ich in dieser Textdatei einer bestimmten Syntax folge, kann ich die Datei von einem Programm im Hintergrund (namens pandoc) in hübsch formatierte Word-, html- oder PDF-Dateien umwandeln. Details findet ihr hier: http://rmarkdown.rstudio.com.

Es gibt oben den **Knit**-Knopf (englisch für stricken und das Symbol ist ein Wollknäul mit Stricknadeln), der das Dokument umwandelt. Wer faul ist wie ich, benutzt den Shortcut Strg+Umschalt+k.

Code chunks sind die Bereiche, in denen ich Code unterbringen kann. Man erstellt am einfachsten einen mit dem Shortcut Strg+Alt+i.

Sie folgen einer bestimmten Logik und werden durch drei Hochkommas (``` = die Art, die man auf der deutschen Tastatur neben dem Back-Pfeil findet) und einer öffnenden geschweiften Klammer mit einem r darin geöffnet. Daneben kann ich den Code Chunk

benennen. Kann voll praktisch sein, wenn ich später mal darauf referenzieren möchte. Dahinter kann man noch Parameter für den Code Chunk setzen, z. B. ob der Code mit als Code in das Word-Dokument übertragen werden werden soll (echo = TRUE) oder nicht (echo = FALSE) oder falls ein Diagramm erstellt wird, wie groß es sein soll. Diese Parameter werden mit Komma voneinander getrennt, weitere findet ihr unter: https://rmarkdown.rstudio.com/lesson-3.html. Man kann sie aber auch händisch einstellen, wenn man auf das kleine Zahnradsymbol klickt. Nach diesen Parametern folgt jedoch eine schließende geschweifte Klammer und der Bereich für Code beginnt. Er ist immer etwas dunkler hinterlegt. In diese erste Zeile sollte nichts weiter hinter die geschweifte Klammer kommen.

Der Code Chunk muss aber auch beendet werden. Dafür braucht man wieder die drei Hochkommas alleine in einer Zeile.

```
# das hier ist ein Code Chunk. Ich sollte also nur Code hinein schreiben
#alles was nicht Code ist, muss auskommentiert werden, sonst will pandoc es gerne als Code behandeln und fängt an zu weinen, weil es sehr fehlerhafter Code wäre
```

Außerhalb des Code Chunks weiß pandoc jedoch nicht, dass was ich schreibe, Code sein soll.

library(archdata) bleibt also komplett wirkungslos. Es ist nur Text.

Deswegen also noch einmal, diesmal im Code Chunk:

```
library(archdata)
library(ggplot2)
data("BACups")
data("Snodgrass")
data("Bornholm")
#install.packages("tidyr")
#install.packages("dplyr")
#brauchen wir später, drum installiert es doch gleich am Anfang.
library(tidyr)
library(dplyr)
```

Die Informationen, die jetzt aufploppen “Attaching package”, “objects are masked” etc brauchen uns nicht zu interessieren. Hier geht es um Paketabhängigkeiten und -überlagerungen.

Was man aber machen kann, ist Variablen, die man erstellt hat, im Text einbetten. Man brauch ein Hochkomma ` r “Variablenname” und wieder ein Hochkomma ´ einfach mitten im Fließtext. Weiter unten kommt ein Beispiel.

Den Text kann man formatieren. zB lassen sich Überschriften mit dem Rautensymbol markieren, Listen mit einem einfach Minus-Symbol und Text durch Unterstriche *kursiv* und durch zwei Sternchensymbole **fett** drucken.

Man kann in Markdown mit einem Literaturverwaltungsprogramm zusammenarbeiten und sich automatisch Literaturverzeichnisse generieren lassen, man kann Bilder einfügen, die keine R-Diagramme sind, man kann eigentlich alles was man braucht. Sogar Präsentationen, Poster und Websiten lassen sich so generieren.

Jetzt noch ein paar weitere *R-basics*:

Grundlegende Zeichen:

- <- mit einem kleinen Pfeil weisen wir einen Wert / eine Berechnunge / einen Vektor oder sogar einen Dataframe einer Variablen zu. Wie im Mathe-Unterricht, wo Formeln mit f(x) und y = a*x + b beschrieben wurden. R merkt sich NICHTS, es sei denn, ich weise es einer Variablen zu. Das heißt auch, wenn ich einen Befehl / eine Formel auf einen Datensatz anwende, bleibt das nur langfristig bestehen, wenn ich mit dem Befehl gleichzeitig entweder meinen alten Datensatz überschreibe ODER einen neuen entstehen lasse.

- $ das Dollarzeichen steht zwischen Dataframe und dem Vector im Dataframe: df$vector, damit wählen wir also den Vector an.

