Data Science/-Mining/-Machine Learning mit R für Aktuare

Eine kurze, praktische Einführung

Friedrich Loser, Okt. 2016

Inhaltsverzeichnis

1. Inhalt und Ziel

Was steckt hinter den aktuellen Begriffen und Entwicklungen im analytischen „Big-Data“-Umfeld? Diese Einführung soll beim Kennenlernen und Verstehen lernen helfen und nebenher die praktische Umsetzung der Methoden mit „R“ ermöglichen. Diese Einführung ist für das Selbststudium am privaten PC gedacht. Installationen, die in großen Unternehmen Monate dauern können, erledigen Sie so in Sekunden. Für das Verständnis der Methoden werden lediglich Grundkenntnisse in Linearer Algebra und Linearer Regression vorausgesetzt.

R ist eine Programmiersprache und Softwareumgebung für statistische Berechnungen und graphische Darstellungen. R ist frei verfügbar (GNU-Lizenz), im Bereich Data Science sehr verbreitet und hochaktuell. Viele neu entwickelte Verfahren wurden zuerst in R umgesetzt.

Anstelle des alten Begriffs Statistik sind zahlreiche neue Schlagworte auf dem Markt, einige davon befinden sich im Titel. Was ist daran so neu, was ist anders?

Die traditionellen statistischen Methoden stammen aus dem „Handrechenzeitalter“, das für die meisten Anwender erst vor weniger als 50 Jahren endete. Seit der Computerisierung der Datenanalyse ist mit stark steigender Rechenpower und Datenverfügbarkeit einiges anders:

- Resourcenabhängigkeit: Rechenintensive und datenhungrige Verfahren werden praktikabel
- Kreuzvalidierung: Der neue Goldstandard der Modellparameterbestimmung

- Overfitting vs. Generalisierung: Neue Probleme durch unüberschaubar große Modelle
- Modelldemokratie: Ensembles und Mehrheitsentscheidungen
- Simulation mit Zufallszahlen: Tausende Stichproben vs. traditionelle Verteilungsannahmen
- Viel hilft viel: Mehr Daten schlagen einen schlauen Algorithmus

Kurzum: Es geht um den Triumph der Daten über die Theorembeweise (J. Friedman, in StatLearning-SP).

Dementsprechend wird in den folgenden Beschreibungen weniger auf die Theorie und mehr auf die praktische Umsetzung mit R an echten und simulierten Daten eingegangen.

Abschnitt A soll einen Überblick geben, zu ersten Schritten mit R anleiten und am Ende zeigen, was aktuariell mit R möglich ist. In Abschnitt B wird eine möglichst breite und kleine Auswahl aus den sehr vielen Methoden des überwachten „Machine Learning" (ML) anhand konkreter R-Skripte vorgestellt. Je mehr Sie bereits wissen, umso mehr werden Sie vermissen. Freuen Sie sich über das, was neu und interessant für Sie ist.

R soll den Anwendern die Arbeit erleichtern und nicht den Computern. Dieser Grundsatz stößt bei größeren Datenmengen an seine Grenzen. Im Abschnitt C wird an aktuellen ML-Anwendungsfällen gezeigt, wie man diese Grenzen in Richtung „Big Data" verschieben kann.

Für einen schnellen Erfolg wird die Methode „erst ausprobieren, dann nachdenken und nachbohren" empfohlen. Viele Fragen lassen sich auf Basis der Beispieldaten und Programme gut beantworten.

2. Begriffe des Statistischen Lernens

Im Titel tauchen gleich drei neue Begriffe auf, die alle mit Datenanalyse und Statistik zu tun haben. Den recht neuen Begriff „Data Science" kann man vielleicht am ehesten in der Abgrenzung von der „Computer Science" (Informatik) verstehen. Der Fokus schwenkt von Computern und Programmen auf die Daten, also auf deren Aufbereitung und Haltung, Analyse und Prognose, Visualisierung und Nutzung. Die Business-Orientierung der „Data Science" ist ein wesentlicher Unterschied zur Statistik. Hat ein Statistiker eine Idee, schreibt er einen Artikel, hat ein Informatiker eine Idee, gründet er eine Firma, schreibt Friedman Ende der 90er Jahre in seinem Weckruf an die Statistiker, kritisiert deren einseitige Liebe zur Mathematik und fordert mehr Beteiligung am „Data Mining" um relevant zu bleiben, siehe www-stat.stanford.edu/~jhf/ftp/dm-stat.pdf . Das passt auch zur Data Science.

Data Mining hat den hohen Anspruch, neue Zusammenhänge und „Erkenntnisschätze" zu entdecken und ist als Begriff etwas aus der Mode gekommen. „Machine Learning" hingegen sucht keine Erkenntnisse, sondern gute datengetriebene Vorhersagen. Statistiker halten den Begriff „Statistical Learning" für passender. Dieses Lernen von den Daten kann grob in überwachtes und unüberwachtes Lernen eingeteilt werden:

- Beim überwachten Lernen wird der Lernalgorithmus an Beispielen mit bekanntem Ergebnis trainiert und dann bei neuen Daten zur Vorhersage verwendet. Dabei handelt es sich überwiegend um Regressions- und Klassifikationsprobleme, für deren Lösung eine Fülle verschiedener Methoden verwendet werden kann, siehe Abschnitt B.
- Beim unüberwachten Lernen gibt es keine vorherzusagende Zielgröße. Das Ziel ist die Beschreibung von Zusammenhängen und Mustern zwischen den beschreibenden Merkmalen.

Die beschreibenden Merkmale werden als „features" oder „input" bezeichnet, die Zielvariable als „label", „output" oder „target". Mit „training" wird der Prozess der Schätzung der Modellparameter bezeichnet. Es werden die aus der Regressionsrechnung bekannten Evaluationsfunktionen und Optimierungsverfahren verwendet. Auf Abweichungen wird bei den betroffenen Modellen eingegangen.

3. Quellen und weiterführende Links

Diese kurze Einführung kann die verwendeten Methoden nur skizzieren. Für die Vertiefung wird das anwenderorientierte Buch „An Introduction to Statistical Learning with Applications in R“ [ISLR] der Autoren James, Witten, Hastie und Tibshirani sehr empfohlen. Zur Vereinfachung des Nachschlagens sind die in den folgenden Kapiteln dargestellten Methoden und R-Skripte so weit wie möglich an die Konventionen und Beschreibungen in diesem Buch angelehnt.
Auf Basis dieses Buches bieten Trevor Hastie und Robert Tibshirani den sehr empfehlenswerten Online-Kurs *StatLearning-SP* an. Der Kurs, die Vorlesungsfolien und sogar eine pdf-Version des Buches sind aktuell frei verfügbar, siehe https://statlearning.class.stanford.edu/. Das gebundene Buch kann man natürlich auch kaufen, es ist sein Geld wert.

Einige Ergänzungen, vor allen zu Künstlichen Neuronalen Netzen, orientieren sich an der zweiten Auflage des etwas mathematischeren Lehrwerks „The Elements of Statistical Learning. Data Mining, Inference, and Prediction“ von Hastie, Tibshirani und Friedman (2008). Die pdf-Version dieses Klassikers ist 764 Seiten stark und hier zu finden: http://statweb.stanford.edu/~tibs/ElemStatLearn/. Die genannten Stanford-Professoren haben wesentliche wissenschaftliche Beiträge geleistet, stellen R-Funktionen zur Verfügung und zeigen die mathematischen Zusammenhänge zwischen den Methoden auf.

Einen gänzlich anderen Ansatz verfolgt das Buch „Machine Learning für Dummies“ von Mueller und Massaron (Juli 2016). Es enthält einige gute, einfache Erläuterungen und viele aktuelle Anwendungsbeispiele, die in R und Python umgesetzt sind.

Weitere interessante Online-Course zu den Themen Machine Learning und R werden zum Teil kostenlos von coursera.org und datacamp.com angeboten. Auf coursera.org kann an einem Kurs über Neuronale Netze von Geoffrey Hinton teilgenommen werden. Für MATLAB-Kenner könnte der Kurs https://www.coursera.org/learn/machine-learning von Andrew Ng interessant sein.

In den letzten Jahren hat sich die Wettbewerbsplattform kaggle.com als „home of data science“ profiliert. Für Hundertausende „Kaggler“ ist es auch eine Jobbörse mit Ausbildungs-, Forums- und Referenzfunktion. Die Teilnahme an den laufend angebotenen Wettbewerben ist kostenlos, die Preisgelder liegen gegenwärtig zwischen 25.000 und 50.000 USD. Folgende Wettbewerbe mit Versicherungsdaten haben bisher stattgefunden:

- Allstate Claim Prediction Challenge (2011)
- As the World Churns (Deloitte 2013): Which customers will leave an insurance company?
- Liberty Mutual Group (2014): Predict expected fire losses for insurance policies
- Homesite Home Insurance (2015): Which customers will purchase a quoted insurance plan?
- Prudential Life Insurance (2016): Can you make buying life insurance easier?
- State Farm (August 2016): Can computer vision spot distracted drivers?

Aktuell läuft der Allstate-Rekrutierungswettbewerb „How severe is an insurance claim“.

Für abgeschlossene Wettbewerbe werden die besten Lösungen auf http://blog.kaggle.com/ vorgestellt. Neben echten, harten Wettbewerben bietet kaggle.com auch Übungswettbewerbe wie „Titanic: Learning from Disaster“ an und hostet zahlreiche gute mit R umgesetzte Tutorials.

Eine interessante Informationsquelle zu den Themen „Data Mining, Analytics, Big Data, and Data Science“ ist KDnuggets, sozusagen Bild für Big Data. Dort findet man Artikel zu neuer Software, zu Studien, Kongressen und zu den wirklich interessanten Fragen wie „Verdient man mehr mit R oder Python?“ oder „20 Fragen, mit denen man einen Fake Data Scientist im Bewerbungsgespräch erkennt“. In den aktuellen Top-Listen findet man auch eine Empfehlung und Kurzbeschreibung des vielgeklickten Videos „ Introduction to Data Science with R“ von David Langer, siehe http://www.kdnuggets.com/2016/10/top-10-data-science-videos-youtube.html .

4. R-Umgebung erzeugen, R kennen lernen und ausprobieren

Mit folgenden Schritten kann in wenigen Minuten eine komfortable R-Umgebung aufgebaut werden:

a) Die aktuelle R-Version von https://cran.r-project.org/bin/windows/base/ herunterladen und als 64-Bit-Variante installieren (gibt es auch für Linux und OS X).

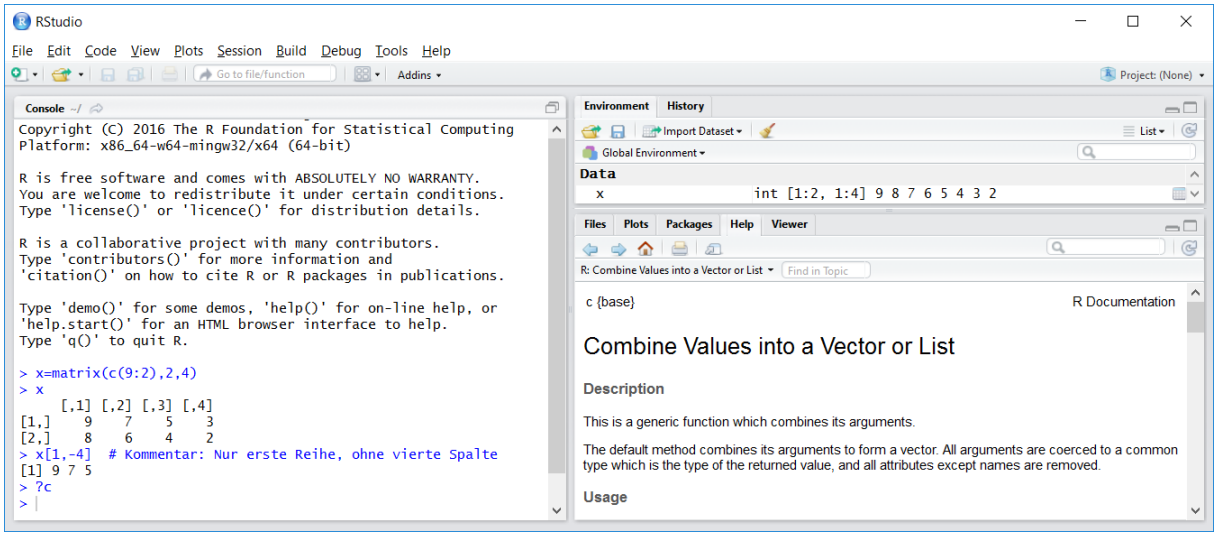
b) Die ebenfalls kostenlose integrierte Entwicklungsumgebung und graphische Benutzeroberfläche RStudio von https://www.rstudio.com herunterladen und installieren.

Nach dem Start von RStudio können im Console-Fenster R-Kommandos eingegeben werden:

- Die Benutzereingaben erfolgen nach dem Prompt **>** (unten blau hervorgehoben)
- Die Befehlszeilen werden durch die Enter-Taste ausgeführt
- Mit den Pfeil-Tasten kann durch die zuletzt eingegebenen Befehle navigiert werden
- Die auf ein # folgenden Zeichen werden nicht interpretiert (Kommentierung)
- Durch ?Funktionsname wird die Beschreibung im Hilfefenster angezeigt

R unterscheidet zwischen Groß- und Kleinschreibung. Wertzuweisungen erfolgen in R traditionell mit '<-'. Seit Langem ist auch '=' möglich. In Benennungen wird gerne '.' anstelle von '_' verwendet.

In R ist alles, was existiert, ein Objekt und alles, was passiert, ein Funktionsaufruf (sagte J. Chambers). Als erstes Objekt legen wir eine Matrix x an. Über Klammern können einzelne Elemente und Bereiche dieser Matrix angesprochen werden:



Übung: Wenden Sie auf x die Funktionen `mean`, `median`, `min`, `max`, `sd`, `var(as.vector(x))` und `sort` an.

Ein weiteres wichtiges Objekt ist der data frame. Damit können unterschiedliche Merkmalstypen, wie sie in Datensätzen regelmäßig vorkommen, in einem Objekt gespeichert werden. Die Elemente können wie bei einer Matrix angesprochen werden.
Dazu bitte die folgenden Anweisungen (Skript) kopieren, in die Console einfügen und Enter drücken:

```
##R 4.1  Werden die folgenden Zeilen in RStudio nur als eine Zeile eingefügt? Tipp: Einen anderen pdf-Reader (z.B. Foxit-Reader) verwenden
dat <- data.frame(sex=c("m","f","m","f"),age=c(23,45,67,89),class=c("1","3","3","2"))
dat
ran <- c(1,3,4)
dat[ran,]        # Auswahl von Datensätzen
dat[-ran,]       # Komplement dazu (z.B. Testdatei)
dat[,-2]         # Ohne zweites Merkmal (age)
##t (Ende R-Skript, weiter im Text. R-Skripte können über den Suchtext "##R" angesteuert werden)
```

Für ein erstes Ausprobieren ist die Console gut geeignet. Regelmäßige Nutzer legen über das „File"-Menü eine neue Datei an oder öffnen bereits existierende R-Skripte. Ein viertes Fenster erscheint nun links oben. Der Code von oben kann nun dort eingefügt werden. Die Zeile, in der sich der Cursor befindet oder alternativ ein markierter Bereich kann dann ausschnittweise über den "Run"-Button (oder Strg+Enter) ausgeführt werden.

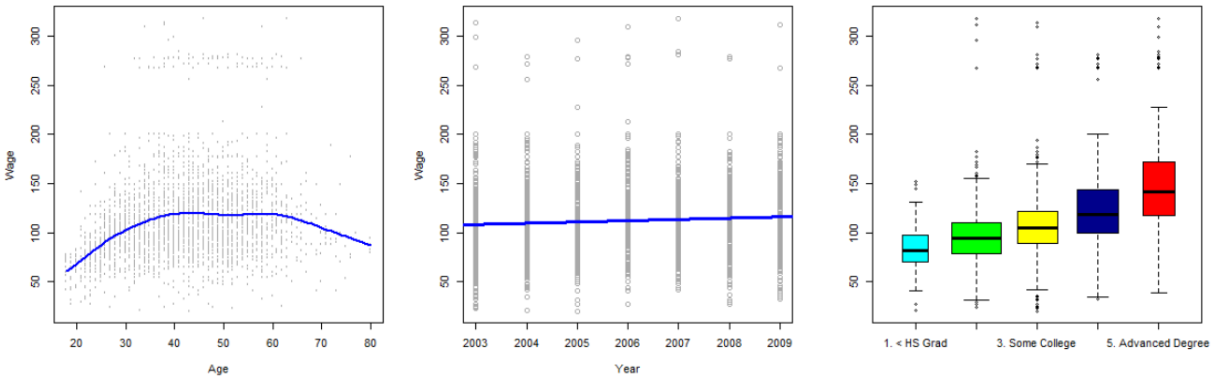
Als nächstes wollen wir Funktionsaufrufe kennen lernen und ein paar Zufallszahlen erzeugen:

```
##R 4.2 Tipp: Wenn sie einen guten pdf-Reader haben, können Sie das komplette Dokument markieren, in RStudio einfügen und ausschnittweise ausführen
set.seed(3)                     # Startwert setzen (für reproduzierbare Ergebnisse sorgen)
x=rnorm(100)                    # 100 standardnormalverteilte Pseudozufallszahlen erzeugen
sum(x)                          # Summe (Nahe Null wäre gut)
length(x)                       # Länge des Vektors (100)
y=x+rnorm(100,mean=50,sd=.2)
cor(x,y)                        # Korrelationkoeffizient
plot(x,y)
##t
```

Die oben verwendeten Funktionen werden bereits mit der R-Installation mitgeliefert. Jeder R-Nutzer kann weitere, eigene Funktionen definieren und ausführen.
Eine Vielzahl von Funktionen kann gebündelt über Bibliotheken, sogenannte Packages geladen werden. Sehr einfach geht das in RStudio über das Menü „Tools“, Unterpunkt „Install Packages …“. Dort jetzt `ISLR` eingeben und das Package ISLR installieren. Dieses Package enthält unter anderem die Datei `Wage`, die wir gleich benötigen.

Ein wichtiger Aspekt der Datenwissenschaft ist die Visualisierung von Daten und Zusammenhängen. Das folgende R-Skript berechnet für US-amerikanische Einkommensdaten einen altersabhängigen Smoothing Spline sowie eine lineare zeitliche Trendlinie und stellt diese zusammen mit weiteren Merkmalen graphisch dar.

```
##R 4.3
library(ISLR)
attach(Wage)        # Nun kann Prefix Wage$ bei Variablennamen weggelassen werden
par(mfrow=c(1,3))   # drei Grafiken nebeneinander

# 1. Altersabhängigkeit mit smoothing spline
plot(age, wage, xlab="Age", ylab="Wage" ,cex=.5, col="darkgrey ")
fit=smooth.spline(age, wage ,df=7)
lines(fit, col="blue", lwd=2)

# 2. Zeitabhängigkeit mit linearem Modell (Funktion lm)
plot(year, wage, col="darkgrey ", xlab="Year", ylab="Wage")
lm.fit=lm(wage ~ year)
abline(lm.fit, lwd=3, col="blue")

# 3. Bunter Boxplot zur Ausbildungsabhängigkeit
plot(education, wage, col=c("cyan","green","yellow","darkblue","red"), varwidth =T)
detach(Wage) # verhindert spätere Fehlermeldungen „The following objects are masked from ...“
##t
```



Im Buch [ISLR] befindet sich auf Seite 2 eine recht ähnliche Darstellung (ohne R-Skript).

Die oben verwendeten statistischen Funktionen `lm` und `smooth.spline` sowie die im weiteren Verlauf benutzen Funktionen `kmeans`, `prcomp` und `glm` sind Teil der Bibliothek `stats`, die zum Basislieferumfang von R gehört. Eine lange Liste der stats-Funktionen erhält man mit folgender Anweisung:
`library(help="stats")`

5. Titanic-Daten einlesen, visualisieren und aufbereiten

Als erstes Klassifikationsbeispiel werden wir die Überlebenschancen der Passagiere beim Untergang der Titanic untersuchen. Die Passagierdaten können von https://www.kaggle.com/c/titanic/data heruntergeladen werden. Wer sich nicht gleich registrieren möchte, kann diese Dateien über das Tutorial von Curt Wehrley auf github.com beziehen. Ein weiteres, auf kaggle.com hervorgehobenes Tutorial stammt von Trevor Stephens. Die im Folgenden knapp dargestellten Schritte können in diesen Titanic-R-Tutorials ausführlich nachvollzogen werden.

