Universität Hamburg

Proseminar: Programmieren in R

Plotten mit GGPlot2

Anne Kunstmann

Betreuer: Jakob Lüttgau

28. September 2016

Zusammenfassung

Im Mittelpunkt dieses Berichts soll der Umgang mit dem R-basierten Datenvisualisierungspaket GGPlot2 stehen. Ziel ist es dabei, einen anfängerfreundlichen Einstieg ins Plotting mit diesem Paket zu ermöglichen. Die Einführung erfolgt unter dem Aspekt einem Einsteiger den natürlichen Arbeitsfluss beim Erstellen eines Plots nachempfinden zu lassen und ihn so Schritt für Schritt zu einem detaillierten, komplexen Graphen zu führen. Letztendlich soll ein fertiger Graph exportiert und genutzt werden können.
Augenmerk wird insbesondere auf elementare Grundlagen wie der Aufbau der *ggplot*-Funktion, ihre Gestaltungsmöglichkeiten und nützliche *geometric* Funktionen gelegt. Ausschnitthaft finden sich in dieser Abhandlung auch kurze Einblicke in GGPlot2's Entstehung, seine internen Funktionen sowie leicht fortgeschrittene Abschnitte zum Thema Facetting und statistischer Transformation wieder.
Es sei außerdem erwähnt, dass der Fokus dieser Abhandlung bewusst auf die praktische Anwendung des GGPlot2-Pakets ausgerichtet ist und dass das Erlernen des Umgangs und Begreifen der Intuitivität der Datenvisualisierung im Mittelpunkt steht. Zahlreiche Codebeispiele und Plots veranschaulichen die einzelnen Kapitel und geben Vorlagen, um diese selbst auszuprobieren.

Inhaltsverzeichnis

1 Einführung

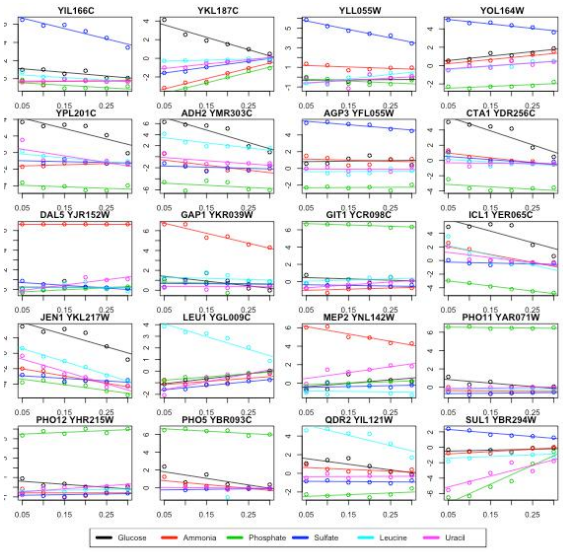
GGPlot2 ist ein in R verwendetes Datenvisualisierungspaket, das im Jahre 2005 veröffentlicht wurde und sich großer Beliebtheit erfreut. Hadley Wickham, neuseeländischer Statistiker und Entwickler des Pakets, wurde unter anderem für GGPlot2 mit dem John Chambers Award im Jahr 2006 ausgezeichnet. Basierend auf den Grundlagen des Buches „The Grammar of Graphics“ von Leland Wilkinson baute Wickham sein Datenvisualisierungspaket auf und übernahm die grundlegenden Thesen des Buches, die auch heute noch als richtungsweisend für die Statistik gelten. [1, 2]

Abbildung 1: Code in R mit nativen Funktionen
[http://varianceexplained.org/r/why-I-useggplot2/]

```
par(mar = c(1.5, 1.5, 1.5, 1.5))

colors <- 1:6
names(colors) <- unique(top_data$nutrient)

# legend approach from http://stackoverflow.com/a/10391001/712603
m <- matrix(c(1:20, 21, 21, 21, 21), nrow = 6, ncol = 4, byrow = TRUE)
layout(mat = m, heights = c(.18, .18, .18, .18, .18, .1))

top_data$combined <- paste(top_data$name, top_data$systematic_name)
for (gene in unique(top_data$combined)) {
    sub_data <- filter(top_data, combined == gene)
    plot(expression ~ rate, sub_data, col = colors[sub_data$nutrient], main
    for (n in unique(sub_data$nutrient)) {
        m <- lm(expression ~ rate, filter(sub_data, nutrient == n))
        if (!is.na(m$coefficients[2])) {
            abline(m, col = colors[n])
        }
    }
}

# create a new plot for legend
plot(1, type = "n", axes = FALSE, xlab = "", ylab = "")
legend("top", names(colors), col = colors, horiz = TRUE, lwd = 4)
```

Laut Leland lässt sich jeder Datensatz, egal welcher Komplexität, leicht darstellen, wenn man ihn sinnvoll in Ästhetik und Geometrie unterteilt. Mittels dieser Differenzierung können Aufgabenbereiche getrennt und Zuständigkeiten an verschiedene Aspekte des Plots übergeben werden. [1, 2]

Abbildung 2: R nativer Plot
[http://varianceexplained.org/r/why-I-useggplot2/]



Sobald komplexere Daten visualisiert werden sollen, zeigen sich die GGPlot2’s Vorteile in vollen Zügen. Durch das feste Schema, das es dem Programmierer vorgibt, kann dieser quasi jeden Plot nach einem Muster abarbeiten. Das führt unter anderem dazu, dass der Code meist sehr einfach gehalten ist und bei nachträglichen Änderungen nur wenigen Abwandlungen unterzogen werden muss. Bei Farb- oder Formeinsatz innerhalb des Plots wird automatisch eine Legende berechnet, die ebenfalls jederzeit manuell gestaltbar ist. Dies steht ganz im Gegensatz zu R nativen Plots, die durchweg individuell durchprogrammiert werden

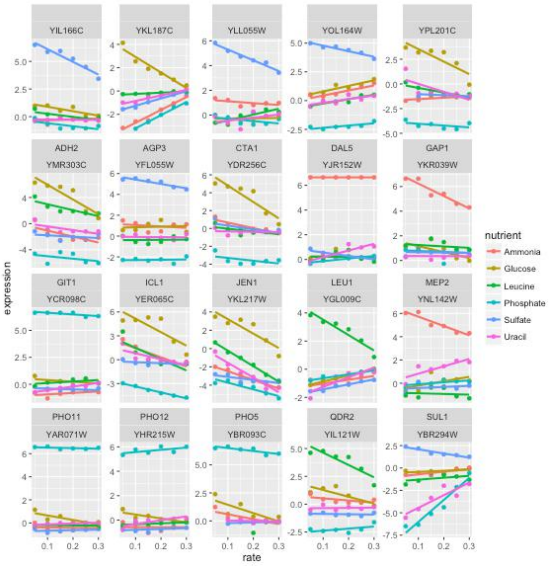
müssen, was an sich zeitintensiv sein und großen Aufwand nach sich ziehen kann, sollten nachträgliche Änderungen an der Darstellung nötig sein. [3] Zur Verdeutlichung der Unterschiede sollen hier beispielhaft Code und Visualisierung über R und über GGPlot2 gegenübergestellt werden.

Der R native Code in Abbildung 1 ist verhältnismäßig umfangreich. Im Quellcode finden sich Programmierstrukturen wie Schleifen und if-Abfragen wieder, die gerade Anfängern oft Schwierigkeiten bereiten. Die Visualisierung (Abbildung 2) selbst ist schwer einsehbar, so überdecken sich gegenseitig Graphen oder haben keine flüssigen Übergänge ihres Verlaufs. Es ist fraglich, ob der Aufwand und die Masse an Code in einem ausgeglichenen Verhältnis zur heute nicht mehr standardgemäßen Visualisierung stehen. Im Vergeich dazu die Präsentation der Darstellung (Abbildung 4) und der dazugehörige Quellcode in Abbildung 3 unter Nutzung von GGPlot2. Hier ist der Code kurz und prägnant. Keine der vorher komplizierteren Programmierstrukturen finden sich im GGPlot2-Code wieder. Er wirkt insgesamt übersichtlicher. Dies gilt auch für die Visualisierung, die flüssige Verläufe der Graphen zeigt und durch Kontrastsetzung die Lesbarkeit des Plots erhöht. Noch unerwähnt blieben bisher außerdem GGPlot2's Vorteile bezüglich seiner Bandbreite an visualisierbaren Graphen sowie das beinahe unbeschränkte Erweitern des Pakets durch weitere Pakete. Die Grenzen GGPlot2's beginnen erst bei individuelleren Gebieten wie beispielsweise der Graphentheorie, interaktiven oder 3D-Graphiken. Diese können bisher nicht visualisiert werden. [13]

Abbildung 3: Code mit GGPlot2-Funktionen
[http://varianceexplained.org/r/why-I-useggplot2/]

```
ggplot(top_data, aes(rate, expression, color = nutrient)) +
    geom_point() +
    geom_smooth(method = "lm", se = FALSE) +
    facet_wrap(~name + systematic_name, scales = "free_y")
```

Abbildung 4: Plot mit GGPlot2
[http://varianceexplained.org/r/why-I-useggplot2/]



2 Preprocessing

3.1 Datenimport

Um Datensätze schnell und einfach zu importieren, bietet R die *read*-Funktion an. Mit Hilfe nur weniger Parameter lassen sich Datensätze verschiedenster Dateitypen importieren und verarbeiten. [21]

Beispiel: Angenommen eine *csv*-Datei namens *airquality* befindet sich auf dem verwendeten Rechner und diese soll importiert werden.