- Klammern ( ) sind grundlegend, weil ich bei jeder Funktion, die ich aufrufe erst den Funktionsnamen / den Befehl schreibe und dahinter in Klammern, worauf er sich bezieht, also auf welche Daten der Befehl angewandt werden soll.

- in anderen Kontexten umgeben Klammern ( ) immer eine Sinneinheit. D. h. bei der Erstellung eines Diagramms, dass die Informationen, die zu aes gehören in Klammern hinter aes geschrieben werden, sie dürfen nicht hinter die schließende Klammer rutschen. Passiert manchmal.

- Eckige Klammern []: Über sie haben wir noch nicht geredet. Sie sind spannend, weil man mit ihnen Zeilen, Spalten und Felder eines Dataframes anwählen kann. Ein kleines Beispiel:

Will ich in dem Datensatz BACups zB die allererste Information (1. Zeile, 1. Spalte, was steht da?) herausholen, geht das so:

```
BACups_1_1 <-BACups[1,1]
# Es ist genau ein Wert der Variable BACups_1_1 zugewiesen worden. Sie ist rechts unter "Values" zu finden
```

Ich kann aber auch die gesamte erste Zeile auslesen:

```
BACups_1 <-BACups[1,]
# BACups_1 ist rechts unter "data", weil es sich um einen Vector handelt. 1 observation, 6 variables steht daneben.
```

Natürlich lassen sich auch Spalten auswählen

```
BACups_x_1 <-BACups[,1]
# es sollte geklappt haben
```

Folgerichtig kann man sich merken: In der eckigen Klammer hinter dem Datensatz kann man mit der ersten Zahl die Zeile bestimmen und mit der zweiten Zahl hinter einem Komma die Spalte. Gerade Spalten haben häufig Namen, die kann man für die Auswahl auch nutzen. Aber dazu kommt später noch ein Beispiel.

Negativauswahl gibt es natürlich auch. Also: Gib mir alles außer diese Spalte:

```
BACups_vieles <- BACups[, -2]
# alles außer Spalte 2 ist jetzt dem neuen Datensatz BACups_vieles zugewiesen worden
```

Ganz toll ist auch die Auswahlmöglichkeit “von a bis x”. Das geht mit Doppelpunkt:

```
BACups_x <-BACups[c(10:20),]
# Schaut euch an, was entstanden ist.
```

Das wird jetzt richtig tricky: Ich muss R fast immer, wenn ich eine Reihe von Werten hintereinander angebe, die zusammengehören, dem Programm sagen, dass sie das tun. Dafür benutzt man das “c()” in der Klammer sind dann die Angaben zusammengefasst. Mit Doppelpunkt sage ich dann vom 10. bis zum 20. Wert hätte ich gern die Zeilen ausgewählt (weil das die Zahlen VOR dem Komma sind).

Alles klar soweit?

Was ich also hier auch machen kann, ist Werte in den Rmd-Text einbetten. Der Median von der Spalte RD ist 12.05. Geschrieben habe ich nicht die Zahl sondern: ` r median(BACups$RD)` und das Programm hat es sozusagen “on the fly” im Text für mich ausgerechnet.

Das ist suuuuper praktisch, weil man, wenn sich irgendwelche Daten ändern, nicht mehr unbedingt den Text ändern muss.

Dafür darf man nicht vergessen zu überprüfen, ob die Interpretation noch zu den neuen Zahlen passt, aber meistens ändern sich ja nur Kleinigkeiten.

Dann:

Skalenniveaus

Wir hatten über **Skalenniveaus** geredet und ich hatte auch erwähnt, dass es dafür äquivalente in R gibt. Was ich euch nicht gezeigt hatte, war, wie man das überprüfen kann und wie man vllt eine Information in R ändert.

Der class-Befehl zeigt, welches Datenformat die Daten haben.

- boolean: Das sind TRUE / FALSE - Angaben, man braucht sie häufig innerhalb von Funktionen, um bestimmte Parameter einzustellen

- factor und character sind nominal, wobei character Buchstaben enthalten muss, factor nicht. Ein factor hat “level”, das sind die Werte, die in dem Vektor, der die class factor hat, enthalten sind
- ordered factor ist ordinal
- numeric: metrisch
- Es gibt zwei Typen numeric, nämlich: integer (ganze Zahlen) und double (Kommazahlen, Dezimalzahlen)

Diese Typen findet man mit dem Befehl typeof heraus.

```
class(Bornholm)
#> [1] "data.frame"

class(Bornholm$Site)
#> [1] "character"

class(Bornholm$Number)
#> [1] "numeric"
typeof(Bornholm$Number)
#> [1] "double"
```

Ordinale Daten sind irgendwie logischerweise in R einfach factor-Daten, deren Werte (level) eine Ordnung zugewiesen bekommen haben. Das kann man recht einfach mit dem Befehl ordered.