Die beiden heruntergeladenen Trainings- und Testdateien (siehe Validierungskonzept in Kap. 10) wollen wir nun als data frames in R einlesen und einen Blick auf und in die Daten werfen:

```
##R 5.1
test  <- read.csv("D:/downloads/titanic_test.csv")        #  Pfad/Name anpassen. Windows:
train <- read.csv("D:/downloads/titanic_train.csv")       #       "\" durch "/" ersetzen.
dim(test)     # Anzahl Datensätze und Merkmale anzeigen
dim(train)
str(train)    # Struktur des data frame anzeigen
head(train)   # Die ersten Zeilen andrucken
##t
```

Die Trainingsdatei umfasst demnach 891 Passagiere (Zeilen) und 12 Merkmale (Spalten) und beginnt mit folgenden Zeilen:

```
> head(train)   # Die ersten Zeilen andrucken
  PassengerId Survived Pclass                                                 Name    Sex Age SibSp Parch           Ticket    Fare Cabin Embarked
1           1        0      3                              Braund, Mr. Owen Harris   male  22     1     0        A/5 21171  7.2500              S
2           2        1      1 Cumings, Mrs. John Bradley (Florence Briggs Thayer) female  38     1     0         PC 17599 71.2833   C85        C
3           3        1      3                               Heikkinen, Miss. Laina female  26     0     0 STON/O2. 3101282  7.9250              S
4           4        1      1       Futrelle, Mrs. Jacques Heath (Lily May Peel) female  35     1     0           113803 53.1000  C123        S
5           5        0      3                             Allen, Mr. William Henry   male  35     0     0           373450  8.0500              S
6           6        0      3                                     Moran, Mr. James   male  NA     0     0           330877  8.4583              Q
```

Die Testdatei ist typischerweise etwas kleiner und umfasst 418 Passagiere und nur 11 Merkmale, da das fehlende Merkmal `Survived` für diese Daten vorhergesagt und als Wettbewerbsbeitrag auf kaggle.com hochgeladen werden soll.

Das bekannte Motto „Frauen und Kinder zuerst“ könnte beim Untergang der Titanic eine Rolle gespielt haben. Daher starten wir mit den Überlebenshäufigkeiten nach Geschlechtern:

```
##R 5.2
attach(train)              # Nun kurze Variablennamen (z.B. Sex statt train$Sex) möglich
table(Sex, Survived)                  # Absolute Häufigkeiten
prop.table(table(Sex, Survived),1)    # Relative Häufigkeiten (1: Zeilensumme=100%)
aggregate(Survived ~ Pclass + Sex, data=train, FUN=function(x) {sum(x)/length(x)})
##t
```

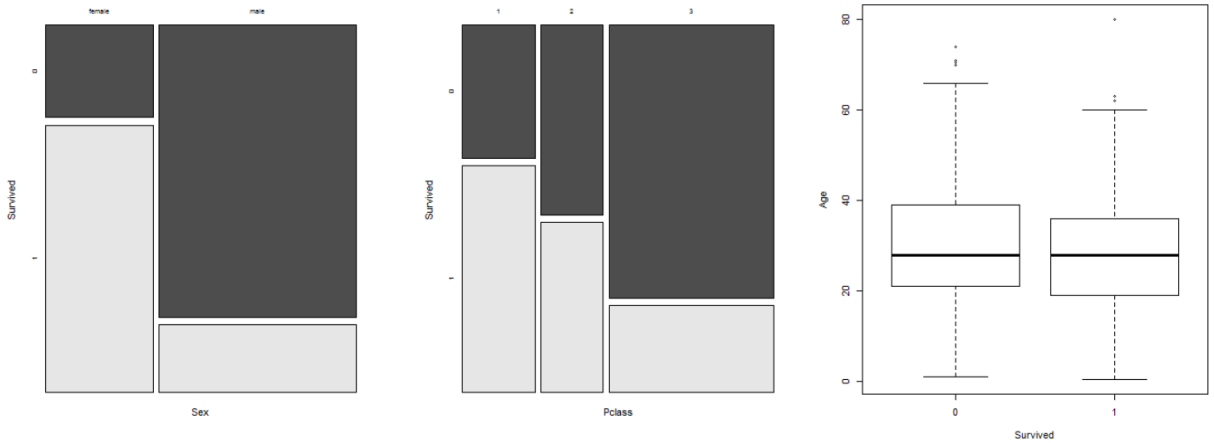
```
> table(Sex, Survived)                # Absolute Häufigkeiten
        Survived
Sex        0   1
  female  81 233
  male   468 109
> prop.table(table(Sex, Survived),1)   # Relative Häufigkeiten (1: Zeilensumme=100%)
        Survived
Sex            0         1
  female 0.2579618 0.7420382
  male   0.8110919 0.1889081
> aggregate(Survived~ Pclass + Sex, data=train, FUN=function(x) {sum(x)/length(x)})
  Pclass    Sex  Survived
1      1 female 0.9680851
2      2 female 0.9210526
3      3 female 0.5000000
4      1   male 0.3688525
5      2   male 0.1574074
6      3   male 0.1354467
```

Danach haben nur 19% der Männer, jedoch 74% der Frauen insgesamt, in der ersten Klasse sogar 97% der Frauen überlebt.

Die Überlebenshäufigkeiten können über verschiedene Plot-Funktionen visualisiert werden. Nach Installation des Package `vcd` kann die Funktion `mosaicplot` verwendet werden.

```
##R 5.3
library(vcd)                                  # für mosaicplot
par(mfrow=c(1,3))
mosaicplot(Sex ~ Survived, main="", shade=F, color=T)
mosaicplot(Pclass~Survived,main="", shade=F, color=T)
boxplot(Age ~ Survived, main="",xlab="Survived", ylab="Age")
detach train
##t
```

Die Mosaicplots visualisieren hier neben der Überlebenshäufigkeit auch die Gruppengröße (Breite):



Perfekte Daten kommen nur in Lehrbeispielen vor. Reale Daten enthalten fehlerhafte Angaben, fehlende Angaben, viel zu detaillierte Angaben sowie Angaben, die weiter aufbereitet und in einen fachlichen Zusammenhang gebracht werden sollten, zum Beispiel:

- aus GPS-Koordinaten Entfernungen berechnen
- aus Einwohnerzahlen und Gemeindeflächen die Bevölkerungsdichte berechnen
- aus Diagnosen mit Grouper-Software Krankheitsgruppe und Schweregrad ermitteln
- bei Umsatzprognosen Anzahl der Arbeitstage mit einem „Ferienfaktor“ gewichten
- aus Textfeldern auswertbare Merkmale ableiten

R ist für dieses sogenannte „feature engineering“ gut geeignet.

Fehlende Angaben werden in R durch „NA“ gekennzeichnet und können über die Funktion is.na() gefunden werden. Auf diese Weise könnten Datensätze ohne Altersangabe entfernt werden: `Train <- train[!is.na(train$Age),]  # .` Bei Dateien mit vielen Merkmalen kann das allerdings dazu führen, dass nur noch wenige bis gar keine Datensätze mehr übrig bleiben. Daher werden fehlende Werte oft durch passende (Mittel-)Werte ersetzt.

Im folgenden Beispiel soll aus dem vollen Namen der Titanic-Passagiere der Titel abgeleitet werden*. Da sich das feature engineering zwischen den Trainings- und Testdaten nicht unterscheiden darf, werden die beiden Datensätze zuerst zusammengeführt. Im Namensfeld (siehe oben) befindet sich der Titel zwischen dem Nachnamen und dem Vornamen und wird praktischerweise durch die Trennzeichen “,“ und “.“ eingerahmt. Daher kann mit der Funktion `strsplit()` der Name recht einfach in mehrere Teile zerlegt werden. So funktioniert das für den ersten Passagier:

```
##R 5.4
test$Survived <- NA                    # In der Testdatei das Leerfeld Survived anlegen
full <- rbind(train, test)             # Trainings- und Testdatei aneinanderhängen
full$Name <- as.character(full$Name)   # Name zu Textfeld machen
full$Name[1]                           # „Braund, Mr. Owen Harris“ auswählen
strsplit(full$Name[1], split='[,.]')   # ... seinen Namen splitten
strsplit(full$Name[1], split='[,.]')[[1]][2]  # ... das 2. Namenselement abgreifen („Mr.“)
##t
```

Das soll nun auf alle Datensätze angewandt werden. Dazu wird die letzte Zeile als Funktion über eine Vektorliste ausgeführt:

```
##R 5.5
full$Title <- sapply(full$Name, FUN=function(x) {strsplit(x, split='[,.]')[[1]][2]})
full$Title <- sub(' ', '', full$Title)     # noch 1. Leerzeichen entfernen
table(full$Sex, full$Title)                #  ... und das Ergebnis ansehen
##t
```

Diese Titel können weiter zusammengefasst und schließlich in den Prognosemodellen verwendet werden.

* Kurzfassung für Eilige: `full$Title=gsub('(.*,)|(\\..*)','',full$Name)`

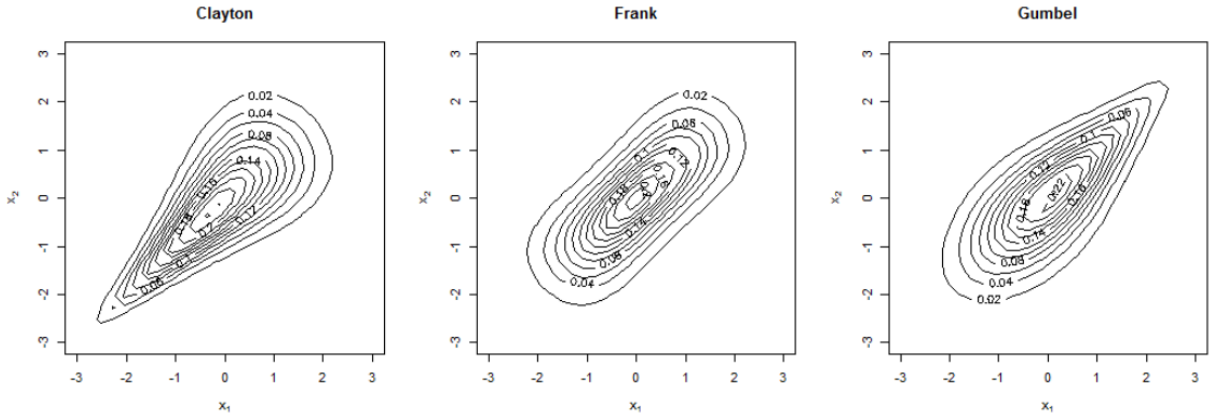
6. R für den Aktuaralltag: Excel, SAS und Funktionen für Aktuare

Im vorangegangenen Kapitel wurden csv-Dateien in R eingelesen. Umgekehrt kann man mit R natürlich auch csv- und txt-Dateien erzeugen und in Excel einlesen. Es geht aber auch bequemer. Über das Excel-Add-In RExcel ist sogar eine direkte Excel-R-Interaktion möglich.

SAS-Nutzer können inzwischen R in IML sowie in den Enterprise Miner einbinden und so in R umgesetzte neue Verfahren im SAS-Prozessfluss verwenden und vergleichen, näheres siehe sas.com. Über das Package `foreign` können SAS-, SPSS- und Stata-Dateien in R gelesen werden.

Für aktuarielle Anwendungen gibt es einige R-Packages: `lifecontingencies` richtet sich an Life-Actuaries; `actuar` enthält eine Vielzahl aktuarieller Funktionen aus den Gebieten Risikotheorie, Credibility-Theorie und eine Fülle endlastiger Verteilungen.
Für die Modellierung von Abhängigkeitsstrukturen stellt das Package `copula` einige nützliche Funktionen und Verfahren für gängige elliptische und archimedische Copulas bereit. Als kleines Beispiel erzeugen wir hierzu Contour-Plots (jeweils τ=0,5):

```
##R 6.1
library(copula)
c1=mvdc(copula = archmCopula(family = "clayton", param = 2), margins=c("norm","norm"),
        paramMargins=list(list(mean = 0,sd = 1), list(mean=0,sd= 1)))
c2=mvdc(copula = archmCopula(family ="frank",param = 5.736), margins=c("norm","norm"),
        paramMargins=list(list(mean = 0,sd = 1), list(mean=0,sd= 1)))
c3=mvdc(copula = archmCopula(family = "gumbel",  param = 2), margins=c("norm","norm"),
        paramMargins=list(list(mean = 0,sd = 1), list(mean=0,sd= 1)))
par(mfrow=c(1,3))
contour(c1,dMvdc,main="Clayton Copula",xlim = c(-3, 3), ylim = c(-3, 3))
contour(c2,dMvdc,main="Frank Copula",  xlim = c(-3, 3), ylim = c(-3, 3))
contour(c3,dMvdc,main="Gumbel Copula", xlim = c(-3, 3), ylim = c(-3, 3))
##t
```



Für Schaden-Aktuare könnte das Package `ChainLadder` interessant sein. Das Run-Off-Dreieck `MW2014` stammt aus „Claims Run-Off Uncertainty: The Full Picture" von Merz & Wütherich (2014) und ist zusammen mit folgendem Beispiel im Package enthalten:

```
##R 6.2
library(ChainLadder)
W <- MackChainLadder(MW2014, est.sigma="Mack")
plot(W)
CDR(W)
##t
```

Diese wenigen Zeilen sorgen für mehr Plots und Ergebnisse, als wir hier darstellen können. Das Package `ChainLadder` enthält etliche weitere Datensätze, Funktionen und Verfahren.

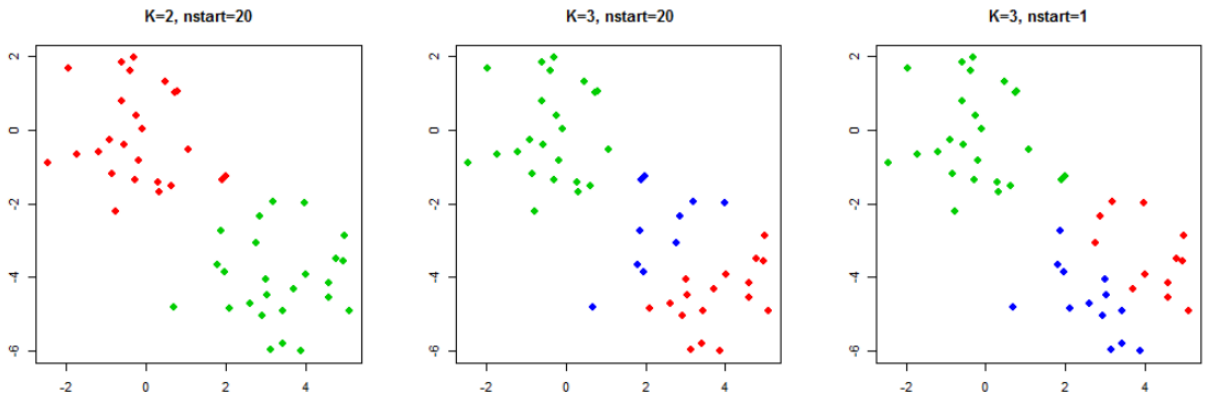
Für fast alle Packages gibt es auf cran.r-project.org ein Benutzerhandbuch sowie eine etwas ausführlichere Dokumentation.

7. Unüberwachtes Lernen: Clusteranalyse und Hauptkomponentenanalyse

Die Güte einer gefundenen Lösung kann beim unüberwachten Lernen nicht nach ähnlich objektiven Kriterien wie beim überwachten Lernen bewertet werden. Unüberwachtes Lernen ist daher oft Teil der explorativen Datenanalyse und wird zur Dimensionsreduzierung und Gruppenbildung eingesetzt.

Für das Auffinden möglichst homogener Untergruppen in den Daten gibt es eine Vielzahl unterschiedlicher Clusterverfahren. Recht einfach und schnell ist das k-means-Verfahren. Dabei wird zunächst jeder Beobachtung eine zufällige Clusternummer von 1 bis k zugeordnet. Iterativ wird für jeden der k-Cluster der Schwerpunkt berechnet und dann jede Beobachtung dem am wenigsten entfernt liegenden Schwerpunkt zugeordnet. Optimierungsziel ist eine möglichst geringe Varianz innerhalb der Cluster. Der beschriebene Algorithmus kann in einem lokalen Minimum stecken bleiben und muss daher häufig genug wiederholt werden. Als beste Lösung wird diejenige mit der geringsten within-Varianz ausgewählt. Ein Beispiel mit zwei simulierten Clustern:

```
##R 7.1                                                    Quelle: ISLR, Ch 10.5
set.seed(2)
x=matrix(rnorm(50*2),ncol=2)
x[1:25,1]=x[1:25,1]+3           # Zweiten Cluster nach rechts verschieben
x[1:25,2]=x[1:25,2]-4           # Zweiten Cluster nach unten verschieben

# k-means-Clusteranalyse mit K=2 und 20 Läufen
km.out=kmeans(x,2,nstart =20)   # Die Clusteranalyse durchführen
km.out$cluster                  # enthält Clusterzuordnungen
par(mfrow=c(1,3))               # drei Grafiken nebeneinander
plot(x, col=(km.out$cluster +1), main="K=2, nstart=20", xlab="", ylab="",pch=20,cex=2)

# Vergleich mit K=3 und 20 Läufen
km3a=kmeans(x,3,nstart =20)
km3a$tot.withinss               # Ergebnis:  97.9793
plot(x, col=(km3a$cluster +1), main=" K=3, nstart=20", xlab="", ylab="", pch=20, cex=2)

# dazu Vergleich mit nur einem Lauf
set.seed(3)                     # dem Zufall nachhelfen ...
km3b=kmeans(x,3,nstart =1)
km3b$tot.withinss               # Ergebnis: 104.3319
plot(x, col=(km3b$cluster +1), main="K=3, nstart=1", xlab="", ylab="", pch=20, cex=2)
##t
```

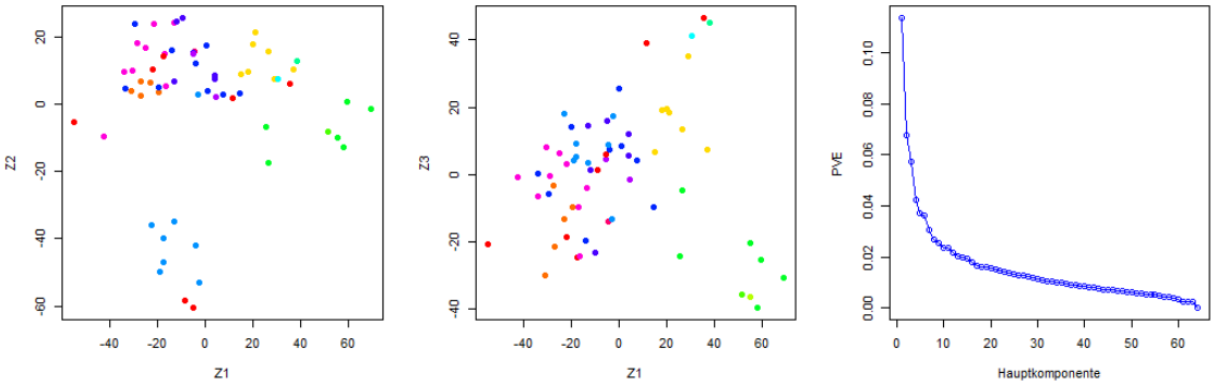


Ein weiteres populäres Verfahren des unüberwachten Lernens ist die Hauptkomponentenanalyse (PCA): Die Hauptkomponenten sind normalisierte Linearkombinationen aller Merkmale. Die erste Hauptkomponente ist diejenige mit der höchsten Varianz, jede weitere Hauptkomponente steht orthogonal zu den zuvor berechneten Hauptkomponenten. Die Hauptkomponenten sind also unkorreliert. Dieses Verfahren hat das Potential, bei Datensätzen mit sehr vielen, hoch korrelierten Merkmalen einen großen Teil der Varianz mit wenigen Dimensionen zu erklären.