Codebeispiel 1: Datenimport einer csv-Datei

```
1 airquality <- read.csv("airquality.csv", header = T)
```

Die *read*-Funktion wird durch den „.“-Operator mit dem zu importierenden Dateiformat *csv* verbunden und aufgerufen, als Parameter wird der Funktion lediglich der Dateiname *airquality.csv* als String übergeben und durch Angabe eines booleschen Wertes *T* oder *F* (auch ausgeschrieben als *true* oder *false* möglich) wird die Übernahme der Spalten- und Zeilenüberschriften, also der *header*, des Datensatzes bestimmt. [21] Der Datensatz wird dann in eine Variable namens *airquality* abgespeichert und ist von nun an in der R-Umgebung verfügbar. Hier ein kurzer Ausschnitt des Datensatzes über die *head*-Funktion, der in den nächsten Kapiteln fast ausschließlich verwendet werden wird. Der *airquality*-Datensatz ist eigentlich bereits nativ in R enthalten. Der aufgeführte Datenimport soll lediglich die Funktion beispielhaft veranschaulichen. Für genauere Ausführungen zum Thema Datenimporte sollte Isabella Trans Paper gelesen werden.

Abbildung 5: Ausschnitt der importierten Daten

```
> head(airquality)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
```

Nach dem Import ist es sinnvoll zu entscheiden, welche Daten genutzt werden sollen. Wichtig werden im weiteren Verlauf dieses Papers besonders die Spalten der Windstärke, Temperatur, des Monats und Tages.

3.2 Datenunterscheidung

Die Statistik kennt zweierlei Arten von Merkmalen - die stetigen und diskreten. Stetige Daten zeichnen sich durch eine überabzählbare Anzahl an Merkmalen aus, hierzu zählen Angaben wie Größe einer Person oder Temperaturen in Grad Celsius. Sei die Körperhöhe eines Menschen beispielsweise 172 cm. Da dies ein stetiges Merkmal ist, können wir die eigentliche Körperhöhe der Person immer genauer bestimmen. Angenähert wäre die Größe zum Beispiel 1,72348 cm, aber sie könnte noch endlos exakter festgelegt werden.

Anders die diskreten Daten. Sie beinhalten eine abzählbare Menge unendlicher Merkmale. Als Beispielsmerkmale könnten hier Geschlechter, Religionszugehörigkeiten oder Monate genannt werden. Diskrete Attribute der Monate sind beispielsweise Januar, Februar, März oder April. Es ist keine genauere Annäherung an eines der Attribute möglich, es kann nur Februar oder März sein, nichts dazwischen.
Die Unterscheidung von Datentypen wird später recht relevant, um sinnvolle und aussagekräftige Visualisierungen für einen Plot auszuwählen. [25]

3.3 Datenmanipulation

GGPlot2 ist grundsätzlich auf sogenannte „long format“ Daten ausgelegt, das bedeutet, seine Visualisierungen werden umso aussagekräftiger mehr je mehr Daten zu einem bestimmten Merkmal vorliegen. Um die Repräsentativität von „wide format“ Datensätzen zu erhöhen, also Datensätze mit sehr vielen verschiedenen Merkmalen, gibt es in R die Pakete *reshape2* und *plyr*, die häufig zur Datenmanipulation verwendet werden. Datenmanipulation meint hierbei das Schichten, Gruppieren und Zusammenfassen verschiedener Daten. [22, 23, 24]

Beispiel: Angenommen man möchte feststellen, in welchem Monat des *airquality*-Datensatzes die höchste Durchschnittstemperatur herrschte. Durch Anwendung des *plyr*-Pakets ist das leicht realisierbar.

Codebeispiel 2 : Ermitteln der Durchschnittstemperatur pro Monat

```
airquality_means <- ddply(airquality, .(Month), summarise,
    meanTemp = mean(Temp))
# Vereinfachen des Datensatzes auf Monat und mittlere
    Temperatur
```

Die *ddply*-Funktion erhält den zu verarbeitende Datensatz als Parameter, es wird die Abhängigkeit der Temperatur vom Monat bestimmt und die mittlere Temperatur während eines Monats berechnet. Ausschnitt des Endergebnis ist dabei der unten stehende Datensatz.

Abbildung 6: Zusammengefasstes Dataframe

```
> head(airquality_means)
  Month meanTemp
1     5 65.54839
2     6 79.10000
3     7 83.90323
4     8 83.96774
5     9 76.90000
```

Das genaue Vorgehen sowie die Zusammensetzung und Funktionsweisen der *ddply*-Funktion würde den Rahmen dieses Papers sprengen. Die Möglichkeit dieses Vorgehens sollte jedoch bekannt sein. Bei weiterem Interesse am Thema kann Vu Hung Quans Darlegung des Stoffgebietes gelesen werden.

3 Grundlagen

4.1 Grundfunktionen

GGPlot2 stellt zwei grundlegende Funktionen zur Verfügung. Eine davon ist eine sehr anfängerfreundliche Funktion namens *qplot*, kurz für *quick plot*. Sie folgt nicht dem typischen GGPlot2-Schema und ist eher als Einstieg ins Plotting gedacht, ermöglicht jedoch dieselbe Plotvisualisierung wie die *ggplot*-Funktion. Ihr Aufbau und ihre Gestaltung wird lediglich durch verschiedenste Parameter bestimmt. [5, 7]

Codebeispiel 3: Die allgemeine qplot-Funktion

```
qplot(x, y, data =, color =, shape =, fill=, size=, alpha=,
  geom=, method=, formula=, facets=, xlim=, ylim=, xlab=,
  ylab=, main=, sub=)
```

Die Bedeutung und Anwendung dieser Parameter klären sich im Verlauf dieses Papers. Da, wie bereits erwähnt, die *qplot*-Funktion nicht den „Grammar of Graphics"-Grundsätzen folgt, soll auf diese nicht weiter eingegangen werden.

Das eigentliche Augenmerk liegt, wie eben schon erwähnt, auf der *ggplot*-Funktion, also dem *grammar of graphics plot*. Sie setzt sich aus zwei Hauptkomponenten zusammen - den *geometric object*-Funktionen (kurz *geom-Funktionen*) und den *aesthetics*-Funktionen (kurz *aes*-Funktionen). Beide unterscheiden sich grundlegend in ihrer Funktionalität.
Die *geom*-Funktionen können als das formgebende Merkmal gesehen werden, bestimmen also die allgemeine Darstellung der Daten. Mittels der *geometric objects* kann entschieden werden, ob die Daten beispielsweise als Boxplot (*geom_boxplot*) oder Streudiagramm (*geom_point*) visualisiert werden sollen. Sie wird grundsätzlich außerhalb der *ggplot*-Funktion implementiert und durch den „+"-Operator mit dem restlichen Code verbunden. Wichtig: Die *geom*-Funktion bestimmen weder Farbe noch Form in Bezug auf Abhängigkeiten! Die Verantwortung hierfür trägt die *aes*-Funktion.
Die *aes*-Funktion wird immer innerhalb *ggplot* implementiert, kann aber auch innerhalb der *geom*-Funktionen auftreten. Sie unterteilt und unterscheidet Daten und verdeutlicht Abhängigkeiten untereinander. Ihre möglichen Parameter können variieren, abhängig von den verwendeten *geometric objects*. [4, 6, 13]

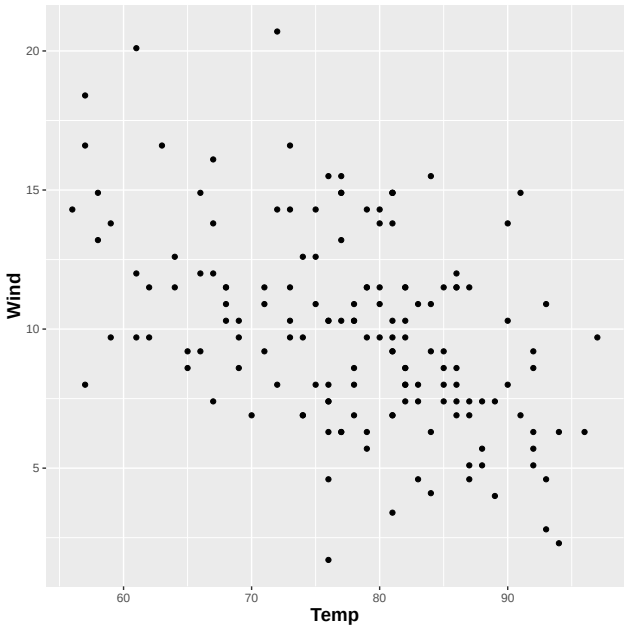
Beispiel: Nach dem Installieren des Pakets und Einbinden der *ggplot2*-Bibliothek soll nun unter Verwendung des Datensatzes *airquality* der erste Plot erstellt werden. Es gilt herauszufinden, ob die Windstärke eines Tages abhängig von der jeweiligen Tagestemperatur ist.