Gerade so etwas wie Periodenbezeichnungen kann man mit seinem archäologischen Fachwissen wunderbar in die richtige Reihenfolge bringen. Mit dem levels-Befehl lässt sich überprüfen, wie diese Reihenfolge aussieht.

```
class(Bornholm$Period) # welcher Datentyp ist der vector Period des Datensatzes Bornholm ?
#> [1] "factor"
# schaut euch den einfach mal an (zB mit View())
# wie ihr seht, sind die Perdioden mit 1a und 1b, 2a und 2b etc kodiert.
# Denen ist also eine aphabetische Reihenfolge schon inhärent. Deswegen muss ich keine Reihenfolge angeben, wenn ich sie ordnen möchte:
Bornholm$Period <- ordered(Bornholm$Period)

class(Bornholm$Period) #checken: hat sich was verändert?
#> [1] "ordered" "factor"

levels(Bornholm$Period) #das sind die Werte in der geordneten Reihenfolge
#> [1] "1a" "1b" "2a" "2b" "2c" "3a" "3b"
```

Weil ich es kann, bringe ich jetzt noch einmal Unordnung in diese Reihenfolge:

```
Bornholm$Period <- ordered(Bornholm$Period, levels = c("3a", "1b", "1a", "2c",
"3b", "2a", "2b"))
class(Bornholm$Period)
#> [1] "ordered" "factor"
levels(Bornholm$Period)
#> [1] "3a" "1b" "1a" "2c" "3b" "2a" "2b"
```

Was für eine Macht! Das Chronologie-System ist zerstört!

Und noch etwas kann ich: Zahlen in Text umwandeln!

Manchmal ist das sinnvoll, nicht jede Zahl als Zahl zu verstehen, sondern als Abkürzung für eine Kategorie. DAs geht so:

```
class(Bornholm$Number)
#> [1] "numeric"
Bornholm$Number <-  as.character(Bornholm$Number)
class(Bornholm$Number)
#> [1] "character"
```

Aber, naja, wir wollen ja nicht, dass wir gleich falsche Ergebnisse bekommen, deshalb benutzen wir jetzt folgenden Trick:

Der Vektor Bornholm$Period wurde von uns mit der falschen Reihenfolge überschrieben. Und der Bornholm$Number mit einem anderen Datentyp als er vorher war.

Was passiert, wenn wir einfach die Daten noch einmal neu einladen?

Also den data-Befehl noch einmal benutzen?

Ha! Damit überschreiben wir die falsch geordneten Daten einfach wieder. Solange wir unsere Originaldaten irgendwo haben und diese NICHT ÜBERSCHREIBEN (deswegen legen wir sie immer in einem anderen Ordner ab als die Daten, die wir aus R hinaus wieder irgendwo abspeichern, nämlich im Ordner raw data). Die Originaldaten sind im Paket archdata unangetastet geblieben, weil wir sie nicht exportiert haben.

Und man braucht das alles nicht neu zu schreiben oder so, nein, man scrollt in seinem RMarkdown-Dokument einfach nach oben und wiederholt den Befehl, der da noch irgendwo steht.

Tatsächlich ist das relativ häufig notwendig, wenn man versucht etwas umzusetzen, was man noch nie gemacht hat und ein paar unterschiedliche Dinge ausprobieren möchte…. muss… ;-)

Allgemeiner Tipp: Was funktioniert, immer erst einmal stehen lassen, kopieren und an der Kopie rumprobieren bis es klappt. Wenn man die Lösung irgendwann hat, kommentiert man sie sich und löscht alles, was vorher nicht geklappt hatte.

Jetzt aber zu den Diagrammen. Es geht im Folgenden um

- ggplot- Logik
- Liniendiagramme
- Dichtediagramme
- Facettierungen
- geom_raster
- ggplot-Hilfen

ggplot- Logik!

Zur Wiederholung empfehle ich folgenden Link: https://r-intro.tadaa-data.de/book/visualisierung.html Ja, ich bin zu faul um das abzutippen. Gleichzeitig lernt ihr so aber auch eine weitere gute Ressource kennen, die auf R auf deutsch erklärt und frei zugänglich ist.

Es gibt eine große Onlinecommunity zu R, man findet eigentlich zu jeder Frage eine Antwort. Vieles ist aber tatsächlich auf englisch, deswegen ist es eine gute Idee, auch mal auf englisch zu googeln. Beim googeln sollte man übrigens immer neben R auch den Paketnamen zu der eigentlichen Frage / Stichworten eingeben.