Wir zeigen das nun am Datensatz NCI60, der 64 Krebszelllinien mit jeweils 6.830 Messwerten (p) zu Genexpressionen enthält. Die Dimensionsanzahl ist also drastisch höher als die Anzahl der Fälle. Man

kann davon ausgehen, dass zahlreiche Merkmale hoch korreliert sind. Eine Visualisierung von Zusammenhängen ist aufgrund der extrem hohen Zahl an möglichen Scatterplots jedoch kaum durchführbar. Die Hauptkomponenten der Messwerte werden hier mit der Funktion `prcomp` berechnet. Das Merkmal Krebsart wird dabei nicht verwendet. Für die wichtigsten Hauptkomponenten werden Scatterplots erstellt und Zusammenhänge visualisiert. Für die Einfärbung der Plots nach Krebsarten legen wir die Funktion `Cols` an.

```
##R 7.2                                                    Quelle: ISLR, Ch 10.6
library (ISLR)                          # Enthält die NCI60-Daten.
nci.labs=NCI60$labs                     # labs: Krebsart
nci.data=NCI60$data                     # data: Messwerte
dim(nci.data)                           # Anzahl Spalten und Zeilen anzeigen
table(nci.labs)                         # Verteilung der Krebs-Typen
pr.out=prcomp(nci.data , scale=TRUE) # Hauptkomponenten berechnen
summary(pr.out)                         # Std und Varianzanteil für 64 Komponenten
Cols=function(vec) { cols=rainbow(length(unique(vec)))
  return(cols[as.numeric (as.factor(vec))]) }
par(mfrow=c(1,3))
plot(pr.out$x[,1:2], col=Cols(nci.labs), pch=19, xlab="Z1",ylab="Z2")
plot(pr.out$x[,c(1,3)], col=Cols(nci.labs), pch=19, xlab="Z1",ylab="Z3")
pve=pr.out$sdev^2/sum(pr.out$sdev^2)      # Anteil erklärter Varianz
plot(pve, type="o", ylab="PVE", xlab="Hauptkomponente", col="blue")
##t
```



Die 14 verschiedenen Krebstypen sind durch unterschiedliche Farben dargestellt. Im linken und mittleren Bild kann man Clusterbildungen bei gleichen Krebsarten erkennen. Im rechten Bild ist der Anteil der erklärenden Varianz für die Hauptkomponenten 1 bis 64 dargestellt. In Summe erklären die ersten drei Hauptkomponenten 20%, die ersten 12 Hauptkomponenten 50% der Varianz.

Die Hauptkomponenten könnten nun als erklärende Merkmale (anstelle der Originalmerkmale) in einem Prognoseverfahren verwendet werden. Das kann bei Regressionsrechnungen mit hoch korrelierten Merkmalen zu deutlich stabileren und übersichtlicheren Modellen führen.

Die Hauptkomponenten werden mit der Methode der Singulärwertzerlegung berechnet. Die Singulärwertzerlegung spielt auch eine wichtige Rolle bei der Berechnung von personalisierten Empfehlungen, siehe Kapitel 17.

8. Regression: Multivariate lineare Regression, Interaktionen und Polynome

Die weiteren Kapitel stellen Verfahren zum überwachten Lernen vor. Um von bekanntem Grund zu starten, beginnen wir mit einer multivariaten Regression. Dazu muss zunächst das R-Package `MASS` installiert werden. Dieses Package enthält eine große Sammlung an Datensätzen und Funktionen, darunter die Datei `Boston`, in der von 506 Bezirken um die US-amerikanische Stadt Boston die mittleren Hauspreise `medv` sowie 13 erklärende Merkmale enthalten sind.
Wir wollen sehen, wovon der Hauspreis abhängt und fitten drei lineare Modelle:

```
##R 8.1                                                    siehe auch: ISLR, Ch 3.6
library(MASS)
fit1=lm(medv~.,Boston)           # 1. Lineares Modell: Fit mit allen Merkmalen
summary(fit1)                       # => Adjusted R-squared = 0.7334
fit2=update(fit1,~.-age-indus)   # 2. Lineares Modell: Fit ohne die schwachen Merkmale
summary(fit2)                       # => Adjusted R-squared = 0.7344

# 3. Fit mit qualitativem Merkmal, 2 Interaktionen und 2 Polynomen (in versch. Schreibweisen)
Boston$zn2=ifelse(Boston$zn>0,'c1','c0') # Qualitatives Merkmal ableiten
fit3=lm(medv~.-age-indus-lstat-zn+zn2*rm+zn2*nox+I(dis^2)+poly(lstat,5),Boston)
par(mfrow=c(1,4))
plot(fit3)
summary(fit3)       # => Ergebnis:
##t
```

```
Call:
lm(formula = medv ~ . - age - indus - lstat - zn + zn2 * rm +
    zn2 * nox + I(dis^2) + poly(lstat, 5), data = Boston)

Residuals:
     Min       1Q   Median       3Q      Max
-12.7383  -2.1037  -0.1833   1.7190  24.8526

Coefficients:
                Estimate Std. Error t value Pr(>|t|)
(Intercept)     4.492e+01  4.716e+00   9.525  < 2e-16 ***
crim           -1.615e-01  2.764e-02  -5.842 9.47e-09 ***
chas            2.333e+00  7.165e-01   3.256 0.001209 **
nox            -2.242e+01  3.245e+00  -6.909 1.53e-11 ***
rm              1.992e+00  3.919e-01   5.084 5.28e-07 ***
dis            -3.415e+00  4.872e-01  -7.009 8.02e-12 ***
rad             3.074e-01  5.421e-02   5.671 2.44e-08 ***
tax            -1.195e-02  2.920e-03  -4.092 5.01e-05 ***
ptratio        -7.588e-01  1.142e-01  -6.642 8.31e-11 ***
black           7.579e-03  2.217e-03   3.418 0.000684 ***
zn2c1          -2.893e+01  5.166e+00  -5.600 3.59e-08 ***
I(dis^2)        2.205e-01  4.133e-02   5.336 1.46e-07 ***
poly(lstat, 5)1 -1.026e+02 6.536e+00 -15.690  < 2e-16 ***
poly(lstat, 5)2  4.419e+01  4.439e+00   9.955  < 2e-16 ***
poly(lstat, 5)3 -1.241e+01  4.335e+00  -2.863 0.004375 **
poly(lstat, 5)4  2.197e+01  4.053e+00   5.421 9.34e-08 ***
poly(lstat, 5)5 -1.754e+01  3.985e+00  -4.401 1.32e-05 ***
rm:zn2c1        3.293e+00  6.534e-01   5.039 6.60e-07 ***
nox:zn2c1       1.545e+01  6.756e+00   2.286 0.022681 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.902 on 487 degrees of freedom
Multiple R-squared:  0.8264,    Adjusted R-squared:  0.82
F-statistic: 128.8 on 18 and 487 DF,  p-value: < 2.2e-16
```

Boston {MASS} — R Documentation

Housing Values in Suburbs of Boston

Description

The `Boston` data frame has 506 rows and 14 columns.

Usage

```
Boston
```

Format

This data frame contains the following columns:

`crim` per capita crime rate by town.

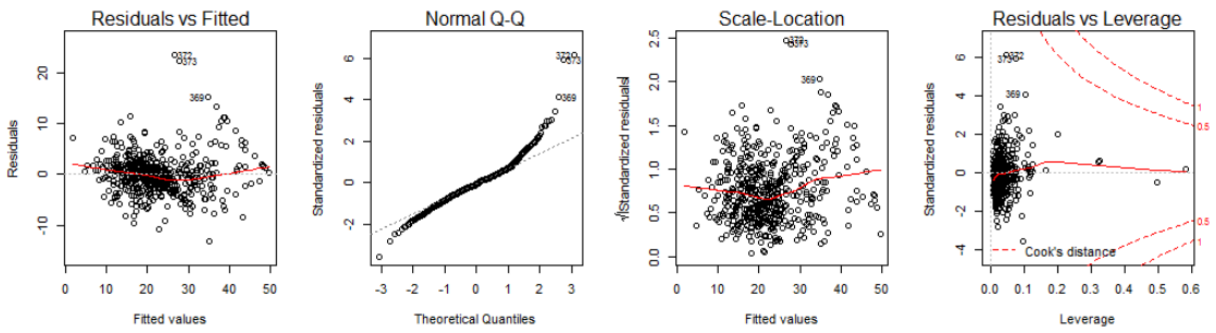
`zn` proportion of residential land zoned for lots over 25,000 sq.ft.

`indus` proportion of non-retail business acres per town.

`chas` Charles River dummy variable (= 1 if tract bounds river; 0 otherwise).

`nox` nitrogen oxides concentration (parts per 10 million).

Wir haben am Ende ein nicht mehr ganz lineares Modell mit vielen hochsignifikanten Parametern und können über 80% der Varianz erklären. Heißt das, es ist es ein gutes Modell?
R macht es den Anwendern sehr einfach, die Güte des Fits in Augenschein zu nehmen: `plot(fit3)`

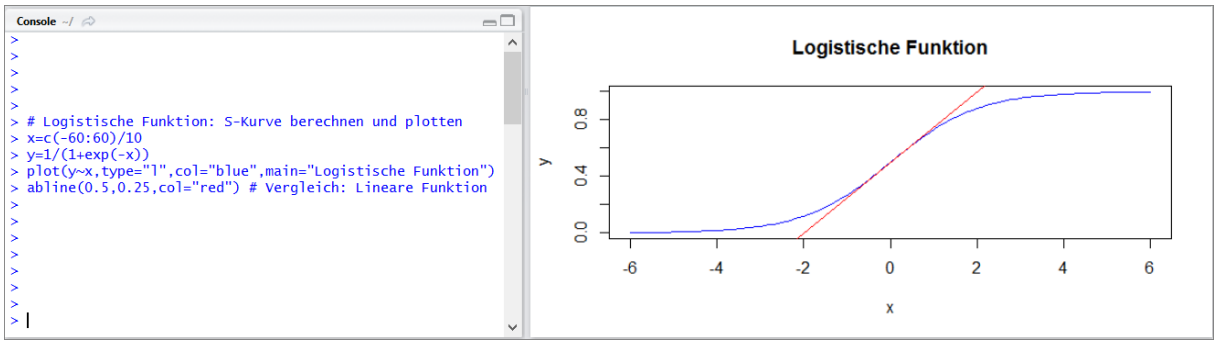


Das Modell beschreibt die Daten noch nicht perfekt. Die Ausreißer werden in den Plots mit Datensatznummer angegeben und können Anhaltspunkte für weitere Interaktionen geben.

Verallgemeinerte lineare Modelle können mit der Funktion `glm` gefittet werden, siehe auch Kapitel 9.

9. Klassifikation: Logistische Regression und Vergleich mit K-Nearest-Neighbors

Bei Klassifikationsaufgaben geht es häufig darum, den Eintritt eines Ereignisses wie Kauf, Kündigung, Kreditausfall oder Unfall vorherzusagen. Die logistische Regression modelliert die Eintrittswahrscheinlichkeit anhand der logistischen Funktion:



Im Gegensatz zur linearen Regression werden hier auch für betragsmäßig sehr große x nur plausible Wahrscheinlichkeiten im Intervall zwischen 0 und 1 berechnet. Als Schätzverfahren wird statt der Kleinste-Quadrate-Methode nun die Maximum-Likelihood-Methode angewandt. Ansonsten überwiegt die Verfahrensverwandtschaft, man erhält eine sehr ähnlich aufgebaute Parameterliste und kann (ziemlich rätselhafte) Residuenplots anschauen.

Die logistische Regression ist ein weit verbreitetes, auch in Prognosewettbewerben erfolgreiches Klassifikationsverfahren und kann in R mit der Funktion `glm` durchgeführt werden. Wir wollen nun eine Kaufentscheidung vorhersagen, und zwar auf Basis von 85 überwiegend soziodemographischen Angaben zu 5.822 Personen, die im belgischen Versicherungsdatensatz `Caravan` enthalten sind. Die Kaufwahrscheinlichkeit beträgt insgesamt nur 6%, wir suchen also eher seltene Ereignisse im Datensatz. Das ist nicht untypisch für die eingangs genannten Beispiele.

Für einen späteren fairen Vergleich mit einem nichtparametrischen Verfahren arbeiten wir mit Training- und Testdaten. Das entscheidende Kriterium ist nun der Klassifikationserfolg bei den Testdaten. Das Modell wird daher ohne die Testdaten entwickelt. Die Parameter werden dann auf die Testdaten angewandt und die Prognose wird mit dem tatsächlichen Kaufverhalten verglichen:

```
##R 9.1                                                    Quelle: ISLR, Ch 4.6.6
library(ISLR)  # Enthält den Datensatz Caravan
dim(Caravan)
attach(Caravan)
summary(Purchase)
test=1:1000
test.Y=Purchase[test]              # test.Y enthält tatsächliches Kaufverhalten

# Logistische Regression (Verallgemeinertes Lineares Modell)
glm.fit=glm(Purchase~.,data=Caravan,family=binomial,subset=-test)
test.probs=predict(glm.fit,Caravan[test,],type="response")
test.pred=rep("No",1000)
test.pred[test.probs >.25]="Yes"    # test.pred enthält prognostiziertes Kaufverhalten
table(test.pred ,test.Y)            # 11/(11+22)=33%
##t
```

Üblicherweise wird erst ab 50% Wahrscheinlichkeit der Interessent als Käufer klassifiziert. Mit 50% ergibt sich hier allerdings kein einziger Treffer. Daher ist der Schwellenwert abgesenkt auf 25%. Von den damit vorhergesagten 33 Käufern haben immerhin 11 tatsächlich gekauft. Die Prognosequote liegt mit 33% deutlich über dem reinen Rateerfolg von 6%. Allerdings sind in der Testdatei 59 Käufer enthalten, von denen nur 11 korrekt vorhergesagt wurden. Insgesamt also eine eher durchwachsene Bilanz. Das kann auch daran liegen, dass die erdrückend große Gruppe „Nichtkäufer“ die Modellanpassung dominiert. Durch eine etwa gleich große Stichprobe von Käufern (alle) und Nichtkäufern (hier ca. jeder 15.te) könnte das seltene Ereignis „Kauf“ angereichert und in

der Folge besser modelliert werden. Bei extrem vielen Nicht-Ereignissen kann dieser Schritt die Berechnungen stark beschleunigen.

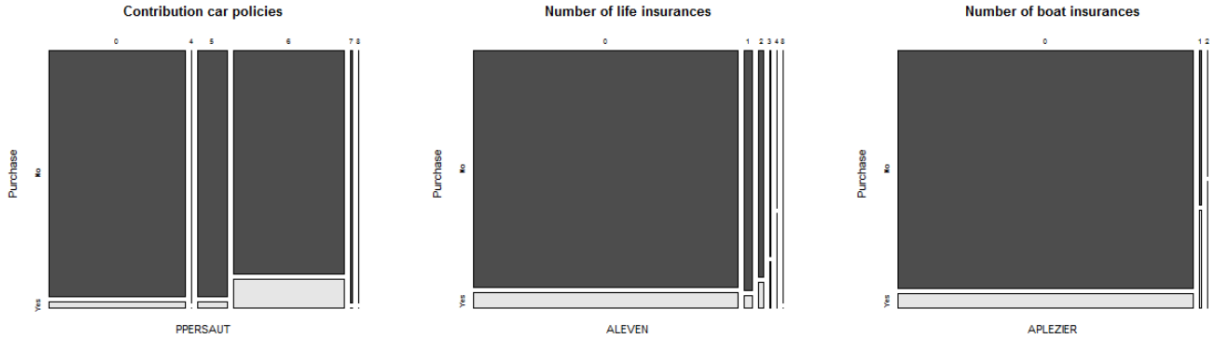
Als Alternative zu einem Regressionsmodell, dessen umfangreiche Parameterliste wir uns hier noch nicht einmal angeschaut haben, wollen wir nun die gänzlich parameterfreie K-Nächste-Nachbarn-Methode (KNN) ausprobieren und die Ergebnisse vergleichen. Bei diesem Verfahren wird die durchschnittliche Distanz der K Beobachtungen im Trainingsdatensatz, die dem interessierenden Punkt y am nächsten liegen, berechnet. Im zweidimensionalen Fall entspricht dies einer kreisförmigen Nachbarschaft. Damit diese „rund“ ist, müssen die Input-Variablen standardisiert werden. Je höher K, umso mehr Nachbarn werden in die Durchschnittsberechnung einbezogen und umso größer sind der Glättungseffekt und die Varianz. Für die Trainingsdaten ist K=1 die optimale Lösung, denn der Schätzfehler für den Abstand zu sich selbst ist 0. Das KNN-Verfahren ist daher anfällig für das „Overfitting“ und ein Fall für die im folgenden Kapitel beschriebenen Sampling-Methoden.

Im Gegensatz zu den bisher verwendeten Prognose-Methoden, bei denen zuerst ein Modell gefittet und auf Basis dieses Modells dann Vorhersagen gemacht werden („scoring“), benötigt das KNN-Verfahren keinen Modellbildungsschritt. Es gibt kein Modell, daher wird für das Scoring der gesamte Trainingsdatensatz benötigt. Für jede Beobachtung im Testdatensatz wird die durchschnittliche Distanz zu den K nächsten Nachbarn in der Trainingsdatei berechnet. Bei Klassifikationsaufgaben gewinnt die Mehrheit.

Die Standardisierung und das KNN-Klassifikationsverfahren wenden wir auf nun auf den Caravan-Datensatz an, um das Kaufverhalten vorherzusagen:

```
##R 9.2                    Achtung: Installiert automatisch die class-Bibliothek          Quelle: ISLR, Ch 4.6.6
# Falls nicht vorhanden das Package class installieren
if (!"class" %in% rownames(installed.packages())) {install.packages("class")}
library("class")
std.X=scale(Caravan[,-86])                  # Alle numerischen Variablen standardisieren
test=1:1000
train.X=std.X[-test,]
test.X=std.X[test,]
train.Y=Purchase[-test]
test.Y=Purchase[test]
set.seed(1)
knn.pred=knn(train.X,test.X,train.Y,k=5)
table(knn.pred,test.Y)                      # 4/(4+11)=27%
##t
```

Das Ergebnis fällt hier schlechter als bei der logistischen Regression aus. Das könnte an linearen Zusammenhängen in den Daten liegen, siehe `summary(glm.fit)` sowie die signifikanten Merkmale:
`mosaicplot(PPERSAUT~Purchase,main="Contribution car policies",shade=F,color=T)` ...



Das KNN-Verfahren liefert gute Klassifikationsergebnisse bei hochgradig nichtlinearen Entscheidungsgrenzen. Der optimale Wert für K kann über die im folgenden Kapitel beschriebenen Methoden gefunden werden.

10. Sampling: Training und Test, Cross-Validation, Bootstrapping und Standardfehler

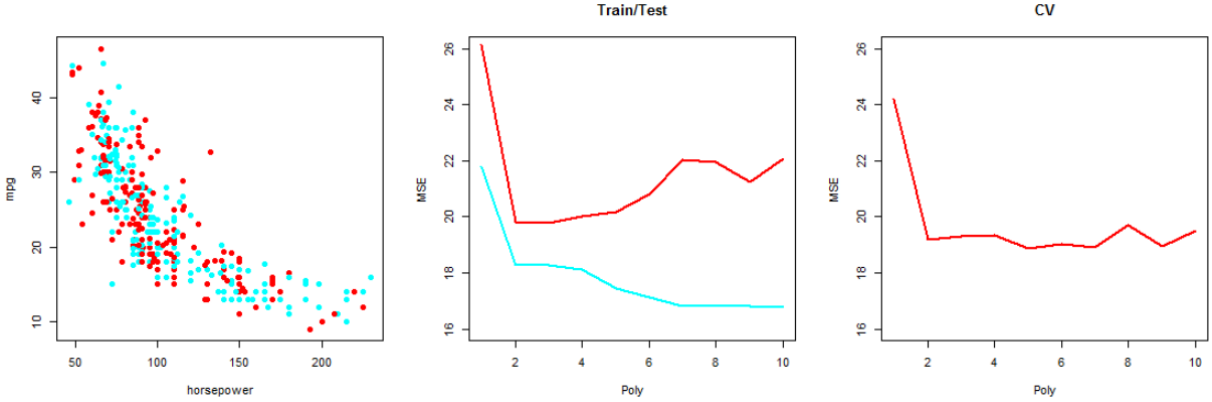
Eine gute Prognose ist auf Basis der Trainingsdaten sehr einfach möglich. Es reicht, sich die Beispiele zu merken. Das fundamentale Ziel des statistischen Lernens ist jedoch, über die Beispiele hinaus zu verallgemeinern. Hier spielen die Sampling-Verfahren eine entscheidende Rolle.