Codebeispiel 4 : Minimale ggplot-Funktion

```
install.packages("ggplot2") #Paket installieren
library(ggplot2)            #Bibliothek einbinden

#Frage: Existiert irgendeine Art von Relation zwischen der
   Temperatur und der Windstärke?
ggplot(airquality, aes(x = Temp, y = Wind)) + geom_point()
```

Der Datensatz *airquality* wird an *ggplot* übergeben ebenso wie die *aes*-Funktion. Letztere beinhaltet die in Abhängigkeit gesetzten Vektoren des *airquality*-Datensatzes, nämlich *Temp* und *Wind*. Zuletzt wird lediglich die *geom_point*-Funktion hinzugefügt, um den Plot als Streudiagramm zu visualisieren. Nach der Auführung des Quelltextes ergibt sich der folgende Plot.

Abbildung 7: Streudiagramm



Die x- und y- Achsenbezeichnung entsprechen den Vektoren *Wind* und *Temp* des Datensatzes, die Achsenskalierungen haben sich den Werten angepasst und insgesamt wurden die Daten durch Punkte visualisiert. Bei Betrachten des Plots lässt sich bereits erahnen, dass tatsächlich ein Zusammenhang zwischen Windstärke und Temperatur herrschen könnte. Wie aber kann dieser Zusammenhang deutlicher hervorgehoben werden?

4.2 Regressionen

Die *geom_smooth*-Funktion ist eine geometrische Funktion, die Trends und Abhängigkeiten veranschaulicht, indem sie die Regressionskurve samt Konfidenzintervall innerhalb der Graphik berechnet. [13]

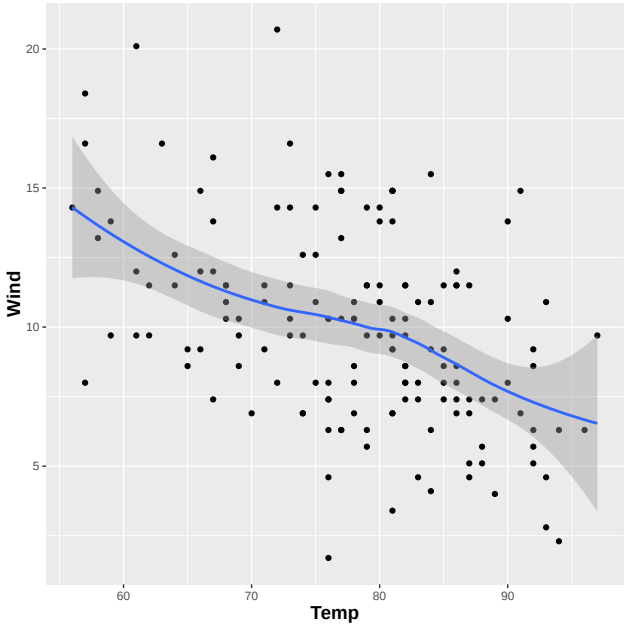
Beispiel: Hinzugefügt werden soll eine standardmäßige *geom_smooth*-Funktion, die dem Plot durch eine Regressionkurve mitsamt Konfidenzintervall mehr Aussagekraft verleiht.

Codebeispiel 5: default-Regressionskurve

```
ggplot(airquality, aes(x = Temp, y = Wind)) + geom_point() +
    geom_smooth()
```

Am Quellcode selbst hat sich im Vergleich zu Codebeispiel 4 nur wenig verändert. Lediglich *geom_smooth* wurde hinzugefügt und durch den „+“-Operator mit dem vorherigen Code verbunden. Im Plot jedoch macht sich die geringfügige Codeänderung bemerkbar.

Abbildung 8: Streudiagramm mit Standard-Regressionkurve



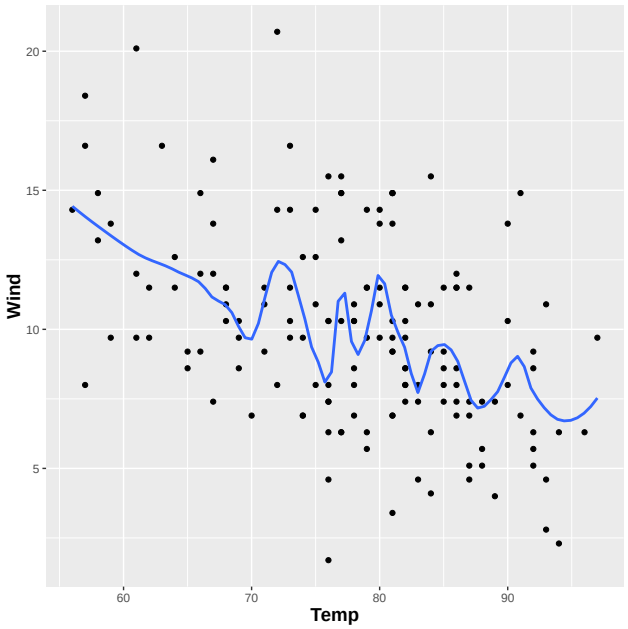
Durch Hinzufügen weiterer Parameter kann die *geom_smooth*-Funktion nach Belieben angepasst werden. Drei der wichtigsten sind hierbei *span*, *se* und *method*. *span* legt dabei die Empfindlichkeit der Regressionkurve fest. Bei einem Dezimalwert nahe 0 wird die Regressionskurve stark auf Ausreißer reagieren und dementsprechend sehr kurvig ausfallen, bei einem deziamlen Wert gegen 1 zeigt sie sich hingegen robust gegenüber ungewöhnlichen Werten. Der Parameter *se* erwartet einen booleschen Wert. Ist er *true* (per default ist *se* immer *true*) wird automatisch das Konfidenzintervall dem Plot hinzugefügt. Wird allerdings *false* übergeben, so wird in der Graphik kein Konfidenzintervall angezeigt. *method* bestimmt die Art der Regression, so kann als String beispielsweise *„lm“* übergeben werden, um eine lineare Regression in den Plot zu zeichnen, oder aber *„loess“*, um eine lokale Regression einzuführen. *method* kann zudem selbst durch einen weiteren Parameter namens *formula* verändert werden. Aus thematischen Gründen werde ich aber auch diesen Teil nicht weiter ausführen, da zur Nutzung von *formula* teilweise die Einbindung weiterer Pakete nötig ist. [8, 13]

Beispiel: Statt der default-Regressionkurve soll eine lokale Regression mit sehr großer Empfindlichkeit auf Ausreißer und ohne Konfidenzintervall für den Datensatz *airquality* verwendet werden.

Codebeispiel 6: lokale empfindliche Regressionskurve ohne Konfidenzintervall

```
ggplot(airquality, aes(x = Temp, y = Wind)) + geom_point() +
    geom_smooth(span = 0.2, se = F, method = "loess")
```

Abbildung 9: Benutzerdefinierte Regressionkurve



Eine Regressionskurve unter diesen gewählten Parametern mag zwar nicht besonders hilfreich sein, um eine Relationen aufzudecken, aber sie verdeutlichen klar die Auswirkungen auf den Plot bei benutzerdefinierten Angaben.

4.3 Labels

Von Fall zu Fall kann es manchmal sinnvoll sein, die unterschiedlichen Datenpunkte mit Bezeichnungen zu versehen, um bessere Orientierung zu ermöglichen oder einfach nur die Repräsentativität zu steigern. Dies wird GGPlot2-intern durch eine weitere geometrische Funktion mit Namen *geom_text* geregelt. [13, 15]

Beispiel: Die einzelnen Punkte des Streudiagramms sollen mit den Tagen bezeichnet werden, an denen die Messungen stattfanden.