Wenn man eine Antwort nicht versteht, nicht verwirren lassen: Häufig gibt es mehrere Lösungswege. Wie in einer normalen Sprache, gibt es auch in Programmiersprachen unterschiedliche Wege um das gleiche auszudrücken. Sollte immer der gleiche Lösungsweg vorgeschlagen werden und man versteht ihn nicht, dann versucht euch über die R-Hilfe die einzelnen Schritte des Lösungsweges anzuschauen. Oder fragt jemanden, manchmal reicht schon ein zweites Paar Augen, um ein Problem zu lösen.

Auf zu den Diagrammtypen jetzt:

Liniendiagramme!

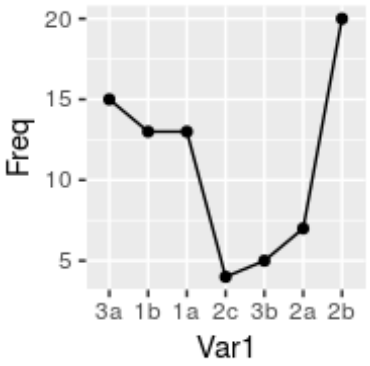
Liniendiagramme sind nur sinnvoll, wenn auf der X-Achse eine Abfolge erstellt werden kann, also mindestens ordinale Daten abgetragen werden können. Für zeitliche Entwicklungen eignen sie sich super, aber es muss beachtet werden, dass jedem X-Wert nur ein Y-Wert zurgeordnet werden darf. Das ist der Grund, warum wir in dem Bsp erst eine neue Tabelle bauen müssen.

Im Bornholmer Datensatz gibt es eine schöne Abfolge von verschiedenen Perioden und ich will einfach nur darstellen, wie viele Fundstellen es pro Periode gibt. Vielleicht benutze ich das für eine Bevölkerungsdichtenrekonstruktion, wer weiß.

Der table-Befehl zählt, wie häufig eine Variable mit einer anderen vorkommt. Da ich nur eine Variable angeben, zählt er einfach, wie häufig diese vorkommt. Da ggplot nicht mit dem Format table arbeiten kann, wandeln wir bh_table noch in einen data frame um.

Schaut euch die Tabelle bh_table einmal mit View an, dann versteht ihr, warum ich im Liniendiagramm die Variablen so gewählt habe, wie sie da stehen.

```
bh_table <- table(Bornholm$Period) # Häufigkeiten zählen
bh_table <- as.data.frame(bh_table) # als Dataframe überspeichern


ggplot(data = bh_table)+
  geom_point(aes(x = Var1, y = Freq))+
  geom_line(aes(x = Var1, y = Freq, group = 1)) # group = 1 ist für geom_line wichtig, weil es sonst Daten gruppieren soll. Ich habe die Gruppierung aber schon vorher mit dem table-Befehl erledigt.
```




```
# ein Punktdiagramm über einem Liniendiagramm "markiert" die Stellen, wo ich Datenpunkte habe
# wie ihr seht, kann man unterschiedliche Diagrammtypen übereinander plotten!
```

Aufgabe: Das ist zwar ein ordentliches Diagramm, aber die Beschriftung ist eher hässlich. Bitte kramt den Code der letzten Sitzung raus und beschriftet die Achsen angemessen.

Viel Erfolg!

Liniendiagramm mit mehreren Linien und der notwendige Umstellungsspaß mit den Daten

Häufig will ich ja gar nicht nur eine Linie darstellen, sondern mehrere Verläufe vergleichen. Selbstverständlich geht das auch mit R. Mit R geht aaaaalles.

Ich muss allerdings erst ein bisschen die Daten in eine Form bringen, mit der ich arbeiten kann. Ich möchte jetzt gern wissen, wie häufig welche Fundtypen in welcher Periode auftauchen. Die Fundtypen sind die ganzen komischen Kürzel im Datensatz Bornholm.

Was ich kreiieren möchte, ist eine Tabelle mit den Spalten: Periode, Fundtypus, Häufigkeit in der Periode. Site und Number interessieren mich nicht mehr. Ich wandle ein “breites” Datenformat in ein “langes” um. Allgemein geht ggplot lieber mit langen Datensätzen um als mit breiten.

Was meine ich damit? Was passiert hier?

Vielleicht erklärt dieser Blogpost mehr: http://archaeoinformatics.net/r-seperate-gather-spread/ Das Vorgehen "gather" brauchen wir hier auch: Es ist etwas komplizierter als die Tabelle vorhin, deswegen benutzen wir die beiden neuen Pakete tidyr und dplyr. Beide gehören zu einer R-Philosophie, die sich tidyverse nennt. Das ist eine Reihe von Paketen, die gut miteinander zurecht kommen und ähnliche Syntaxen verwenden. ggplot gehört auch dazu.