In den vorangegangenen Kapiteln wurden bereits die Unterteilung der Daten in Trainings- und Teststichprobe verwendet. Das Modell wird an die Trainingsdaten „trainiert“, also angepasst. Die Prognosegüte wird dann auf Basis der Testdaten bewertet. Diesen Validierungsansatz wollen wir jetzt näher anschauen.

Overfitting ist nicht nur ein Problem von parameterfreien Verfahren. Auch die klassische Regression läuft bei zu vielen Parametern ins Overfitting. Das zeigen wir am Beispiel eines älteren US-amerikanischen Datensatzes mit 392 verschiedenen Autotypen, für die wir ein Modell für den Kraftstoffverbrauch in Abhängigkeit von der Motorleistung erstellen:

```
##R 10.1                                                                    siehe auch: ISLR, Ch 5.3.1
library(ISLR)
library(boot)        # enthält cv.glm
attach(Auto)
summary(Auto)        # (der mpg-Mittelwert von 23,45 miles per gallon entspricht 10L/100km)
set.seed(1)
train=sample(392,196) # Train/Test: train enthält zuf. Nummern für die Hälfte der Datensätze
par(mfrow=c(1,3))
plot(horsepower,mpg, pch=19,col=rainbow(ifelse(train>0,2,1)))         # s.u. Graphik Links
mse=matrix(rep(0,10*2),2)
for (i in 1:10) {
  lm.fit=lm(mpg~poly(horsepower,i),data=Auto, subset=train)
  mse[1,i]=mean((mpg -predict(lm.fit ,Auto))[train ]^2)      # Standardfehler: Trainingsdaten
  mse[2,i]=mean((mpg -predict(lm.fit ,Auto))[-train ]^2)     # Standardfehler: Testdaten
}                                                                  # s.u. Graphik Mitte
plot(mse[1,],col="cyan",lwd=2,type="l",ylim=c(16,26),xlab="Poly",ylab="MSE",main="Train/Test")
lines(mse[2,] ,col="red",lwd=2)

# K-fache Cross-Validation mit cv.glm (Beschreibung auf der nächsten Seite)
set.seed(17)
cv.error=rep(0,10)
for (i in 1:10) {glm.fit=glm(mpg~poly(horsepower,i),data=Auto)  # Polynom 10.ter Ordnung
  cv.error[i]=cv.glm(Auto ,glm.fit ,K=10)$delta [1] }           # 10-fache Cross-Validation
# Ergebnis der Kreuzvalidierung :                                    s.u. Graphik Rechts
plot(cv.error ,col="red",lwd=2,type="l",ylim=c(16,26),xlab="Poly",ylab="MSE",main="CV")
##t
```

Der Kraftstoffverbrauch wird dabei als Reichweite „miles per gallon“ angegeben (rot: Testdaten):



Linke Graphik: Reichweite mpg in Abhängigkeit von der Motorleistung des Fahrzeugs.
Mittlere Graphik: Während der Standardfehler in der Trainingsdatei mit steigender Modellkomplexität immer weiter sinkt, zeigt der Standardfehler der Testdatei, dass ein Polynom zweiten Grades völlig ausreicht und die Varianz ab dem dritten Grad wieder ansteigt.

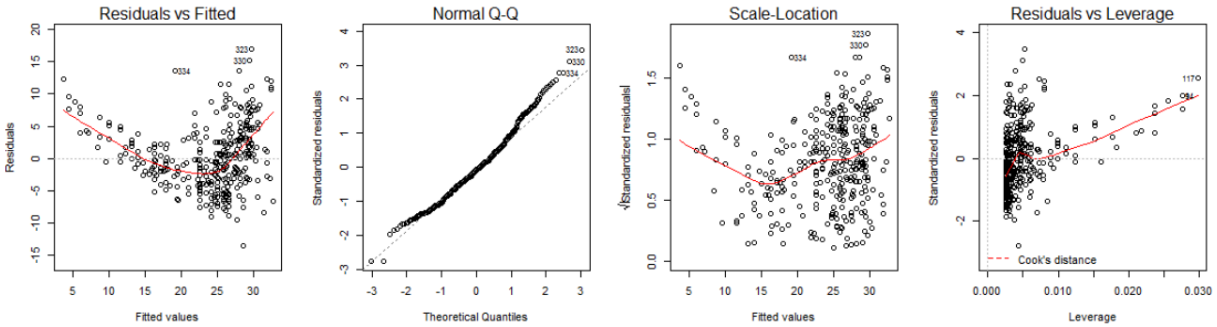
Als nächstes wollen wir den Trainings- und Testdatenansatz verallgemeinern zur K-fachen Kreuz-validierung (Cross-Validation, CV). Hier wird jedem Datensatz eine Zufallszahl zwischen 1 und K zugeordnet. Von diesen K überschneidungsfreien Teilgruppen wird jede einmal als Testdatei verwendet, die Modellbildung also K mal auf Basis aller anderen Teildatensätze durchgeführt und das Ergebnis gemittelt. Das Ergebnis ist für unser Polynom in der vorherigen Graphik rechts abgebildet. Dazu wurde der Datensatz in 10 gleich große Stichproben unterteilt und iterativ jede Stichprobe einmal zur Testdatei. Das zu testende Modell wurde jeweils an den restlichen Daten (90%) trainiert.

Einen anderen Weg geht das Bootstrap-Verfahren. Hier wird aus der Originaldatei eine große Anzahl von Stichproben gezogen, und zwar „mit Zurücklegen“. Datensätze aus der Originaldatei können also mehrfach, einfach oder gar nicht in einer Stichprobe vorkommen. Dieses nichtparametrische Verfahren kann dazu benutzt werden, den Standardfehler der Koeffizienten eines linearen Modells über die Streuung der gezogenen Stichproben zu ermitteln. Das Ergebnis vergleichen wir mit den klassischen Standardfehlern:

```
##R 10.2                                                    Quelle: ISLR, Ch 5.3.4
set.seed(1)
boot.fn=function(data,index) return (coef(lm(mpg~horsepower,data=data,subset=index)))
boot(Auto,boot.fn,1000)                          # Ergebnis:
# Bootstrap Statistics :
#     original    bias         std. error
# t1* 39.9358610  0.0269563085 0.859851825
# t2* -0.1578447 -0.0002906457 0.007402954

# Vergleich mit Standardfehlern der Regressionskoeffizienten eines linearen Modells:
summary(lm(mpg~horsepower,data=Auto))$coef       # Ergebnis:
#                   Estimate   Std. Error   t value   Pr(>|t|)
#       (Intercept) 39.9358610 0.717498656  55.65984 1.220362e-187
#       horsepower  -0.1578447 0.006445501 -24.48914  7.031989e-81
par(mfrow=c(1,4))
plot((lm(mpg~horsepower,data=Auto)))             # Residuenplots für den klassischen Weg:
##t
```

Beide Wege führen zur gleichen Regressionsgerade. Die über das Bootstrap-Verfahren mit 1.000 Stichproben ermittelten Standardfehler sind hier allerdings 20% bzw. 15% größer. Was ist nun richtig?

Platt gesagt beruht die Berechnung des klassischen Standardfehlers auf der Annahme, dass es einen rein linearen Zusammenhang gibt, das Modell also passt. In der linken Graphik auf der vorherigen Seite sieht man bereits einen nichtlinearen Zusammenhang zwischen `horsepower` und `mpg`. Die Residuenplots zeigen ebenfalls, dass das lineare Modell die Daten nicht ausreichend gut beschreibt:



Im Ergebnis werden die Residuen des linearen Modells sowie die Varianz aufgeblasen.

Der Bootstrap-Ansatz ist frei von diesen und weiteren Annahmen und sollte daher die zutreffendere Schätzung für den Standardfehler abgeben. Für das passendere, quadratische Modell stimmen die Schätzungen für den Standardfehler zwischen Bootstrap und Regressionsstatistik besser überein.

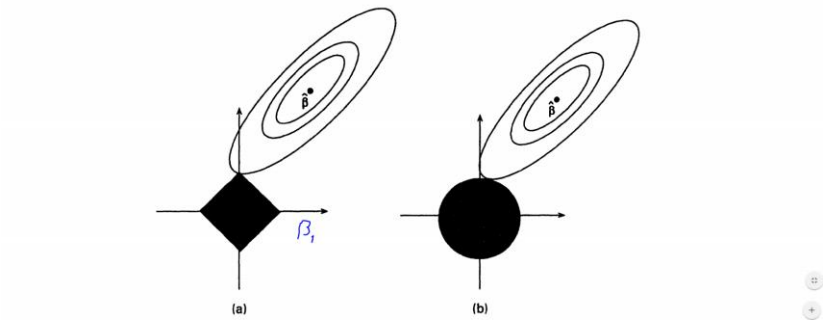
In der ML-Praxis haben sich Bootstrapping für die Berechnung des Standardfehlers und Cross-Validation für die Verhinderung des Overfitting durchgesetzt.

11. Regularisierung linearer Modelle: Ridge-Regression und LASSO

Die Regularisierung (oder Penalisierung) ist ein relativ neues Konzept, die Vorhersagegenauigkeit und Modellinterpretierbarkeit von linearen Modellen, insbesondere bei Datensätzen mit sehr vielen Variablen, zu erhöhen. Da dieses Konzept auch bei anderen Modellklassen (siehe folgende Kapitel) verwendet wird, wollen wir es genauer ansehen.

Die Methoden LASSO (Least Absolute Shrinkage and Selection Operator) und Ridge-Regression funktionieren ähnlich wie die klassische Regressionsmethode der kleinsten Quadrate. Allerdings wird die zu minimierende Residuenquadratsumme RSS um eine sogenannte „shrinkage penalty" ergänzt: Bei Ridge-Regression wird $\lambda \Sigma\beta_j^2$, bei LASSO wird $\lambda \Sigma|\beta_j|$ addiert (Details: j>0, std. Merkmale).
Für λ=0 bleibt es beim Kleinste-Quadrate-Schätzer für ein lineares Modell, allerdings mit allen Merkmalen und damit der Neigung zum Overfitting. Mit wachsendem λ schrumpfen die β_j, die Modellkomplexität wird zusammengedampft, so dass am (unendlichen) Ende alle β_j =0 sind.
Die optimale Modellkomplexität wird durch Kreuzvalidierung ermittelt, λ übernimmt die Rolle des Tuningparameters. Ziel ist ein optimaler „Bias-Variance Tradeoff".

Im Falle des Lasso werden mit steigendem λ immer mehr β_j =0. Dadurch findet zusätzlich eine Parameterselektion statt. Diese ganz besondere und nützliche Eigenschaft wird im Artikel http://statweb.stanford.edu/~tibs/lasso/lasso.pdf von Robert Tibshirani (1996) anhand des folgenden Bildes für den Lasso (a) im Vergleich zur Ridge-Regression (b) erläutert:



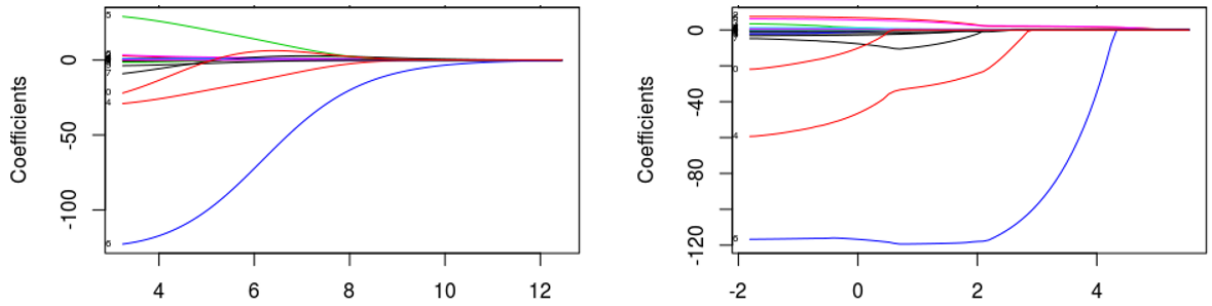
Die Ellipsen repräsentieren hier Regionen mit konstantem RSS. Die schwarzen Flächen sind der durch das „Budget" s (hier zweidimensional: $|\beta_1| + |\beta_2| \leq s$ bzw. $\beta_1^2 + \beta_2^2 \leq s$) abgedeckte Raum. Gesucht ist der erste Punkt, bei dem die Ellipse auf die schwarze Fläche trifft. Die Lasso-Restriktion führt zu Ecken auf den Achsen und genau dort ist häufig der erste Kontaktpunkt und damit im Beispiel oben β_1 =0. In höher dimensionalen Räumen kann das gleichzeitig an mehreren Achsen passieren.

Wir wollen nun auf Basis des Baseball-Datensatzes `Hitters` das Gehalt amerikanischer Baseball-spieler vorhersagen und dazu Ridge-Regression und Lasso mit der Funktion `glmnet` anwenden. Dazu muss das R-Package `glmnet` installiert werden, das bereits Parallelisierung (`foreach` von Revolution Analystics) verwendet und sehr schnell ist, aber bei der Windows-Installation etwas widerspenstig sein kann und beispielsweise einen fortan 90 compiler vermisst (Alternative: Auf Linux ausweichen).

```
##R 11.1                                                    siehe  auch: ISLR, Ch 6.6
library(ISLR)
library(glmnet)
Hitters=na.omit(Hitters)                  # Datensätze mit fehlenden Werten entfernen
with(Hitters,sum(is.na(Salary)))          # Zähle Datensätze ohne Salary. Ergebnis: 0
dim(Hitters)                              # Ergebnis: Noch 263 Datensätze, 20 Merkmale
x=model.matrix(Salary~.-1,data=Hitters)   # Matrix, ohne Salary. Bildet num. Dummy-Merkmale
y=Hitters$Salary

# Schrumpfen der Koeffizienten:
par(mfrow=c(1,2))
set.seed(1)
fit.ridge=glmnet(x,y,alpha=0)             # Ridge-Ression fitten
plot(fit.ridge,xvar="lambda",label=TRUE)  # Schrumpfen der Koeffizienten plotten
```

```
fit.lasso=glmnet(x,y)                    # Lasso fitten. Default: alpha=1
plot(fit.lasso,xvar="lambda",label=TRUE)  # Ergebnis x: log(Lamba)
##t
```

Die Koeffzienten (β_j) für Ridge-Regression (links) und Lasso (rechts) in Abhängigkeit von log(λ):



Wir wollen nun durch Kreuzvalidierung (CV) das optimale Modell finden und λ „tunen“:

```
##R 11.2                                                    siehe  auch: ISLR, Ch 6.6
# x-Matrix und y-Vektor für glmnet anlegen. Trainings- und Testdatei erzeugen
x=model.matrix(Salary~.,Hitters)[,-1] # Matrix ohne Salary. Bildet num. Dummy-Merkmale
y=Hitters$Salary
library(glmnet)
grid=10^seq(10,-2,length=100)          # Gitter von Lamba=0.01 bis 10^10 anlegen
set.seed(1)                            # Reproduzierbare Trainings- und Testdaten erzeugen
train=sample(1:nrow(x),nrow(x)/2)
test=(-train)
y.test=y[test]

# Ridge Regression: Elasticnet mixing parameter „alpha“ = 0
ridge.mod=glmnet(x[train,],y[train],alpha=0,lambda=grid,thresh=1e-12) # Werte fürs Gitter
set.seed(1)
cv.out=cv.glmnet(x[train,],y[train],alpha=0)
plot(cv.out)
bestlam=cv.out$lambda.min              # Lambda mit dem kleinsten Standardfehler
ridge.pred=predict(ridge.mod,s=bestlam,newx=x[test,])
mean((ridge.pred-y.test)^2)                              # Ergebnis: 96016
out=glmnet(x,y,alpha=0)                                  # Fit mit gesamtem Datensatz
predict(out,typ="coefficients",s=bestlam)[1:20,]         # Prognose mit bestlam

# The Lasso: Elasticnet mixing parameter „alpha“ = 1 (default)
lasso.mod=glmnet(x[train,],y[train],alpha=1,lambda=grid) # Für grid-Suche
set.seed(1)
cv.out=cv.glmnet(x[train,],y[train],alpha=0)
plot(cv.out)
bestlam=cv.out$lambda.min
lasso.pred=predict(lasso.mod,s=bestlam,newx=x[test,])
mean((lasso.pred-y.test)^2)                              # Ergebnis: 100743
out=glmnet(x,y,alpha=1,lambda=grid)
predict(out,typ="coefficients",s=bestlam)[1:20,]         # CV-Ergebnis Lasso-Coefficients:
##t
(Intercept)       AtBat        Hits       HmRun        Runs         RBI       Walks       Years      CAtBat       CHits
 18.5394844   0.0000000   1.8735390   0.0000000   0.0000000   0.0000000   2.2178444   0.0000000   0.0000000   0.0000000
     CHmRun       CRuns        CRBI       CWalks      LeagueN   DivisionW     PutOuts     Assists      Errors   NewLeagueN
  0.0000000   0.2071252   0.4130132   0.0000000   3.2666677 -103.4845458   0.2204284   0.0000000   0.0000000   0.0000000
```

In diesem Beispiel spricht der geringere Schätzfehler für Ridge-Regression (α=0), die deutlich geringere Parameteranzahl und damit bessere Modellinterpretierbarkeit für Lasso (α=1).

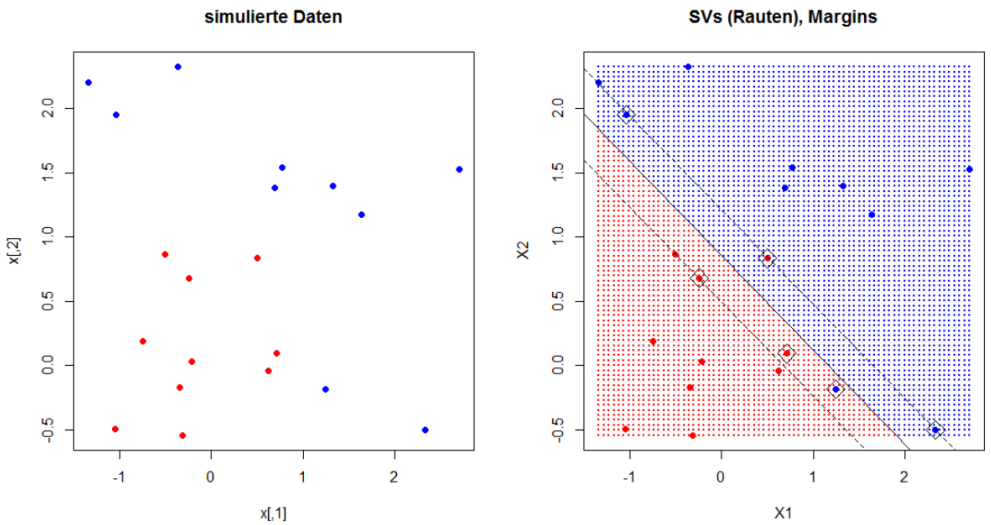
Beide Verfahren haben ihre Vor- und Nachteile. Zur Kombination der Vorteile addiert das Verfahren „Elastic Net“ beide Penalty-Terme. Auch hier besteht ein aktives Forschungsfeld, erst 2014 wurde die Äquivalenz zum im folgenden Kapitel beschriebenen Verfahren „Support Vector Maschine“ aufgezeigt.

12. Support Vector Classifier und Support Vector Machine (SVM), ROC-Kurve

Bei Klassifikationsaufgaben wird für die abhängige Variable eine möglichst trennscharfe Hyperebene im Merkmalsraum gesucht, im zweidimensionalen Fall also eine Trennlinie, im einfachsten Fall eine Gerade. Diese Gerade wird auf Basis der am dichtesten liegenden Datenpunkte in der Grenzregion berechnet. Ein Datenpunkt, der die Berechnung der Trennlinie „unterstützt“, wird als „Support Vector“ bezeichnet. Das Besondere an diesem Verfahren ist, dass weiter von der Grenzlinie entfernte Punkte keinen Einfluss haben.