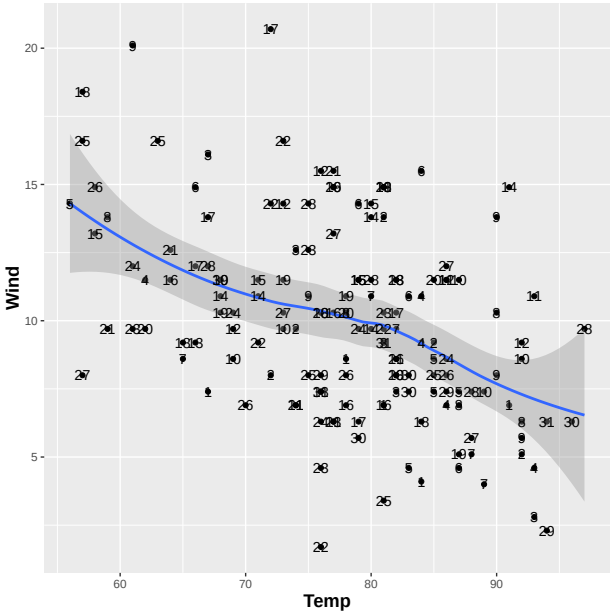
Codebeispiel 7: geom_text()-Funktion

```
ggplot(airquality, aes(x = Temp, y = Wind)) + geom_point() +
    geom_smooth() + geom_text(aes(label = Day))
#Nummerierung anhand der Messungstage auf den Punkten
```

Auch hier wird nur die *geom_text*-Funktion hinzugefügt, der vorherige Code bleibt ansonsten unverändert. Die neue geometrischen Funktion implementiert allerdings eine *aes*-Funktion, über die durch den Parameter *label* der Vektor *Day* des *airquality*-Datensatzes übergeben wird.

Beim Betrachten des Plots fällt auf, dass sich naheliegende Datenpunkte teils gegenseitig überschreiben können und besonders „long format“ Daten schnell an Übersichtlichkeit verlieren können. Außerdem erschwert die direkte Platzierung der Labels über den Punkten die Lesbarkeit.
Abhilfe wurde durch das Paket *ggrepel* geschafft, dass die Beschriftungen der Punkte daneben anordnet. Bei eng beieinander liegenden Punkten werden außerdem automatisch Verbindungslinien vom Label zu zugehörigen Punkt gezogen, um Überschneidungen zu vermeiden. [16]

Abbildung 10: Label durch native GGPlot2-Funktion

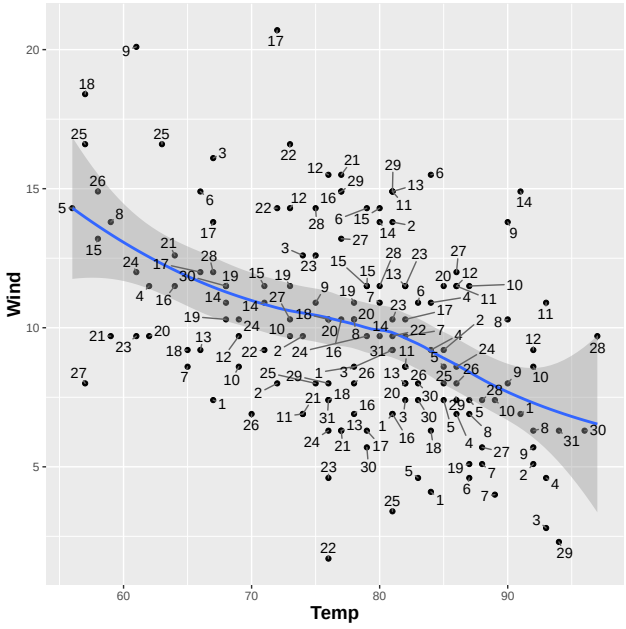


Beispiel: Nochmals sollen die einzelnen Punkte des Streudiagramms mit den dazugehörigen Tagen gekennzeichnet werden.

Codebeispiel 8: geom_text_repel()

```
install.packages("ggrepel")
library(ggrepel) #Einbinden der Bibliothek
ggplot(airquality, aes(x = Temp, y = Wind)) + geom_point() +
   geom_smooth() + geom_text_repel(aes(label = Day))
#Nummerierung anhand der Messungstage neben den Punkten
```

Abbildung 11: Labeling mit GGRepel



Dasselbe Vorgehen wie im Codebeispiel 7 findet sich hier wieder, lediglich das Einbinden der *ggrepel*-Bibliothek ist und das anschließende Nutzen von *geom_text_repel* statt *geom_text* muss beachtet werden.

Der Plot wirkt nun geordneter und übersichtlicher als vorher. Aufgrund der automatischen Verbindungslinien können auch eng beieinander liegende Datenpunkte gut voneinander differenziert werden.

4 Gestaltung

5.1 Mapping und Setting

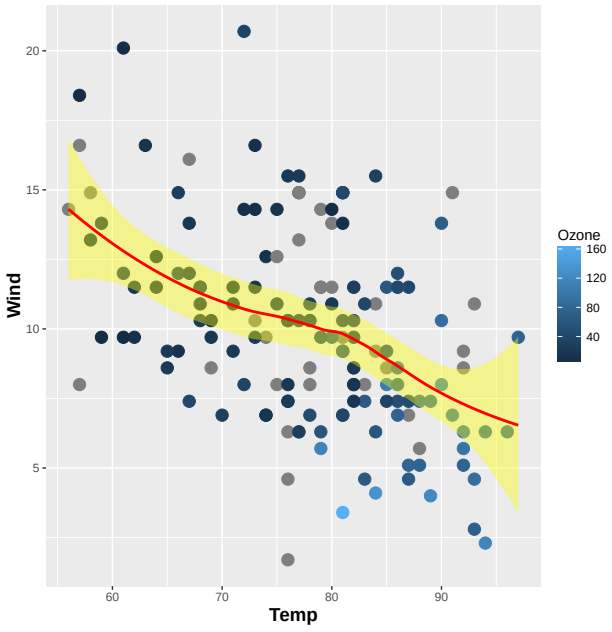
Gestaltungstechnisch bietet GGPlot2 zweierlei Konzepte innerhalb der *geom*- und *aes*-Funktionen an. *Mapping* und *Setting* sind zentral, um dem Plot durch Farben, Formen und Größen Aussagekraft zu verleihen. Sie unterscheiden sich jedoch grundlegend in ihrer Verwendung. Das sogenannte *Setten* betrifft immer die Übergabe von konstanten Werten als Parametern. Durch das Setzen von Konstanten erhält der Plot immer ein einheitliches Erscheinungsbild unabhängig von bestehenden Abhängigkeiten zwischen Variablen. [7, 13]
Vom *Mappen* wird gesprochen, wenn als Parameter variable Werte übergeben werden. Beim *Mapping* entstehen automatisch Legenden, die von GGPlot2 generiert werden und z.B. die Farbverläufe oder verschiedene Formen innerhalb der Graphik erläutern. Der Einfluss beider Gestaltungsmittel soll einmal veranschaulicht werden. [7, 13]

Beispiel: Ein farblich gut differenzierbarer Graph soll erstellt werden, der zusätzlich den Ozongehalt in Abhängigkeit der Temperatur und Windstärke offen legt.

Codebeispiel 9: Mapping und Setting

```
ggplot(airquality, aes(x = Temp, y = Wind, colour = Ozone))
  + geom_point(size = 4) + geom_smooth(fill = "yellow",
  colour = "red") #Mapping: Ozone, Setting : yellow, red, 4
```

Abbildung 12: Mapping und Setting



Bis auf einige hinzugefügte Parameter ist die Zusammensetzung der Funktion gleich geblieben. Es fällt auf, dass innerhalb der *aesthetic* Funktion ein Parameter *colour* (auch amerikanische Schreibweisen wie *color* sind in GGPlot2 zulässig) auftaucht und diesem der *Ozone*-Vektor aus dem *airquality*-Datensatzes übergeben wird. Dieser Vektor ist eine Variable, die für den blauen Farbverlauf der Datenpunkte sorgt. Anhand der Legende lässt sich erkennen: Dunkelblaue Datenpunkten stehen für niedrige Ozonwerte, je hellblauer die Färbung jedoch wird, desto höher auch der