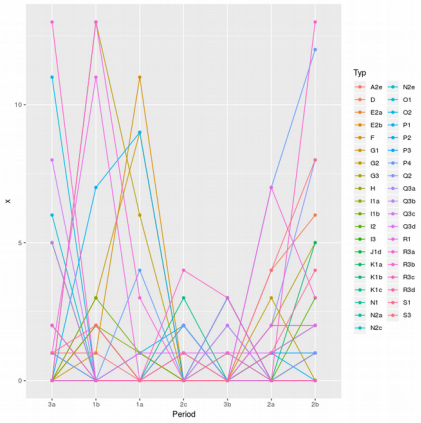
Sie benutzen ein neues Zeichen, das in R base keine Rolle spielt: Die "Pipe" %>%

Pipes sind auch aus anderen Programmiersprachen bekannt. Sie sagen eigentlich nur "was ich gerade in dieser Zeile gemacht habe, übertrage ich auch in die nächste" und man spart sich eine Reihe von "Zwischensicherungen" in Variablen.

Als Bsp mach ich diese Umformung der Daten auf beide Weisen:

Zuerst oldschool:

```
Bornh1  <- Bornholm[, -c(1:2)] #mit eckigen Klammern kann ich aus dem Datensatz Bornholm bestimmte Spalten entfernen: - Spalte 1 bis 2

Bornh2 <-  gather(Bornh1, key = "Typ", value = "Haeufigkeit", "N2c":"A2e") # das ist der Code der die Umformung vornimmt.
#Schaut euch Bornh2 einmal an, damit ihr versteht, was passiert. Manchmal gibt es jetzt gleiche Periode und Typ mit unterschiedlichen Häufigkeiten. Das muss noch einmal zusammengefasst werden: aggregate!

Bornh3 <- aggregate(Bornh2$Haeufigkeit, by = list(Typ = Bornh2$Typ, Period = Bornh2$Period), FUN = sum)
# jetzt ist dummerweise Haeufigkeit in x umbenannt worden

ggplot(data = Bornh3)+
  geom_point(aes(x = Period, y = x, color = Typ))+
  geom_line(aes(x = Period, y = x, color = Typ, group = Typ)) # group bestimmt, welche Punkte verbunden werden
```



Jetzt das gleiche in tidy code. Wie ihr seht, liegt der Unterschied v.a. darin, dass ich nicht dauernd neue Variablen benenne:

```
Bornholm %>%
  select(-Site, -Number) %>%
  gather(key = "Typ", value = "x", "N2c":"A2e") %>%
  group_by(Typ, Period)%>% #ich gruppiere mein Daten, wie bei aggregate
  summarize(Haeufigkeit = sum(x))%>% #ich summiere sie jetzt und geb den
Namen Haeufigkeit für die Spalte
  ggplot()+
  geom_point(aes(x = Period, y = Haeufigkeit, color = Typ))+
  geom_line(aes(x = Period, y = Haeufigkeit, color = Typ, group = Typ))

# Nach tidyverse-Logik ist das leichter zu lesen. Was denkt ihr?
```

(insert same picture here)...

Aber oje oje, weg vom Code, hin zum Plot: Was ist denn da passiert?

Schöne Idee war das ja, mit den Typenhäufigkeiten nach Periode, aber...

Wir erkennen ein Problem: Zu viele Informationen auf einmal sind keine gute Idee.

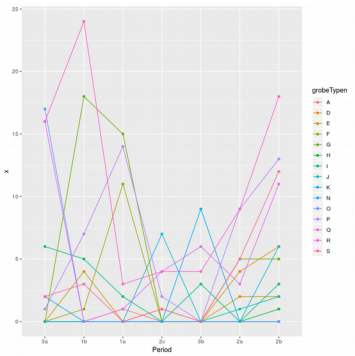
Können wir die Daten also vllt ein bisschen gruppieren?

Ich bin dafür, dass wir nur die Großbuchstaben der Typenbezeichnungen benutzen, weil ich davon ausgehe, dass das irgendwelche Übergruppen darstellen könnte. Ich arbeite dafür mit Bornh2 weiter.

Drei Schritte braucht es: 1. Ich brauche eine neue Spalte, in der die neuen Gruppentypenbezeichnungen eingetragen werden , 2. Ich muss dort die richtigen Typen eintragen – in diesem Fall kann ich einfach nur den ersten Buchstaben aus den Typenbezeichnungen behalten Wie das geht, hab ich ergoogelt, sowas hab ich vorher nicht gebraucht – und 3. die neuen Gruppen müssen zusammengerechnet werden mit der Funktion aggregate.

```
#1. neue Spalte
Bornh2$grobeTypen  <- Bornh2$Typ # hiermit erstelle ich eine neue Spalte, die
genau den gleichen Inhalt hat, wie die Typ-Spalte

#2. ersten Buchstaben erhalten (Buchstaben 1 bis 1 erhalten)
# wir brauchen ein neues Paket namens "stringr". Bitte installiert es.
library(stringr)

Bornh2$grobeTypen <- str_sub(Bornh2$Typ, 1,1) # wir nehmen die Worte in
Bornh2$Typ und benutzen nur Buchstabe 1 bis 1

#3. Daten wieder zusammenfassen:
# es sind die gleichen Datensaetze mit unterschiedlichen Haeufigkeiten
```