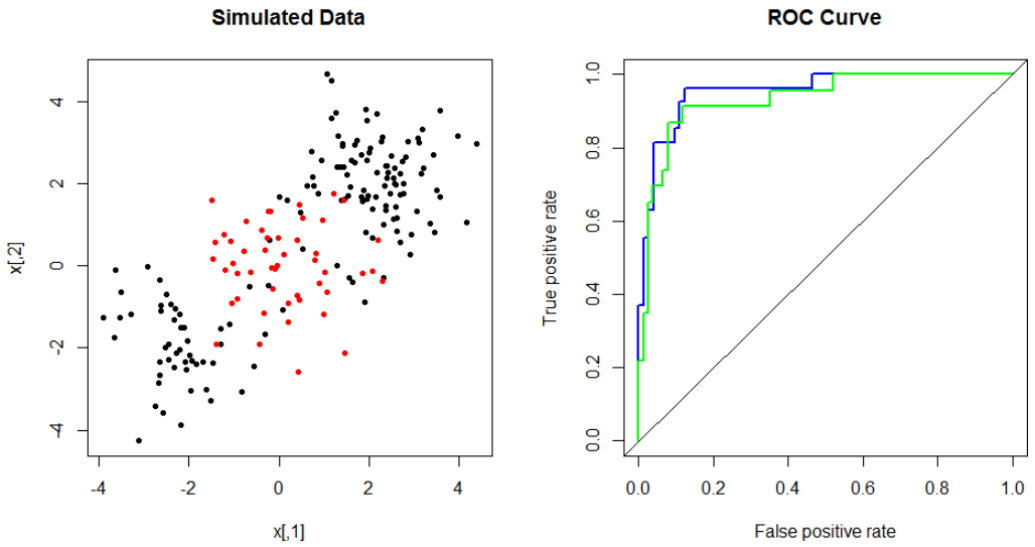
Die Lage eines einzelnen Datenpunktes kann großen Einfluss auf die Lage der Trennlinie haben. Liegt der Datenpunkt auf der „falschen“ Seite, ist genau genommen gar keine Trennlinie möglich. Damit das Verfahren auch in diesem nicht seltenen Fall angewandt werden kann, wird ein „Budget“ C für Grenzverletzungen akzeptiert. Dieser Kostenparameter wird über Kreuzvalidierung getunt, wir kennen das bereits aus dem letzten Kapitel (was dem einen sein λ, ist dem andern sein C).
Wir wollen diesen linearen „Support Vector Classifier“ nun an Übungsdaten berechnen und zeigen:

```
##R 12.1                                                    Quelle: statLearning, ch09.Rmd
library(e1071)
set.seed(10111)            # probieren geht über studieren ...
x=matrix(rnorm(40),20,2)   # zweidimensionale Daten mit unscharfer Trennlinie anlegen
y=rep(c(-1,1),c(10,10))
x[y==1,]=x[y==1,]+1        # Für etwas Abstand zwischen den Clustern sorgen
par(mfrow=c(1,2))
plot(x,col=y+3,pch=19, main="simulierte Daten")
svmfit=svm(y~.,data=data.frame(x,y=as.factor(y)),kernel="linear",cost=10,scale=FALSE)
print(svmfit)                                          # Ergebnis: 6 Support Vectors

# gepunkteten Hintergrund anlegen, Support Vectors markieren und Margins plotten
make.grid=function(x,n=75){  grange=apply(x,2,range)
  x1=seq(from=grange[1,1],to=grange[2,1],length=n)
  x2=seq(from=grange[1,2],to=grange[2,2],length=n)
  expand.grid(X1=x1,X2=x2)}
xgrid=make.grid(x)
ygrid=predict(svmfit,xgrid)
beta=drop(t(svmfit$coefs)%*%x[svmfit$index,])       # Jetzt geht's um die Linien
beta0=svmfit$rho
plot(xgrid,main="SVs (Rauten), Margins",col=c("red","blue")[as.numeric(ygrid)],pch=20,cex=.2)
points(x,col=y+3,pch=19)                            # Datenpunkte einzeichnen
points(x[svmfit$index,],pch=5,cex=2)                # Rauten um „Support Vectors“ malen
abline(beta0/beta[2],-beta[1]/beta[2])              # Linien für „Margin“ ziehen
abline((beta0-1)/beta[2],-beta[1]/beta[2],lty=2)
abline((beta0+1)/beta[2],-beta[1]/beta[2],lty=2)    # Aufgehübschtes Ergebnis:
##t
```



Eine „Support Vector Machine“ ist die Erweiterung dieses Konzepts um Kernel. Dadurch können auch nichtlineare Grenzen mit vergleichsweise geringem Rechenaufwand bestimmt werden. Für folgendes Beispiel mit einer nichtlinearen Grenzlinie verwenden wir einen radialen Kernel und ermitteln durch Kreuzvalidierung den optimalen Cost-Parameter C sowie den optimalen Kernel-Formparameter γ.

```
##R 12.2                                                    Quelle: ISLR, Ch 9.6
set.seed(1)
x=matrix(rnorm (200*2) , ncol=2)     # 2D-Daten mit nichtlinearer Klassengrenze anlegen
x[1:100,]=x[1:100,]+2
x[101:150 ,]=x[101:150,]-2
y=c(rep(1,150) ,rep(2,50))
dat=data.frame(x=x,y=as.factor(y))
par(mfrow=c(1,2))
plot(x,col=y,main="Simulated Data",pch=20)
train=sample(200,100)                 # 50% Training-Sample anlegen

#Kreuzvalidierung mit tune() für Cost- und gamma-Gitter durchführen
set.seed(1)
tune.out=tune(svm, y~.,data=dat[train,],kernel="radial",ranges=list(cost=c(0.1,1,10,100,1000),gamma=c(0.5,1,2,3,4)))
summary(tune.out)
table(true=dat[-train,"y"],pred=predict(tune.out$best.model,newdata=dat[-train,])) # Test-Ergebnis: 10% Fehlklassifikation

# ROC-Kurve für bestes Modell plotten (gamma=2, cost=1)
library(ROCR)
rocplot=function(pred ,truth,...) { predob=prediction(pred,truth)
perf=performance(predob,"tpr","fpr")
plot(perf,...) }
svmfit=svm(y~.,data=dat[train,],kernel="radial",gamma=2,cost=1,decision.values=T)
pred.trn=attributes(predict(svmfit,dat[train,],decision.values=T))$decision.values
rocplot(pred.trn,dat[train,"y"],main="ROC Curve",col="blue",lwd=2)
pred.tst=attributes(predict(svmfit,dat[-train,],decision.values=T))$decision.values
rocplot(pred.tst,dat[-train,"y"],add=T,col="green",lwd=2)
abline(0,1)
##t
```



Für hohes Gamma erhält man eine 100% perfekte Klassifikation bei den Trainingsdaten und sehr schlechte Ergebnisse für die Testdaten, also perfektes Overfitting. Je kleiner C und γ, umso glatter wird die Grenzlinie.

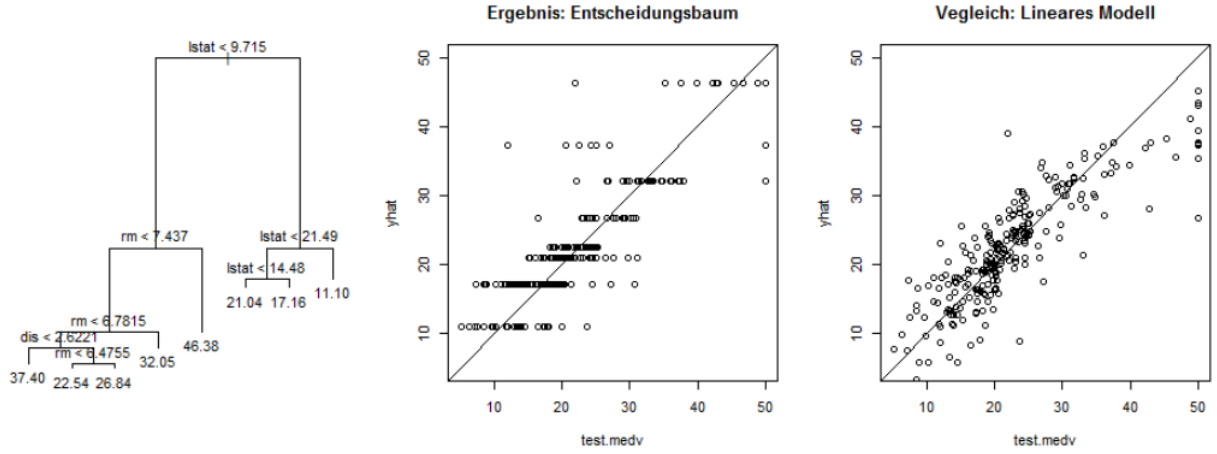
Die Klassifikationsgüte kann über sogenannte ROC-Kurven veranschaulicht werden. Dazu werden die falsch positiv vorhergesagten Werte gegen die korrekt positiv vorhergesagten Werte geplottet. Die ROC-Kurven von Trainings- (blau) und Testdaten (grün) des besten Modells unterscheiden sich in unserem Beispiel kaum noch. Overfitting konnte also verhindert werden. Je weiter sich die beiden Kurven in die obere linke Ecke drängen, desto besser ist das Klassifikationsergebnis.

13. Entscheidungsbäume: Splits, Pruning, Bagging und Random Forests

Klassische Entscheidungsbäume werden gerne für erste Analysen angewendet. Sie eignen sich sowohl für Regressions- als auch für Klassifikationsprobleme, sind einfach zu verstehen und einfach zu interpretieren. Zudem sind sie anspruchslos gegenüber Merkmalsarten und kommen auch mit fehlenden Werten gut zurecht. Und es gibt einen großen Baukasten von mächtigen Ausbaustufen.

Wir starten mit dem Grundprinzip des rekursiven binären Teilens. Bei einem Regressionsproblem wird für jedes Merkmal der Splitpunkt gesucht, der zum größten Rückgang der RSS führt. Der erste Split wird dann mit dem besten Merkmalsplitpunkt durchgeführt. Im Beispiel unten mit den bereits bekannten Hauspreisdaten wird für die gewählte Trainingsstichprobe ein optimaler erster Splitpunkt für `lstat`<9,715 ermittelt. Der Sozialstatus `lstat` hat also zusammen mit der ermittelten Splitgrenze den größten Einfluss auf den Hauspreis `medv`. Dieser Prozess wird nun für alle entstehenden „Äste“ solange wiederholt, bis ein Stopkriterium greift. Dies ist hier nach 8 Endknoten („Blätter“) erreicht. Entscheidungsbäume werden allgemein nach unten wachsend dargestellt und können mit dem R-Package `tree` gerechnet und visualisiert werden.

```
##R 13.1                                                    siehe auch: Kap.08 und ISLR, Ch 8.5
library(MASS)
require(tree)  # Synonym für library(tree)
set.seed(1)
train = sample(1:nrow(Boston), nrow(Boston)/2)    # 50%-Trainingsstichprobe definieren
test.medv=Boston[-train,"medv"]
tree.boston=tree(medv~.,Boston,subset=train)      # Entscheidungsbaum rechnen
plot(tree.boston)
text(tree.boston,pretty=0)
yhat=predict(tree.boston,newdata=Boston[-train,]) # Modell auf Testdaten anwenden
plot(yhat~test.medv,main="Ergebnis: Entscheidungsbaum",xlim=c(5,50),ylim=c(5,50))
print(paste("RMSE:",sqrt(mean((yhat - test.medv)^2))))             # Ergebnis: RMSE=5.005
abline(0,1)

# Zum Vergleich: Lineares Modell (siehe Kap. 8)
fit2=lm(medv~.-age-indus,Boston,subset=train)
summary(fit2)
yhat.lm=predict(fit2,newdata=Boston[-train,])
print(paste("RMSE:",sqrt(mean((yhat.lm - test.medv)^2))))          # Ergebnis: RMSE=5.120
plot(yhat.lm~test.medv,main="Vergleich: Lineares Modell",xlim=c(5,50),ylim=c(5,50))
abline(0,1)
##t
```



Obwohl es sich um ein Regressionsproblem handelt und für den Hauspreis nur acht verschiedene Werte vorhergesagt werden, ist das Prognoseergebnis nicht schlechter als das vom zweiten linearen Modell aus Kapitel 8. In beiden Fällen wird der Hauspreis auf etwa 5.000$ genau geschätzt (RMSE).

Klassifikationsbäume können in gleicher Weise durch rekursives binäres Teilen aufgebaut werden. Ziel sind möglichst reine Knoten in Bezug zum abhängigen Merkmal.

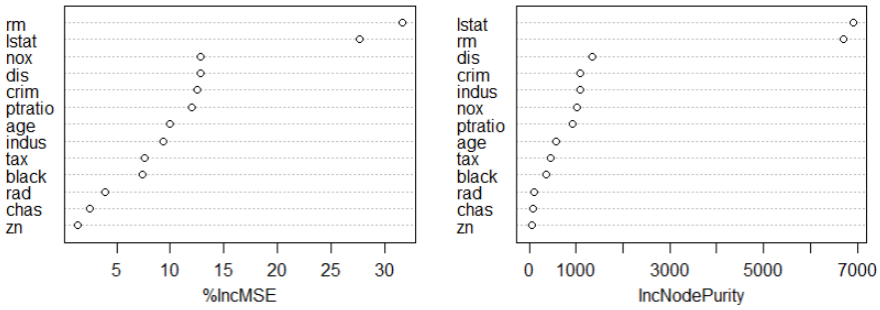
Unser erster Entscheidungsbaum ist „links unten“ etwas ins Overfitting gelaufen. Overfitting ist für Entscheidungsbäume recht typisch, sie neigen zum „verwachsen“ und sollten zurückgeschnitten werden. Dieses „Pruning“ kann durch die Einbeziehung der Testdaten (die dann nicht mehr neutral sind) oder durch Kreuzvalidierung durchgeführt werden. Der Tuningansatz gleicht dem aus Kapitel 11. Ein zurückgeschnittener Entscheidungsbaum sieht ganz normal aus und ist genauso einfach zu interpretieren. Das wird sich bei den folgenden Methoden stark ändern.

Durch Bootstrapping (Kap. 10) kann Varianzreduktion erreicht werden. Diesen Umstand verwendet die Bagging-Methode, bei der B verschiedene Bootstrap-Trainingsstichproben gezogen werden und dazu jeweils ein Entscheidungsbaum mit allen p Merkmalen gerechnet wird. Diese B Entscheidungsbäume führen zu B Prognoseergebnissen, aus denen bei Regressionsproblemen ein Durchschnittswert berechnet und bei Klassifikationsproblemen eine Mehrheitsentscheidung gefällt wird.

Die Methode „Random Forest“ geht noch einen Schritt weiter und dekorreliert die Entscheidungsbäume dadurch, dass jeder Split auf Basis einer neuen zufälligen Auswahl von m Prädiktormerkmalen durchgeführt wird. Gemäß der Faustformel m $\approx \sqrt{p}$ werden für nicht allzu schmale Datensätze die Mehrheit der Merkmale, und damit auch häufig genug die stärksten Merkmale von der Split-Berechnung ausgeschlossen. Dadurch wachsen die Bäume viel unterschiedlicher und sind unkorrelierter.

Das Verfahren Random Forest (und für m=p Bagging) kann mit dem Package `randomForest` angewendet werden. Wir wollen nun sehen, ob dieses Verfahren zu einer besseren Prognose des Hauspreises führt:

```
##R 13.2                                                                    Quelle: ISLR, Ch 8.5
library(randomForest)
set.seed(1)
rf.boston=randomForest(medv~.,data=Boston,subset=train,mtry=6,importance=TRUE)
yhat.rf=predict(rf.boston,newdata=Boston[-train,])
print(paste("RMSE:",sqrt(mean((yhat.rf - test.medv)^2))))          # Ergebnis: RMSE=3.389
varImpPlot(rf.boston)      # Variable Importance plotten:
##t
```



Im Ergebnis wird die Varianz mehr als halbiert (von 25 auf unter 12), der Standardfehler sinkt entsprechend. Die Varianzreduktion wird also beeindruckend gut erreicht. Die wichtigsten Merkmale sind unverändert `lstat` und `rm`.

Die Funktion `randomForest` rechnet dazu 500 (Default-Wert) teilweise sehr unterschiedliche Bäume. Deren Mittelung führt zwar zu einer guten Prognose, verhindert aber eine einfache Darstellung als Entscheidungsbaum. Das ist in diesem Vergleich der Hauptnachteil von Bagging und Random Forest. Diesem Nachteil haben aber auch andere leistungsfähige Verfahren.

Bagging und die Weiterentwicklung Random Forest setzen voll auf den Zufall und die Unabhängigkeit einer großen Anzahl von Stichproben und erreichen damit das Ziel der Varianzreduktion.

Geht das auch anders, und vielleicht noch besser?

14. Gradient Boosting Machines, Sparse Data und XGBOOST

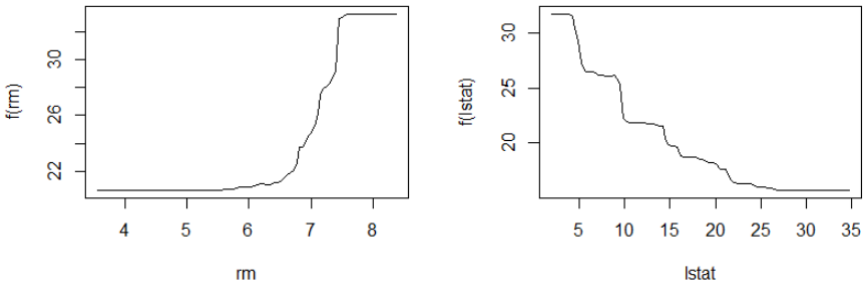
Boosting ist ein allgemeiner Ansatz, der für viele Verfahren angewendet werden kann. Generell wird hier aus einer Vielzahl von sogenannten schwachen Lernern ein starker Lerner erzeugt. Boosting von Regressionsbäumen verwendet wie Bagging eine hohe Anzahl an Entscheidungsbäumen. Diese sind beim Boosting jedoch voneinander abhängig und sehr klein. Jeder Baum hängt von den zuvor erzeugten Bäumen ab und versucht die Prognosegenauigkeit zu erhöhen. Gradient Boosting Machines können Klassifikation, Regression und Ranking durchführen. Die Gewichte werden durch ein Gradientenverfahren ermittelt. Es wird (auch hier) eine Kostenfunktion mit Schrumpfungs-parameter verwendet. Siehe auch die sehr gute und anschauliche Einführung von Tianqi Chen: http://homes.cs.washington.edu/~tqchen/pdf/BoostedTree.pdf (Autor von XGBOOST).

Boosting hat drei Tuning-Parameter:

1. Die Anzahl B der Entscheidungsbäume.
2. Die Anzahl d der Splits je Entscheidungsbaum. Hiermit wird die Komplexität bestimmt. Oft ist eine hohe Anzahl B von Baumstümpfen (d=1, sehr schwacher Lerner) erfolgreich.
3. Den Schrumpfungsparameter λ. Dieser regelt die Lernrate und liegt typischerweise zwischen 0,001 und 0,01. Für ein sehr kleines λ kann ein großes B erforderlich sein.

Wir verwenden nun die Funktion `gbm` aus dem gleichnamigen Package und fitten mit der Option `distribution="gaussian"` „geboostete“ Regressionsbäume (für binäre Klassifikationsbäume: `distribution="bernoulli“`), wie zuvor am Beispiel der Hauspreisdaten. Die Funktion verwendet dazu Friedmans „gradient boosting machine“. Es werden 5.000 Bäume mit jeweils 4 Splits berechnet:

```
##R 14.1                                                    Quelle: ISLR, Ch 8.5
library(MASS)
set.seed(1)
train = sample(1:nrow(Boston), nrow(Boston)/2)    # 50%-Trainingsstichprobe definieren
library(gbm)
set.seed(1)
boost.boston=gbm(medv~.,data=Boston[train,],distribution="gaussian",n.trees=5000,interaction.depth=4)
summary(boost.boston)
yhat.boost=predict(boost.boston,newdata=Boston[-train ,],n.trees=5000)
print(paste("RMSE:",sqrt(mean((yhat.boost - test.medv)^2))))        # Ergebnis: RMSE=3.442
par(mfrow=c(1,2))                    # Ergebnis (hier für 5.000 Trees):
plot(boost.boston,i="rm")            # Der mittlere Hauspreis steigt mit der Raumanzahl
plot(boost.boston,i="lstat")         # und fällt mit steigendem Anteil der Sozial-Haushalte
##t
```



Die Standardfehler liegt bei den Testdaten auf dem Niveau der Methode Random Forest. Man kann den Fit weiter verbessern und den shrinkage-Parameter λ vom voreingestellten Wert 0,001 auf 0,02 hochsetzten (`shrinkage = 0.02, verbose=0`) und erzielt dadurch ein Test-RMSE=3.172. Das ist der bisher niedrigste Wert. Die Optimierung der drei Parameter B, λ sowie d sollte durch Kreuz-validierung erfolgen.