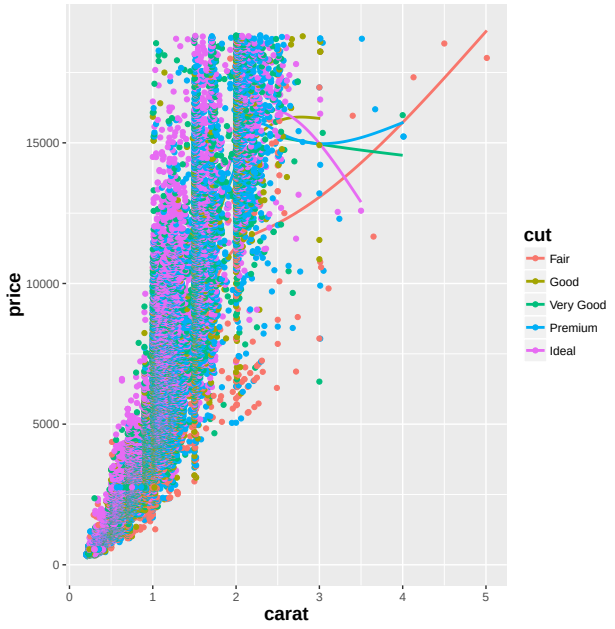
Ozongehalt. Es kann leicht erkannt werden, welche Temperatur bei welcher Windstärke herrschte und wie das Ozon sich an diesem Tag verhielt. Die Variable *Ozone* vereinfacht also das Erkennen von Zusammenhängen, sie ist *gemappt*. Das Einsetzen von beispielsweise Farbverläufen ermöglicht es also Code einzusparen und die Übersichtlichkeit des Plots zu erhalten. Graphiken können sehr schnell sehr komplex werden, so dass es sich nicht anbietet für jedes Merkmal einen eigenen Graphen zu erstellen. Es sollte eher entschieden werden, worauf das Augenmerk der Graphik gelegt wird und um dieses Thema herum können durch Gestaltungskonzepte weitere Merkmale ergänzt werden.
Gesettet werden im Quellcode die Parameter *size* innerhalb *geom_point*, dort wird die Größe der Punkte des Streudiagramms bestimmt, sowie die Parameter *fill* und nochmals *colour* innerhalb der *geom_smooth*-Funktion, durch denen Füllfarbe des Konfindenzintervalls und Farbe der Regressionskurve bestimmt wird. Es ergeben sich - wie bereits erwähnt - keine Veränderungen und Farb- oder Größenverläufe, da beim *Setten* von Konstanten auf keinerlei Zusammenhänge geachtet wird. [8, 13]

Die Wirkung des *Mappings* kann sehr unterschiedliche Auswirkungen auf den Plot haben und schnell zu einem Thema für fortgeschrittenere Programmierer werden. Hier ein Beispiel:

Codebeispiel 10: Mapping innerhalb ggplot

```
1 ggplot(diamonds, aes(x=carat, y=price, colour=cut))
  + geom_point() + geom_smooth(se=F)
```

Abbildung 13: Mapping innerhalb ggplot

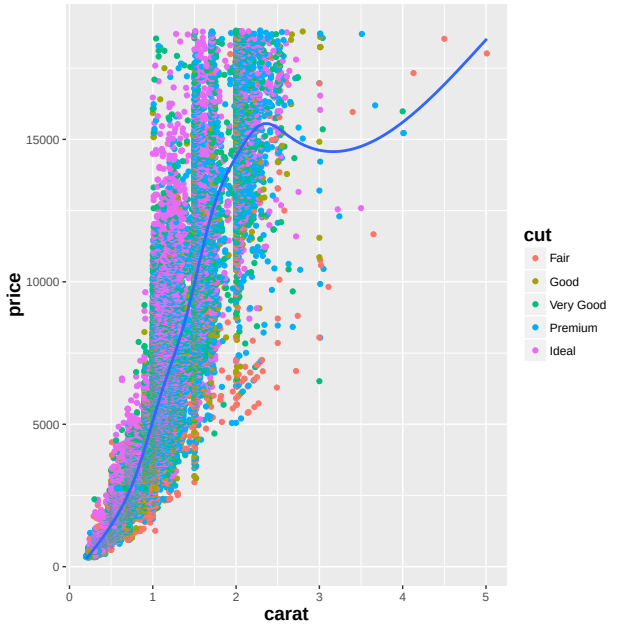


Die Graphik repräsentiert Diamanten, die nach Karat, Preis und Schliff unterteilt werden. *Gemappt* wird die Variable *cut* innerhalb der *ggplot*-Funktion, daraufhin wird jedem Schliff eines Diamanten eine bestimmte Farbe zugeordnet. Die *geom_smooth*-Funktion kalkuliert nun für jeden einzelnen Schliff die individuelle Regressionskurve in der entsprechenden Farbgebung.

Codebeispiel 11: Mapping innerhalb geom_smooth

```
ggplot(diamonds, aes(x=carat, y=price))
+ geom_point(aes(colour=cut)) + geom_smooth(se=F)
```

Abbildung 14: Mapping innerhalb geom_smooth



Obwohl sich innerhalb des Codes das *Mapping* nur von der der *ggplot*-Funktion in die *geom_point*-Funktion verlagert hat, ist das Ergebnis doch deutlich anders. Die Diamanten werden zwar noch immer durch verschiedene Farben in ihren Schliffen unterschieden, jedoch betrachtet *geom_smooth* alle Datenpunkte nun als Ganzes und berechnet lediglich eine Regressionskurve für den gesamten Datensatz.
Besonders das *Mapping* ist also ein mächtiges und facettenreiches Werkzeug, das meiner Meinung nach einige Übung benötigt, um ausreichend verstanden und angewandt zu werden.

5.2 Scaling und Achsenverhätnisse

Neben der Gestaltung des Datensatzes bietet GGPlot2 auch native Funktionen an mit deren Hilfe das Koordinatensystem samt Legende umgestaltet werden kann. Wichtig ist hierbei, dass man zwischen *stetigen* und *diskreten* Plots unterscheidet, da einige Funktionen nur auf Merkmale einer Art anwendbar sind. Funktionen, die sich ausschließlich auf stetige Merkmale beziehen können, sind meist gekennzeichnet durch ein „*continuous*“ innnerhalb des Funktionsnamens, bei diskreten Merkmale ist das Schlagwort „*discrete*“. Da die volle Bandbreite der verschiedenen Gestaltungsmöglichkeiten schwer zusammenzufassen sind, sollen hier nur anhand von Beispielen Funktionen bekannt gemacht und erklärt werden. [10, 11, 19]

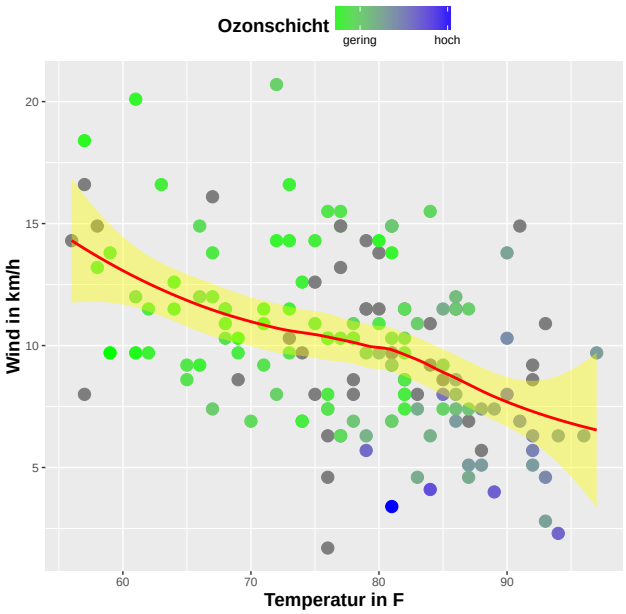
Beispiel: Eine über dem Plot positionierte Legende, mit blau-grünem Farbverlauf, benutzerdefiniertem Titel, Abschnittsgrößen und Abschnittsbezeichnungen sowie neue Achsenbezeichnungen der Koordinatenachsen sollen erstellt werden.

Codebeispiel 12: Umgestaltung der Legende

```
g1 + theme(legend.position="top") + scale_colour_continuous(
    name="Ozonschicht", breaks=c(40, 160), labels=c("gering",
    "hoch"), low="green", high="blue" ) + scale_x_continuous("
    Temperatur in F") + scale_y_continuous("Wind in km/h")
```

Innerhalb *g1* sei die bekannte Funktion des *airquality*-Datensatzes gespeichert, durch Hinzufügen der *theme*-Funktion wird die Legendenposition durch „*top*“ bestimmt. Neben *legend.position*, kennt *theme* noch eine Vielzahl an weiteren Parametern wie *legend.text*, *axis.text*, *axis.title* usw. [18]
scale_colour_continuous sorgt für die Legendengestaltung an sich. Über *name* wird der Legendentitel bestimmt, *breaks* erhält über einen Vektor die Anweisung die Legende in eine bestimmte Anzahl von Abschnitten an bestimmten Punkten zu unterteilen. [11, 19]
Über *labels* können dann diese Abschnitte mit Werten und Bezeichnungen versehen werden. *low* und *high* bestimmen den Farbverlauf der Legende und letztendlich wird von *scale_y_continuous* und *scale_x_continuous* für eine benutzdefinierte y- bzw. x- Achsenbezeichnung gesorgt. Achsenbezeichnungen stehen zwar nicht im direkten Zusammenhang mit Legenden, unterstützen aber die Lesbarkeit der Legende.