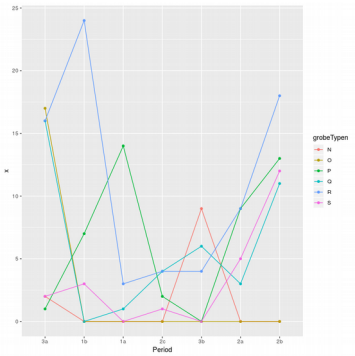
```
entstanden, die muessen noch mal zusammengefasst werden

Bornh3 <- aggregate(Bornh2$Haeufigkeit, by = list(grobeTypen =
Bornh2$grobeTypen, Period = Bornh2$Period), FUN = sum)

# wieso hier ein neuer Code Chunk? Reine Gewohnheit meinerseits, damit ich einen
besseren Überblick behalte, trenne ich nach längeren Umwandlungen gern den
Code für die Grafik nochmal ab.
# neues Diagramm

ggplot(data = Bornh3)+
  geom_point(aes(x = Period, y =x, color = grobeTypen))+
  geom_line(aes(x = Period, y = x, color = grobeTypen, group = grobeTypen))
```



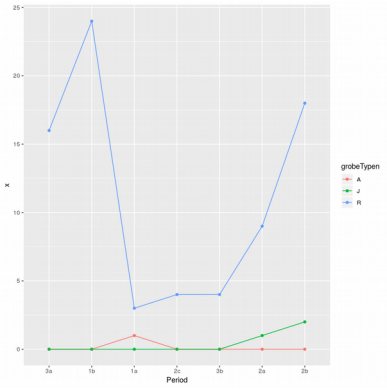
Das sieht doch gleich viel besser aus. Auch wenn immer noch etwas viel vllt.

Ich kann den Datensatz auch noch weiter verkleinern, in dem ich mir zB nur bestimmte Typen rauspicke. Dafür gibt es unterschiedliche Möglichkeiten. Die subset-Funktion, mit der ich dann auch wieder neue Datensätze erstelle oder das Filtern:

```
Bornh3%>%
  filter(grobeTypen > "M")%>% # nur die groben Typen, die "größer als M" sind, also
nach M im Alphabet kommen
  ggplot()+
  geom_point(aes(x = Period, y =x, color = grobeTypen))+
  geom_line(aes(x = Period, y = x, color = grobeTypen, group = grobeTypen))
```

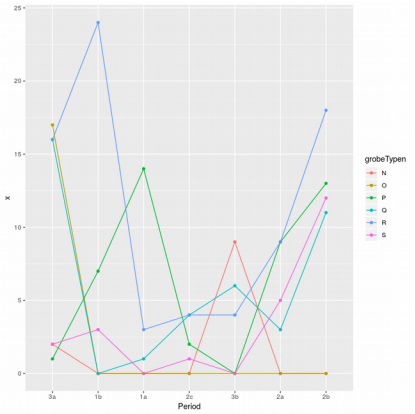


```
#oder
Bornh3%>%
  filter(grobeTypen == "A" | grobeTypen== "J" | grobeTypen == "R")%>% #bitte nur die Zeilen, wo grobe Typen A, J oder R ist. Das Zeichen für ODER ist ein senkrechter Strich und findet sich links unten auf der deutschen Tastatur.
  ggplot()+
  geom_point(aes(x = Period, y =x, color = grobeTypen))+
  geom_line(aes(x = Period, y = x, color = grobeTypen, group = grobeTypen))
```



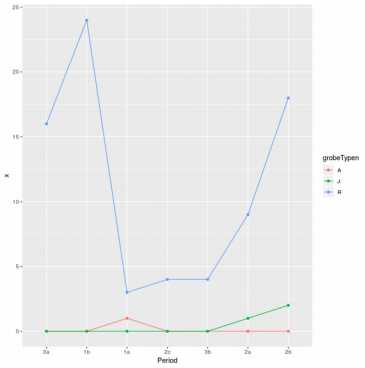
Das gleiche mit subset:

```
Bornh4 <- subset(Bornh3, Bornh3$grobeTypen > "M") #ich nehme eine Auswahl von Bornh3 und zwar da, wo Bornh3$grobe Typen größer als M ist und weise diesem Datensatz die Variable Bornh4 zu

ggplot(data = Bornh4)+
  geom_point(aes(x = Period, y =x, color = grobeTypen))+
  geom_line(aes(x = Period, y = x, color = grobeTypen, group = grobeTypen))
```




```
Bornh5 <- subset(Bornh3, Bornh3$grobeTypen == "A" | grobeTypen== "J" | grobeTypen == "R")

ggplot(data = Bornh5)+
```

```
  geom_point(aes(x = Period, y =x, color = grobeTypen))+
  geom_line(aes(x = Period, y = x, color = grobeTypen, group = grobeTypen))
```