Im Jahr 2015 hat das Package `XGBOOST` (e**X**treme **G**radient **Boost**ing) durch zahlreiche Erfolge bei kaggle-Wettbewerben auf sich aufmerksam gemacht. XGBOOST nutzt Parallelisierung und ist sehr schnell. Es kann auf die gleiche Weise wie alle anderen bisherigen Packages installiert werden.

Die Autoren von XGBOOST stellen ein R-Tutorial mit einer Klassifikationsaufgabe zur Verfügung: https://github.com/dmlc/xgboost/blob/master/R-package/vignettes/xgboostPresentation.Rmd Dieses R Markdown-Dokument kann als raw-file heruntergeladen werden. Alternativ können Sie die unten angegebene Kurzfassung ausführen.

Es soll vorhergesagt werden, welche Pilze giftig sind. Im Package sind bereits eine Trainingsdatei sowie eine Testdatei mit den Angaben zu 8.124 Pilzen in einer 80%/20%-Aufteilung enthalten. Die beiden Dateien enthalten zu jedem Pilz die Angabe, ob er giftig ist sowie 126 beschreibende 0/1-Merkmale (Dummy-Kodierung). Die 126 Dummy-Variablen werden aus 22 Pilzeigenschaften abgeleitet. Beispielsweise ist die Information des 1. Pilzmerkmals „Kappenform“, für das es sechs Kategorien gibt (cap-shape: bell=b, conical=c, convex=x, flat=f, knobbed=k, sunken=s) in den sechs 0/1-Merkmalen ‘cap-shape=bell’ bis ‘cap-shape=sunken’ abgespeichert, wobei die zutreffende Eigenschaft wie üblich mit 1 kodiert ist. Von den sechs Merkmalen zur Kappenform sind also immer fünf mit der 0 kodiert. Entsprechend haben jeweils 104 der 126 Merkmale den Wert 0. Daher sind hier die Daten bereits platzsparend als sogenannte „Sparse-Matrix“ ohne die Nullsätze gespeichert.

Durch diese Vorgehensweise kann übrigens auch bei Versicherungsdaten der Platzbedarf von zahlreichen spärlich gefüllten 0/1-Kennzeichen im Datawarehouse massiv reduziert werden.

Wir führen nun ein Tree-Boosting und ein Regression-Boosting mit XGBOOST durch:

```
##R 14.2
library(xgboost)
data(agaricus.train, package='xgboost'); train <- agaricus.train
data(agaricus.test, package='xgboost'); test <- agaricus.test
class(train$data)[1]           # Ergebnis: "dgCMatrix" - sparse matrix
class(train$label)             # Ergebnis: "numeric": label=1 bedeutet „giftig“.
summary(as.matrix(test$data))  # Ergebnis: u.a. min, mean, max für 126 Pilzeigenschaften
dtrain=xgb.DMatrix(data = train$data, label = train$label)

# Boosting Trees (default: booster=“gbtree“)
set.seed(1)
bst=xgboost(data=dtrain,max_depth=2,eta=1,nrounds=2,objective="binary:logistic")
pred <- predict(bst, test$data)
prediction <- as.numeric(pred > 0.5)     # Aus Regressionsergebnis Klassifikation machen
err <- mean(as.numeric(pred > 0.5) != test$label) # Anteil Fehlklassifikationen berechnen
print(paste("test-error=", err))         # Ergebnis: "test-error= 0.0217"
table(prediction ,test$label)            # Kreuztabelle ausgeben

# Regression Boosting (hier Logistische Regression)
set.seed(1)
bst=xgboost(data=dtrain,max_depth=2,booster="gblinear",nrounds=2,objective="binary:logistic")
pred <- predict(bst, test$data); prediction <- as.numeric(pred > 0.5)
err <- mean(as.numeric(pred > 0.5) != test$label)            # Ergebnisse:
print(paste("test-error (lineares Boosting)=", err))
table(prediction ,test$label)
```

```
##t
 > print(paste("test-error (lineares Boosting)=", err))
 [1] "test-error (lineares Boosting)= 0.00434512725015518"
 > table(prediction ,test$label)

 prediction   0   1
          0 830   2
          1   5 774
```

Die Vorhersage des linearen Boosting stimmt bei den Testdaten bereits nach zwei Iterationen zu 99,6%! Das ist mit Abstand die beste Prognose, die wir bisher gemacht haben. Derartig hohe Werte sind bei Ereignisprognosen im Versicherungsbereich (Kauf, Kündigung, Unfall, etc.) nahezu ausgeschlossen und typischerweise ein Hinweis auf einen Fehler in der Datenaufbereitung.

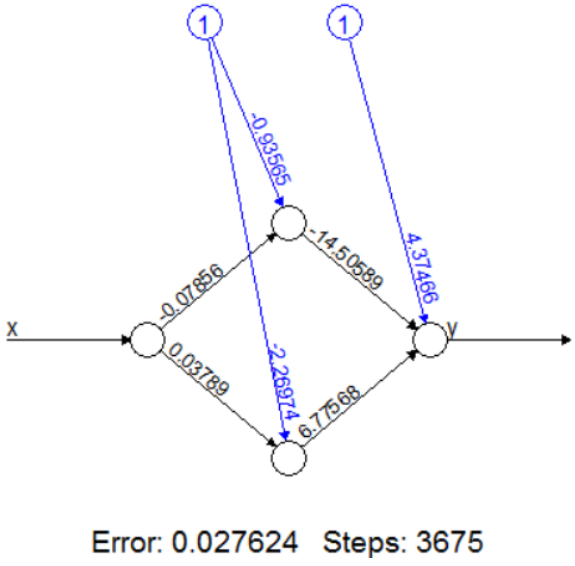
Bei der Beurteilung der Giftigkeit von Pilzen und bei modernen optischen Erkennungsverfahren kann jedoch die Fehlerrate nicht klein genug sein, siehe Kapitel 18.

15. Künstliche Neuronale Netze (1): Einführung, einfache Regressionsmodelle

Aus Sicht der Statistik ist ein künstliches neuronales Netz (NN) ein mehrschichtiges, üblicherweise nichtlineares Regressions- oder Klassifikationsmodell. Einige Beiträge und Begriffe zu dieser Verfahrensklasse haben die Neurowissenschaften und das Forschungsfeld Künstliche Intelligenz beigesteuert. Neuronale Netze gehören inzwischen zu den leistungsfähigsten Prognosemodellen.

Wir wollen uns hier nicht mit der Entstehungsgeschichte und frühen, limitierten Modellen befassen, sondern als einfaches erstes Beispiel die Quadratwurzel näherungsweise über ein kleines NN abbilden, das Netz nachrechnen und dabei die nötigen Begriffe und Zusammenhänge kennen lernen. Das NN trainieren wir mit der Funktion `neuralnet` aus dem gleichnamigen Package und verwenden die plot-Funktion zur Veranschaulichung der Netzstruktur und der berechneten Gewichte:

```
##R 15.1
library(neuralnet); set.seed(1)
x=runif(30,min=0,max=100)          # 30 Zufallszahlen zwischen 0 und 100 simulieren,
y=sqrt(x)                          # die Quadratwurzel berechnen und beides
train=data.frame(cbind(x,y))       # in eine Trainingsdatei schreiben

# Neuronales Netz trainieren: Multi-Layer Perceptron mit 1 „hidden layer“ und 2 „Neuronen“
set.seed(1)
net=neuralnet(y~x,train,hidden=2) # Default: act.fct="logistic"
print(net)
plot(net) # Grafische Ausgabe der Netzstruktur und Gewichte:
##t
```



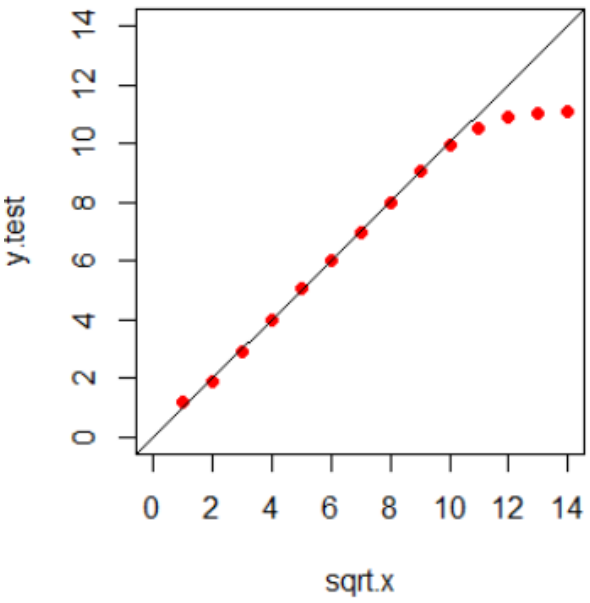
Die linke Seite des Modells stellt die erklärenden Variablen dar (hier die Variable x) und wird als „Input Layer“ bezeichnet. Die rechte Seite stellt die Prognosemerkmale dar (hier y) und wird „Output Layer“ genannt.
Die Lagen dazwischen werden als „Hidden Layer“ bezeichnet. Dieses Modell hat eine hidden layer mit zwei Elementen, den sogenannten Neuronen. Ganz entscheidend ist hierbei die auf die Neuronen wirkende Aktivierungsfunktion, typischerweise die logistische Funktion (siehe auch Kap. 9), die das nichtlineare Verhalten bewirkt.
In diesem Feed-Forward-Modell hängen die rechts folgenden Schichten nur von früheren Schichten ab, es gibt keine Rückkoppelung.

Wir wollen nun für die Zahl x=100 die Prognose (Quadratwurzel) dieses Netzes nachrechnen:
`y = 4,375 + -14,506/(1+EXP(-(100*-0,0786+-0,94))) + 6,776/(1+EXP(-(100*0,0379+-2,27))) = 9,93`

Für die geringe Anzahl an Trainingsdaten und das sehr kleine NN ist das gar nicht schlecht. Allerdings werden hier 3.675 Schritte zur Bestimmung der sieben Gewichte benötigt. Mit acht Neuronen wird eine drastisch genauere Prognose erreicht, und das mit nur 1373 Schritten.
Mit mehr Datensätzen steigt die Anzahl der benötigten Schritte überproportional stark an.

Das Ergebnis testen wir nun sowohl innerhalb als auch außerhalb des Trainingsbereiches des NN:

```
##R 15.2
t=((1:14)^2)          # Testdaten mit Quadratzahlen erzeugen
pred=compute(net,t); y.test=as.vector(pred$net.result); sqrt.x=sqrt(t)
test=data.frame(cbind(sqrt.x,y.test))
plot(test,pch=19,col="red",xlim=c(0,14),ylim=c(0,14))
abline(0,1)           # Graphik:
##t
```



Im Trainingsbereich, vor allem zwischen 2 und 9, passt das kleine Modell recht gut. Mit mehr Trainingsdaten und mehr Neuronen könnte eine noch deutlich bessere Annäherung an die Quadratwurzel erreicht werden. Außerhalb des Trainingsbereiches funktioniert ein größeres Modell aber nicht besser.

Unser einfaches Modell konvergiert aufgrund der logistischen Aktivierungsfunktionen für große positive Werte gegen 11,15 (4,375 + 6,776, Modell-Obergrenze für die Quadratwurzel).

Neuronale Netze haben Schwierigkeiten mit Ereignissen außerhalb ihres Trainings-bereiches. Das trifft aber auch auf andere neue und alte Verfahren (siehe Polynome) zu.

Sofern genügend Trainingsdaten sowie Rechenpower vorhanden sind und eine Modellinterpretation unerheblich ist, gelten Neuronale Netze aktuell als das empfehlenswerteste und leistungsfähigste Prognoseverfahren.

Das wollen wir natürlich an unserer Regressions-Benchmark, den Hauspreisdaten, sehen. In diesem Fall verwenden wir ein neuronales Netz mit der Topologie 13-5-3-2-1, also den 13 Input-Variablen, der Output-Variablen Hauspreis und dazwischen drei verborgene Schichten. Für Modelle mit mehr als einer verborgenen Schicht wird oft der Begriff „deep learning“ verwendet. Nun, bringt das was?

```
##R 15.3
require(MASS)
set.seed(1)
smp=sample(1:nrow(Boston),nrow(Boston)/2)
train=Boston[smp,]
test=Boston[-smp,]
mins=apply(train,2,min)                    # Normalisierung/Skalierung
rng=apply(train,2,max) - apply(train,2,min)
sc_train=scale(train,mins,rng)
sc_test=scale(test, mins,rng)
formula=paste("medv ~", paste(colnames(Boston[1:13]),collapse='+'))
nn=neuralnet(formula, data=sc_train, hidden=c(5,3,2), linear.output=T)
predictions=compute(nn, sc_test[,1:13])
predicted_values=(predictions$net.result * rng[14]) + mins[14]
print(paste("RMSE:",sqrt(mean((test[,14] - predicted_values)^2))))   # Ergebnis: RMSE=3.334
##t
```

Das Ergebnis ist gut, der Standardfehler liegt bei den Testdaten auf dem Niveau von Random Forest und Boosting. Einen Quantensprung in der Prognosekraft kann man an diesem kleinen Beispieldaten-satz und Modell namens „Multi Layer Perceptron (MLP)“ zumindest noch nicht erkennen.

Zudem ist das Modell mit drei verborgenen Schichten für so einen kleinen und überwiegend linear transformierbaren Datensatz verdächtig tief. Da hier die Trainings-und Testdaten gleich groß sind, sollte man bei einem stabilen Modell durch das Vertauschen der Stichproben (oben Vorzeichen bei `[+/-smp,]` vertauschen) einen vergleichbar geringen Standardfehler erhalten. Das ist hier nicht der Fall, mit `RMSE: 4.6867` ist die Varianz fast doppelt so groß.
Hier könnte also mit Blick auf das Validierungsergebnis optimiert worden sein. Diese Art des Overfitting ist ein bekanntes Problem der öffentlichen Rangliste bei kaggle-Wettbewerben, siehe http://gregorypark.org/blog/Kaggle-Psychopathy-Postmortem/ .

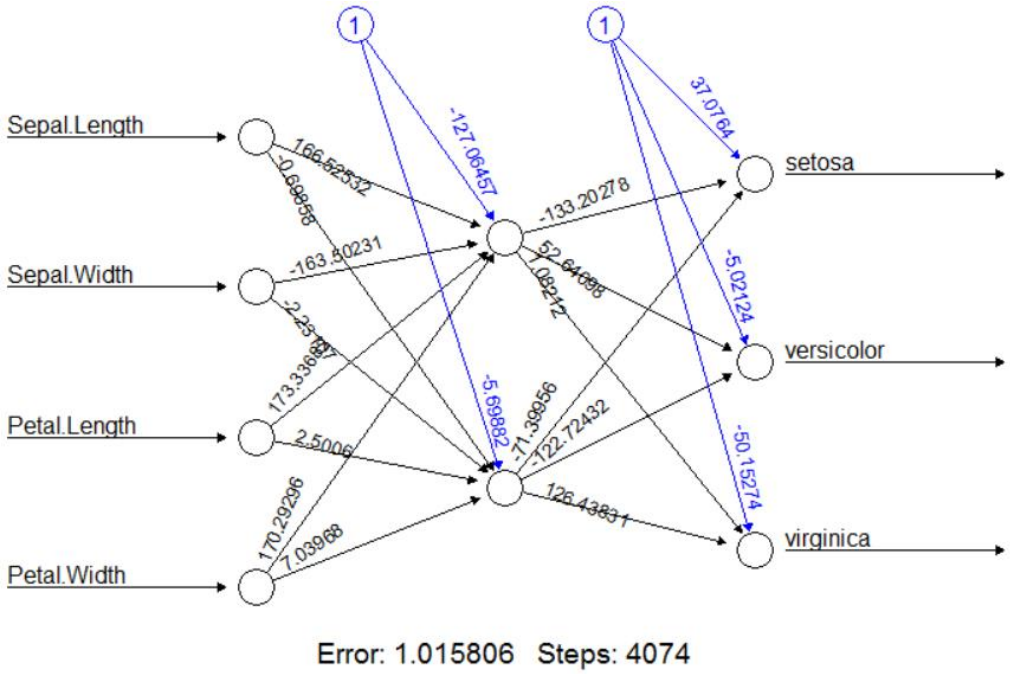
16. Künstliche Neuronale Netze (2): Multidimensionale Klassifikation, weitere Netztypen

Bevor wir zur Paradedisziplin der neuronalen Netze, der Bilderkennung kommen, wollen wir mit einem einfachen Neuronalen Netz eine multidimensionale Klassifikation am bekannten Iris-Datensatz, an dem R.A. Fisher seine Diskriminanzanalyse vorgestellt hat, durchführen.

Der Datensatz `iris` ist in der Bibliothek `MASS` enthalten und umfasst für 150 verschiedene Pflanzen je vier Messwerte zu Längen und Breiten der Blätter sowie die Angabe, um welche der drei Arten setosa, virginica oder versicolor es sich handelt.

Wir verwenden für das NN die gleiche Funktion und auch die gleiche hidden layer mit nur zwei Neuronen wie beim Wurzel-Beispiel im vorangegangenen Kapitel. Allerdings ergibt sich durch die vier Input-Variablen und drei Output-Variablen ein etwas komplexeres Neuronales Netz:

```
##R 16.1
require(MASS)
require(neuralnet)
spec = model.matrix( ~ Species - 1, data=iris )
colnames(spec) = c("setosa", "versicolor", "virginica")

# Datensatz teilen und normalisieren
set.seed(1)
smp = sample(1:nrow(iris), 100)
d_train = iris[smp, 1:4]
d_test  = iris[-smp, 1:4]
mins    = apply(d_train, 2, min)
maxs    = apply(d_train, 2, max)
sc_train = cbind(scale(d_train,mins,maxs-mins),spec[smp,])
sc_test  = cbind(scale(d_test ,mins,maxs-mins),spec[-smp,])
summary(sc_train)

# NN Multi-Layer-Perceptron (MPL) trainieren. 1 hidden layer, voll verknüpft
set.seed(1)
n=names(iris)
f=as.formula(paste("setosa+versicolor+virginica~",paste(n[!n %in% "Species"],collapse="+")))
net=neuralnet(f, data=sc_train, hidden=c(2), linear.output=F)
plot(net)          # Grafische Ausgabe der Netzstruktur und Gewichte:
##t
```



In den weiteren Schritten wird das neuronale Netz auf die Testdaten übertragen, die Klassifikationsentscheidung auf Basis des höchsten Prognosewertes getroffen und der Testfehler berechnet:

```
##R 16.2
pred = compute(net, sc_test[,1:4])
y_pred = apply(pred$net.result,1,which.max)  # Höchsten Wert auswählen,
y_true = apply(sc_test[,5:7],1,which.max)    # auf Testdaten anwenden und
confusion_matrix = table(y_true, y_pred)     # Genauigkeit berechnen:
print(confusion_matrix)
print(paste("Accuracy:",sum(diag(confusion_matrix))/sum(confusion_matrix)))
```

```
##t
> head(pred$net.result)
   [,1]          [,2]                                [,3]
1     1 0.005741347327 0.00000000000000000000001898786506
5     1 0.005795409208 0.00000000000000000000001880435072
6     1 0.005495686772 0.00000000000000000000001986795975
11    1 0.005881211319 0.00000000000000000000001852012400
12    1 0.005541638201 0.00000000000000000000001969731200
17    1 0.005650910538 0.00000000000000000000001930282601
> print(confusion_matrix)
      y_pred
y_true  1  2  3
     1 18  0  0
     2  0 14  0
     3  0  2 16
> print(paste("Accuracy:",sum(diag(confusion_matrix))/sum(confusion_matrix)))
[1] "Accuracy: 0.96"
```

Die Zeilensumme der Vorhersagewerte übersteigt leicht 100%. Das ist zwar ungefährlich, aber auch unschön und ließe sich durch Verwendung einer Softmax-Funktion vermeiden.