Abbildung 15: Benutzerdefinierte Legende



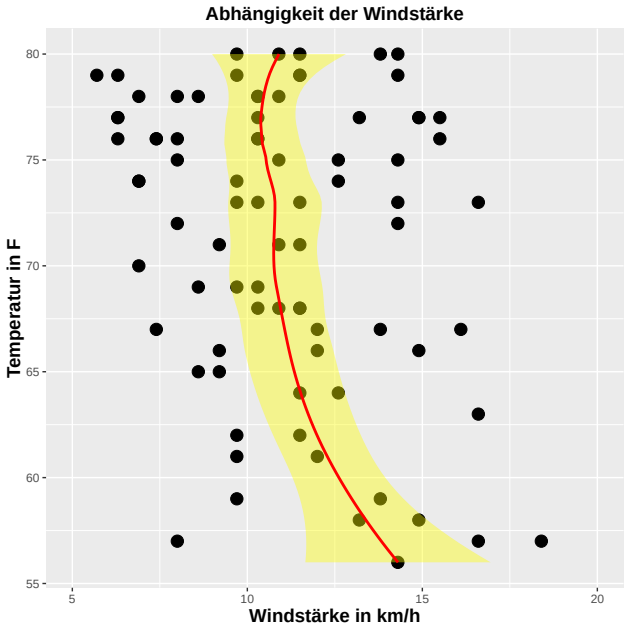
Analog zu den hier aufgeführten stetigen Funktionen existieren auch dieselben diskreten Funktionen, es muss lediglich „*continuous*“ im Funktionsnamen durch „*discrete*“ ausgetauscht werden, um diskrete Legenden oder Achsen zu benennen. [10]

Beispiel: Erstellen Plot mit Überschrift, benutzerdefinierten Achsenbezeichnungen, deren Achsenbegrenzung manuell festgelegt worden ist, und vertauschten Achsenabhängigkeiten.

Codebeispiel 13: Umgestaltung des Koordinatensystems

```
g1 + labs(title="Abhängigkeit der Windstärke",
x="Temperatur in F", y="Windstärke in km/h")
+ lims(x=c(NA, 80), y=c(5,20)) + coord_flip()
```

Abbildung 16: Benutzerdefiniertes Koordinatensystem



labs sorgt, wie der Name bereits verrät, für sämtliche Bezeichnungen des Plots. Innerhalb dieser Funktion können über *title*, *x* und *y* die Plotüberschrift sowie die Achsenbezeichnungen festgelegt werden (einzeln wäre dies auch möglich über die Funktionen *ggtitle*, *xlab* und *ylab*). *lims*, ebenfalls quasi selbsterklärend, kümmert sich um die Achsenbegrenzungen, welche als Vektor übergeben werden (*xlim* und *ylim* als einzelne Funktionen möglich). Durch das Verkleinern oder Vergrößern der Achsenbegrenzungen kann es passieren, dass wichtige Teile des Plots herausgeschnitten werden. Um dies zu verhindern, kann stattdessen auch die Funktion *coord_cartesian(xlim, ylim)* verwendet, welche in einen bestimmten Achsenabschnitt hereinzoomt. *coord_flip* sorgt letztendlich für das Vertauschen der Abhängigkeiten. x- und y-Achse werden einfach gewechselt. [10, 17]

5.3 Facets

Wie bereits erwähnt kann ein Plot sehr schnell komplex werden und damit auch unübersichtlich. *Facets* bieten in GGPlot2 ein weiteres Konzept, um die Durchschaubarkeit zu erhalten und sogar zu steigern. Um *Facets* verwenden zu können, sollte ein diskretes Merkmal verwendet werden. Zu jedem einzelnen Wert des diskreten Merkmals wird dann eine dazugehörige Graphik erstellt. Genutzt werden dafür die Funktionen *facet_grid*, die es sogar ermöglicht zwei diskrete Merkmale einzubinden, und *facet_wrap*, die zwar nur ein diskretes Merkmal sinnvoll einbinden kann, aber benutzerdefiniert auf die Anordnung der Plots reagieren kann. [8, 9]

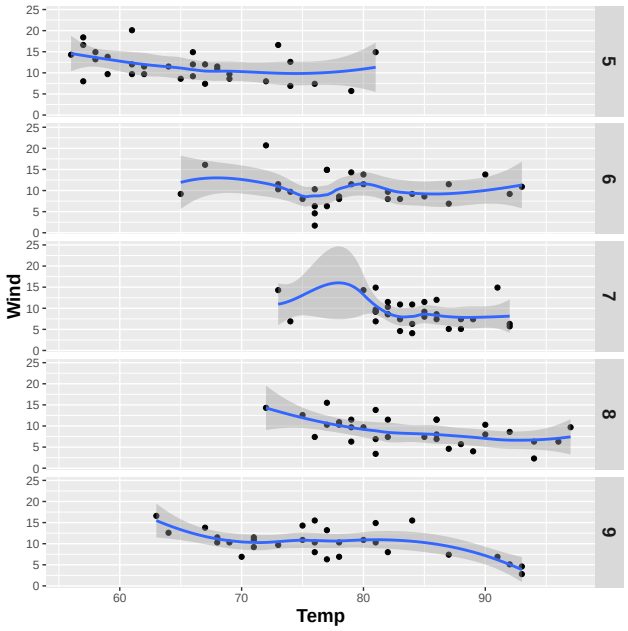
Beispiel: Es soll für jeden einzelnen Monat eine eigene Grafik im Plot angelegt werden, um die Regressionskurven miteinander zu vergleichen.

Codebeispiel 14: Vertikales facet_grid mit Month

```
ggplot(airquality, aes(x=Temp, y=Wind)) + geom_point()
+ geom_smooth() + facet_grid(Month~.)
```

Die Funktion *facet_grid* erwartet als Parameterübergabe mindestens ein Merkmal und darin inbegriffen die Anordnung des Plots. Die Notation *Month~ .* bedeutet, sortiere nach *Month* und ordne es vertikal an, dann sorge für kein Merkmal, das horizontal angeordnet werden soll („.“ ist das Füllsymbol für ein leeres Merkmal),

Abbildung 17: Vertikale Anordnung von *Month*

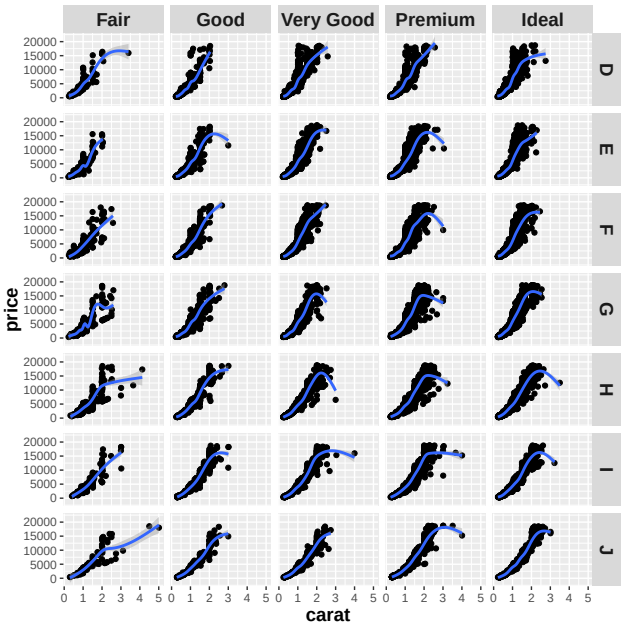


Das Lesen der Plots mag eine kurze Eingewöhnungszeit brauchen, aber alle Datenpunkte, die innerhalb eines Monats gemessen worden sind, stehen nun gemeinsam innerhalb einer Graphik und bilden ihre individuelle Regressionskurve. In vertikale Richtung gestaffelt, verlaufen die Monate 5 bis 9.

Auch sehr große Datensätze, verlieren nicht an Übersichtlichkeit bei der Nutzung von *facet_grid*. Hier einmal ein Beispiel wie es aussehen kann, wenn zwei diskrete Merkmale einem Plot hinzugefügt werden.