Ob ich subset nehme oder filter, liegt ähnlich wie bei der Entscheidung oben zwischen tidy verse und old school R v.a. daran, ob ich mit dem reduzierten Datensatz noch häufiger arbeiten werde. Wenn ja, dann weise ich ihm lieber eine Variable zu, weil die Chance dann auch nicht ganz schlecht steht, dass ich noch weiß, was das für eine Variable ist (Bornh4 ist ein ganz schlechter Name. Sinnvoll wäre Bornh_M oder so gewesen). Wenn ich das aber nur ein-zweimal für eine Visualisierung mache, dann nutze ich filter.

Welche Auswahlen jedoch archäologisch relevant ist, entscheidet ihr als Expert*innen im Feld!

Jetzt reicht es aber auch langsam mit den Liniendiagrammen, machen wir doch mal was mit Dichte.

Dichtediagramme!

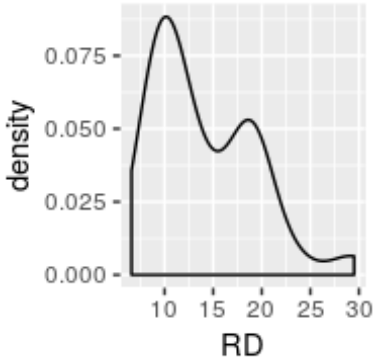
Wie die funktionieren, habt ihr hoffentlich gelesen.

Bei Dichtediagrammen wird die x-Achse wieder metrisch. Das geht also NICHT mit den Periodenangaben, mit denen wir die ganze Zeit eben rumgespielt haben.

Wir könne stattdessen wieder mit den Bronzezeitlichen Tassen arbeiten.

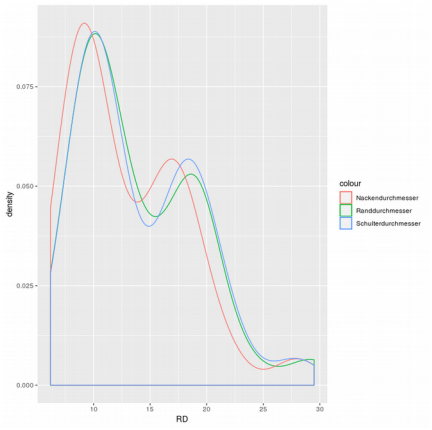
Die Funktion in R heißt density.

```
ggplot(data = BACups)+
  geom_density(aes(x = RD))
```



Das war doch ganz einfach. Aber jetzt möchte ich gern die unterschiedlichen Messungen an den Tassen im Vergleich sehen.

```
# Ich kann ganz einfach den geom_density-Befehl mehrmals rufen

ggplot(data = BACups)+
  geom_density(aes(x = RD, col ="Randdurchmesser"))+
  geom_density(aes(x = ND, col = "Nackendurchmesser"))+
  geom_density(aes(x = SD, col = "Schulterdurchmesser"))
```

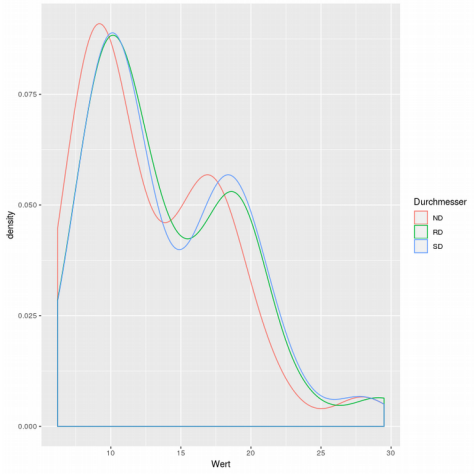


Das funktioniert zwar, aber es ist nicht gerade “elegant”.

Elegant wär es, erst die Daten so umzuformen, dass sich in ggplot dann mit möglichst wenig Befehlen meine Grafik darstellen lassen kann.