Während „iris setosa“ (1) von unserem kleinen Neuronalen Netz sicher erkannt wird, werden zwei „iris virginica“ (3) von unserem Modell fälschlich als „iris versicolor“ (2) klassifiziert.
Mit zwei Neuronen in der hidden layer beträgt die Genauigkeit bei den Testdaten damit 96%. Für die Berechnung der 19 Gewichte werden immerhin 4.074 Schritte benötigt. Mit jedem zusätzlichen Neuron erhöht sich hier die Anzahl der zu berechnenden Gewichte um #Input + #Output + Bias, also acht. Mehr Gewichte bedeuten nicht unbedingt mehr Schritte. Wenn das Modell durch die Hinzunahme eines weiteren Neurons schneller konvergiert, kann die Anzahl der Schritte sogar fallen.
Das ist hier der Fall, mit fünf Neuronen werden 43 Gewichte in 2.875 Schritten berechnet, die Genauigkeit beträgt ebenfalls 96%. Mit noch mehr Neuronen sinkt die Genauigkeit wieder.
Bei dieser kleinen Teststichprobe von nur 50 Pflanzen hängen die erzielten Ergebnisse allerdings stark vom gewählten Startwert des Zufallszahlengenerators ab.

Soweit zu NNs mit kleinen Datensätzen und der Funktion `neuralnet` samt schönen Visualisierungen von Netzstrukturen und Gewichten. `neuralnet` ist als Funktion in R programmiert, siehe Source-Code https://github.com/cran/neuralnet/blob/master/R/neuralnet.r . Es ist vergleichsweise übersichtlich, transparent und gut nachvollziehbar. Die Funktion `neuralnet` hat neben diesen Vorteilen auch die konzeptionellen Nachteile der speicherhungrigen in-memory-Ausführung auf nur einen Rechenkern. Zudem wird aktuell nicht die Aktivierungsfunktion ReLU („Rectified Linear Units“, typischerweise umgesetzt als softplus-Funktion) angeboten. Für große nichtlineare Datensätze und entsprechend große, tiefe Modelle brauchen wir eine sehr schnelle, parallel arbeitende und RAM-schonende Funktion, siehe Kapitel 19.

Neben den oben vorgestellten Feed-Forward-Netzen können rückgekoppelte, sogenannte rekurrente neuronale Netze (rnn) sehr leistungsfähig sein, siehe Kap. 22.

Eine recht neue und sehr erfolgreiche Regularisierungsmethode ist „dropout“, bei der ein beträchtlicher Teil der Neuronen zufallsgesteuert vom Training ausgeschlossen wird.

In diesem dynamischen Forschungs- und Anwendungsfeld werden gerade viele alte und neue Ideen mit enormer Rechenpower ausprobiert. Man kann es bereits als Ingenieursdisziplin ansehen, bei der das gerade technisch machbare am erfolgreichsten ist. Und das ändert sich aktuell sehr schnell.

17. Personalisierte Empfehlungen: Kollaboratives Filtern und Singulärwertzerlegung

Personalisierte Empfehlungen von Gütern, Informationen und Menschen sind online inzwischen allgegenwärtig. Kritiker sprechen bereits von einer Filterblase und sehen die Gefahr, dass Nutzer nur noch stark vorgefiltert Informationen, Meinungen und „Wahrheiten" wahrnehmen, weniger differenzieren können, selbstbestärkend einseitiger denken und zu extremeren Ansichten neigen.

Personalisierte Empfehlungen sind wirtschaftlich bedeutsam. So erzielt Amazon ein Drittel und Netflix sogar zwei Drittel seines Umsatzes über daten- und algorithmengetriebene Empfehlungen, für näheres siehe folgendes Video https://www.youtube.com/watch?v=bLhq63ygoU8 aus dem Jahr 2014 (Alternative: http://de.slideshare.net/xamat/recommender-systems-machine-learning-summer-school-2014-cmu). Spätestens seit der öffentlichkeitswirksamen Auslobung des Netflix-Preises von einer Million USD im Jahre 2006 für diejenigen, die eine 10% bessere Treffsicherheit bei Filmempfehlungen als Netflix selbst erzielen können, sind Empfehlungen ein bekanntes ML-Thema. Diesem Preis und dem entstandenen intensiven Wettbewerb werden einige ML-Fortschritte zugeschrieben.

Wir wollen nun Filmempfehlungen auf Basis der Filmdatenbank „MovieLens" erstellen. Das R-Package `recommenerlab` stellt mit der Datei `MovieLens` einen kleinen Auszug aus der Filmdatenbank bereit und enthält die Funktion `similarity`, mit der wir den Cosinus-Winkel zwischen Bewertungs-vektoren berechnen und damit Filme mit einem ähnlichen Publikumsgeschmack erkennen:

```
##R 17.1
library(recommenderlab)               # Die Filmbewertungen der Nutzer ansehen:
data(MovieLense)  # Daten bereitstellen. Die Datei enthält 99.392 Filmbewertungen
print(MovieLense) # Matrixgröße/-art anzeigen: n=943 User (Zeilen), p=1664 Filme (Spalten)
print(table(as.vector(as(MovieLense, "matrix"))))  # "Sterneverteilung" (1 bis 5) anzeigen
summary(colCounts(MovieLense))     # Filmbewertungen je Nutzer: Median 59.73
summary(rowCounts(MovieLense))     # Bewertungen je Film:       Median 105.4
print(colCounts(MovieLense[,1:5])) # Anzahl der Bewertungen für die ersten  5 Filme
print(colMeans(MovieLense)[1:5])   # Durchschnittsrating (Spaltendurchschnitt) -"-

# Kollaboratives Filtern: Cosinus-Ähnlichkeiten zu Film 1 (Toy Story) berechnen
cos.film1=similarity(MovieLense[,1],MovieLense[,-1],method="cosine",which="items")
colnames(cos.film1)[which(cos.film1>0.69)] # Filme mit der höchsten Ähnlichkeit. Ergebnis:
##t
[1] "Star Wars (1977)"          "Return of the Jedi (1983)"
```

Amazon-Mitarbeiter berichteten bereits im Jahre 2003 über die Anwendung des „Item-to-Item Collaborative Filtering" bei ihrem Empfehlungen. Siehe folgenden, viel zitierten Bericht https://www.cs.umd.edu/~samir/498/Amazon-Recommendations.pdf .

Als nächstes wollen wir vorhersagen, ob sich eine Person für einen bestimmten Film interessiert und den Erfolg diese Vorhersage überprüfen. Dazu verwenden wir die Singulärwertzerlegung (SVD), die beim Netflix-Preis als erfolgreiche Methode für Empfehlungen bekannt wurde, und zerlegen unsere n*p-Filmbewertungsmatrix A folgendermaßen: $A = U\ D\ V^T$. Die Komponenten der User-Matrix U werden dann in einem Random-Forest-Modell (Kap. 13) als erklärende Merkmale für das Ereignis „Film angesehen" verwendet. Die SVD führen wir mit der Funktion `irlba` aus dem gleichnamigen R-Package durch. Diese Funktion ist auch für große Matrizen geeignet, siehe https://www.youtube.com/watch?v=ipkuRqYT8_I .

```
##R 17.2
library(irlba) # Mindestanzahl an Bewertungen für Filme und Nutzer stellen:
rm=MovieLense[rowCounts(MovieLense)>10,colCounts(MovieLense)>50]
rm.n=normalize(rm,row=TRUE)  # Bewertungen je Nutzer standardisieren
rm.dm=as(rm.n,"matrix")      # Werte in eine "normale" (dense) Matrix schreiben
rm.dm[is.na(rm.dm)]=0        # Fehlende Werte auf 0 setzen.
film<-39                     # Film, hier "Star Wars", auswählen
print(paste("Choosen film:", colnames(rm.dm)[film]))
print(colCounts(rm[,film]))  # (583 von 943 Nutzern haben den Film bewertet)
label=as.factor(as.numeric(rm.dm[,film]!=0))  # Antwortvektor (1: Film gesehen) erzeugen
```

```
## Singulärwertzerlegung ohne den ausgewählten Film durchführen
SVD=irlba(rm.dm[,-film],nv=50,nu=50) # nv,nu: Anzahl rechte,linke Singulärvektoren
#rotation <- data.frame(movies=colnames(rm.dm[,-film]),SVD$v)
print(dim(rm.dm));print(dim(SVD$u));print(dim(SVD$v)) # Check der Matrixgrößen
print(length(SVD$d))

# Random Forest trainieren und testen, Prognosegüte bewerten
library(randomForest)
set.seed(1)
train=sample(1:length(label),500) # Trainingsstichprobe mit 500 von 943 Nutzern
u.t=as.data.frame(SVD$u[train,])  # Trainingsdaten mit den U-Komponenten (User)
rf=randomForest(label[train] ~.,data=u.t,importance=TRUE)
u.v=as.data.frame(SVD$u[-train,]) # Testdaten erzeugen
pred=predict(rf,newdata=u.v,n.trees=model$n.trees) # Testdaten scoren
c.m=table(label[-train],pred)     # Confusion Matrix berechnen und ausgeben
precision=c.m[2,2]/sum(c.m[,2]);recall=c.m[2,2]/sum(c.m[2,]);print(c.m)
print(paste("Precision:",round(precision,3),"  Recall:",round(recall,3))) # Ergebnis:
##t
[1] "Precision: 0.786   Recall: 0.867"
```

Auf Basis der anderen Filme, die eine Person angesehen und bewertet hat, kann man also recht gut vorhersagen, ob sich diese Person für einen bestimmten Film interessiert und entsprechende Empfehlungen machen.

Mit recht ähnlichen Methoden, wie sie oben für Filmempfehlungen beschrieben sind, werden die von Spotify vorgeschlagenen Playlists erstellt. Da häufig keine Bewertungsangaben des Hörers vorliegen wird der Musikgeschmack überwiegend auf Basis der angehörten Songs ermittelt. Nun gibt es deutlich mehr Musikstücke und Hörer als Filme und Filmkonsumenten. Daher ist die Bewertungs-matrix von der üppigen Größenordnung 10^7 Hörer mal 10^7 Songs. Von den Zellen diese Sparse-Matrix ist zwar nur ein verschwindend geringer Anteil besetzt, das sind immerhin 25 Milliarden Einträge. Näheres siehe: http://www.slideshare.net/erikbern/music-recommendations-mlconf-2014 (ab Folie 15).

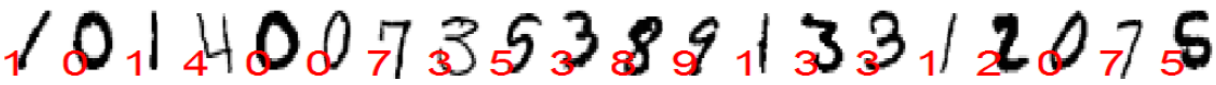
Die SVD setzt die Normalverteilungsannahme voraus. Diese Voraussetzung ist nicht immer gegeben. Das im Netflix-Wettbewerb ebenfalls erfolgreiche Verfahren „Restricted Boltzmann Machine“ kommt ohne diese Annahme aus. Es handelt sich dabei um ein spezielles neuronales Netzwerk.

18. Entziffern mit Random Forest und XGBOOST, Laufzeiten, kaggle.com

Bei den bisherigen Anwendungsbeispielen handelte es sich um recht kleine Datensätze. Die Anzahl der erforderlichen Berechnungen war entsprechend gering und daher die Laufzeit der Fits und Trainings unerheblich.

Das ändert sich nun, wir wenden uns der Bilderkennung zu. Ein viel getesteter Referenzdatensatz sind die 70.000 Bilder von handschriftlichen, sauber getrennten Ziffern der MNIST-Database (Mixed National Institute of Standards and Technology). Diese Bilddaten werden von kaggle.com als einfach einzulesende Trainings- und Testdaten mit einer Auflösung von 28x28 Bildpunkten bereitgestellt: https://www.kaggle.com/c/digit-recognizer/data (Alternative ohne Anmeldung: siehe Kap. 5).
Der greyscale-Pixelwert bewegt sich ganzzahlig zwischen 0 für weiß und 255 für schwarz. Wir lesen nun die Daten mit der Funktion `read_csv` aus dem Package `readr` ein und schauen uns die ersten 20 Ziffern an:

```
##R 18.1
library(readr)                                  # Nötig für read_csv
train=read_csv("D:/downloads/mnist_train.csv") # 42000 Images mit 784 Bildpunkten + label
test =read_csv("D:/downloads/mnist_test.csv")  # 28000 Images mit 784 Bildpunkten
par( mfrow = c(1,20)) # in RStudio Plot-Fenster entsprechend anpassen. Nun Pixel aufbereiten:
for (i in 1:20){ y = as.matrix(train[i, 2:785]); dim(y) = c(28, 28)
  image( y[,nrow(y):1], axes = FALSE, col = gray(255:0/255) )
  text( 0.2, 0, train[i,1], cex = 4, col = 2, pos = 3 ) } # und Ziffernwert rot überblenden:
##t
```



```
> table(train$label) # Verteilung der 42.000 Ziffern:

   0    1    2    3    4    5    6    7    8    9
4132 4684 4177 4351 4072 3795 4137 4401 4063 4188
```

Es ist keine weitere Datenaufbereitung, kein „feature engineering“ erforderlich. Es reicht ein gutes Modell und Rechenpower.

Das folgende R-Skript hat das Ziel, mit der Methode Random Forest (rf) ein Klassifikationsmodell mit 10 verschiedenen Klassen zu trainieren, das Modell auf die Testdaten anzuwenden und eine Score-Datei für die Teilnahme am Übungswettbewerb „Digit Recognizer“ auf kaggle.com zu erstellen.
Zu Vergleichszwecken wird die Laufzeit des Trainings gemessen. Aus Laufzeitgründen werden zunächst nur 10.000 Datensätze der Trainingsstichprobe verwendet. Die Trainingszeit kann trotzdem rechnerabhängig ein paar Minuten betragen.

```
##R 18.2                Achtung: Ein paar Minuten Laufzeit möglich. Für kurzen Test ntree=2 setzen
library(randomForest)
set.seed(1)
rows=sample(1:nrow(train),10000)               # Liste von 10000 Zufallszahlen
labels=as.factor(train[rows,]$label)           # Spalte „label“ enthält Ziffer
train=train[rows,-1]          # je Obs. 28*28 Bildpunkte (pixel0 bis pixel783), ohne label
tic <- proc.time()                             # Startzeit merken
rf=randomForest(train, labels, xtest=test, ntree=250)
print(proc.time() - tic)                       # Training-Laufzeit des Random Forest
pred=data.frame(ImageId=1:nrow(test), Label=levels(labels)[rf$test$predicted])
head(pred)
write_csv(pred,"D:/downloads/mnist_submission_rf1.csv")   # auf kaggle.com submitten
##t
```

Zur kaggle-Teilnahme:
Der Vortrag „Winning Data Science Competitions“ https://www.youtube.com/watch?v=ClAZQI_B4t8 von Jeong-Yoon Lee gibt einen Überblick über Wettbewerbe, nennt „best practices“ und erläutert

mit einem guten Schaubild um Minute 12 den Zusammenhang von Trainings-, Test- und Submissiondaten sowie die unterschiedlichen kaggle-Ranglisten (Public vs. Private Leaderboard).

Bei unserer ersten Submission auf https://www.kaggle.com/c/digit-recognizer/submissions/attach für den oben genannten Übungswettbewerb haben wir mit der hochgeladenen Datei einen Score von 0.951 erzielt. Das bedeutet, dass 5% der Ziffern von unserem Random Forest nicht korrekt vorhergesagt wurden. Für eine Ziffernerkennung ist das nicht gut, knapp 80% der aktuell über 1.000 Wettbewerbsteilnehmer haben eine bessere Prognose erzielt.

Die Verringerung der Fehlerrate von 5% auf unter 1% soll bei Apps, die Bild- oder Spracherkennung verwenden, den Unterschied zwischen Misserfolg und Welterfolg bedeuten können.

Daran wollen wir arbeiten. Also machen wir einen weiteren Versuch, nun mit allen Trainingsdaten und mit 1.000 Entscheidungsbäumen (Achtung: Das ist nur auf Power-PCs empfehlenswert).
Die Fehlklassifikationsrate fällt dadurch auf 3,3%, wir nähern uns von unten dem Mittelfeld des kaggle-Wettbewerbs. Allerdings mussten wir dafür schon lästig lange auf das Ergebnis warten und haben im Task Manager gesehen, dass die CPU-Auslastung am Windows-PC (rechnerabhängig) nur zwischen 12% und 30% liegt. Die Gesamtauslastung ist sogar umso schlechter, je mehr Rechenkerne der PC hat! Das ist auch nicht gut.

Dieses Problem von klassischen R-Funktionen, die nur auf einem einzigen PC-Rechenkern ausgeführt werden und dabei auch noch reichlich Arbeitsspeicher verlangen, hat auch die R-Funktion `gbm`.
XGBOOST hingegen ist hinter den R-Kulissen in C++ programmiert und nutzt die OpenMP-Schnittstelle für die Parallelverarbeitung der Prozessoren. Daher führen wir das folgende Boosting mit XGBOOST durch und lesen nun die MNIST-Daten mit der Standardfunktion `read.csv` ein.

```
##R 18.3          Achtung: Ein paar Minuten Laufzeit möglich. Für kurzen Test nrounds=2 setzen
library(xgboost)
train=read.csv("D:/downloads/kaggle-mnist_train.csv") # 42000 Images, 784 Bildpunkte + label
test =read.csv("D:/downloads/kaggle-mnist_test.csv")  # 28000 Images, 784 Bildpunkte
train.images=train[,-1]        # Bilddaten und Ziffern in verschiedene Objekte schreiben
train.digits=train[,1]

# Boosting Trees (default: booster="gbtree", eta=0.3, max_depth=6)
set.seed(1)
tic <- proc.time()             # Startzeit merken
mod=xgboost(data=xgb.DMatrix(model.matrix(~.,data=train.images),label=train.digits),
       num_class=10, nrounds=100, early.stop.round=3,
       params=list(objective="multi:softmax",eval_metric="merror"))
print(proc.time() - tic)       # Training-Laufzeit xgboost berechnen
pred=predict(mod,newdata=xgb.DMatrix(model.matrix(~.,data=test)))
write.csv(data.frame(ImageId=1:nrow(test),Label=pred),"mnist_xgb.csv",quote=F,row.names=F)
##t
```

Die CPU-Auslastung liegt mit `XGBOOST` bei konstant 100% auf allen Rechenkernen.
Nach 56 von maximal 100 vorgegebenen Runden greift das Early-Stop-Kriterium, hier bei drei aufeinander folgenden Runden mit weiterer Verschlechterung. Die Fehlklassifikationsrate beträgt gleich beim ersten Versuch nur 3,3%, der Trainingslauf ist ein Mehrfaches schneller als bei unseren rf-Versuchen. In einem weiteren Versuch mit der Evaluationsmetrik "mlogloss" sowie 250 Trainingsrunden und der proportional verlängerten Laufzeit wird die Fehlklassifikationsrate auf 2,7% gesenkt. Wir nähern uns damit dem oberen Drittel des Wettbewerbs. Für ein paar Minuten Laufzeit mit Standardeinstellungen ist das nicht übel.

XGBOOST stellt für die erforderliche Optimierung der Tuning-Parameter durch Cross-Validation die Funktion `xgb.cv` zur Verfügung. Auf Cross-Validation wird auch im oben genannten Video mit einem guten Schaubild eingegangen (ca. 57'). Gegen Ende des Videos wird das Ensemble von 64 Modellen (darunter 26 Gradient-Boosting-Modelle), das den KDD Cup 2015 gewonnen hat, erläutert.

19. Handschrifterkennung mit MXNet: Deep Learning und Convolutional Neural Nets

MXNet.io bezeichnet sich als flexible und effiziente Bibliothek für Deep Learning. MXNet wurde Ende 2015 vorgestellt und hat teilweise die gleichen Autoren wie XGBOOST, siehe https://arxiv.org/abs/1512.01274 . MXNet unterstützt aktuell sieben Programmier- und Skriptsprachen, darunter auch R. Es läuft auf Windows, Linux, OS X und auf den großen Cloud-Plattformen. Es rechnet sowohl auf CPUs als auch auf GPUs (Graphikkarten) und ist sehr schnell. Zur Wahl stehen Neuronale Netze von Typ „feed-forward", „recurrent" und „convolutional".

Mit MXNet wollen wir Deep-Learning-Methoden auf die MNIST-Handschriftdaten von kaggle.com anwenden. Dazu verwenden wir zwei Beispiele aus der MXNetR-Einführung von Tong He (der bereits XGBOOST auf R umgesetzt hat): http://dmlc.ml/rstats/2015/11/03/training-deep-net-with-R.html . Die Installationsprozedur des R-Packages `mxnet` ist etwas anders als bisher, für CPU-Computing aber noch sehr einfach umsetzbar.