Die Abbildung 16 zeigt nochmals einen Ausschnitt aus dem *diamonds*-Datensatz, der nativ in GGPlot2 vorhanden ist. Dieser Plot ist von vier Variablen abhängig: *price*, *carat*, *cut* (horizontal) und *color* (vertikal). Dementsprechend würde die Funktion *facet_grid(color ~ cut)* zu unserem Code hinzugefügt werden, um das dargestellte Muster zu erreichen.

Abbildung 18: Horizontale und vertikale Anordnung zweier diskreter Merkmale



Neben *facet_grid* existiert die *facet_wrap*-Funktion. Mit Hilfe dieser kann der Programmierer eigenhändig bestimmen, wie viele Teilgraphiken eines Plots er innerhalb einer Zeile oder Spalte darstellen lassen möchte.

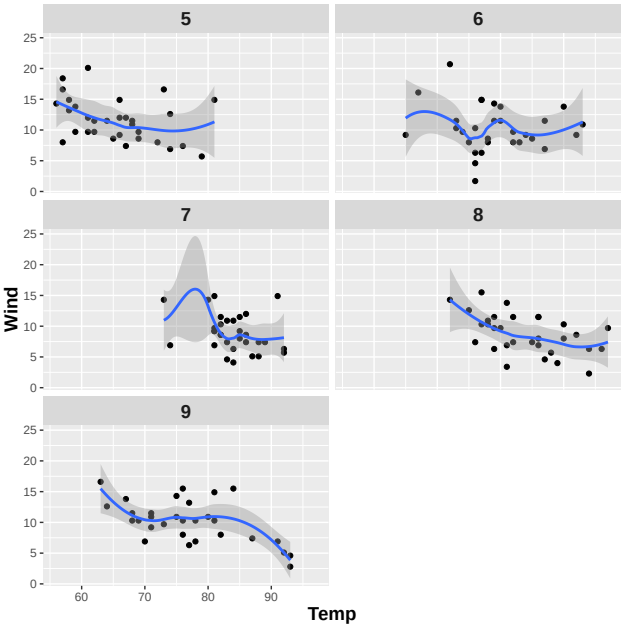
Beispiel: Erstellen eines Facet-Plots basierend auf dem *Month*-Merkmal, der immer genau 3 Graphiken in einer Spalte anzeigt.

Codebeispiel 15: facet_wrap mit 3 Graphiken pro Spalte

```
ggplot(airquality, aes(x=Temp, y=Wind)) + geom_point()
+ geom_smooth() + facet_wrap(~Month, nrow = 3)
```

Über *facet_wrap* wird der eine Parameter übergeben, an dem das *Facetting* ausgeführt werden soll. Dies erfolgt durch die Notation *~Month*, also ähnlich dem *facet_grid*. Es sollte aber darauf geachtet werden, dass *facet_wrap* in den meisten Fällen nur auf ein einziges diskretes Merkmal ausgelegt ist und daher kein Füllsymbol wie „.“ benötigt. Nach der Übergabe des Merkmals, wird über den Parameter *nrow* die Anzahl der Plots pro Spalte festgelegt, die pro Spalte aneinandergereiht werden sollen. Analog dazu existiert *ncol*, um die Anzahl an Plots pro Zeile festzulegen. [9]

Abbildung 19: Anordnung der Graphiken nach Spaltenanzahl



5 Sonstiges

6.1 Statistische Tranformation

Statistische Tranformation wird immer genau dann wichtig, wenn die standardmäßigen *geom*-Funktionen nicht mehr den Ansprüchen des Programmierers entsprechen. Grob beschrieben greift der Programmierer in die Zusammensetzung der Parameter einer Funktion ein und verändert sie nach seinem Belieben. Da aber die Zusammensetzung jeder einzelnen geometrischen Funktion individuell ist, kann auch in diesem Abschnitt nur sehr ausschnitthaft das Thema besprochen werden. [8, 13]

Beispiel: Erstellen eines Balkendiagrammes, das die Durchschnittstemperatur der verschiedenen Monate darstellt mit Hilfe der in Abschnitt 2.3 manipulierten Daten.

Codebeispiel 16: Fehlerhafter Code für ein Balkendiagramm

```
ggplot(airquality_means, aes(x=Month, y=meanTemp))+geom_bar()
```

```
Error: stat_count() must not be used with a y aesthetic.
```

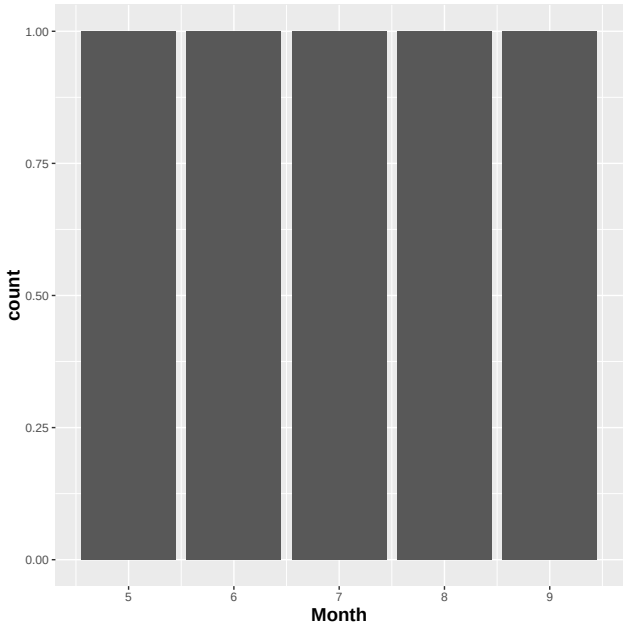
Bei dem Versuch nach den gewohnten Regeln ein Balkendiagramm zu erstellen, wird eine Fehlermeldung geworfen, die darüber informiert, dass ein Parameter namens *stat_count* nicht zusammen mit dem Parameter *y* verwendet werden darf. Dies ist ein gutes Beispiel dafür, dass die möglichen Parameter der *aes*-Funktion abhängig von der jeweiligen *geom*-Funktion ist.

Codebeispiel 17: Standardmäßiges Balkendiagramm

```
1 ggplot(airquality_means, aes(x=Month)) + geom_bar()
```

Durch das Entfernen des *y* Parameters kompiliert der Code schließlich problemlos. Das entstandene Schaubild mag möglicherweise überraschend wirken, denn auf der x-Achse stehen zwar die verschiedenen Monate aus *Month*, aber auch y-Achse hat automatisch ein Merkmal zugeordnet bekommen - *count*, ein Zähler wie viele Werte für einen bestimmten Monat vorliegen. Per default wird also ohne jegliches Zutun ein Wert an den *y* Parameter übergeben, dieser ist damit belegt. Codebeispiel 15 ist also fehlgeschla-

Abbildung 20: Standardmäßiges Balkendiagramm

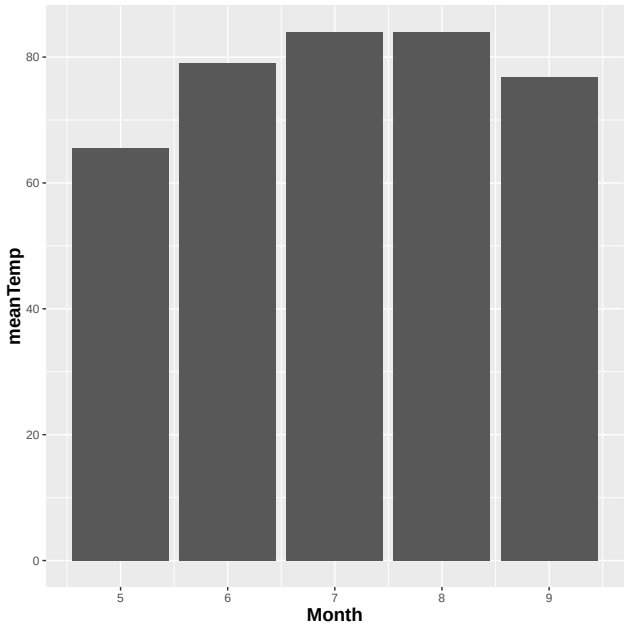


gen, weil zwei unterschiedliche Werte an einen einzigen Parameter übergeben wurden. [13]
Geändert werden kann dies, indem durch den Parameter *stat*, dessen Ausgangswert immer *„count “* ist, ein neuer Wert mit dem Schlüsselwort *„identity“* überschrieben wird.