Wir brauchen also wieder die Umformungen mit tidyr. Der Schritt ist relativ einfach. Ich nehm die Spalten, die mich interessieren sortiere die Werte dieser Spalten neu, so dass ich eine Spalte habe, in der steht die ehemalige Überschrift der Spalte und einen Spalte, in der der dazugehörige Wert steht. Das ist wieder der gather-Befehl.

```
BACups%>%
  gather(key = "Durchmesser", value = "Wert", "RD", "ND","SD") %>%
  ggplot()+
  geom_density(aes(x = Wert, col= Durchmesser))
```

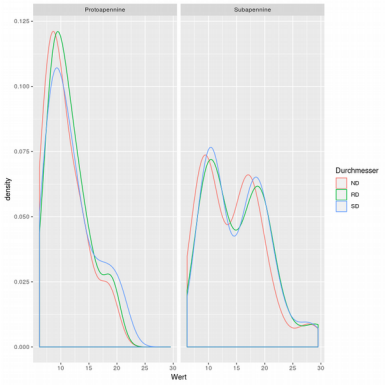



```
# genauso viele Zeilen Code aber schicker.
```

Facettierung!

Jetzt wird es nochmal richtig cool. Der Dichteplot eben, den nochmal nach unterschiedlichen Phasen anzulegen, das wär gut oder?

```
BACups%>%
  gather(key = "Durchmesser", value = "Wert", "RD", "ND","SD") %>%
  ggplot()+
  geom_density(aes(x = Wert, col= Durchmesser))+
  facet_grid(.~Phase)
```

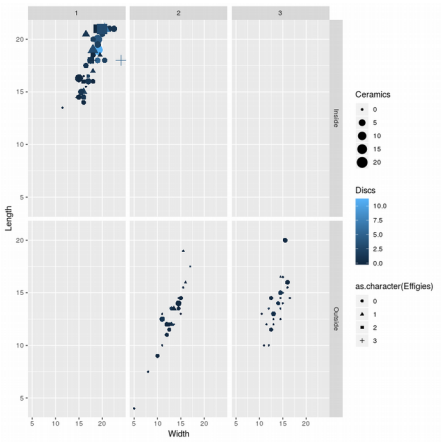


Nur eine einzige Zeile Code mehr und schaut es euch an: Interessantes Ergebnis oder? Die ganzen hohen Durchmesser-Werte kommen fast alle aus der subappeninen Phase. Interessant!

Noch ein anderes Bsp fürs Facettieren. Nehmen wir doch mal den Snodgrass-Datensatz mit den Häusern. Da gibt es zwei nominale Attribute, die man bei der “Facettierung” geenüber stellen kann.

Achja. Und ich benutz mal alle Variablen eines Streudiagramms, die mir einfallen... Versucht mal durchzublicken.

```
ggplot(data = Snodgrass)+
  geom_point(aes(x = Width, y = Length, col= Discs , shape = as.character(Effigies), size = Ceramics))+
          facet_grid(Inside~Segment)
```

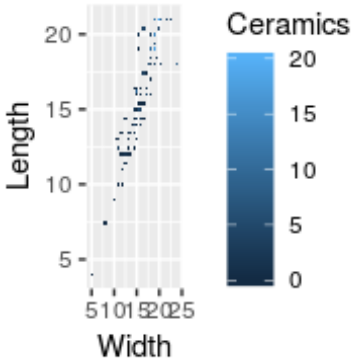



```
# Was passiert hier alles?
```

geom_raster

Irgendjemand wollte etwas über geom_raster erfahren. Mein ggplot-Schummelzettel sagt mir, dass ich damit drei Variablen darstellen kann. Schauen wir doch mal:

```
ggplot(data = Snodgrass)+
  geom_raster(aes(x = Width, y = Length, fill = Ceramics))
```



Ja, nicht uninterssant, aber spannender mit dichteren Datensätzen , denke ich. Die Daten werden sozusagen “gerastert”, also gleichmäßig abgetragen und eingefärbt nach einer dritten Variable.

Last comments

GGplot hat noch viel viel mehr Möglichkeiten. Um einen Überblick zu bekommen, empfehle ich den Blogpost hier zu lesen, der vorführt, wie sich so eine Visualisierung entwickeln kann und am Ende richtig richtig gut aussieht:
https://cedricscherer.netlify.com/2019/05/17/the-evolution-of-a-ggplot-ep.-1/

Hilfen, um mit R und ggplot zurechtzukommen sind:

- die Schummelzettel: https://www.rstudio.com/wp-content/uploads/2015/06/ggplot2-german.pdf
- dieses R-Intro-Buch: https://r-intro.tadaa-data.de/book/visualisierung.html
- das deutsche Wikibook zu R: https://de.wikibooks.org/wiki/GNU_R und
- das englische R-Cookbook: http://www.cookbook-r.com/

Das Nachschlagewerk für RMarkdown:

- offizielle website: https://rmarkdown.rstudio.com/index.html
- The Definitive Guide: https://bookdown.org/yihui/rmarkdown/