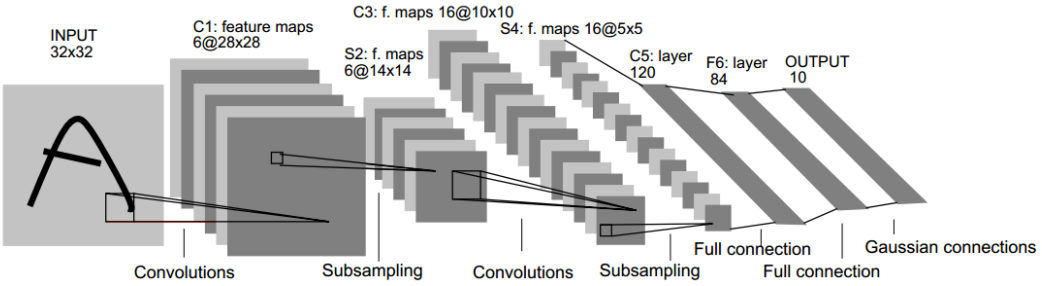
In unserer ersten Anwendung berechnen wir ein zweilagiges Multi Layer Perceptron mit 128 Neuronen in der ersten und 64 Neuronen in der zweiten hidden layer. Für beide Schichten verwenden wir die Aktivierungsfunktion ReLU, die zu schneller Konvergenz und zu einem schlanken Modell führt. Mit der Softmax-Funktion werden die 10 Outputs gleich auf 1 normiert.

```
##R 19.1         Achtung: Längere Laufzeit möglich. Für ersten Test num.nround=1 (statt 30) setzen
install.packages("drat", repos="https://cran.rstudio.com")
drat:::addRepo("dmlc")
install.packages("mxnet")
require(mxnet)
train <- read.csv('D:/downloads/kaggle-mnist_train.csv', header=TRUE)
test  <- read.csv('D:/downloads/kaggle-mnist_test.csv', header=TRUE)
train <- data.matrix(train); test <- data.matrix(test)
train.x <- train[,-1]      ; train.y <- train[,1]
train.x <- t(train.x/255)  ; test <- t(test/255)
table(train.y)

# Konfiguration des vollständig verknüpften Netzwerks
data <- mx.symbol.Variable("data")                                  # Input-Layer
fc1  <- mx.symbol.FullyConnected(data, name="fc1", num_hidden=128)
act1 <- mx.symbol.Activation(fc1, name="relu1", act_type="relu")
fc2  <- mx.symbol.FullyConnected(act1, name="fc2", num_hidden=64)
act2 <- mx.symbol.Activation(fc2, name="relu2", act_type="relu")
fc3  <- mx.symbol.FullyConnected(act2, name="fc3", num_hidden=10)   # Output-Layer
softmax=mx.symbol.SoftmaxOutput(fc3, name="sm")
mx.set.seed(1)
model <- mx.model.FeedForward.create(softmax, X=train.x, y=train.y, ctx= mx.cpu(),
       num.round=30, array.batch.size=100, learning.rate=0.07, momentum=0.9,
       eval.metric=mx.metric.accuracy, initializer=mx.init.uniform(0.07),
       epoch.end.callback=mx.callback.log.train.metric(100))

# Vorhersage und Submission
preds <- predict(model, test)
pred.label <- max.col(t(preds))-1; table(pred.label)
submission <- data.frame(ImageId=1:ncol(test), Label=pred.label)
write.csv(submission, file='mnist_mx1fc.csv', row.names=FALSE, quote=FALSE)
##t
```

Das nicht mehr ganz kleine Modell ist in Rekordzeit nach wenigen Sekunden trainiert. Ist es auch gut? Zur Beantwortung dieser Frage laden wir wieder die Score-Datei auf kaggle.com hoch. Ja, das Modell ist unser bisher bestes, wir haben nur noch 2% Fehlklassifikationen und nähern uns den Top 25%.

Unsere MNIST-Klassifikationsbemühungen schließen wir mit einem sogenannten Convolutional Neural Network (CNN) ab. CNNs sind bekannt für ihre gute Performance bei Bilderkennungsaufgaben. Sie nutzen den Umstand, dass räumlich dicht beieinander liegende Punkte in einem Zusammenhang stehen und hoch korreliert sind. Das geschieht durch lokale Verbindungsstrukturen,

gemeinsame Gewichte und ein räumliches Subsampling, siehe die Animation von Yann LeCun http://yann.lecun.com/exdb/lenet/ . Im Artikel von LeCun et al. wird das erfolgreiche LeNet-5 mit folgenden Schichten vorgestellt, siehe auch http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf:



Ein ähnliches CNN setzen wir nun, fast 20 Jahre später, mit MXNet um. Der Fortschritt besteht in der heutigen Rechenpower und der einfachen Umsetzung mit einer halben Seite R-Skript:

```
##R 19.2                Achtung: Längere Laufzeit möglich. Für ersten Test num.nround=1 (statt 20) setzen
Data= mx.symbol.Variable('data')                                  # Definition der Faltungen:
conv1=mx.symbol.Convolution(data=data, kernel=c(5,5), num_filter=20)        # first conv
tanh1=mx.symbol.Activation(data=conv1, act_type="tanh")
pool1=mx.symbol.Pooling(data=tanh1, pool_type="max", kernel=c(2,2), stride=c(2,2))
conv2=mx.symbol.Convolution(data=pool1, kernel=c(5,5), num_filter=50)       # second conv
tanh2=mx.symbol.Activation(data=conv2, act_type="tanh")
pool2=mx.symbol.Pooling(data=tanh2, pool_type="max", kernel=c(2,2), stride=c(2,2))
flatten=mx.symbol.Flatten(data=pool2)
fc1  =mx.symbol.FullyConnected(data=flatten, num_hidden=500)     # first full connection
tanh3=mx.symbol.Activation(data=fc1, act_type="tanh")
fc2=mx.symbol.FullyConnected(data=tanh3, num_hidden=10)          # second full connection
lenet= mx.symbol.SoftmaxOutput(data=fc2) # loss
train.array=train.x; dim(train.array)=c(28, 28, 1, ncol(train.x))
test.array=test    ; dim(test.array)=c(28, 28, 1, ncol(test))
mx.set.seed(0)
tic=proc.time()
model= mx.model.FeedForward.create(lenet, X=train.array, y=train.y,ctx=mx.cpu(),
       num.round=20, array.batch.size=100, learning.rate=0.05, momentum=0.9,
       wd=0.00001,eval.metric=mx.metric.accuracy,
       epoch.end.callback=mx.callback.log.train.metric(100))
print(proc.time() - tic)
preds=predict(model, test.array)
pred.label=max.col(t(preds))-1; table(pred.label)
submission <- data.frame(ImageId=1:ncol(test), Label=pred.label)
write.csv(submission, file='mnist_mx2conv.csv', row.names=FALSE, quote=FALSE)
##t
```

Das Modell läuft trotz der theoretischen Vorteile gegenüber dem voll verknüpften Modell deutlich langsamer durch und kann auch auf einen Power-PC bei voller Auslastung mehrere Minuten Rechenzeit benötigen. Aber es lohnt sich, wir sind im public leaderboard auf kaggle nun in den Top 15%, die Fehlklassifikationsrate schrumpft auf 0,9%!

Die weitere Verbesserung erfordert mühsames Feintuning und die Optimierung von Ensembles. Für Letzteres könnte auf Basis der vorliegenden Modelle ein Mix der CNN-Vorhersagen mit den besten Boosting- und rf-Vorhersagen erfolgen und eine gewichtete Entscheidung durchgeführt werden.

Kritiker bemängeln, dass derartig an die Aufgabe angepasste CNNs nicht zur Idee des „machine learning“ passen und propagieren allgemeinere Methoden wie SVMs (Kap. 12). Im oben genannten Artikel werden weitere Netztopograpien und als alternative Modelle auch SVMs untersucht. Danach können SVMs ohne weitere Anpassungen und mit vergleichbar gutem Klassifikationserfolg benutzt werden, sind allerdings deutlich langsamer und speicherhungriger. Daher verzichten wir hier auf eine Wettbewerbsteilnahme mit SVMs und widmen uns stattdessen der weiteren Verkürzung der Laufzeiten durch die Rechenpower von Graphikkarten.

20. GPU-Computing mit MXNet: Graphikkarten und Software

Aktuelle „Gaming"-Graphikkarten haben inzwischen mehrere Tausend Prozessorkerne. Sie eignen sich daher ganz hervorragend für die Parallelverarbeitung unserer Berechnungen. MXNet verwendet für die GPU-Nutzung die „CUDA deep learning library" cuDNN des Graphikartenherstellers NVidia, die allerdings aus lizenzrechtlichen Gründen nicht von MXNet ins R-Package eingebunden werden kann. Daher müssen wir nun einen weiten, aber auch lehrreichen Weg gehen. Wir benötigen:

- Microsoft Visual Studio 2013 (Community Edition, kostenlos),
- das NVidia CUDA Toolkit 8.0 (1,2 GB),
- das aktuelle MXNet package (R-package/ und nocudnn/) und
- die Bibliothek cuDNN samt Registierung als Entwickler.

Nun folgen wir der Anleitung http://mxnet.io/get_started/setup.html#installing-mxnet-on-a-gpu und führen die zahlreichen erforderlichen Installationsschritte mit etwas Glück und Erfahrung korrekt durch.

Wenn das alles geklappt hat wird es ganz einfach. Das R-Skript 19.2 muss lediglich an den rot markierten Stellen ergänzt bzw. verändert werden:

```
##R 20.1                    Benöigt Objekt lenet (R 19.2) und Daten aus Kap. 19
require(mxnet)
n.gpu=1
device.gpu=lapply( 0:(n.gpu-1),function(i) { mx.gpu(i) } ) mx.set.seed(0)
tic=proc.time()
model= mx.model.FeedForward.create(lenet, X=train.array, y=train.y, ctx=device.gpu,
       num.round=20, array.batch.size=100, learning.rate=0.05, momentum=0.9,
       wd=0.00001,eval.metric=mx.metric.accuracy,
       epoch.end.callback=mx.callback.log.train.metric(100))
print(proc.time() - tic)
preds=predict(model, test.array)
pred.label=max.col(t(preds))-1; table(pred.label)
submission <- data.frame(ImageId=1:ncol(test), Label=pred.label)
write.csv(submission, file='mnist_mx3gpu.csv', row.names=FALSE, quote=FALSE)
##t
```

Am Prognoseergebnis ändert sich dadurch nichts. Die Modellbildung sollte nun rund 10 mal so schnell erfolgt sein. Je nach Leistungsfähigkeitsverhältnis zwischen CPU und GPU sind auch andere Größenordnungen möglich. CPU-Processing kann sogar schneller sein, wie Kutkina und Feuerriegel in http://www.is.uni-freiburg.de/resources/r-oeffentlicher-zugriff/deep-learning-in-r/ zeigen, siehe den MNIST-Vergleich in der dortigen Tabelle 3.

In diesen Vergleich wird auch H2O einbezogen, unser nächstes Thema.

21. Big Data mit H2O: Java-Umgebung einrichten, neuronales Netz trainieren, R vs. FLOW

H2O beschreibt sich als schnelle und skalierbare „open source machine learning platform", siehe http://docs.h2o.ai/h2o/latest-stable/h2o-docs/welcome.html . Im Vergleich zu MXNet verfügt H2O über eine große Bandbreite von ML-Methoden und kann auf Rechnerclustern ausgeführt werden, nutzt jedoch keine GPUs. H20.ai tritt selbstbewußt auf und wendet sich mit seinen Dienstleistungen an Unternehmen, die in Open Source Analytics einen strategischen Vorteil sehen. Darunter auch Versicherungen, siehe Referenzvideos unter http://www.h2o.ai/customers/ . H2O benötigt das Java Runtime Environment und kann in Python, R und der eigenen Oberfläche FLOW verwendet werden.

Folgende Schritte sind erforderlich, um in RStudio das Package `h2o` nutzen zu können:

a) Java Runtime Environment (64 Bit, Version 8) von https://www.java.com/de/download/ herunterladen und installieren.

b) Das aktuelle h2o-zipfile von http://www.h2o.ai/download/ herunterladen, entzippen und im Terminal (Windows: „CMD“ aufrufen, mit „CD“ ins richtige Unterverzeichnis wechseln) H2O mit folgender Anweisung starten:
`java -jar h2o.jar`
Dieses Fenster bleibt bis zum Schluss geöffnet.

c) In RStudio folgende Installationen vornehmen:

```
##R 21.1
# Packages, die von H2O benötigt werden, installieren
if (! ("metods" %in% rownames(installed.packages()))) { install.packages("methods") }
if (! ("statmod" %in% rownames(installed.packages()))) { install.packages("statmod") }
if (! ("stats" %in% rownames(installed.packages()))) { install.packages("stats") }
if (! ("graphics" %in% rownames(installed.packages()))) { install.packages("graphics") }
if (! ("RCurl" %in% rownames(installed.packages()))) { install.packages("RCurl") }
if (! ("jsonlite" %in% rownames(installed.packages()))) { install.packages("jsonlite") }
if (! ("tools" %in% rownames(installed.packages()))) { install.packages("tools") }
if (! ("utils" %in% rownames(installed.packages()))) { install.packages("utils") }

# Das H2O-Package herunterladen, installieren und initialisieren (Stand 15.10.2016)
install.packages("h2o",type="source",repos=(c("http://h2o-release.s3.amazonaws.com/h2o/rel-turing/8/R")))
library(h2o)
localH2O = h2o.init(nthreads=-1)    # -1: alle Kerne nutzen
demo(h2o.kmeans)                    # Demo starten, H2O kennen lernen
##t
```

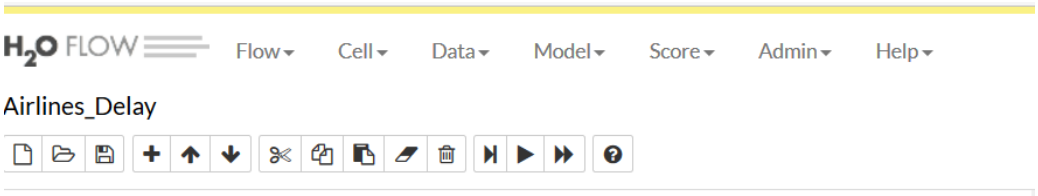
Die Versionszyklen sind noch sehr kurz, siehe https://github.com/h2oai/h2o-3/blob/master/Changes.md

H20 stellt einige Tutorials zur Verfügung, darunter eins über die Vorhersage von Flugverspätungen: https://github.com/h2oai/h2o-tutorials/blob/master/tutorials/intro-to-r-python-flow/intro-to-r.md.R .
Kurzform: Die Airlines-Daten aus /tutorials/data/ herunterladen und folgendes R-Skript ausführen:

```
##R 21.2      Neuronales Netz mit H2O trainieren (Anmerkung: Keine Testdaten)
airlines.hex <- h2o.importFile(path = normalizePath("allyears2k.csv"), destination_frame = "allyears2k.hex")
summary(airlines.hex)              # Überblick über die Airlines-Daten
y <- "IsDepDelayed"                # Kennzeichen für Flugverspätung
x <- c("Dest","Origin","DayofMonth","Year","UniqueCarrier","DayOfWeek","Month","Distance")
dl_model <- h2o.deeplearning(x = x, y = y, training_frame = airlines.hex, distribution = "bernoulli",
        model_id = "H2O_NN_R",epochs = 100, hidden = c(200,200),
        target_ratio_comm_to_comp = 0.02, seed = 12345, variable_importances = T)
auc2 <- h2o.auc(object = dl_model)
print(paste0("AUC of the training set : ", round(auc2, 4)))
print(h2o.varimp(dl_model))
print(h2o.scoreHistory(dl_model))
##t
```

Die Analyse wird über Java an R vorbei ausgeführt. Die Daten liegen nicht in R vor, es wird hier lediglich R-Syntax verwendet und damit R-Nutzern die Möglichkeit geboten, in einer vertrauten Weise die Vorteile von H20 zu nutzen. Die entstandenen Ergebnisdaten können bei Bedarf durch h2o-Funktionen ins „normale“ R überführt und dort weiter verarbeitet werden.

Im Airline-Beispiel wird R eigentlich nicht benötigt. Als Alternative zu R kann nach Schritt b) die Analyse-Oberfläche H2O FLOW über den Browser http://localhost:54321/flow/index.html gestartet und die Analyse über die Click-Oberfläche durchgeführt werden:



H20 kann auf Big-Data-Systemen wie Apache Spark und Hadoop sowie in Cloud-Computing-Umgebungen wie Amazon EC2 ausgeführt werden.

22. Abschluss mit und ohne R

Zum Abschluss möchten wir noch kurz auf einige interessante ML-Themen, die nicht von R abhängen oder deren Umsetzung den Rahmen gesprengt hätte, eingehen.

Ensembles
Unter http://mlwave.com/kaggle-ensembling-guide/ gibt es eine sehr einfache und ansprechende Beschreibung der Ensembling-Ansätze für Wettbewerbe auf kaggle.com. Im dort beschriebenen kleinen Umfang können Ensembles spürbare Verbesserungen erzielen. Inzwischen findet bei den Wettbewerben im Kampf um winzige Verbesserungen häufig eine Materialschlacht mit umfangreichen Ensembles statt, deren praktische Umsetzbarkeit fraglich ist.

Gesichtserkennung
Im Tutorial https://www.kaggle.com/c/facial-keypoints-detection wird gezeigt, wie man mit R in Gesichtern Schlüsselpunkte erkennen kann. Der Gesamtprozess der automatischen Gesichtserkennung und Identifizierung besteht aus den Schritten Gesichter finden, Gesichter ausrichten, Gesichtsmerkmale ermitteln und abgleichen und schließlich den Namen dazu finden, siehe den Überblick in Part 4 von https://medium.com/@ageitgey . Im Analytischen Kern wird dabei ein CNN mit mehreren hundert Millionen Gesichterbildern trainiert.

Github
GitHub.com stellt Softwareentwicklungsprojekte im Web auf seinen Servern bereit. Die Nutzer stehen dabei mit ihren Quellcodes im Mittelpunkt und bilden ein soziales Netz. Für eine lockere Kurzbeschreibung siehe http://t3n.de/news/eigentlich-github-472886/ . Eine Mitmachanleitung befindet sich hier: https://guides.github.com/activities/hello-world/. GitHub ist für öffentliche Projekte kostenlos. Zahlreiche Entwicklerinnen und Entwickler nutzen GitHub für ML-Projekte.

Linux
Einige ML-Bibliotheken und Programme wie H20 funktionieren nur oder am besten mit dem Betriebssytem Linux. Für unsere Zwecke und übliche PCs ist die Linux-Distribution Ubuntu (64 bit) gut geeignet. Falls Sie über einen alten, nicht mehr benötigten PC oder Laptop verfügen, können Sie das gefahrlos ausprobieren und Linux parallel zu Windows oder als alleiniges System installieren, siehe http://www.pcwelt.de/ratgeber/Linux_neben_Windows_installieren_-_so_geht_s-Multiboot-8634306.html .

Python
Python ist eine interpretierte höhere Programmiersprache und hat eine sehr aktive Entwicklergemeinde, die mächtige ML-Bibliotheken erstellt hat und weiter entwickelt. Wer ernsthaft Machine Learning betreiben möchte, kommt an Python als Ergänzung zu R kaum vorbei. Die Einstiegshürde für Python ist höher als für R. Die parallel exisierenden Versionswelten 2.7x und 3.x können Anfängern den Start erschweren.

Tensorflow, Übersetzungen
Die Open-Scource ML-Bibliothek Tensorflow wurde in November 2015 vom Google Brain Team herausgegeben und in kurzer Zeit zum am meisten gefolgten und kopierten ML-Projekt auf GitHub. Zum Kennenlernen gibt es gleich zwei Tutorials auf Basis der MNIST-Daten und am Ende eine ziemlich coole „Tinker"-Anwendung, siehe https://www.tensorflow.org/versions/r0.11/get_started/index.html .
Im Zusammenhang mit Tensorflow berichtet Google von großen Fortschritten bei der maschinellen Übersetzung von Sprache, siehe https://research.googleblog.com/2016/09/a-neural-network-for-machine.html .
Bei der Modellbildung werden rekurrente neurale Netze inzwischen mit ganzen Sätzen trainiert.

Vielen Dank für Ihr Interesse!