Codebeispiel 18: Balkendiagramm mit benutzerdefinierter y-Achse

```
ggplot(airquality_means, aes(x=Month, y=meanTemp))
+ geom_bar(stat="identity")
```

Abbildung 21: Benutzerdefiniertes Balkendiagramm



Nun kann auch der gewünschte Plot erstellt werden und man kann festgestellt, dass im achten Monat die höchste durchschnittliche Temperatur herrschte.
Der sichere Umgang mit statistischen Transformationen erfordert etwas Einarbeitungszeit und eine gewisses Maß an Wissen über die verschiedenen geometrischen Funktionen und ihren Eigenheiten in GGPlot2. Dieses Gebiet ist sicher zu den fortgeschritteneren Themenbereichen im Plotting zu zählen.

6.2 Export von Plots

Nachdem Erstellen des gewünschten Plots kann dieser nun für Dokumente, Präsentationen oder ähnliches eingesetzt werden, dazu ist es aber vorher nötig diesen zu speichern und zu exportieren. *ggsave* ist eine in GGPlot2 vorhandene Funktion, die eben das bietet und den Vorgang simpel gestaltet.

Beispiel: Speichern des zuletzt erstellten Plots im pdf-Format.

Codebeispiel 19: Export als pdf-Datei

```
ggplot(airquality, aes(x=Temp, y=Wind)) + geom_point()
ggsave("meingraph.pdf")
```

GGPlot2 speichert, wenn nicht anders angegeben, immer automatisch den zuletzt erstellten Plot unter dem „Dokumente“-Pfad des Users ab. Der String *„meingraph.pdf“* ist nichts anderes als der Dateiname, unter dem der Plot gespeichert werden soll,

die Dateiendung bestimmt das Format, über den der Export stattfinden soll. Nach Ausführung des Quellcodes kann die Datei mit dem gespeicherten Inhalt sofort im jeweiligen Ordner gefunden und weiterverwendet werden.

Beispiel: Speichern eines vorherigen Plots im png-Format unter einem benutzerdefinierten Pfad.

Codebeispiel 20: Export als png-Datei unter bestimmten Pfad

```
g1 <-ggplot(airquality, aes(x=Temp ,y=Wind)) + geom_point()
g2 <-ggplot(airquality, aes(x= Ozone, y=Temp)) + geom_point()
ggsave("meingraph.png", g1, path="C:/Users/Max Mustermann/
    Desktop/Graphen")
```

Der Graph g1 soll exportiert werden. Dazu muss nach der Angabe von Dateinamen und -format, in das der Plot gespeichert werden soll, die Variable, durch die die Graphik aufgerufen werden kann, übergeben werden, um die Sicherung des zuletzt erstellten Plots zu vermeiden. Der Parameter *path* erwartet als String schließlich die Übergabe eines Datenpfades, in der der Plot *g1* abgelegt werden soll. Nach Ausführung des Quellcodes kann dann die png-Datei im Ordner „Graphen“ des Nutzers gefunden werden.
Neben diesen Parametern existieren noch weitere für *ggsave*. So ist es möglich über *device* nochmals explizit das Dateiformat anzugeben. Dies ist sinnvoll, wenn das geforderte Dateiformat GGPlot2 unbekannt ist und es deshalb die Dateiart nicht automatisch aus dem *title*-Parameter herausfiltern kann. Des Weiteren existieren *scale*, *width* und *height*, über welche Größenverhältnisse des Plots angepasst werden können, sowie *unit*, ein Parameter, der festlegt in welcher Maßeinheit die Größenanpassungen getroffen werden sollen. [4, 20]

6 Zusammenfassung

GGPlot2 lebt von seinem klaren Konzept und der Konsistenz, die es konsequent zeigt. Viele der Gestaltungsmöglichkeiten benötigen kaum Wissen über herkömmliche Programmierstrukturen und ermöglichen auch Einsteigern schon früh sehr komplexe Graphiken zu erstellen. Das Verständnis für das Plotting kommt fast intuitiv, ohne großes Nachdenken. Grundlagen wie der Aufbau der *ggplot-*, *aesthetics-* und *geom*-Funktion bilden mitsamt der verschiedenen Gestaltungsmöglichkeiten durch Einsatz von Konstanten, Variablen, *Facets* oder statistischer Tranformation einen Grundstein zum Ausbau dieses Wissens.
Und obwohl in diesem Paper die essentiellen Grundlagen und Gestaltungsmöglichkeiten besprochen wurden, so ist dies doch nur ein kurzer Einblick in die Möglichkeiten, die GGPlot2 seinem Nutzer bietet. Die volle Bandbreite an Optionen was Entwurf und Gestaltung von Plots betrifft, könnte ganze Bücher füllen. So bietet GGPlot2 noch eine Vielzahl an geometrischen Funktionen an, die einen ganz eigenen Umgang verlangen, oder ist über unzählige weitere Pakete erweiterbar. So stellt beispielsweise das Paket *ggthemes* unterschiedlichste Hintergundfarbgebungen und -thematiken zur Verfügung, um den Plot an bestimmte Verwendungszwecke und Schriftdokumente anzupassen (es existiert ein *ggtheme* Hintergund für farbenblinde Menschen). [12, 14] Teilweise sind Pakete, die GGPlot2 erweitern nochmals durch andere Pakete ergänzbar und so ergibt sich bald ein Baum aus Paketverzweigungen, deren Wurzel GGPlot2 ist. GGPlot2 ist so umfassend, dass es einzelnen Nutzern kaum möglich ist das gesamte Potenzial auszuschöpfen. Das Lernen und Erweitern des eigenen Wissens erfolgt stetig, während ein Plot entworfen wird, und diese Wissensaneignung allein ist unkompliziert und effektiv umsetzbar. Darin liegt der Reiz dieses Datenvisualisierungspakets. Auch in Zukunft wird GGPlot2 für die Statistik wahrscheinlich relevant bleiben. Allein die Tatsache, dass die R Community GGPlot2 allzeit aktualisiert und verbessert, lässt vermuten, dass in den nächsten Jahren GGPlot2's Komplexität noch weiter zunehmen wird. [1]

Abbildung 22: ggthemes Solarized

Literatur

[1] A grammar of graphics: past, present, and future
http://ggplot2.org/resources/2007-past-present-future.pdf

[2] Ggplot2
http://en.wikipedia.org/wiki/Ggplot2

[3] Why I use ggplot2, D. Robinson
http://varianceexplained.org/r/why-I-use-ggplot2/

[4] ggplot2 - Essentials
http://www.sthda.com/english/wiki/ggplot2-essentials

[5] Getting started with qplot
http://ggplot2.org/book/qplot.pdf

[6] Visualizing Data Using ggplot2, D. Robinson
http://varianceexplained.org/RData/lessons/lesson2/

[7] The fundamentals of ggplot2 explained, Alan
http://www.aridhia.com/technical-tutorials/
the-fundamentals-of-ggplot-explained/

[8] Build a plot layer by layer, H. Wickham
https://rpubs.com/hadley/ggplot2-layers

[9] Facets
http://www.cookbook-r.com/Graphs/Facets_%28ggplot2%29/

[10] Axes
http://www.cookbook-r.com/Graphs/Axes_%28ggplot2%29/

[11] Legends
http://www.cookbook-r.com/Graphs/Legends_%28ggplot2%29/

[12] Introduction to ggthemes, J. B. Arnold
https://cran.r-project.org/web/packages/ggthemes/vignettes/ggthemes.html

[13] Introduction to R Graphics with ggplot2
http://tutorials.iq.harvard.edu/R/Rgraphics/Rgraphics.html

[14] Package 'ggthemes', J. B. Arnold
https://cran.r-project.org/web/packages/ggthemes/ggthemes.pdf

[15] Textual annotation
http://docs.ggplot2.org/current/geom_text.html

[16] ggrepel Usage Examples, K. Slowikowski
https://cran.r-project.org/web/packages/ggrepel/vignettes/ggrepel.html

[17] ggplot2 Quick Reference: coord
http://sape.inf.usi.ch/quick-reference/ggplot2/coord

[18] Set theme elements
http://docs.ggplot2.org/current/theme.html

[19] Continuous position scales
http://docs.ggplot2.org/current/scale_continuous.html

[20] Save a ggplot (or other grid object) with sensible defaults
http://docs.ggplot2.org/current/ggsave.html

[21] Quick Introduction to ggplot2, E. Chen
http://blog.echen.me/2012/01/17/quick-introduction-to-ggplot2/

[22] Summarizing data
http://www.cookbook-r.com/Manipulating_data/Summarizing_data/

[23] plyr: Tools for Splitting, Applying and Combining Data, H. Wickham
https://cran.r-project.org/web/packages/plyr/index.html

[24] An introduction to reshape2, S. Anderson
http://seananderson.ca/2013/10/19/reshape.html

[25] Statistik für Betriebswirte I, A. Johannssen
Stetige und diskrete Merkmale
S. 10-12